Planning and Optimization

A1. Organizational Matters

Malte Helmert and Gabriele Röger

Universität Basel

September 17, 2025

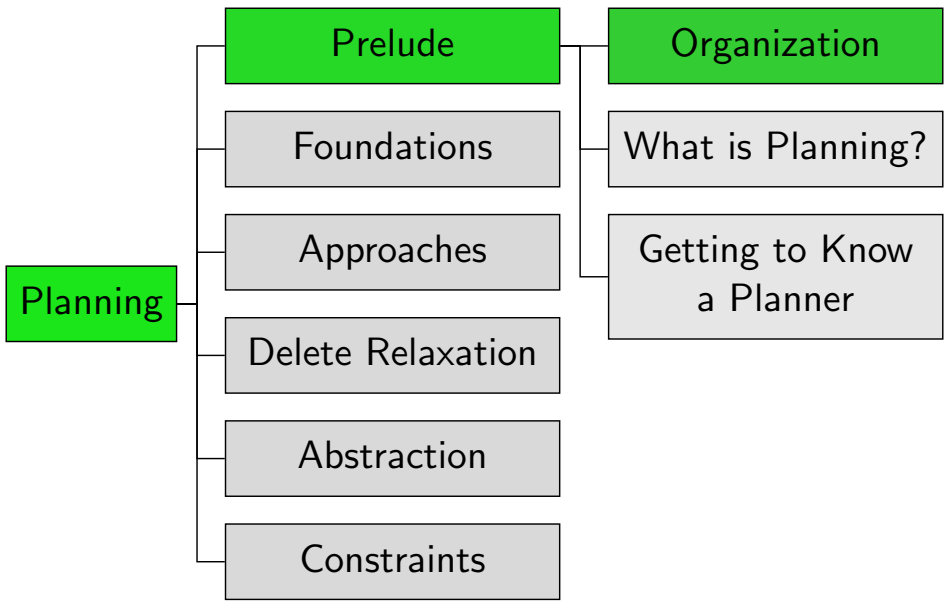
Planning and Optimization
September 17, 2025 — A1. Organizational Matters

A1.1 People & Coordinates

A1.2 Target Audience & Rules

A1.3 Course Content

Content of the Course



A1.1 People & Coordinates

People: Lecturers

Malte Helmert

Gabriele Röger

Lecturers

Malte Helmert

- email: malte.helmert@unibas.ch
- office: room 06.004, Spiegelgasse 1

Gabriele Röger

- email: gabriele.roeger@unibas.ch
- office: room 04.005, Spiegelgasse 1

People: Assistant

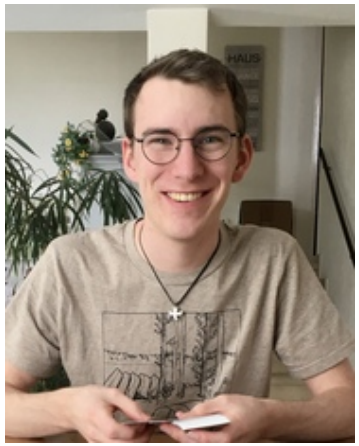
Tanja Schindler

Assistant

Tanja Schindler

- email: tanja.schindler@unibas.ch
- office: room 04.005, Spiegelgasse 1

People: Tutors



Clemens Büchner

Esther Mugdan

Tutors

Clemens Büchner

- email: `clemens.buechner@unibas.ch`
- office: room 04.001, Spiegelgasse 5

Esther Mugdan

- email: `esther.mugdan@unibas.ch`
- office: room 04.001, Spiegelgasse 5

Time & Place

Lectures

- time: Mon 14:15–16:00, Wed 14:15–16:00
- place: room 00.003, Spiegelgasse 1

Exercise Sessions

- time: Wed 16:15–18:00
- place: room 00.003, Spiegelgasse 1

first exercise session: today

Communication Channels

- lecture sessions (Mon, Wed)
- exercise sessions (Wed)
- course homepage
- ADAM workspace
- Discord server (invitation link on ADAM workspace)
- email

registration:

- `https://services.unibas.ch/`
- Please register today to receive all course-related emails!

Planning and Optimization Course on the Web

Course Homepage

`https://dmi.unibas.ch/en/studies/computer-science/course-offer-fall-semester-25/lecture-planning-and-optimization/`

- course information
- slides
- link to ADAM workspace
- bonus materials (not relevant for the exam)

A1.2 Target Audience & Rules

Target Audience

target audience:

- M.Sc. Computer Science
  - Major in Machine Intelligence:
    module Concepts of Machine Intelligence
    module Methods of Machine Intelligence
  - Major in Distributed Systems:
    module Applications of Distributed Systems
- M.A. Computer Science ("Master-Studienfach")
  module Concepts of Machine Intelligence
- M.Sc. Data Science: module Electives in Data Science
- other students welcome

Prerequisites

prerequisites:

- general computer science background: good knowledge of
  - algorithms and data structures
  - complexity theory
  - mathematical logic
  - programming
- background in Artificial Intelligence:
  - Foundations of Artificial Intelligence course (13548)
  - in particular chapters on state-space search

Gaps?

$\rightsquigarrow$ talk to us to discuss a self-study plan to catch up

Exam

- written examination (105 min)
- date and time: January 28, 14:00–16:00
- place: Biozentrum, room U1.131
- 8 ECTS credits
- admission to exam: 50% of the exercise marks
- final grade based on exam exclusively
- no repeat exam (except in case of illness)

Exercise Sheets

exercise sheets (homework assignments):

- solved in groups of two or three ($3 < 4$), submitted in ADAM
- weekly homework assignments
  - released Monday before the lecture
  - have questions or need help?
    $\rightsquigarrow$ assistance provided in Wednesday exercises
  - not sure if you need help?
    $\rightsquigarrow$ start before Wednesday!
  - due following Monday at 23:59
- mixture of theory, programming and experiments
- range from basic understanding to research-oriented

Programming Exercises

programming exercises:

- part of regular assignments
- solutions that obviously do not work: 0 marks
- work with existing C++ and Python code

Exercise Sessions

exercise sessions:

- ask questions about current assignments (and course)
- work on homework assignments
- discuss past homework assignments

Plagiarism

Plagiarism

Plagiarism is presenting someone else's work, ideas, or words as your own, without proper attribution.

For example:

- Using someone's text without citation
- Paraphrasing too closely
- Using information from a source without attribution
- Passing off AI-generated content as your own original work

Long-term impact:

- You undermine your own learning.
- You start to lose confidence in your ability to think, write, and solve problems independently.
- Damage to academic reputation and professional consequences in future careers

Plagiarism in Exercises

- You may discuss material from the course, including the exercise assignments, with your peers.
- But: You have to independently write down your exercise solutions (in your team).
- Help from an LLM is acceptable to the same extent as it is acceptable from someone who is not a member of your team.

Immediate consequences of plagiarism:

- 0 marks for the exercise sheet (first time)
- exclusion from exam (second time)

If in doubt: check with us what is (and isn't) OK before submitting
Exercises too difficult? We are happy to help!

Special Needs?

- We (and the university) strive for equality of students with disabilities or chronic illnesses.
- Contact the lecturers for small adaptations.
- Contact the Students Without Barriers (StoB) service point for general adaptations and disadvantage compensation.

A1.3 Course Content

Learning Objectives

Learning Objectives

- get to know theoretical and algorithmic foundations of classical planning and work on practical implementations
- understand fundamental concepts underlying modern planning algorithms and theoretical relationships that connect them
- become equipped to understand research papers and conduct projects in this area

Course Material

course material:

- slides (online)
- no textbook
- additional material on request

Git Repository

- We use a git repository for programming exercises and for demos during the lecture.
- Setting up the repository is your first task for the exercises.

Demo Examples

When working with the repository, go to its base directory:

Base Directory for Demos and Exercises

```
$ cd planopt-hs25
```

One-time demo set-up (from the base directory)
if the necessary software is installed on your machine:

Demo Set-Up

```
$ cd demo/fast-downward
$ ./build.py
```

Under Construction...

- Advanced courses are close to the frontiers of research and therefore constantly change.
- We are always happy about feedback, corrections and suggestions!

Planning and Optimization

A2. What is Planning?

Malte Helmert and Gabriele Röger

Universität Basel

September 17, 2025

Planning and Optimization
September 17, 2025 — A2. What is Planning?

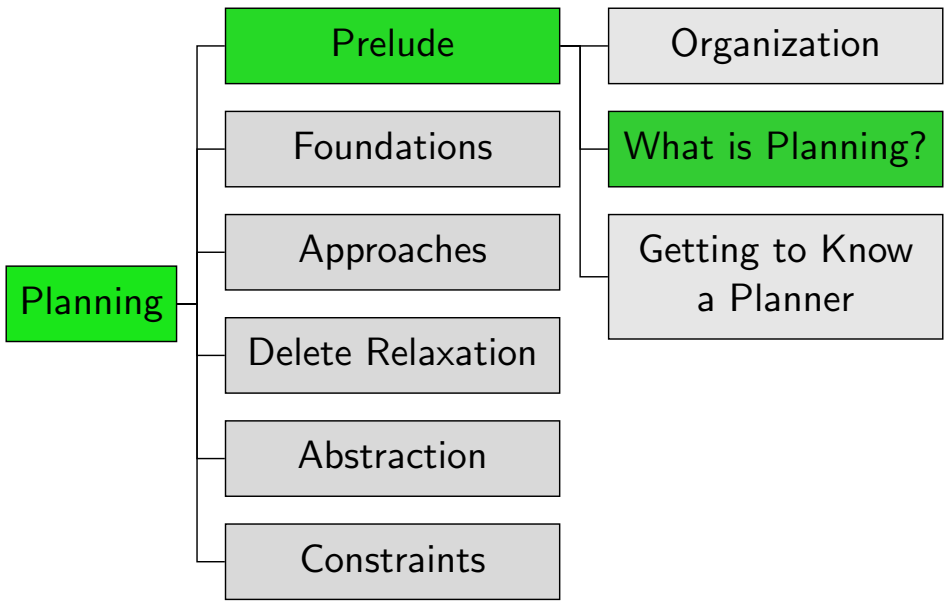
A2.1 Planning

A2.2 Planning Task Examples

A2.3 How Hard is Planning?

A2.4 Summary

Content of the Course



Before We Start...

Prelude (Chapters A1–A3): very high-level intro to planning

- our goal: give you a little feeling what planning is about
- preface to the actual course

$\rightsquigarrow$ main course content (beginning with Chapter B1)
will be mathematically formal and rigorous

- You can ignore the prelude when preparing for the exam.

A2.1 Planning

General Problem Solving

Wikipedia: General Problem Solver

General Problem Solver (GPS) was a computer program created in 1959 by Herbert Simon, J.C. Shaw, and Allen Newell intended to work as a universal problem solver machine.

Any formalized symbolic problem can be solved, in principle, by GPS. [...]

GPS was the first computer program which separated its knowledge of problems (rules represented as input data) from its strategy of how to solve problems (a generic solver engine).

$\rightsquigarrow$ these days called "domain-independent automated planning"
$\rightsquigarrow$ this is what the course is about

So What is Domain-Independent Automated Planning?

Automated Planning (Pithy Definition)

"Planning is the art and practice of thinking before acting."

— Patrik Haslum

Automated Planning (More Technical Definition)

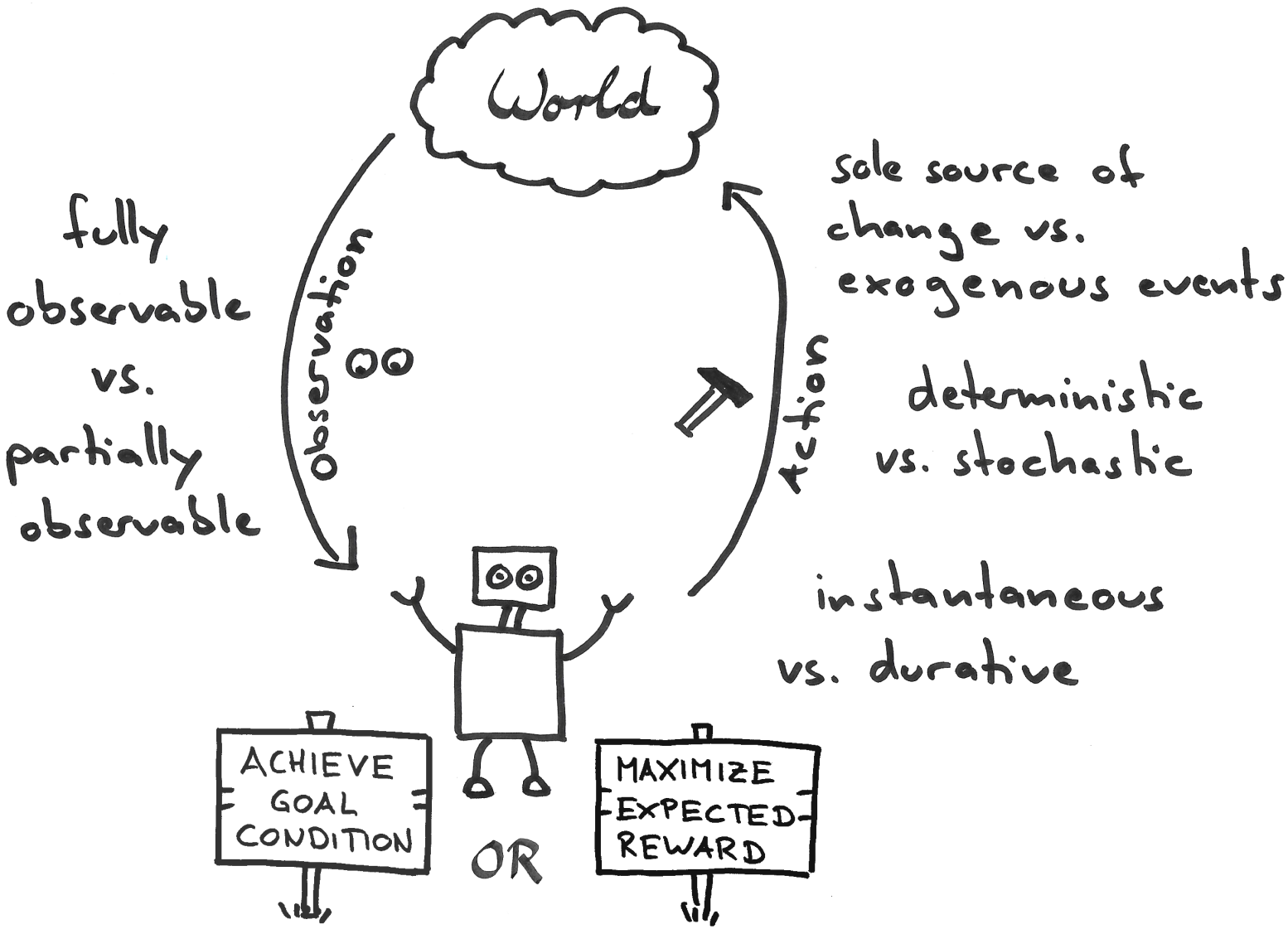
"Selecting a goal-leading course of action
based on a high-level description of the world."

— Jörg Hoffmann

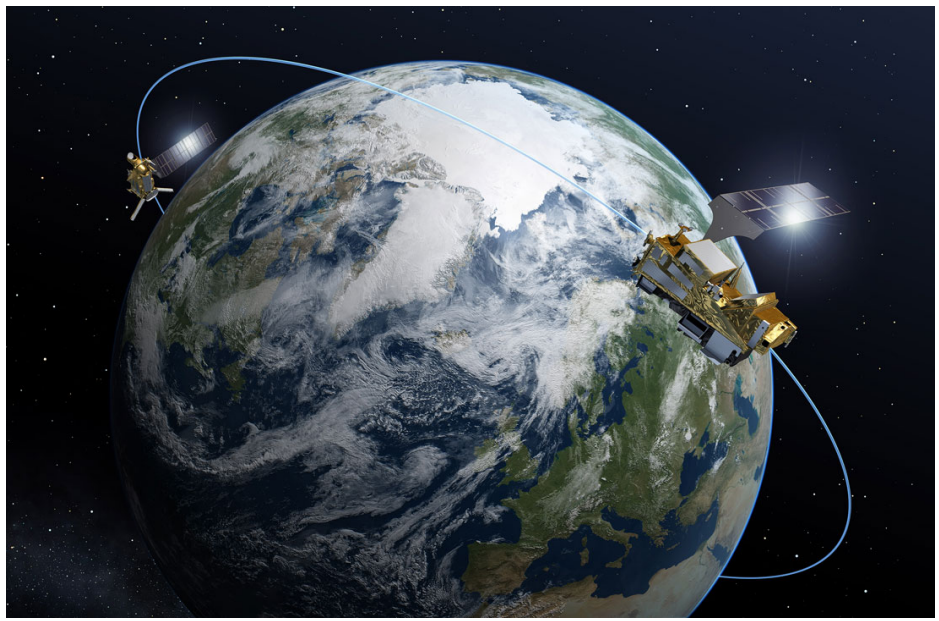
Domain-Independence of Automated Planning

Create one planning algorithm that performs sufficiently well on many application domains (including future ones).

General Perspective on Planning

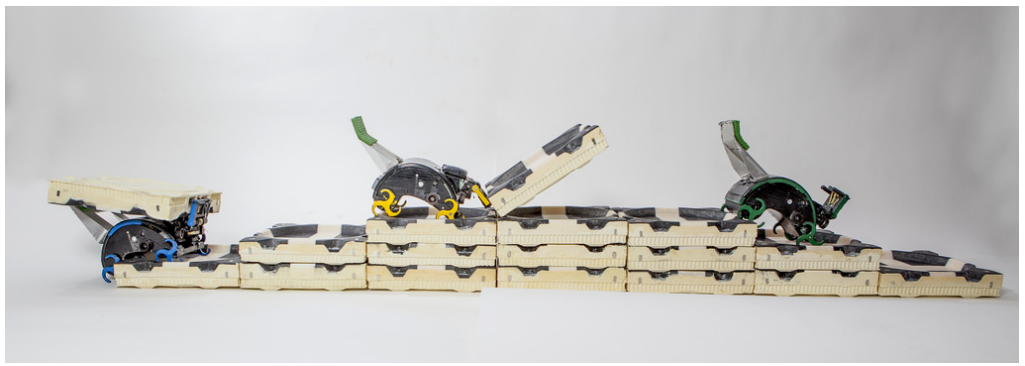


Example: Earth Observation



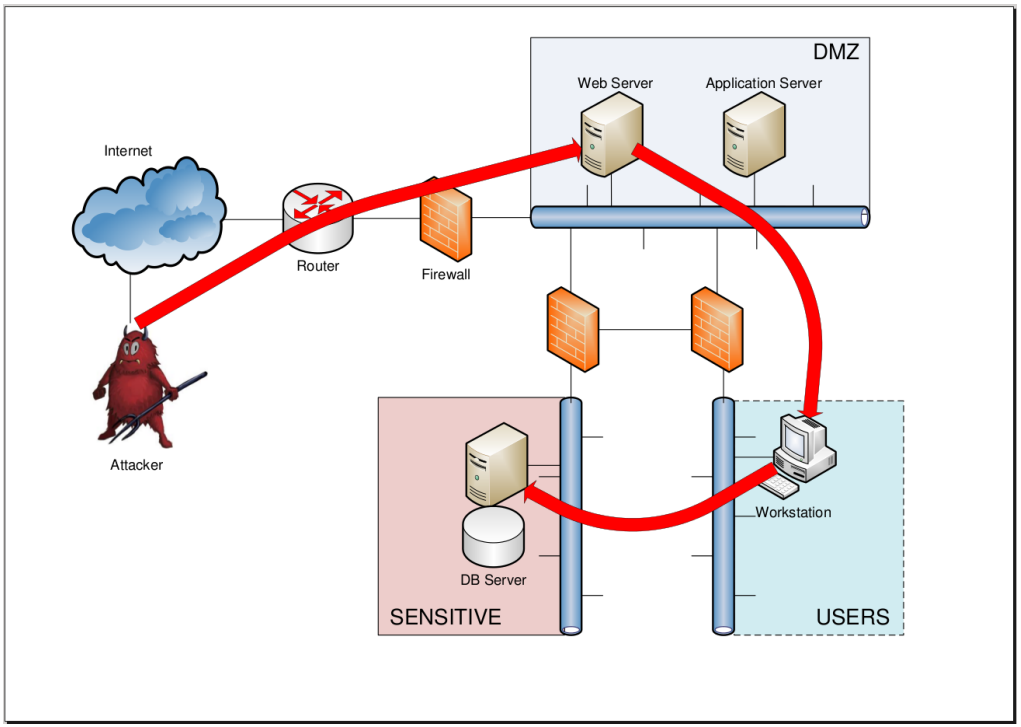
- satellite takes images of patches on Earth
- use weather forecast to optimize probability of high-quality images

Example: Termes



Harvard TERMES robots, based on termites

Example: Cybersecurity

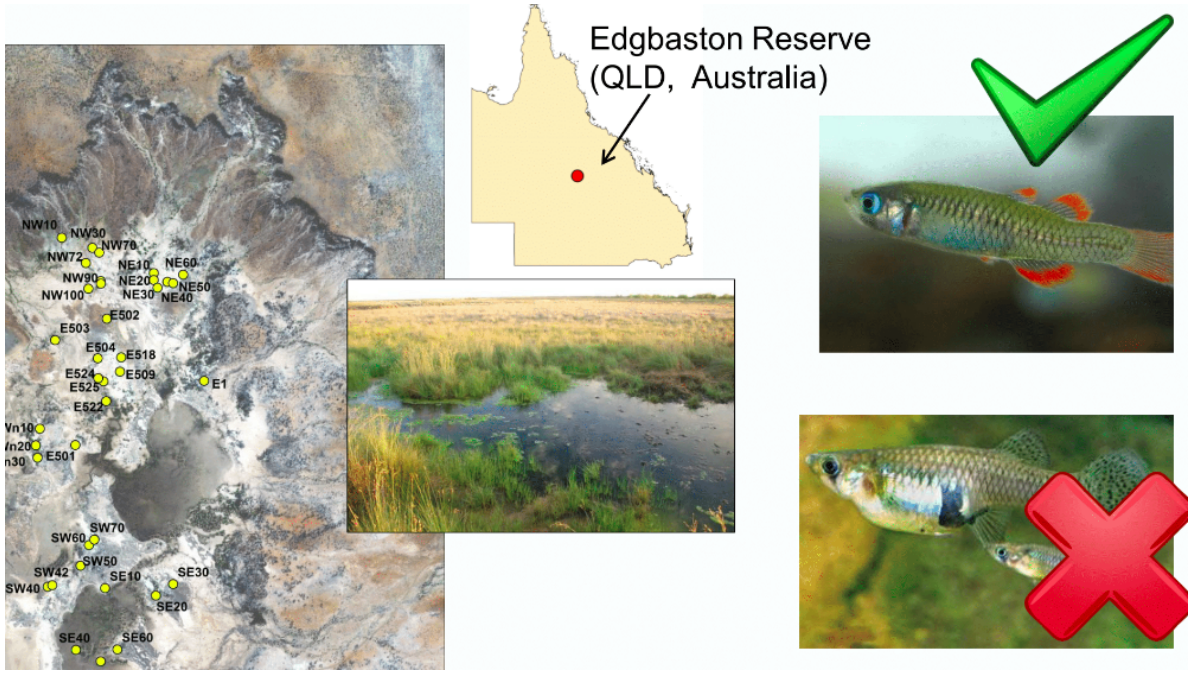


CALDERA automated adversary emulation system

Example: Intelligent Greenhouse

photo © LemnaTec GmbH

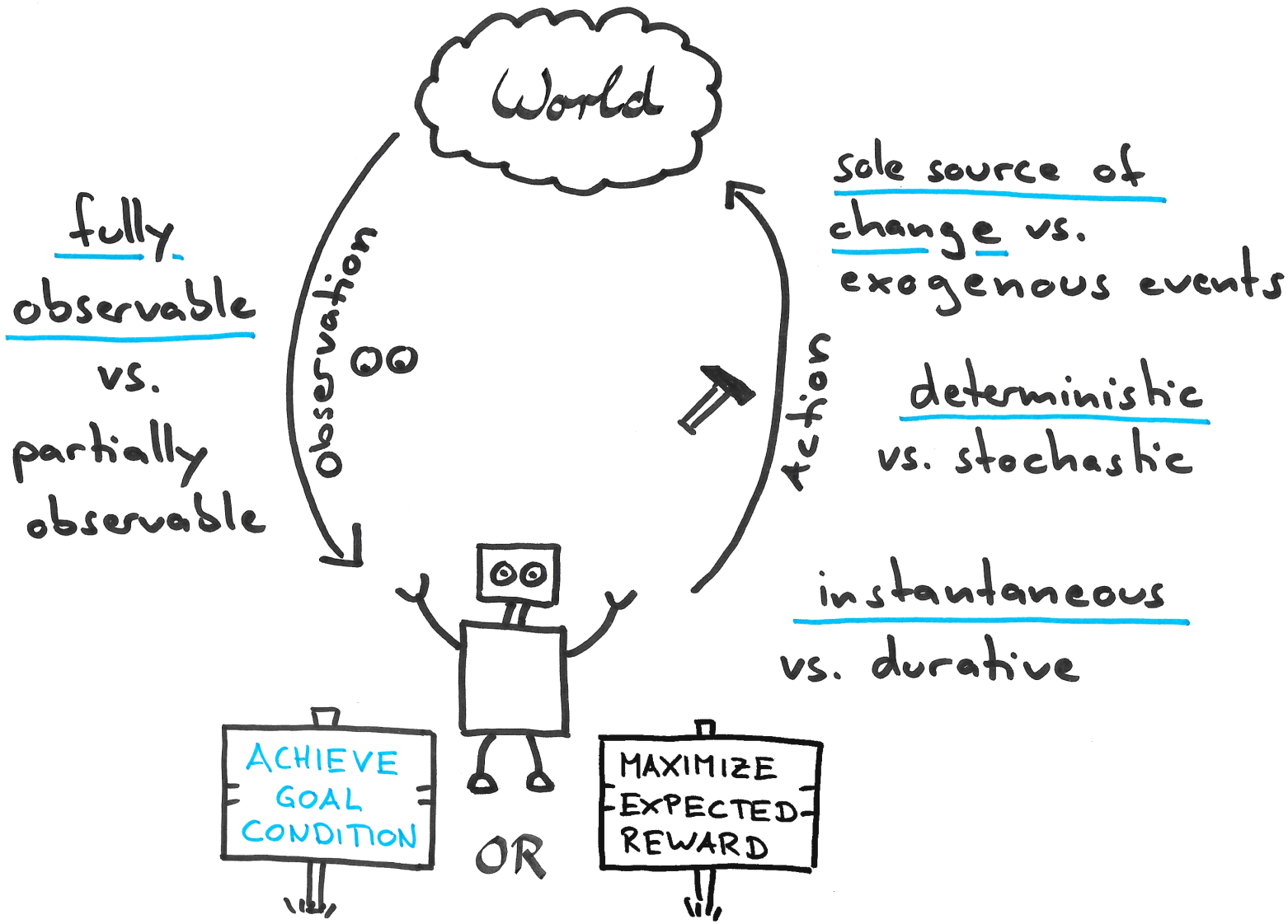
Example: Red-finned Blue-eye



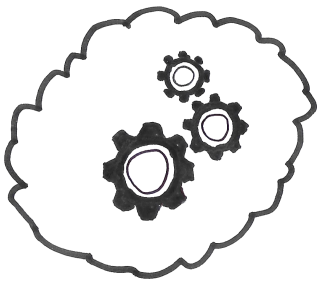
Picture by Iadine Chadès

- red-finned blue-eye population threatened by gambusia
- springs connected probabilistically during rain season
- find strategy to save red-finned blue-eye from extinction

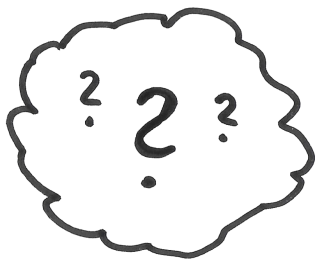
Classical Planning



Model-based vs. Data-driven Approaches



Model-based approaches know
the "inner workings" of the world
$\rightsquigarrow$ reasoning



Data-driven approaches rely only
on collected data from a black-box world
$\rightsquigarrow$ learning

We focus on model-based approaches.

Planning Tasks

input to a planning algorithm: planning task

- initial state of the world
- actions that change the state
- goal to be achieved

output of a planning algorithm:

- plan: sequence of actions taking initial state to a goal state
- or confirmation that no plan exists

$\rightsquigarrow$ formal definitions later in the course

The Planning Research Landscape

- one of the major subfields of Artificial Intelligence (AI)
- represented at major AI conferences (IJCAI, AAAI, ECAI)
- annual specialized conference ICAPS ($\approx$ 250 participants)
- major journals: general AI journals (AIJ, JAIR)

Classical Planning

This course covers classical planning:

- offline (static)
- discrete
- deterministic
- fully observable
- single-agent
- sequential (plans are action sequences)
- domain-independent

This is just one facet of planning.

Many others are studied in AI. Algorithmic ideas often (but not always) translate well to more general problems.

More General Planning Topics

More general kinds of planning include:

- ~~offline~~: online planning; planning and execution
- ~~discrete~~: continuous planning (e.g., real-time/hybrid systems)
- ~~deterministic~~: FOND planning; probabilistic planning
- ~~single-agent~~: multi-agent planning; general game playing; game-theoretic planning
- ~~fully observable~~: POND planning; conformant planning
- ~~sequential~~: e.g., temporal planning

Domain-dependent planning problems in AI include:

- pathfinding, including grid-based and multi-agent (MAPF)
- continuous motion planning

A2.2 Planning Task Examples

Example: The Seven Bridges of Königsberg

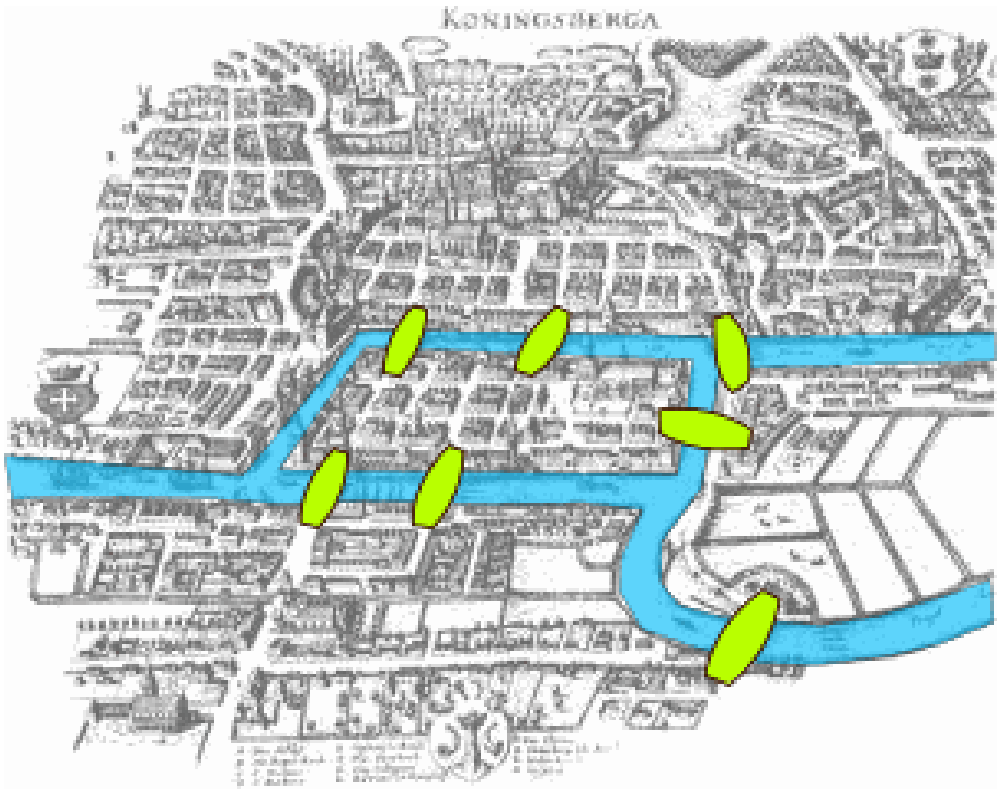


image credits: Bogdan Giuşcă (public domain)

Demo

```
$ ls demo/koenigsberg
```

Example: Intelligent Greenhouse

photo © LemnaTec GmbH

Demo

```
$ ls demo/ipc/scanalyzer-08-strips
```

Example: FreeCell

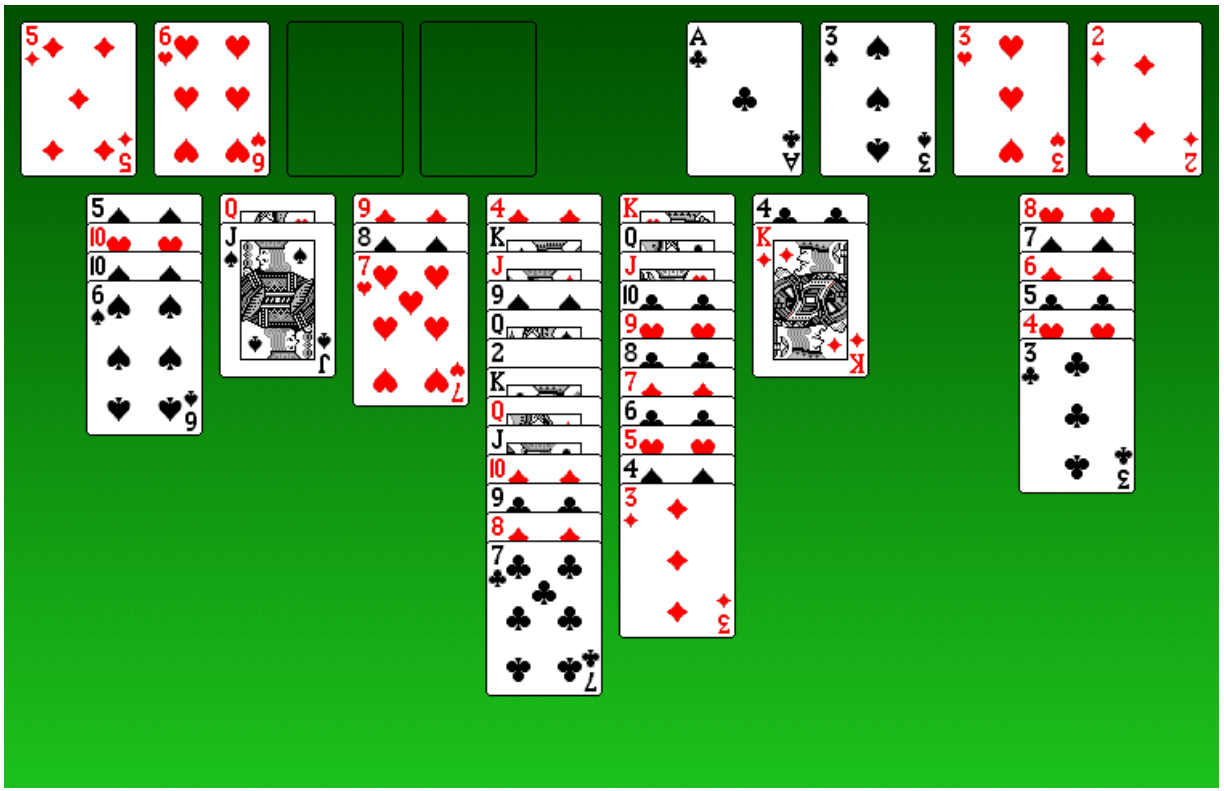


image credits: GNOME Project (GNU General Public License)

Demo Material

```
$ ls demo/ipc/freecell
```

Many More Examples

Demo

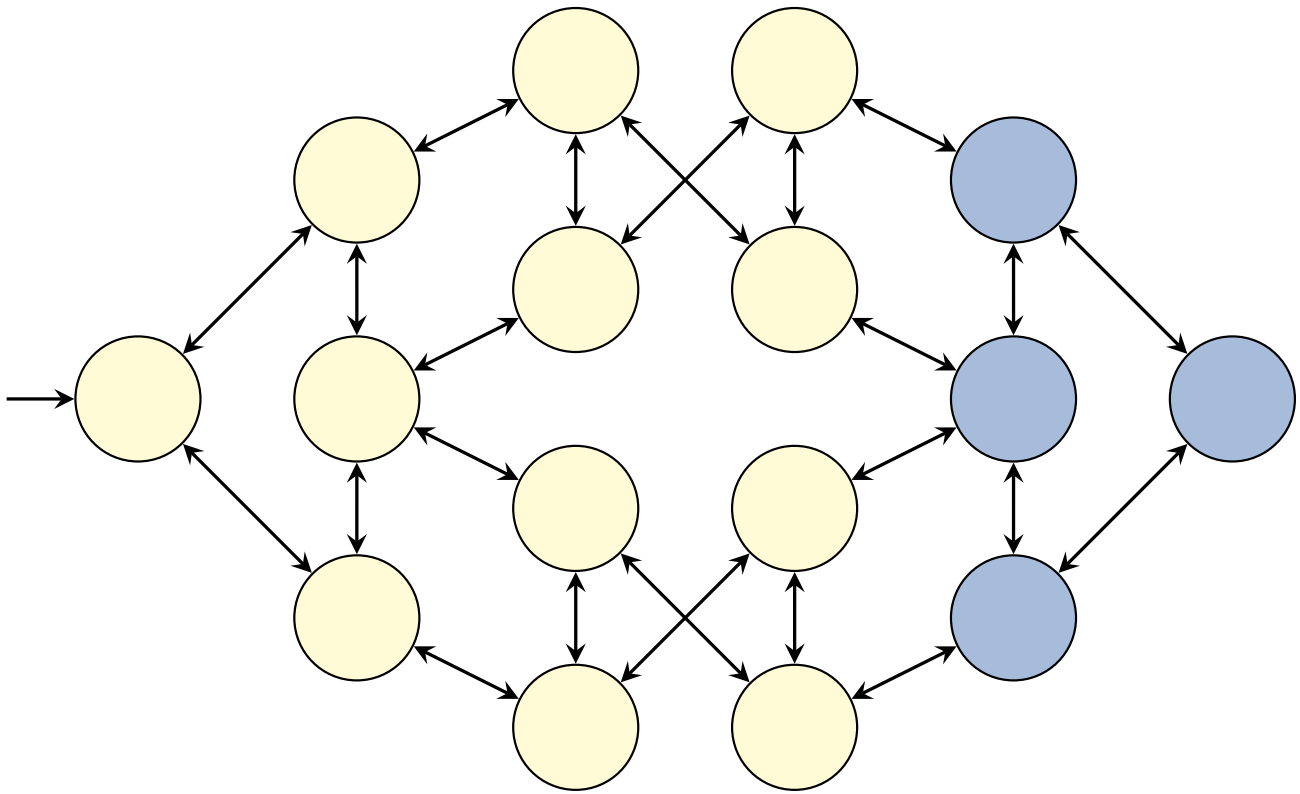
```
$ ls demo/ipc
agricola-opt18-strips
agricola-sat18-strips
airport
airport-adl
assembly
barman-mco14-strips
barman-opt11-strips
barman-opt14-strips
...
```

$\rightsquigarrow$ (most) benchmarks of planning competitions IPC since 1998

A2.3 How Hard is Planning?

Classical Planning as State-Space Search

classical planning as state-space search:



$\rightsquigarrow$ much more on this later in the course

Is Planning Difficult?

Classical planning is computationally challenging:

- number of states grows exponentially with description size when using (propositional) logic-based representations
- provably hard (PSPACE-complete)

$\rightsquigarrow$ we prove this later in the course

problem sizes:

- Seven Bridges of Königsberg: 64 reachable states
- Rubik's Cube: $4.325 \cdot 10^{19}$ reachable states
  $\rightsquigarrow$ consider 2 billion/second $\rightsquigarrow$ 1 billion years
- standard benchmarks: some with $> 10^{200}$ reachable states

A2.4 Summary

Summary

- planning = thinking before acting
- major subarea of Artificial Intelligence
- domain-independent planning = general problem solving
- classical planning = the "easy case"
  (deterministic, fully observable etc.)
- still hard enough!
  $\rightsquigarrow$ PSPACE-complete because of huge number of states
- often solved by state-space search
- number of states grows exponentially with input size

Planning and Optimization

A3. Getting to Know a Planner

Malte Helmert and Gabriele Röger

Universität Basel

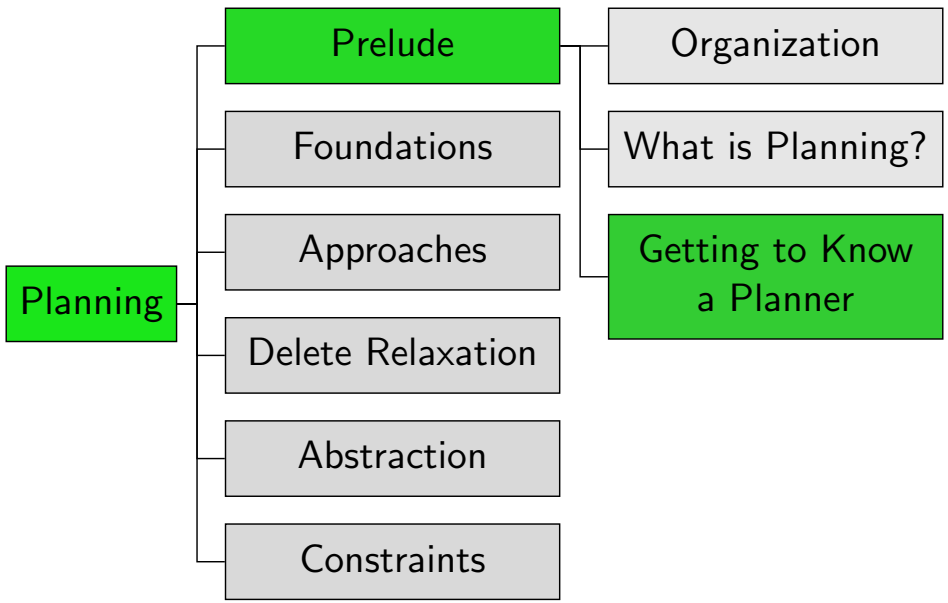
September 22, 2025

A3.1 Fast Downward and VAL

A3.2 15-Puzzle

A3.3 Summary

Content of the Course



A3.1 Fast Downward and VAL

Getting to Know a Planner

We now play around a bit with a planner and its input:

- look at problem formulation
- run a planner (= planning system/planning algorithm)
- validate plans found by the planner

Planner: Fast Downward

Fast Downward

We use the Fast Downward planner in this course

- because we know it well (developed by our research group)
- because it implements many search algorithms and heuristics
- because it is the classical planner most commonly used as a basis for other planners

$\rightsquigarrow$ `https://www.fast-downward.org`

Validator: VAL

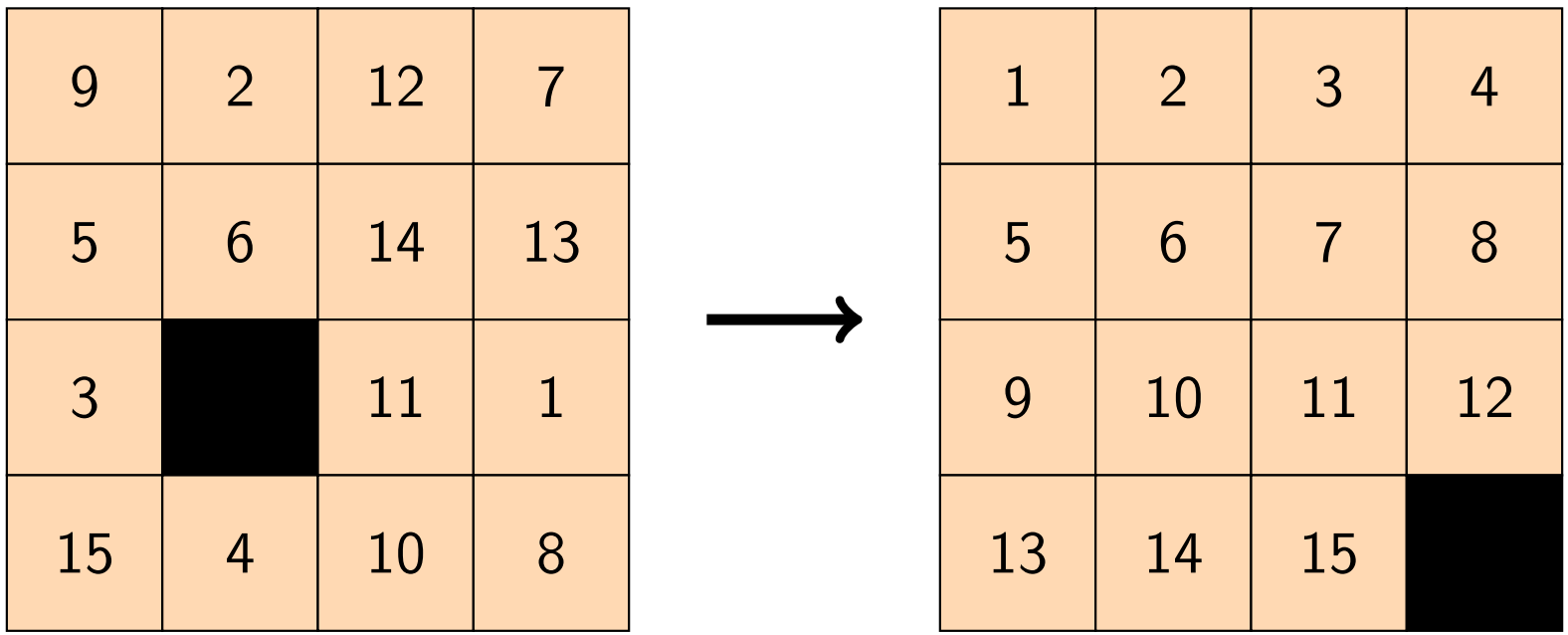
VAL

We use the VAL plan validation tool (Fox, Howey & Long)
to independently verify that the plans we generate are correct.

- very useful debugging tool
- `https://github.com/KCL-Planning/VAL`

A3.2 15-Puzzle

Illustrating Example: 15-Puzzle



Solving the 15-Puzzle

Demo

```
$ cd demo
$ less tile/puzzle.pddl
$ less tile/puzzle01.pddl
$ ./fast-downward.py \
      tile/puzzle.pddl tile/puzzle01.pddl \
      --heuristic "h=ff()" \
      --search "eager_greedy([h],preferred=[h])"
...
$ validate tile/puzzle.pddl tile/puzzle01.pddl \
      sas_plan
...
```

Variation: Weighted 15-Puzzle

Weighted 15-Puzzle:

- moving different tiles has different cost
- cost of moving tile x = number of prime factors of x

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/weight.pddl
$ meld tile/puzzle01.pddl tile/weight01.pddl
$ ./fast-downward.py \
      tile/weight.pddl tile/weight01.pddl \
      --heuristic "h=ff()" \
      --search "eager_greedy([h],preferred=[h])"
...
```

Variation: Glued 15-Puzzle

Glued 15-Puzzle:

- some tiles are glued in place and cannot be moved

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/glued.pddl
$ meld tile/puzzle01.pddl tile/glued01.pddl
$ ./fast-downward.py \
      tile/glued.pddl tile/glued01.pddl \
      --heuristic "h=cg()" \
      --search "eager_greedy([h],preferred=[h])"
...
```

Note: different heuristic used!

Variation: Cheating 15-Puzzle

Cheating 15-Puzzle:

- Can remove tiles from puzzle frame (creating more blanks) and reinsert tiles at any blank location.

Demo

```
$ cd demo
$ meld tile/puzzle.pddl tile/cheat.pddl
$ meld tile/puzzle01.pddl tile/cheat01.pddl
$ ./fast-downward.py \
      tile/cheat.pddl tile/cheat01.pddl \
      --heuristic "h=ff()" \
      --search "eager_greedy([h],preferred=[h])"
...
```

A3.3 Summary

Summary

- We saw planning tasks modeled in the PDDL language.
- We ran the Fast Downward planner and VAL plan validator.
- We made some modifications to PDDL problem formulations and checked the impact on the planner.

Planning and Optimization

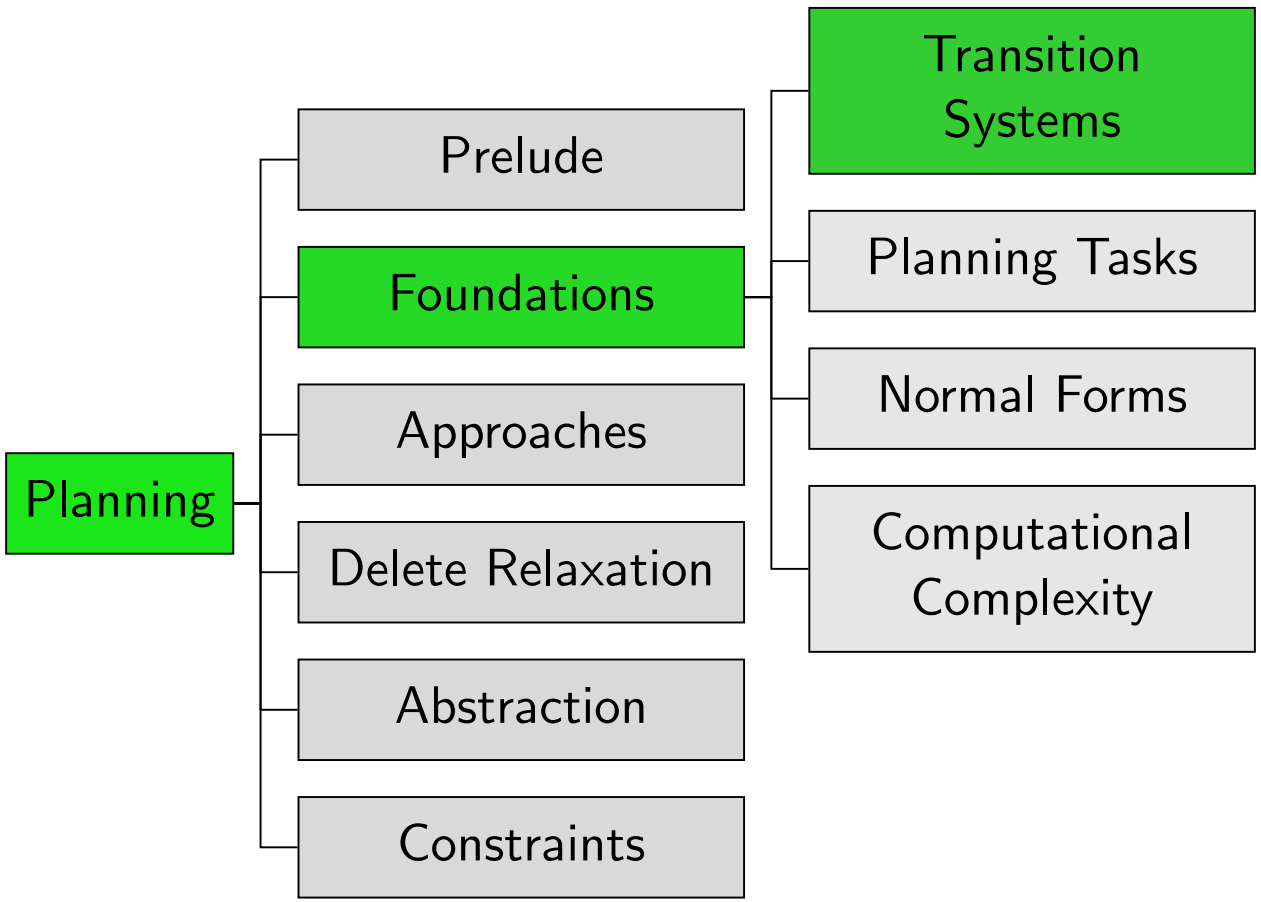
B1. Transition Systems and Propositional Logic

Malte Helmert and Gabriele Röger

Universität Basel

September 22, 2025

Content of the Course



Next Steps

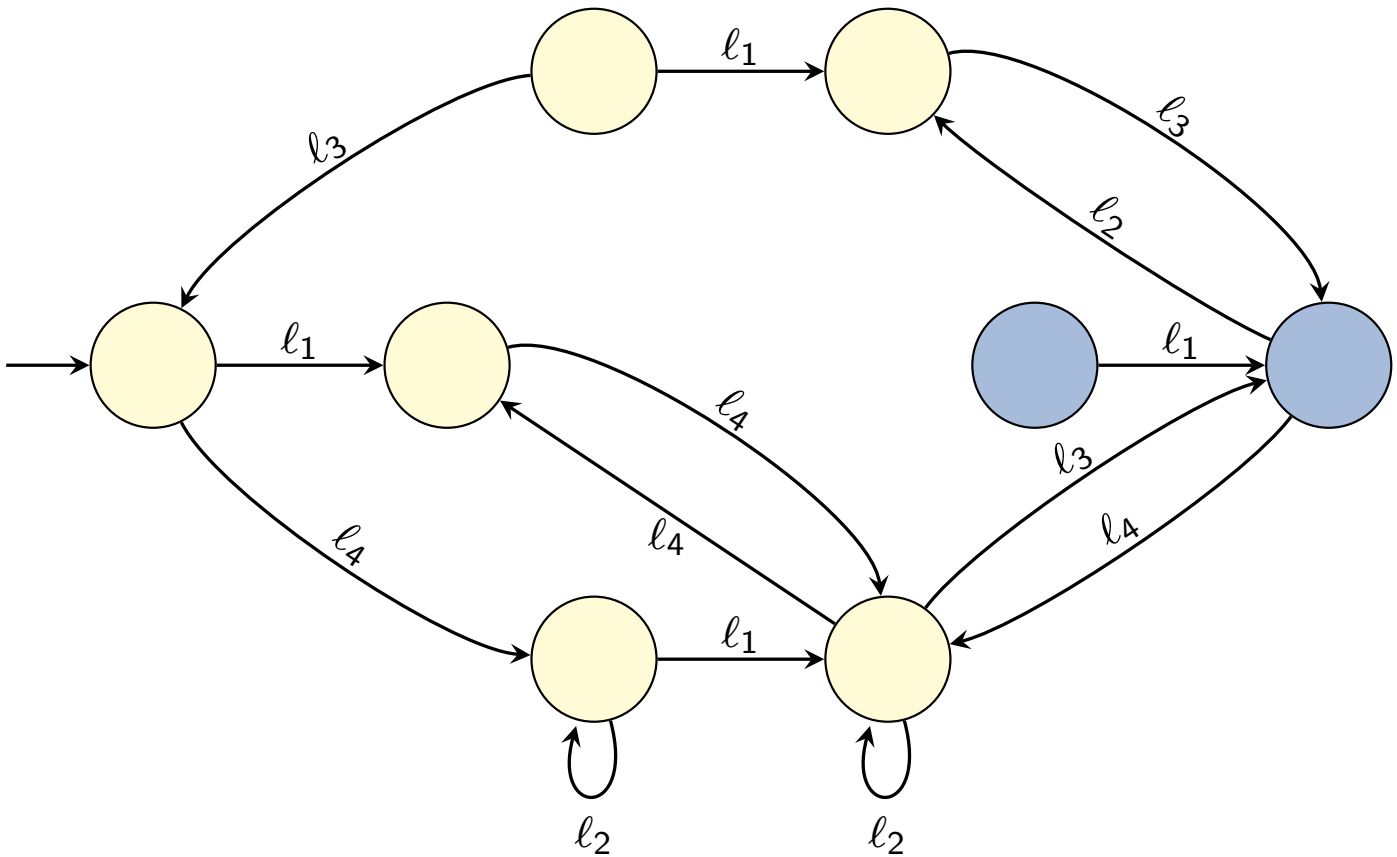
Our next steps are to formally define our problem:

- introduce a mathematical model for planning tasks:
  transition systems
  $\rightsquigarrow$ Chapter B1
- introduce compact representations for planning tasks
  suitable as input for planning algorithms
  $\rightsquigarrow$ Chapter B2

B1.1 Transition Systems

Transition System Example

Transition systems are often depicted as directed arc-labeled graphs with decorations to indicate the initial state and goal states.



$$c(\ell_1) = 1,\ c(\ell_2) = 1,\ c(\ell_3) = 5,\ c(\ell_4) = 0$$

Transition Systems

Definition (Transition System)

A transition system is a 6-tuple $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ where

- S is a finite set of states,
- L is a finite set of (transition) labels,
- $c : L \to \mathbb{R}_0^+$ is a label cost function,
- $T \subseteq S \times L \times S$ is the transition relation,
- $s_0 \in S$ is the initial state, and
- $S_\star \subseteq S$ is the set of goal states.

We say that $\mathcal{T}$ has the transition $\langle s, \ell, s' \rangle$ if $\langle s, \ell, s' \rangle \in T$.

We also write this as $s \xrightarrow{\ell} s'$, or $s \to s'$ when not interested in ℓ.

Note: Transition systems are also called state spaces.

Deterministic Transition Systems

Definition (Deterministic Transition System)

A transition system is called deterministic if for all states s and all labels ℓ, there is at most one state s' with $s \xrightarrow{\ell} s'$.

Example: previously shown transition system

Transition System Terminology (1)

We use common terminology from graph theory:

- s' successor of s if $s \to s'$
- s predecessor of s' if $s \to s'$

Transition System Terminology (2)

We use common terminology from graph theory:

- s' reachable from s if there exists a sequence of transitions $s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n$ s.t. $s^0 = s$ and $s^n = s'$
  - Note: $n = 0$ possible; then $s = s'$
  - $s^0, \ldots, s^n$ is called (state) path from s to s'
  - $\ell_1, \ldots, \ell_n$ is called (label) path from s to s'
  - $s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n$ is called trace from s to s'
  - length of path/trace is n
  - cost of label path/trace is $\sum_{i=1}^{n} c(\ell_i)$

Transition System Terminology (3)

We use common terminology from graph theory:

- s' reachable (without reference state) means reachable from initial state s_0
- solution or goal path from s: path from s to some $s' \in S_\star$
  - if s is omitted, $s = s_0$ is implied
- transition system solvable if a goal path from s_0 exists

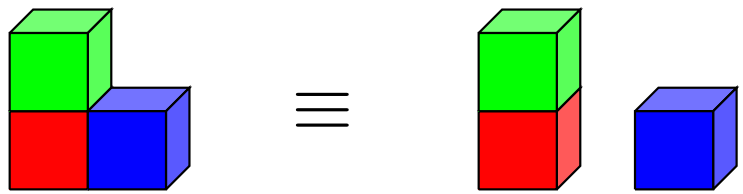
B1.2 Example: Blocks World

Running Example: Blocks World

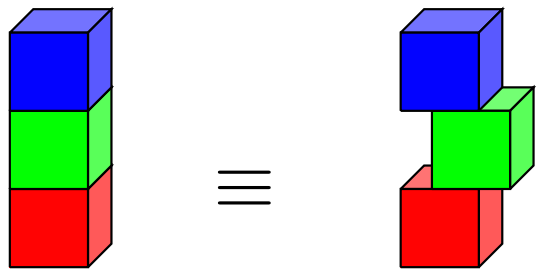
- Throughout the course, we occasionally use the blocks world domain as an example.
- In the blocks world, a number of different blocks are arranged on a table.
- Our job is to rearrange them according to a given goal.

Blocks World Rules (1)

Location on the table does not matter.

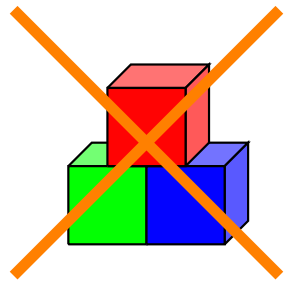


Location on a block does not matter.

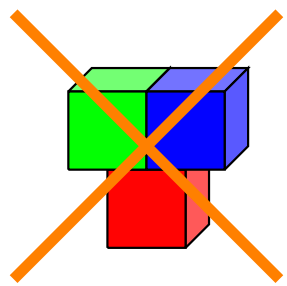


Blocks World Rules (2)

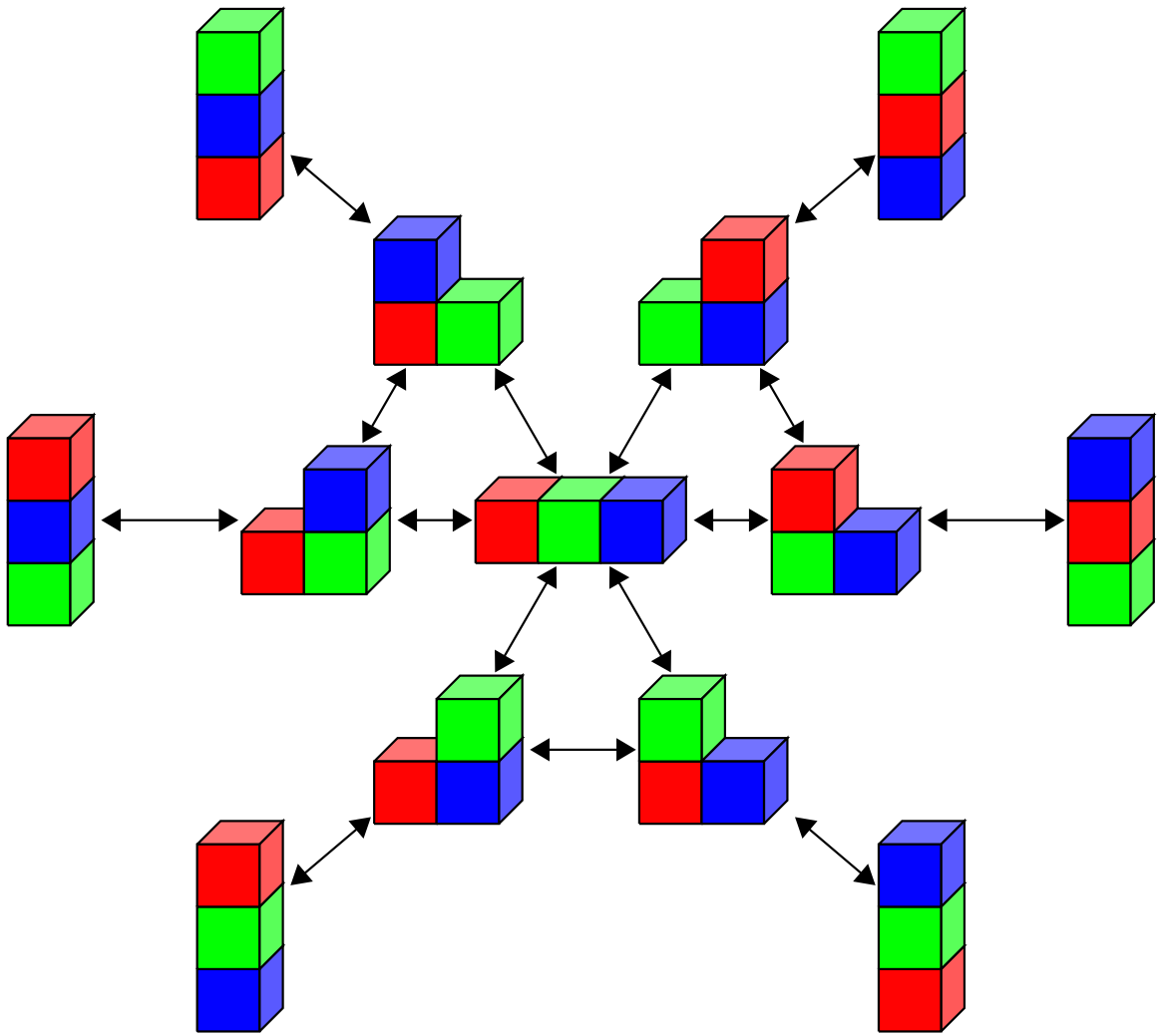
At most one block may be below a block.



At most one block may be on top of a block.



Blocks World Transition System for Three Blocks



Labels omitted for clarity. All label costs are 1. Initial/goal states not marked.

Blocks World Computational Properties

blocks	states	blocks	states
1	1	10	58941091
2	3	11	824073141
3	13	12	12470162233
4	73	13	202976401213
5	501	14	3535017524403
6	4051	15	65573803186921
7	37633	16	1290434218669921
8	394353	17	26846616451246353
9	4596553	18	588633468315403843

- Finding solutions is possible in linear time
  in the number of blocks: move everything onto the table, then construct the goal configuration.
- Finding a shortest solution is NP-complete
  given a compact description of the problem.

The Need for Compact Descriptions

- We see from the blocks world example that transition systems are often far too large to be directly used as inputs to planning algorithms.
- We therefore need compact descriptions of transition systems.
- For this purpose, we will use propositional logic, which allows expressing information about 2^n states as logical formulas over n state variables.

B1.3 Reminder: Propositional Logic

More on Propositional Logic

Need to Catch Up?

- This section is a reminder. We assume you are already well familiar with propositional logic.
- If this is not the case, we recommend Chapters D1–D4 of the Discrete Mathematics in Computer Science course:
  `https://dmi.unibas.ch/en/studies/computer-science/course-offer-hs24/lecture-discrete-mathematics-in-computer-science/`
  - Videos for these chapters are available on request.

Syntax of Propositional Logic

Definition (Logical Formula)

Let A be a set of atomic propositions.

The logical formulas over A are constructed by finite application of the following rules:

- $\top$ and $\bot$ are logical formulas (truth and falsity).
- For all $a \in A$, a is a logical formula (atom).
- If φ is a logical formula, then so is $\neg\varphi$ (negation).
- If φ and ψ are logical formulas, then so are $(\varphi \vee \psi)$ (disjunction) and $(\varphi \wedge \psi)$ (conjunction).

Syntactical Conventions for Propositional Logic

Abbreviations:

- $(\varphi \rightarrow \psi)$ is short for $(\neg\varphi \vee \psi)$ (implication)
- $(\varphi \leftrightarrow \psi)$ is short for $((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$ (equijunction)
- parentheses omitted when not necessary:
  - $(\neg)$ binds more tightly than binary connectives
  - $(\wedge)$ binds more tightly than $(\vee)$, which binds more tightly than $(\rightarrow)$, which binds more tightly than $(\leftrightarrow)$

Semantics of Propositional Logic

Definition (Interpretation, Model)

An interpretation of propositions A is a function $I : A \to \{\mathbf{T}, \mathbf{F}\}$.

Define the notation $I \models \varphi$ (I satisfies φ; I is a model of φ; φ is true under I) for interpretations I and formulas φ by

- $I \models \top$
- $I \not\models \bot$
- $I \models a$ iff $I(a) = \mathbf{T}$ (for all $a \in A$)
- $I \models \neg\varphi$ iff $I \not\models \varphi$
- $I \models (\varphi \vee \psi)$ iff ($I \models \varphi$ or $I \models \psi$)
- $I \models (\varphi \wedge \psi)$ iff ($I \models \varphi$ and $I \models \psi$)

Note: Interpretations are also called valuations or truth assignments.

Propositional Logic Terminology (1)

- A logical formula φ is satisfiable
  if there is at least one interpretation I such that $I \models \varphi$.
- Otherwise it is unsatisfiable.
- A logical formula φ is valid or a tautology
  if $I \models \varphi$ for all interpretations I.
- A logical formula ψ is a logical consequence
  of a logical formula φ, written $\varphi \models \psi$,
  if $I \models \psi$ for all interpretations I with $I \models \varphi$.
- Two logical formulas φ and ψ are logically equivalent,
  written $\varphi \equiv \psi$, if $\varphi \models \psi$ and $\psi \models \varphi$.

Question: How to phrase these in terms of models?

Propositional Logic Terminology (2)

- A logical formula that is a proposition a or a negated proposition $\neg a$ for some atomic proposition $a \in A$ is a literal.
- A formula that is a disjunction of literals is a clause.
  This includes unit clauses ℓ consisting of a single literal and the empty clause $\bot$ consisting of zero literals.
- A formula that is a conjunction of literals is a monomial.
  This includes unit monomials ℓ consisting of a single literal and the empty monomial $\top$ consisting of zero literals.

Normal forms:

- negation normal form (NNF)
- conjunctive normal form (CNF)
- disjunctive normal form (DNF)

B1.4 Summary

Summary

- Transition systems are (typically huge) directed graphs that encode how the state of the world can change.
- Propositional logic allows us to compactly describe complex information about large sets of interpretations as logical formulas.

Planning and Optimization

B2. Introduction to Planning Tasks

Malte Helmert and Gabriele Röger

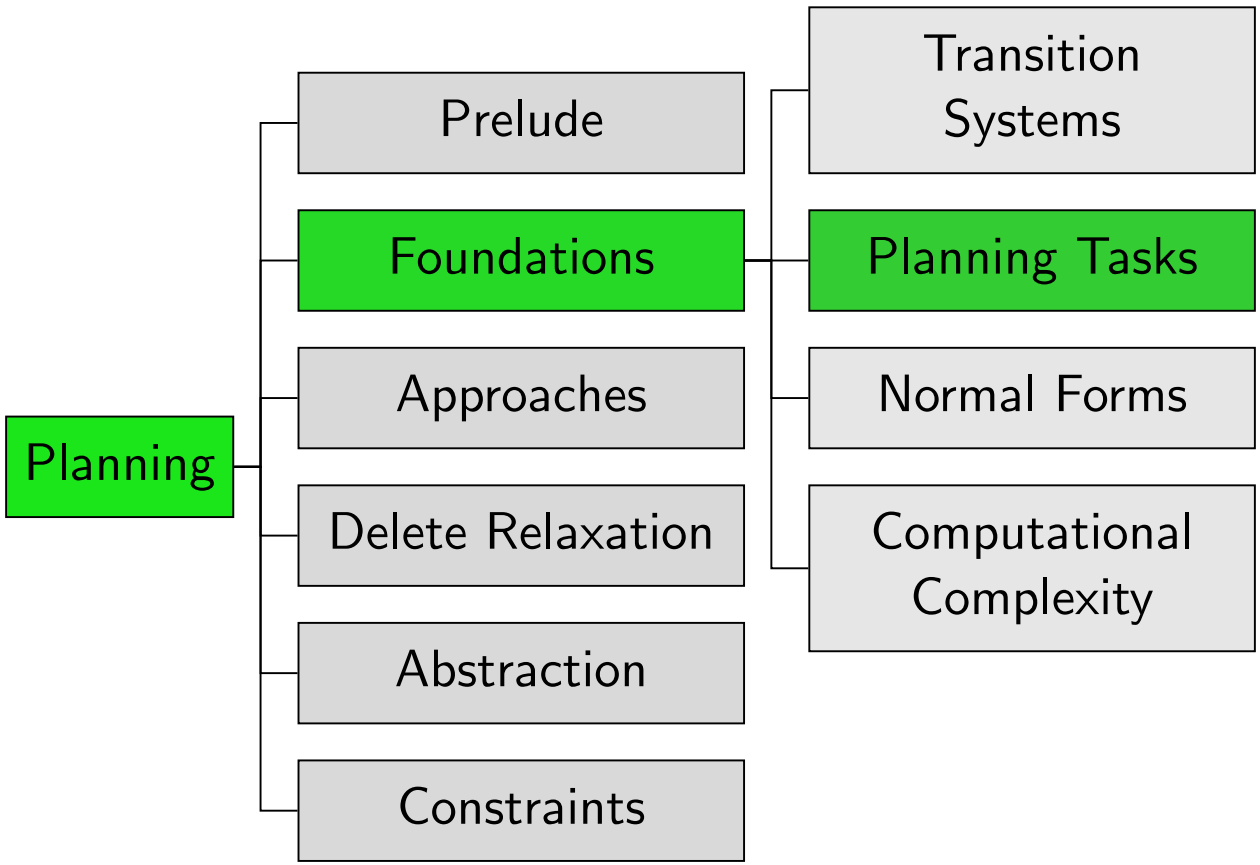
Universität Basel

September 24, 2025

Planning and Optimization

September 24, 2025 — B2. Introduction to Planning Tasks

Content of the Course



B2.1 Introduction

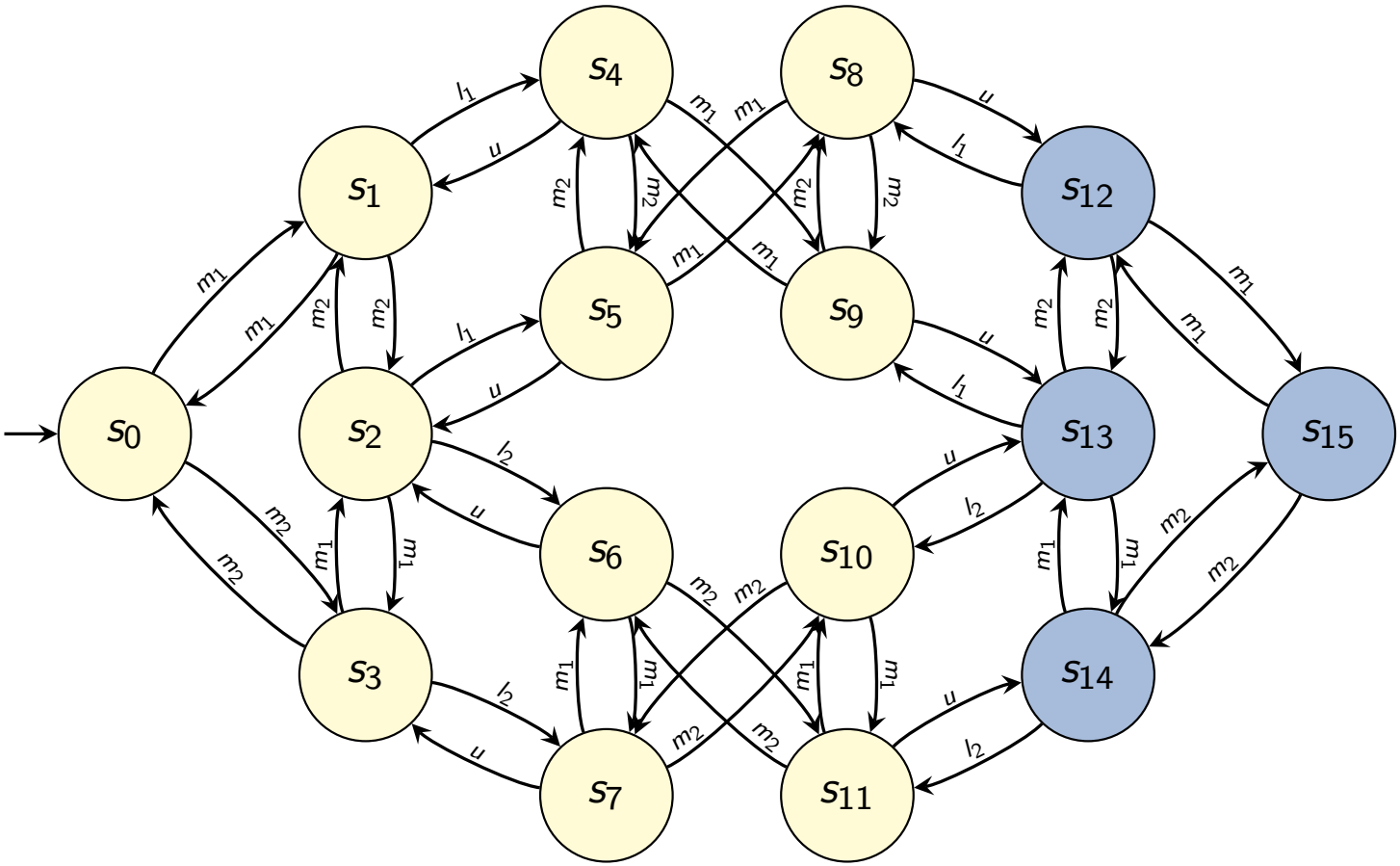
The State Explosion Problem

- We saw in blocks world:
  n blocks $\rightsquigarrow$ number of states exponential in n
- same is true everywhere we look
- known as the state explosion problem

To represent transitions systems compactly,
need to tame these exponentially growing aspects:

- states
- goal states
- transitions

Running Example: Transition System



$c(m_1) = 5,\ c(m_2) = 5,\ c(l_1) = 1,\ c(l_2) = 1,\ c(u) = 1$

B2.2 State Variables

Compact Descriptions of Transition Systems

How to specify huge transition systems
without enumerating the states?

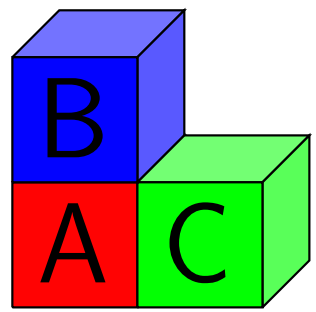
- represent different aspects of the world
  in terms of different (propositional) state variables
- individual state variables are atomic propositions
  $\rightsquigarrow$ a state is an interpretation of state variables
- n state variables induce 2^n states
  $\rightsquigarrow$ exponentially more compact than "flat" representations

Example: n^2 variables suffice for blocks world with n blocks

Blocks World State with Propositional Variables

Example

$$s(\textit{A-on-B}) = \mathbf{F}$$
$$s(\textit{A-on-C}) = \mathbf{F}$$
$$s(\textit{A-on-table}) = \mathbf{T}$$
$$s(\textit{B-on-A}) = \mathbf{T}$$
$$s(\textit{B-on-C}) = \mathbf{F}$$
$$s(\textit{B-on-table}) = \mathbf{F}$$
$$s(\textit{C-on-A}) = \mathbf{F}$$
$$s(\textit{C-on-B}) = \mathbf{F}$$
$$s(\textit{C-on-table}) = \mathbf{T}$$



$\rightsquigarrow$ 9 variables for 3 blocks

Propositional State Variables

Definition (Propositional State Variable)

A propositional state variable is a symbol X.

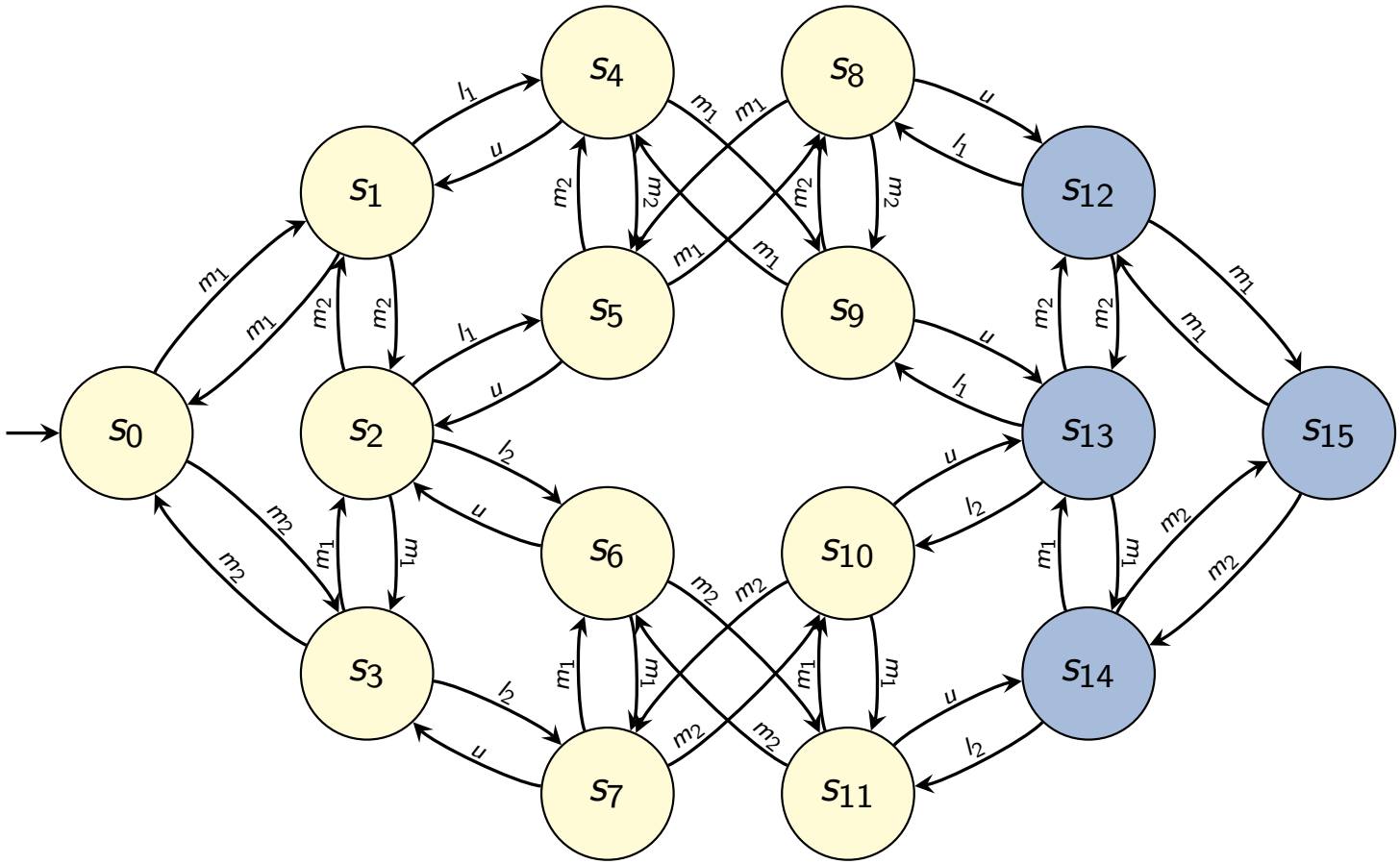
Let V be a finite set of propositional state variables.

A state s over V is an interpretation of V, i.e.,
a truth assignment $s : V \to \{\mathbf{T}, \mathbf{F}\}$.

Running Example: Compact State Descriptions

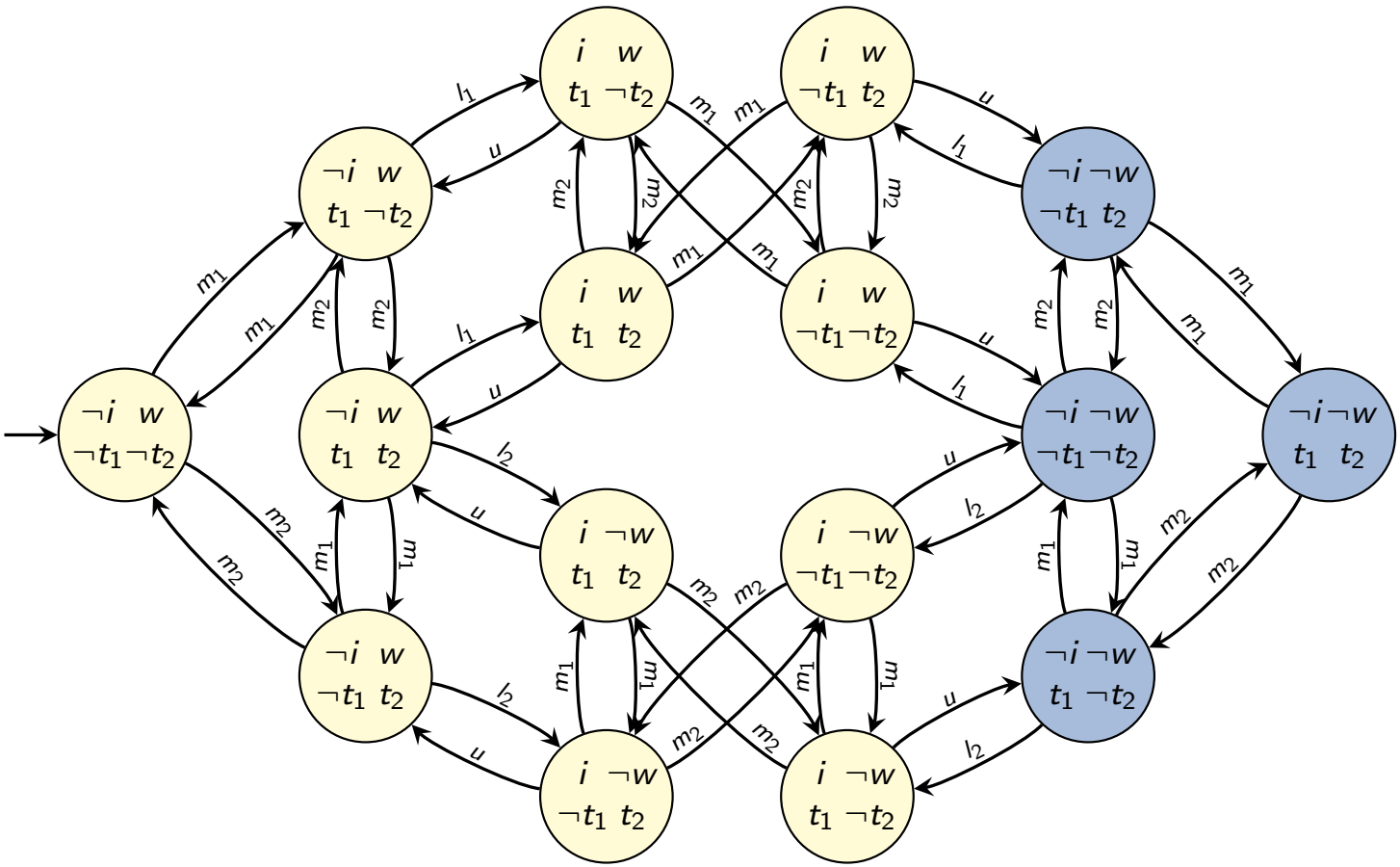
- In the running example, we describe 16 states with 4 propositional state variables ($2^4 = 16$).

Running Example: Opaque States



Running Example: Using State Variables

state variables $V = \{i, w, t_1, t_2\}$



states shown by true literals

example: $\{i \mapsto \mathbf{T}, w \mapsto \mathbf{F}, t_1 \mapsto \mathbf{T}, t_2 \mapsto \mathbf{F}\} \rightsquigarrow i\ \neg w\ t_1\ \neg t_2$

Running Example: Intuition

Intuition: delivery task with 2 trucks, 1 package, locations L and R

transition labels:

- m_1/m_2: move first/second truck
- l_1/l_2: load package into first/second truck
- u: unload package from a truck

state variables:

- t_1 true if first truck is at location L (else at R)
- t_2 true if second truck is at location L (else at R)
- i true if package is inside a truck
- w encodes where exactly the package is:
  - if i is true, w true if package in first truck
  - if i is false, w true if package at location L

B2.3 State Formulas

Representing Sets of States

How do we compactly represent sets of states,
for example the set of goal states?

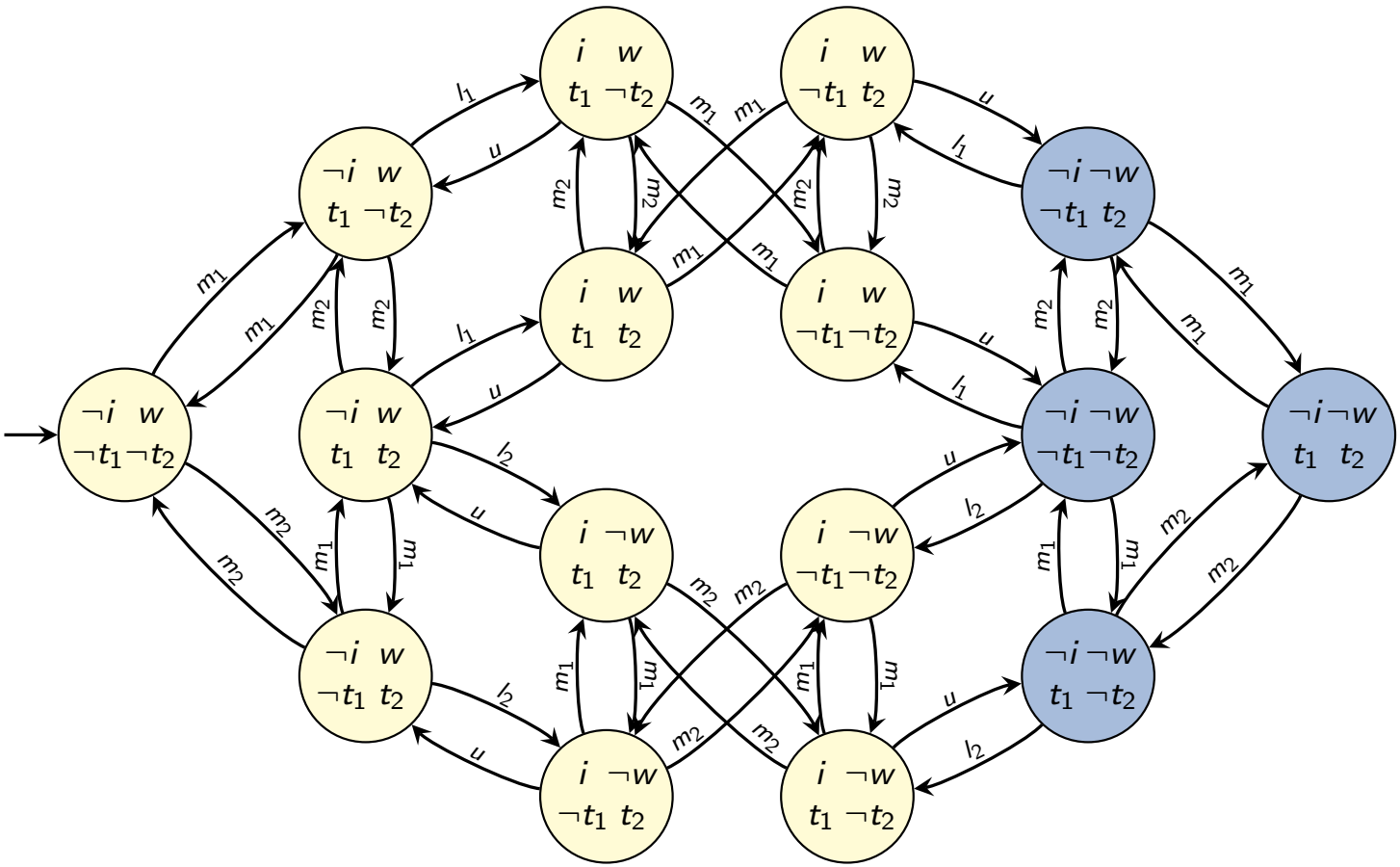
Idea: formula φ over the state variables represents the models of φ.

Definition (State Formula)

Let V be a finite set of propositional state variables.

A formula over V is a propositional logic formula using V
as the set of atomic propositions.

Running Example: Representing Goal States



goal formula $\gamma = \neg i \wedge \neg w$ represents goal states $S_\star$

B2.4 Operators and Effects

Operators Representing Transitions

How do we compactly represent transitions?

- most complex aspect of a planning task
- central concept: operators

Idea: one operator o for each transition label ℓ, describing

- in which states s a transition $s \xrightarrow{\ell} s'$ exists (precondition)
- how state s' differs from state s (effect)
- what the cost of ℓ is

Syntax of Operators

Definition (Operator)

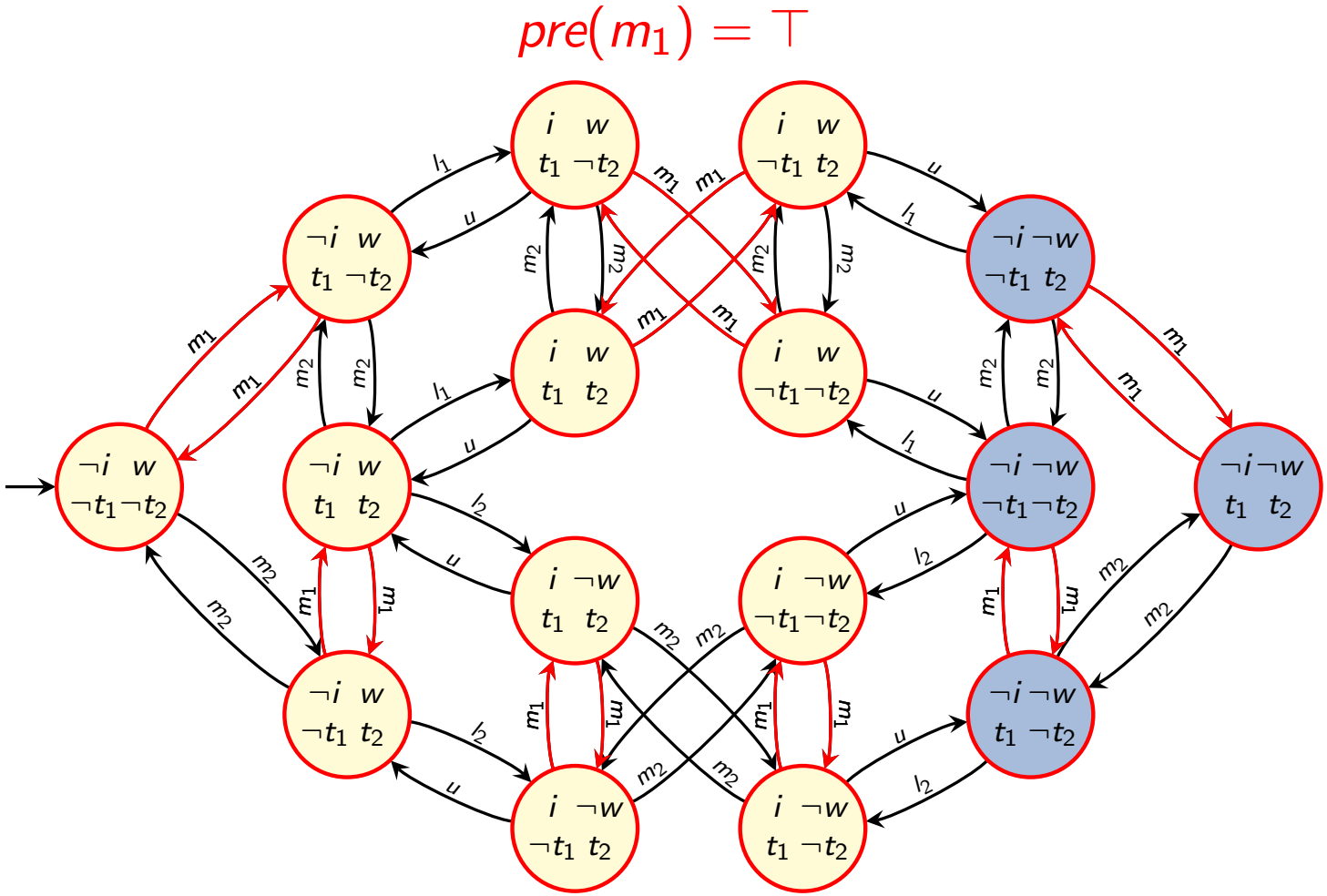
An operator o over state variables V is an object with three properties:

- a precondition $pre(o)$, a formula over V
- an effect $eff(o)$ over V, defined later in this chapter
- a cost $cost(o) \in \mathbb{R}_0^+$

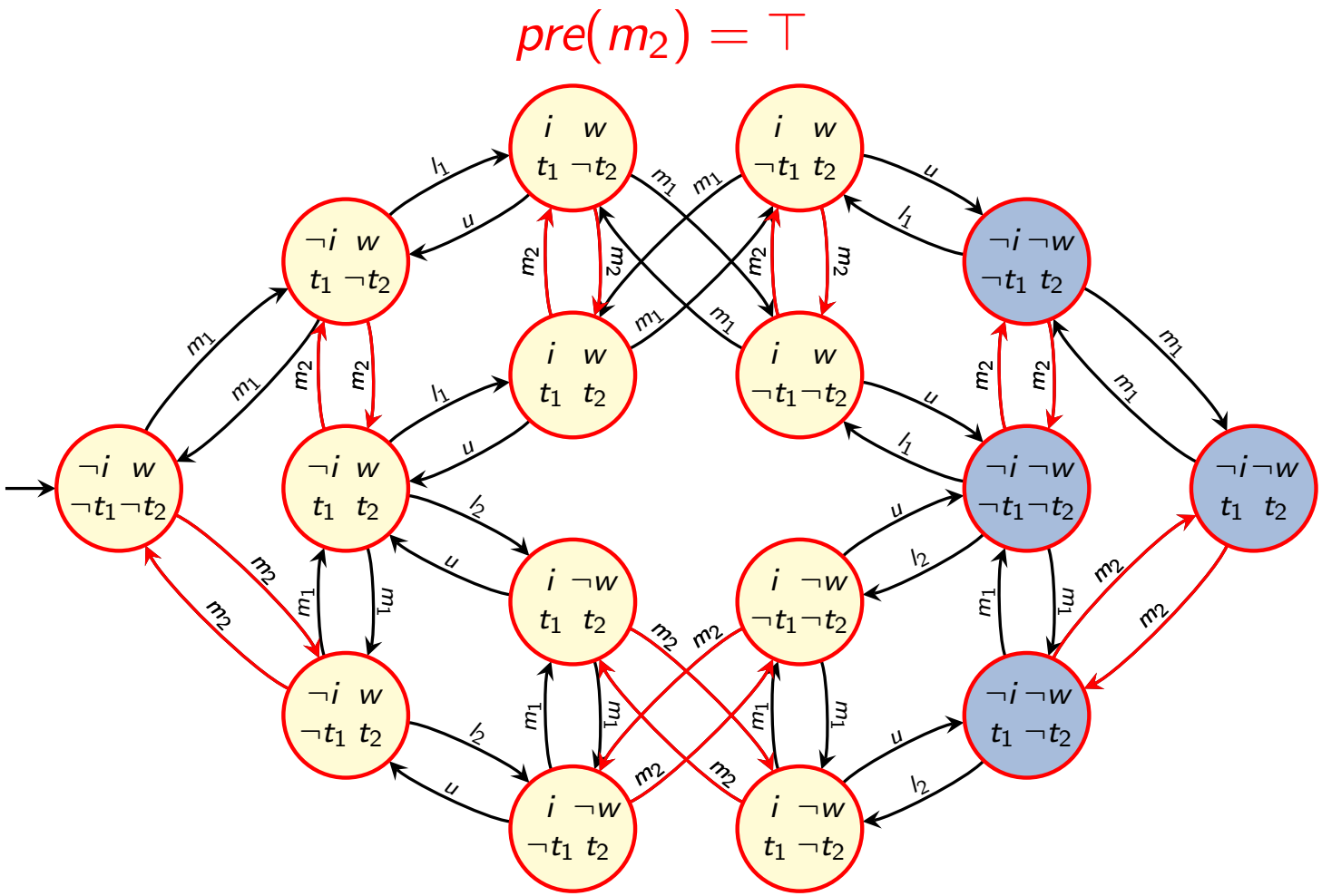
Notes:

- Operators are also called actions.
- Operators are often written as triples $\langle pre(o), eff(o), cost(o) \rangle$.
- This can be abbreviated to pairs $\langle pre(o), eff(o) \rangle$ when the cost of the operator is irrelevant.

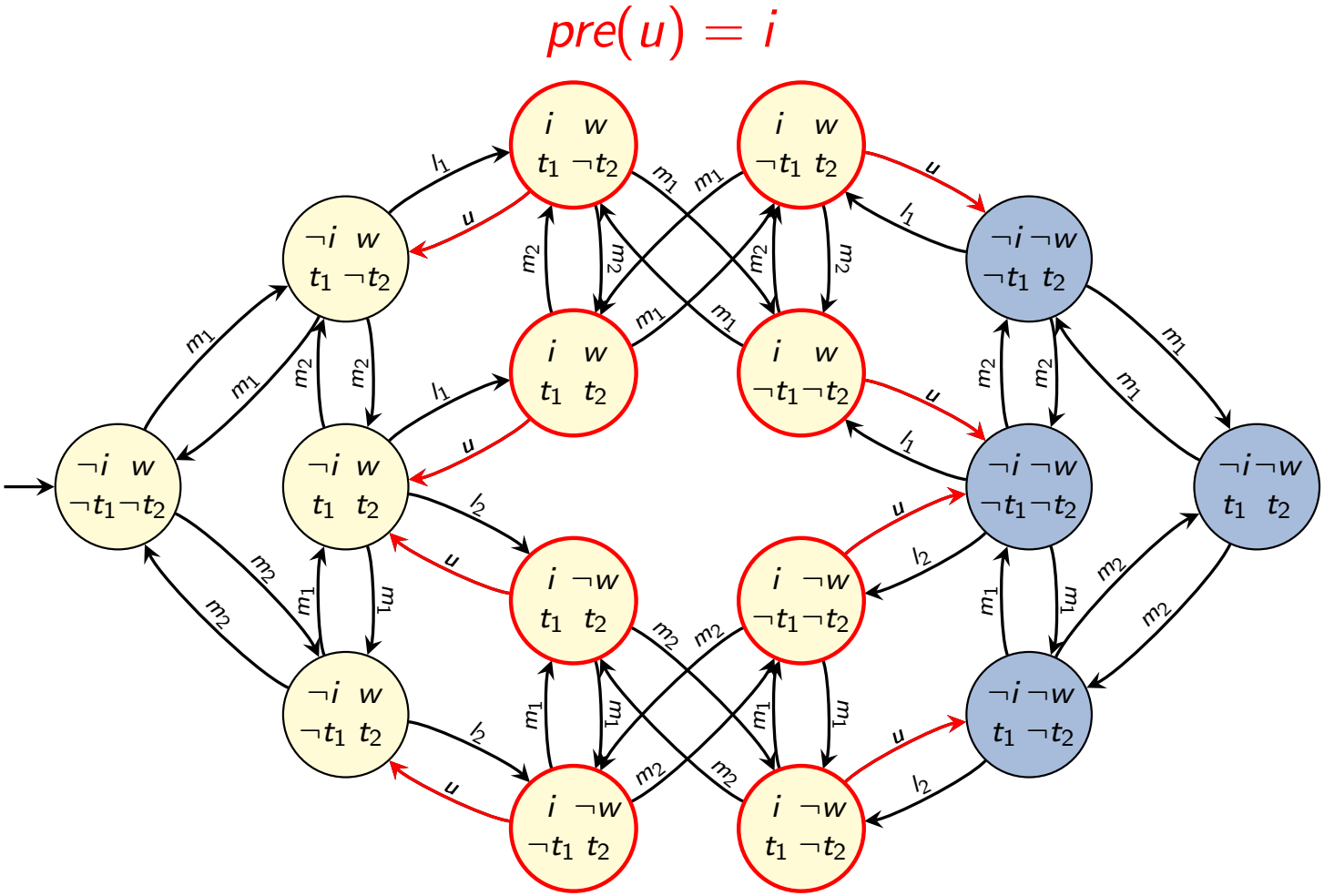
Running Example: Operator Preconditions



Running Example: Operator Preconditions

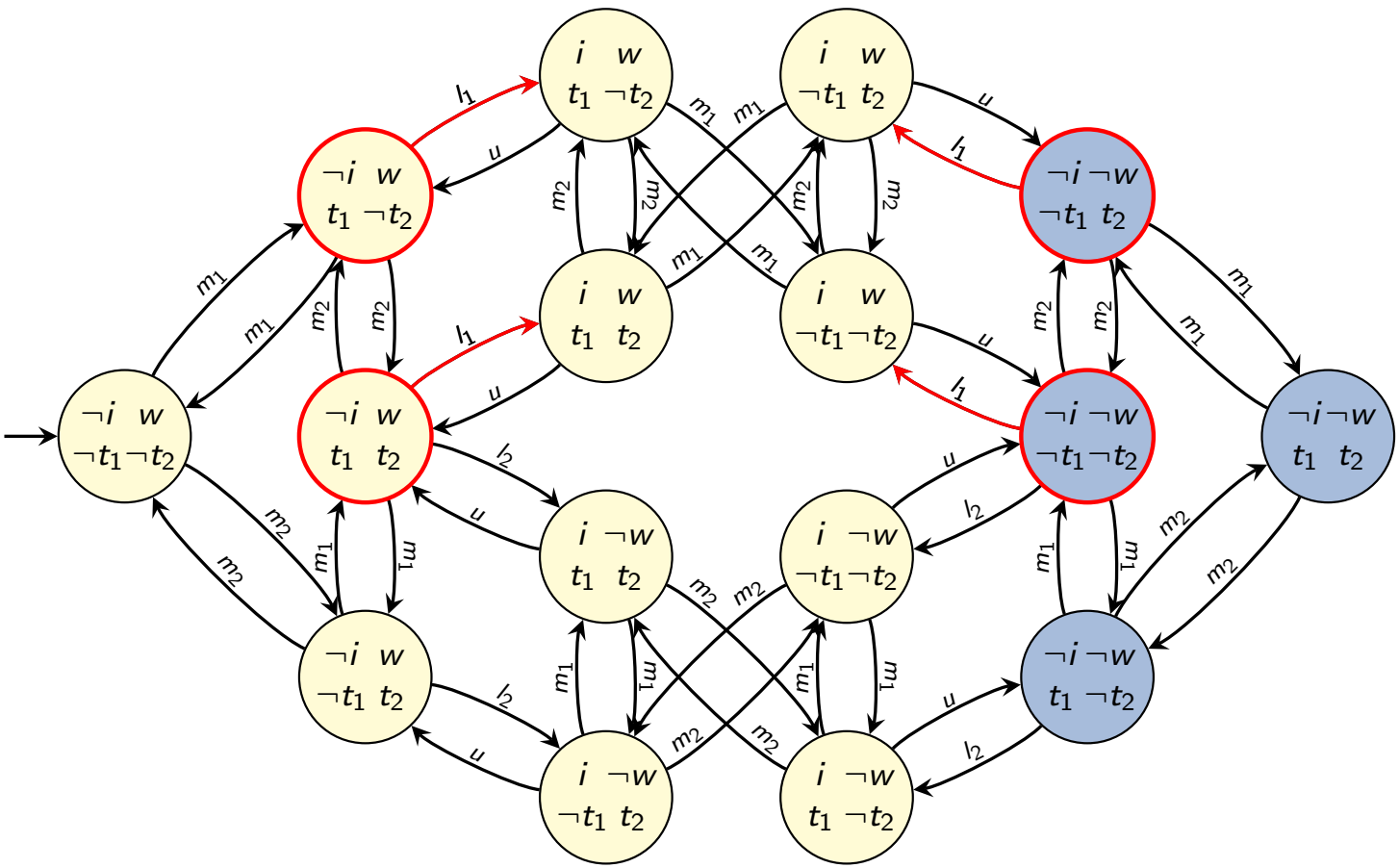


Running Example: Operator Preconditions



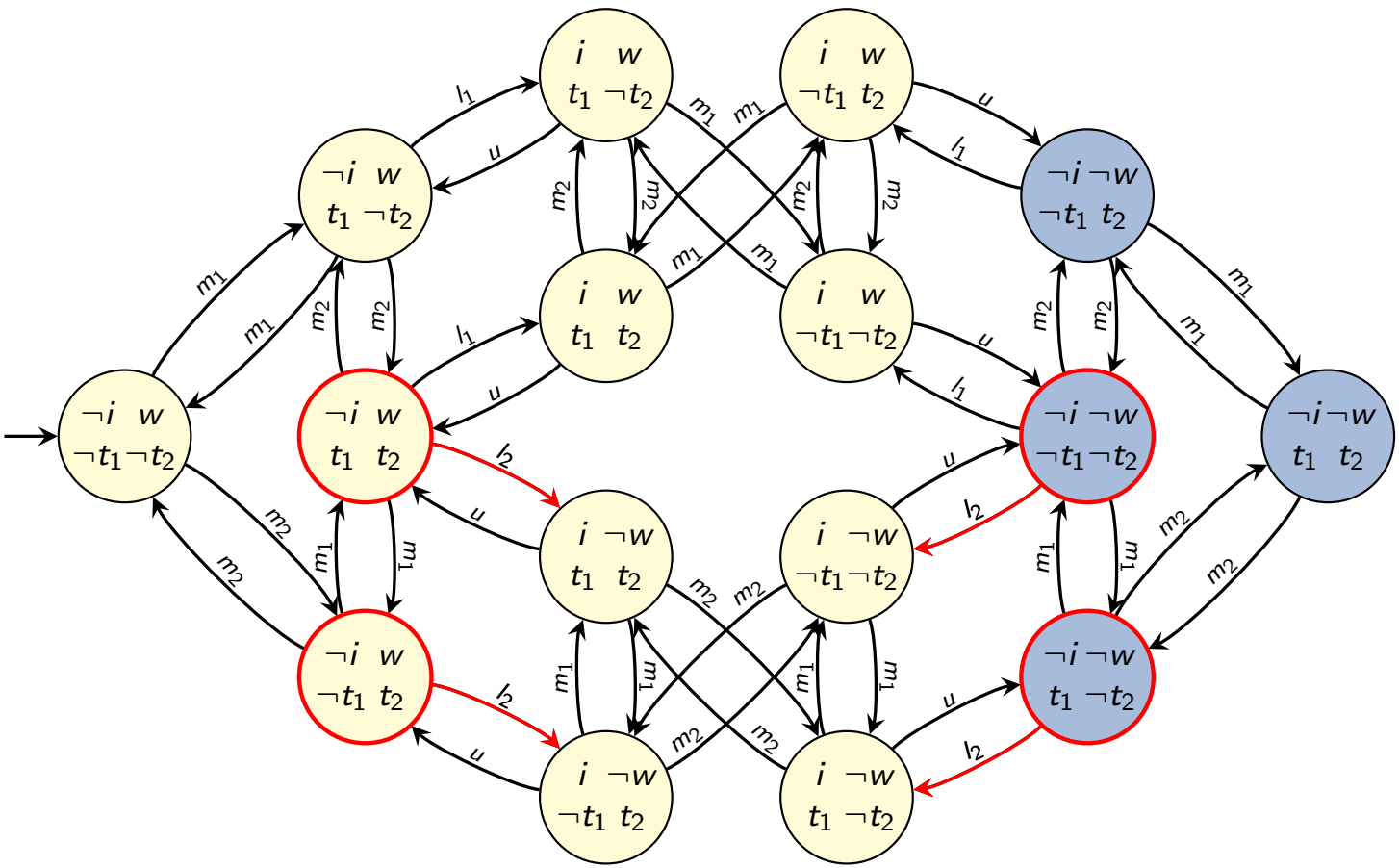
Running Example: Operator Preconditions

$$pre(l_1) = \neg i \wedge (w \leftrightarrow t_1)$$



Running Example: Operator Preconditions

$$pre(l_2) = \neg i \wedge (w \leftrightarrow t_2)$$



Syntax of Effects

Definition (Effect)

Effects over propositional state variables V are inductively defined as follows:

- $\top$ is an effect (empty effect).
- If $v \in V$ is a propositional state variable, then v and $\neg v$ are effects (atomic effect).
- If e and e' are effects, then $(e \wedge e')$ is an effect (conjunctive effect).
- If χ is a formula over V and e is an effect, then $(\chi \rhd e)$ is an effect (conditional effect).

We may omit parentheses when this does not cause ambiguity.

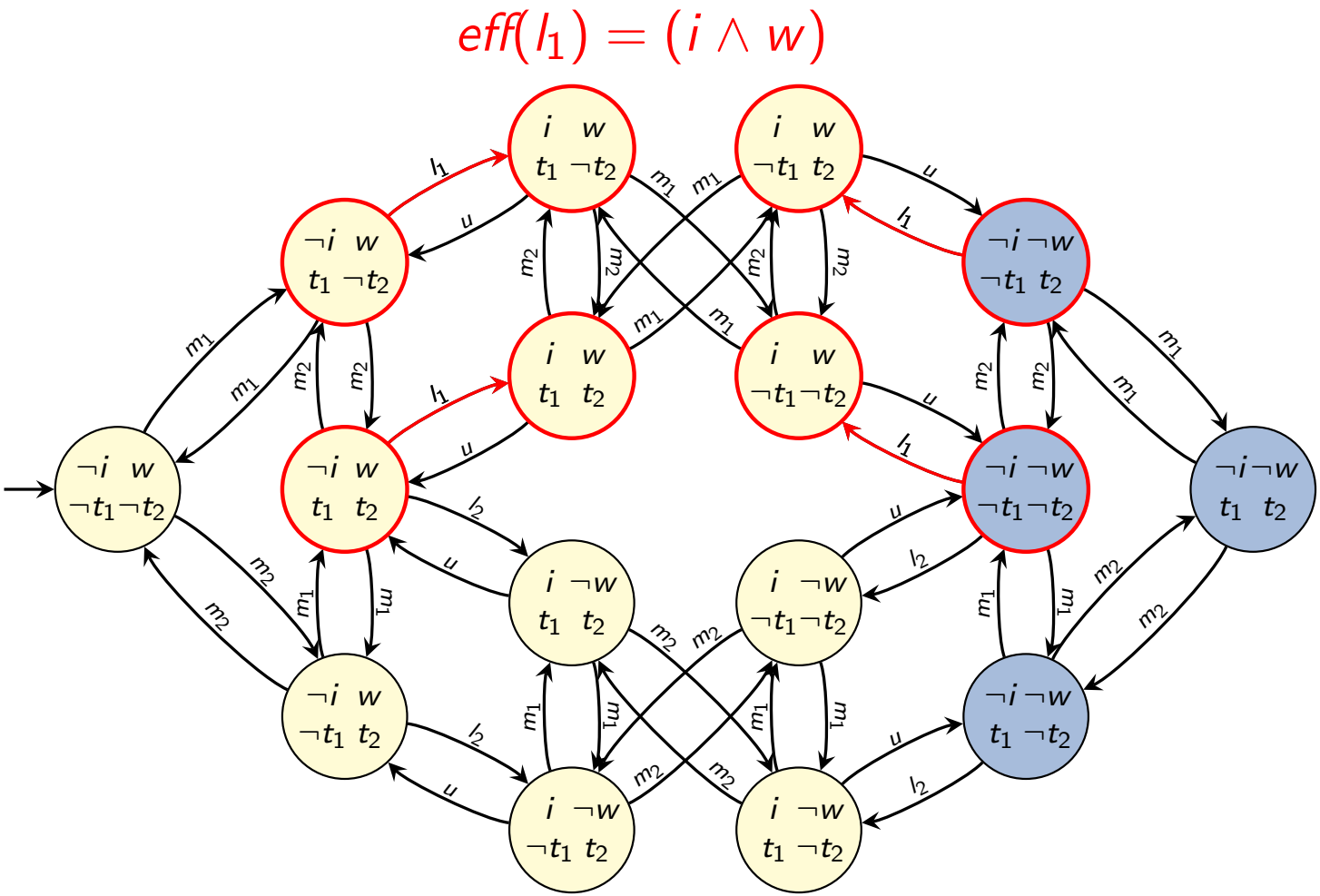
Example: we will later see that $((e \wedge e') \wedge e'')$ behaves identically to $(e \wedge (e' \wedge e''))$ and will write this as $e \wedge e' \wedge e''$.

Effects: Intuition

Intuition for effects:

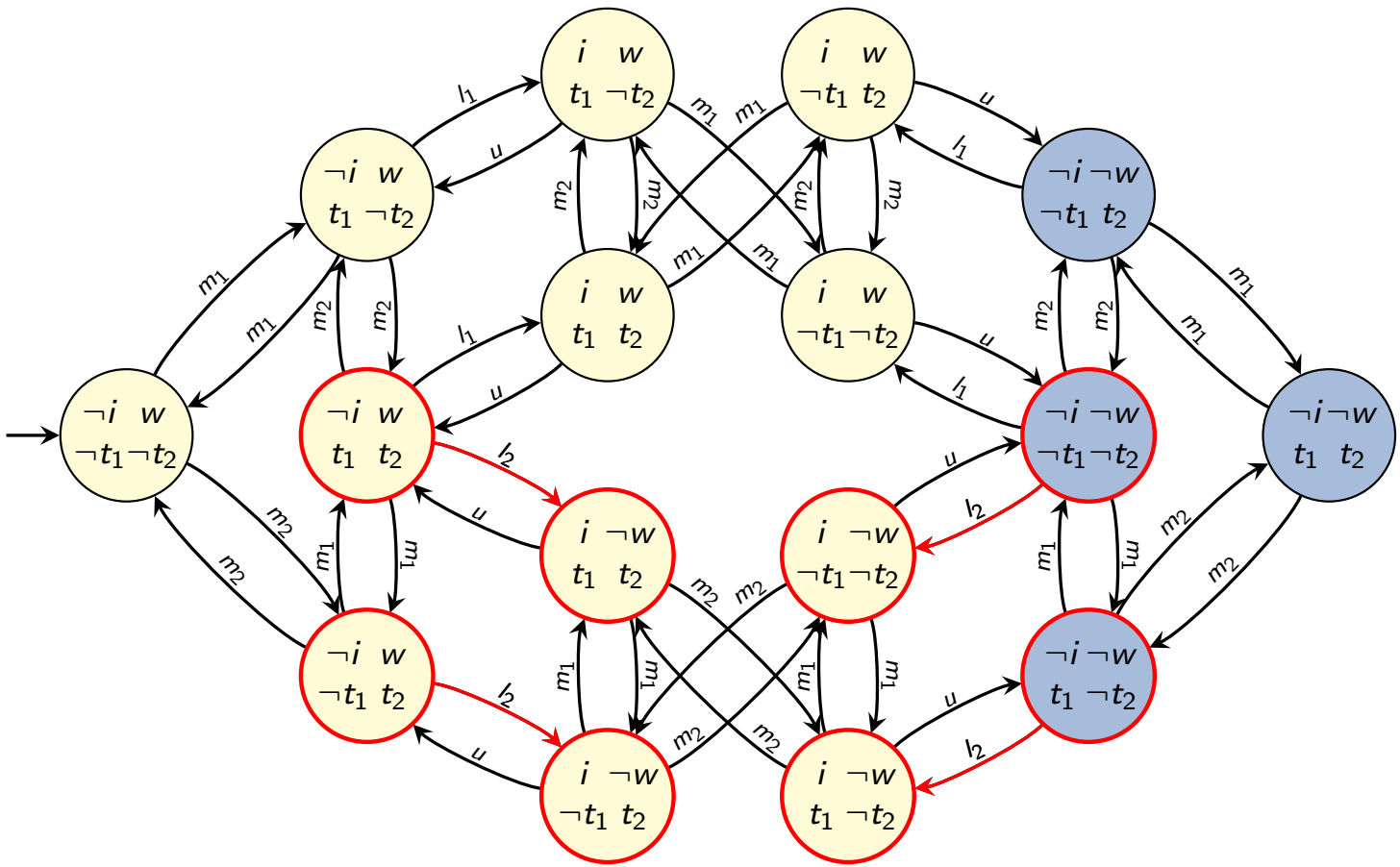
- The empty effect $\top$ changes nothing.
- Atomic effects can be understood as assignments that update the value of a state variable.
  - v means "$v := \mathbf{T}$"
  - $\neg v$ means "$v := \mathbf{F}$"
- A conjunctive effect $e = (e' \wedge e'')$ means that both subeffects e and e' take place simultaneously.
- A conditional effect $e = (\chi \rhd e')$ means that subeffect e' takes place iff χ is true in the state where e takes place.

Running Example: Operator Effects



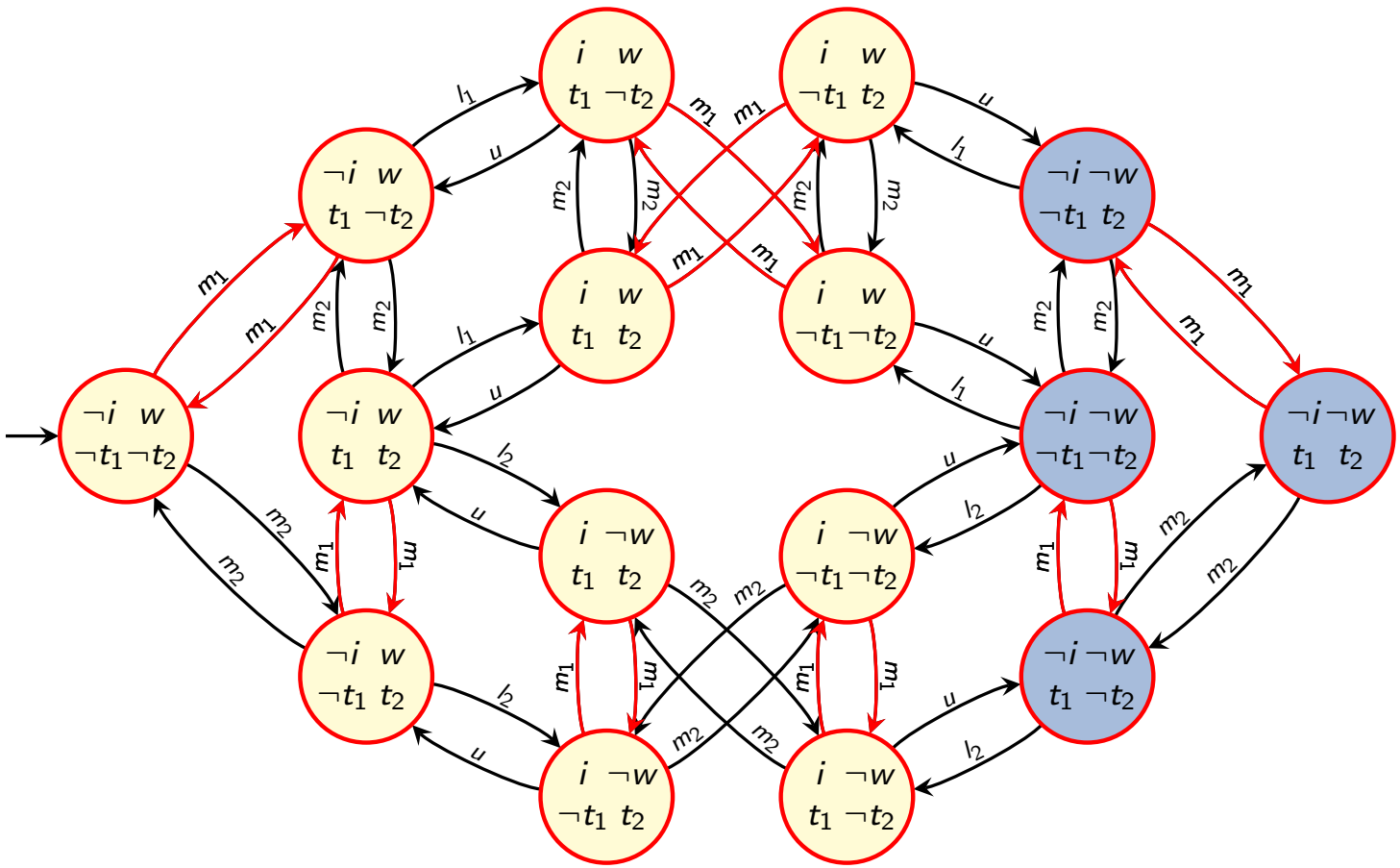
Running Example: Operator Effects

$$eff(l_2) = (i \wedge \neg w)$$



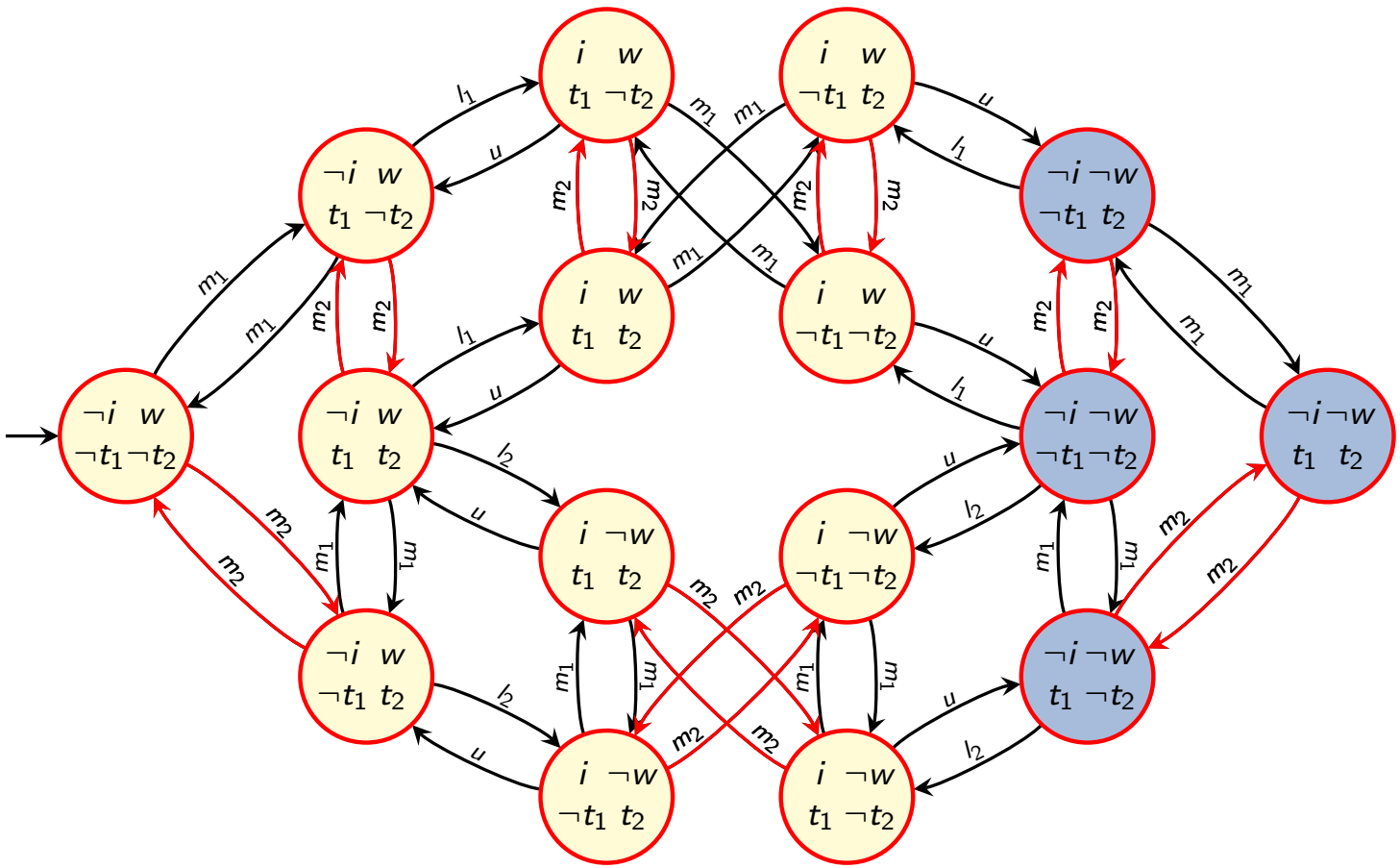
Running Example: Operator Effects

$$eff(m_1) = ((t_1 \rhd \neg t_1) \wedge (\neg t_1 \rhd t_1))$$



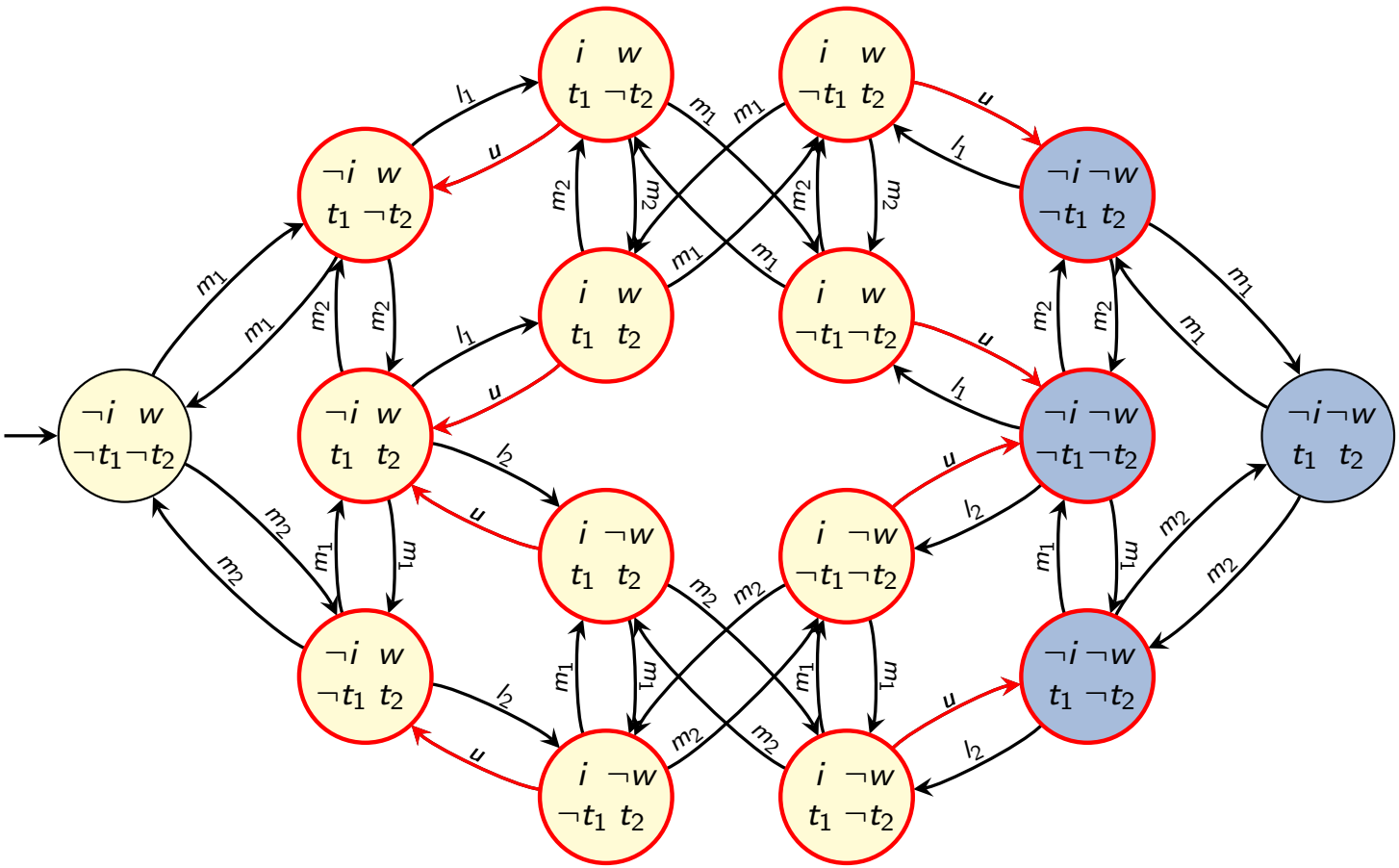
Running Example: Operator Effects

$$eff(m_2) = ((t_2 \rhd \neg t_2) \wedge (\neg t_2 \rhd t_2))$$



Running Example: Operator Effects

$$eff(u) = \neg i \wedge (w \rhd ((t_1 \rhd w) \wedge (\neg t_1 \rhd \neg w))) \\ \wedge (\neg w \rhd ((t_2 \rhd w) \wedge (\neg t_2 \rhd \neg w)))$$



B2.5 Summary

Summary

- Propositional state variables let us compactly describe properties of large transition systems.
- A state is an assignment to a set of state variables.
- Sets of states are represented as formulas over state variables.
- Operators describe when (precondition), how (effect) and at which cost the state of the world can be changed.
- Effects are structured objects including empty, atomic, conjunctive and conditional effects.

Planning and Optimization

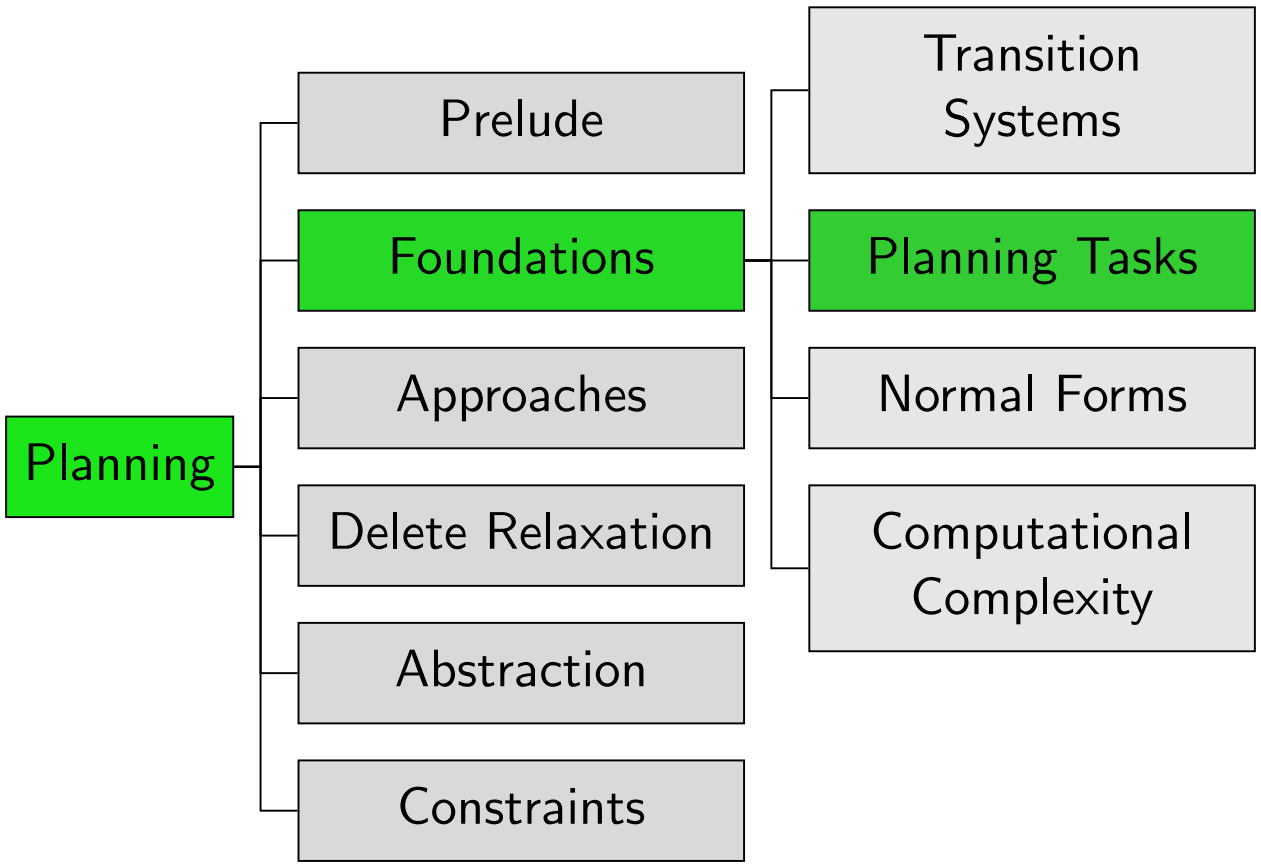
B3. Formal Definition of Planning

Malte Helmert and Gabriele Röger

Universität Basel

September 24, 2025

Content of the Course



B3.1 Semantics of Effects and Operators

Semantics of Effects: Effect Conditions

Definition (Effect Condition for an Effect)

Let ℓ be an atomic effect, and let e be an effect.

The effect condition $\mathit{effcond}(\ell, e)$ under which ℓ triggers given the effect e is a propositional formula defined as follows:

- $\mathit{effcond}(\ell, \top) = \bot$
- $\mathit{effcond}(\ell, e) = \top$ for the atomic effect $e = \ell$
- $\mathit{effcond}(\ell, e) = \bot$ for all atomic effects $e = \ell' \neq \ell$
- $\mathit{effcond}(\ell, (e \wedge e')) = (\mathit{effcond}(\ell, e) \vee \mathit{effcond}(\ell, e'))$
- $\mathit{effcond}(\ell, (\chi \rhd e)) = (\chi \wedge \mathit{effcond}(\ell, e))$

Intuition: $\mathit{effcond}(\ell, e)$ represents the condition that must be true in the current state for the effect e to lead to the atomic effect ℓ

Effect Condition: Example (1)

Example

Consider the move operator m_1 from the running example: $\mathit{eff}(m_1) = ((t_1 \rhd \neg t_1) \land (\neg t_1 \rhd t_1))$.

Under which conditions does it set t_1 to false?

$$\begin{aligned}
\mathit{effcond}(\neg t_1, \mathit{eff}(m_1)) &= \mathit{effcond}(\neg t_1, ((t_1 \rhd \neg t_1) \land (\neg t_1 \rhd t_1))) \\
&= \mathit{effcond}(\neg t_1, (t_1 \rhd \neg t_1)) \lor \\
&\quad \mathit{effcond}(\neg t_1, (\neg t_1 \rhd t_1)) \\
&= (t_1 \land \mathit{effcond}(\neg t_1, \neg t_1)) \lor \\
&\quad (\neg t_1 \land \mathit{effcond}(\neg t_1, t_1)) \\
&= (t_1 \land \top) \lor (\neg t_1 \land \bot) \\
&\equiv t_1 \lor \bot \\
&\equiv t_1
\end{aligned}$$

Effect Condition: Example (2)

Example

Consider the move operator m_1 from the running example: $\mathit{eff}(m_1) = ((t_1 \rhd \neg t_1) \land (\neg t_1 \rhd t_1))$.

Under which conditions does it set i to true?

$$\begin{aligned}
\mathit{effcond}(i, \mathit{eff}(m_1)) &= \mathit{effcond}(i, ((t_1 \rhd \neg t_1) \land (\neg t_1 \rhd t_1))) \\
&= \mathit{effcond}(i, (t_1 \rhd \neg t_1)) \lor \\
&\quad \mathit{effcond}(i, (\neg t_1 \rhd t_1)) \\
&= (t_1 \land \mathit{effcond}(i, \neg t_1)) \lor \\
&\quad (\neg t_1 \land \mathit{effcond}(i, t_1)) \\
&= (t_1 \land \bot) \lor (\neg t_1 \land \bot) \\
&\equiv \bot \lor \bot \\
&\equiv \bot
\end{aligned}$$

Semantics of Effects: Applying an Effect

first attempt:

Definition (Applying Effects)

Let V be a set of propositional state variables.
Let s be a state over V, and let e be an effect over V.

The resulting state of applying e in s, written $s[\![e]\!]$,
is the state s' defined as follows for all $v \in V$:

$$s'(v) = \begin{cases} \mathbf{T} & \text{if } s \models \mathit{effcond}(v, e) \\ \mathbf{F} & \text{if } s \models \mathit{effcond}(\neg v, e) \wedge \neg \mathit{effcond}(v, e) \\ s(v) & \text{otherwise} \end{cases}$$

What is the problem with this definition?

Semantics of Effects: Applying an Effect

correct definition:

Definition (Applying Effects)

Let V be a set of propositional state variables.
Let s be a state over V, and let e be an effect over V.

The resulting state of applying e in s, written $s[\![e]\!]$,
is the state s' defined as follows for all $v \in V$:

$$s'(v) = \begin{cases} \mathbf{T} & \text{if } s \models \mathit{effcond}(v, e) \\ \mathbf{F} & \text{if } s \models \mathit{effcond}(\neg v, e) \wedge \neg \mathit{effcond}(v, e) \\ s(v) & \text{otherwise} \end{cases}$$

Add-after-Delete Semantics

Note:

- The definition implies that if a variable is simultaneously "added" (set to **T**) and "deleted" (set to **F**), the value **T** takes precedence.
- This is called add-after-delete semantics.
- This detail of effect semantics is somewhat arbitrary, but has proven useful in applications.

Semantics of Operators

Definition (Applicable, Applying Operators, Resulting State)

Let V be a set of propositional state variables.
Let s be a state over V, and let o be an operator over V.

Operator o is applicable in s if $s \models \mathit{pre}(o)$.

If o is applicable in s, the resulting state of applying o in s, written $s[\![o]\!]$, is the state $s[\![\mathit{eff}(o)]\!]$.

B3.2 Planning Tasks

Planning Tasks

Definition (Planning Task)

A (propositional) planning task is a 4-tuple $\Pi = \langle V, I, O, \gamma \rangle$ where

- V is a finite set of propositional state variables,
- I is an interpretation of V called the initial state,
- O is a finite set of operators over V, and
- γ is a formula over V called the goal.

Running Example: Planning Task

Example

From the previous chapter, we see that the running example can be represented by the task $\Pi = \langle V, I, O, \gamma \rangle$ with

- $V = \{i, w, t_1, t_2\}$
- $I = \{i \mapsto \mathbf{F}, w \mapsto \mathbf{T}, t_1 \mapsto \mathbf{F}, t_2 \mapsto \mathbf{F}\}$
- $O = \{m_1, m_2, l_1, l_2, u\}$ where
  - $m_1 = \langle \top, ((t_1 \rhd \neg t_1) \wedge (\neg t_1 \rhd t_1)), 5 \rangle$
  - $m_2 = \langle \top, ((t_2 \rhd \neg t_2) \wedge (\neg t_2 \rhd t_2)), 5 \rangle$
  - $l_1 = \langle \neg i \wedge (w \leftrightarrow t_1), (i \wedge w), 1 \rangle$
  - $l_2 = \langle \neg i \wedge (w \leftrightarrow t_2), (i \wedge \neg w), 1 \rangle$
  - $u = \langle i, \neg i \wedge (w \rhd ((t_1 \rhd w) \wedge (\neg t_1 \rhd \neg w))) \wedge (\neg w \rhd ((t_2 \rhd w) \wedge (\neg t_2 \rhd \neg w))), 1 \rangle$
- $\gamma = \neg i \wedge \neg w$

Mapping Planning Tasks to Transition Systems

Definition (Transition System Induced by a Planning Task)

The planning task $\Pi = \langle V, I, O, \gamma \rangle$ induces the transition system $\mathcal{T}(\Pi) = \langle S, L, c, T, s_0, S_\star \rangle$, where

- S is the set of all states over V,
- L is the set of operators O,
- $c(o) = \mathit{cost}(o)$ for all operators $o \in O$,
- $T = \{\langle s, o, s' \rangle \mid s \in S,\ o \text{ applicable in } s,\ s' = s[\![o]\!]\}$,
- $s_0 = I$, and
- $S_\star = \{s \in S \mid s \models \gamma\}$.

Planning Tasks: Terminology

- Terminology for transitions systems is also applied to the planning tasks Π that induce them.
- For example, when we speak of the states of Π, we mean the states of $\mathcal{T}(\Pi)$.
- A sequence of operators that forms a solution of $\mathcal{T}(\Pi)$ is called a plan of Π.

Satisficing and Optimal Planning

By planning, we mean the following two algorithmic problems:

Definition (Satisficing Planning)

Given: a planning task Π
Output: a plan for Π, or **unsolvable** if no plan for Π exists

Definition (Optimal Planning)

Given: a planning task Π
Output: a plan for Π with minimal cost among all plans for Π, or **unsolvable** if no plan for Π exists

B3.3 Summary

Summary

- Planning tasks compactly represent transition systems and are suitable as inputs for planning algorithms.
- A planning task consists of a set of state variables and an initial state, operators and goal over these state variables.
- We gave formal definitions for these concepts.
- In satisficing planning, we must find a solution for a planning task (or show that no solution exists).
- In optimal planning, we must additionally guarantee that generated solutions are of minimal cost.

Planning and Optimization

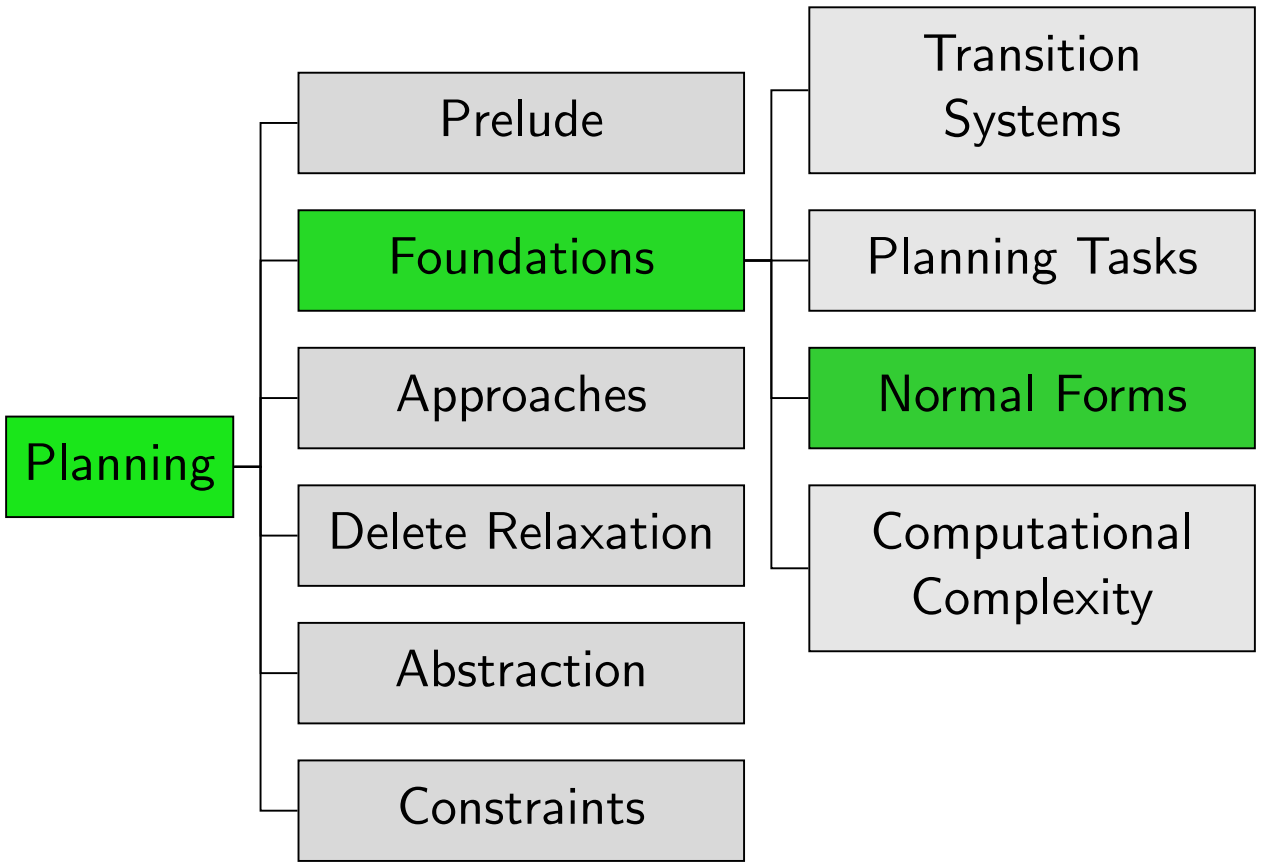
B4. Equivalent Operators and Normal Forms

Malte Helmert and Gabriele Röger

Universität Basel

September 29, 2025

Content of the Course



B4.1 Reminder & Motivation

Reminder: Syntax of Effects

Definition (Effect)

Effects over propositional state variables V are inductively defined as follows:

- $\top$ is an effect (empty effect).
- If $v \in V$ is a propositional state variable, then v and $\neg v$ are effects (atomic effect).
- If e and e' are effects, then $(e \wedge e')$ is an effect (conjunctive effect).
- If χ is a formula over V and e is an effect, then $(\chi \rhd e)$ is an effect (conditional effect).

Arbitrary nesting of conjunctive and conditional effects,
e.g. $c \wedge (a \rhd (\neg b \wedge (c \rhd (b \wedge \neg d \wedge \neg a)))) \wedge (\neg b \rhd \neg a)$
$\rightsquigarrow$ Can we make our life easier?

Reminder: Semantics of Effects

- $\mathit{effcond}(\ell, e)$: condition that must be true in the current state for the effect e to trigger the atomic effect ℓ
- add-after-delete semantics:
  if an operator with effect e is applied in state s
  and we have both $s \models \mathit{effcond}(v, e)$ and $s \models \mathit{effcond}(\neg v, e)$,
  then $s'(v) = \mathbf{T}$ in the resulting state s'.

This is a very subtle detail.

$\rightsquigarrow$ Can we make our life easier?

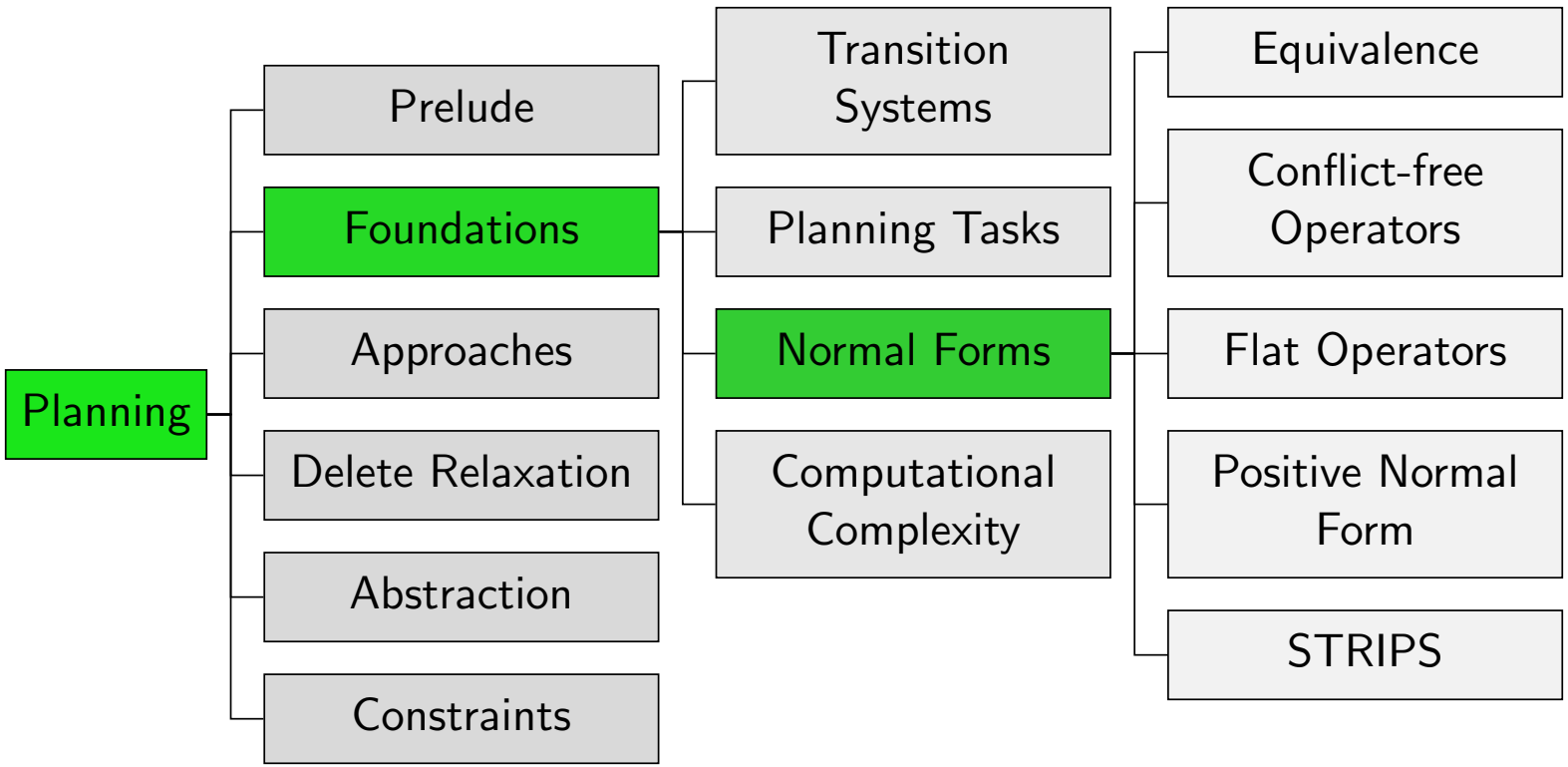
Motivation

Similarly to normal forms in propositional logic (DNF, CNF, NNF), we can define normal forms for effects, operators and planning tasks.

Among other things, we consider normal forms that avoid complicated nesting and subtleties of conflicts.

This is useful because algorithms (and proofs) then only need to deal with effects, operators and tasks in normal form.

Content of the Course



Notation: Applying Operator Sequences

Existing notation:

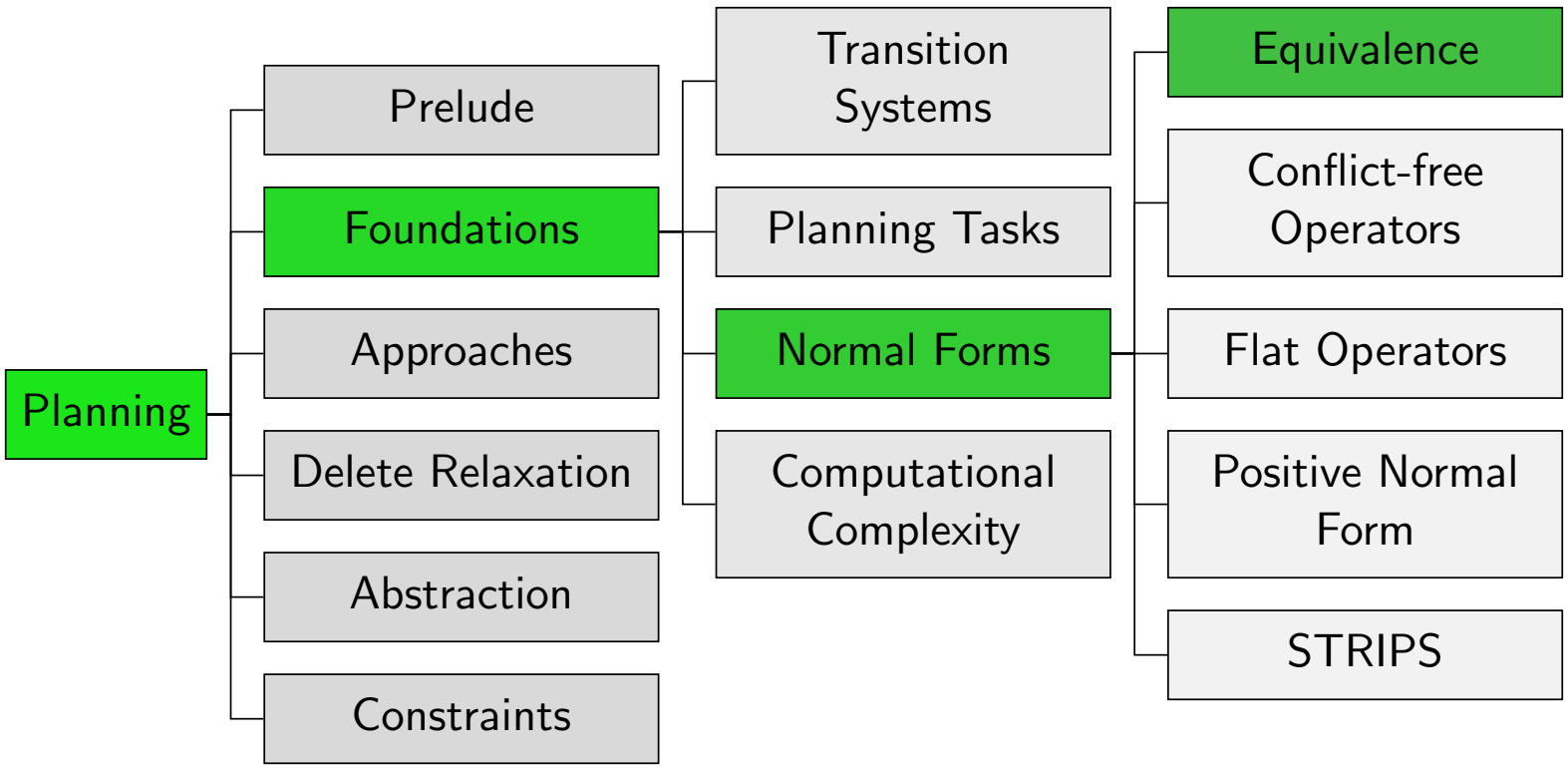
- We already write $s[\![o]\!]$ for the resulting state after applying operator o in state s.

New extended notation:

- For a sequence $\pi = \langle o_1, \ldots, o_n \rangle$ of operators that are consecutively applicable in s, we write $s[\![\pi]\!]$ for $s[\![o_1]\!][\![o_2]\!] \ldots [\![o_n]\!]$.
- This includes the case of an empty operator sequence: $s[\![\langle\rangle]\!] = s$

B4.2 Equivalence Transformations

Content of the Course



Equivalence of Operators and Effects: Definition

Definition (Equivalent Effects)

Two effects e and e' over state variables V are equivalent, written $e \equiv e'$, if $s[\![e]\!] = s[\![e']\!]$ for all states s.

Definition (Equivalent Operators)

Two operators o and o' over state variables V are equivalent, written $o \equiv o'$, if $\mathit{cost}(o) = \mathit{cost}(o')$ and for all states s, s' over V, o induces the transition $s \xrightarrow{o} s'$ iff o' induces the transition $s \xrightarrow{o'} s'$.

Equivalence of Operators and Effects: Theorem

Theorem

Let o and o' be operators with $pre(o) \equiv pre(o')$, $eff(o) \equiv eff(o')$ and $cost(o) = cost(o')$. Then $o \equiv o'$.

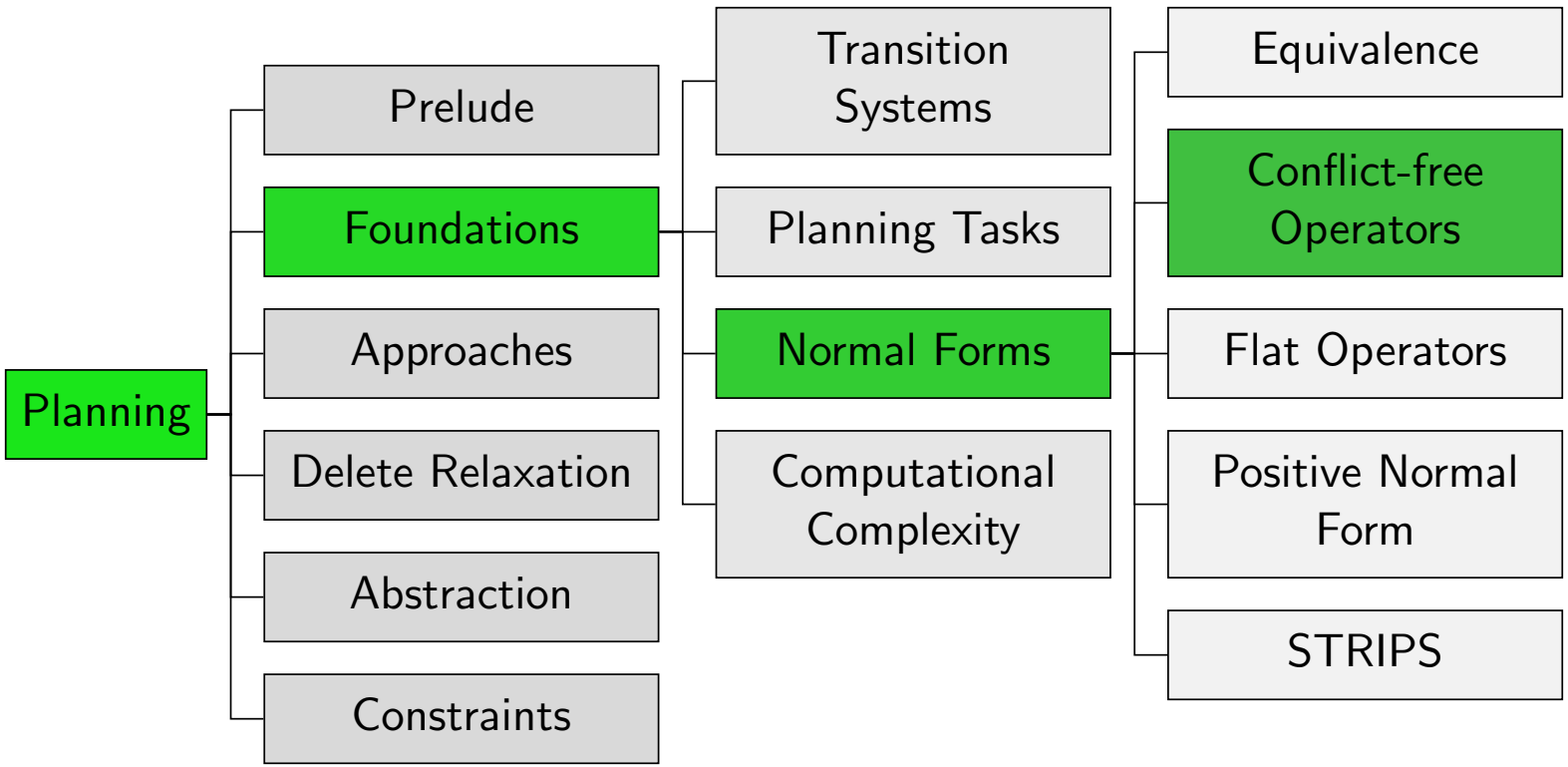
Note: The converse is not true. (Why not?)

Equivalence Transformations for Effects

$$
\begin{aligned}
e \wedge e' &\equiv e' \wedge e && (1)\\
(e \wedge e') \wedge e'' &\equiv e \wedge (e' \wedge e'') && (2)\\
\top \wedge e &\equiv e && (3)\\
\chi \rhd e &\equiv \chi' \rhd e \qquad \text{if } \chi \equiv \chi' && (4)\\
\top \rhd e &\equiv e && (5)\\
\bot \rhd e &\equiv \top && (6)\\
\chi \rhd (\chi' \rhd e) &\equiv (\chi \wedge \chi') \rhd e && (7)\\
\chi \rhd (e \wedge e') &\equiv (\chi \rhd e) \wedge (\chi \rhd e') && (8)\\
(\chi \rhd e) \wedge (\chi' \rhd e) &\equiv (\chi \vee \chi') \rhd e && (9)
\end{aligned}
$$

B4.3 Conflict-Free Operators

Content of the Course



Conflict-Freeness: Motivation

- The add-after-delete semantics makes effects like $(a \rhd c) \wedge (b \rhd \neg c)$ somewhat unintuitive to interpret.

$\rightsquigarrow$ What happens in states where $a \wedge b$ is true?

- It would be nicer if $\mathit{effcond}(\ell, e)$ always were the condition under which the atomic effect ℓ actually materializes (because of add-after-delete, it is not)

$\rightsquigarrow$ introduce normal form where "complicated case" never arises

Conflict-Free Effects and Operators

Definition (Conflict-Free)

An effect e over propositional state variables V
is called conflict-free if $\mathit{effcond}(v, e) \wedge \mathit{effcond}(\neg v, e)$
is unsatisfiable for all $v \in V$.

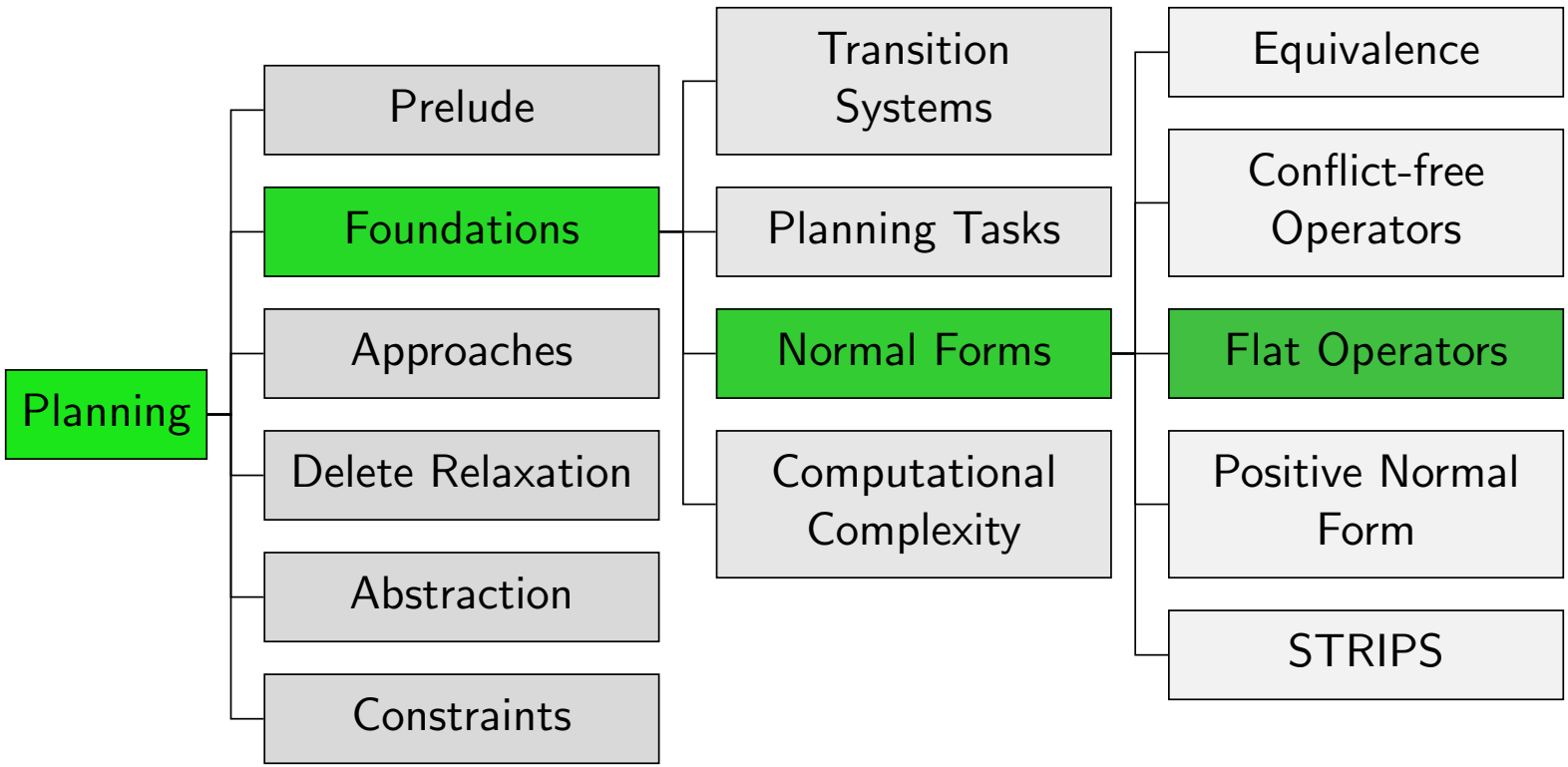
An operator o is called conflict-free if $\mathit{eff}(o)$ is conflict-free.

Making Operators Conflict-Free

- In general, testing whether an operator is conflict-free is a coNP-complete problem. (Why?)
- However, we do not necessarily need such a test. Instead, we can produce an equivalent conflict-free operator in polynomial time.
- Algorithm: given operator o, replace all atomic effects of the form $\neg v$ by $(\neg \mathit{effcond}(v, \mathit{eff}(o)) \rhd \neg v)$. The resulting operator o' is conflict-free and $o \equiv o'$. (Why?)

B4.4 Flat Effects

Content of the Course



Flat Effects: Motivation

- CNF and DNF limit the nesting of connectives in propositional logic.
- For example, a CNF formula is
  - a conjunction of 0 or more subformulas,
  - each of which is a disjunction of 0 or more subformulas,
  - each of which is a literal.
- Similarly, we can define a normal form that limits the nesting of effects.
- This is useful because we then do not have to consider arbitrarily structured effects, e.g., when representing them in a planning algorithm.

Flat Effect

Definition (Flat Effect)

An effect is simple if it is either an atomic effect
or of the form $(\chi \rhd e)$, where e is an atomic effect.

An effect e is flat if it is a conjunction of 0 or more simple effects,
and none of these simple effects include the same atomic effect.

An operator o is flat if $\mathit{eff}(o)$ is flat.

Notes: analogously to CNF, we consider

- a single simple effect as a conjunction of 1 simple effect
- the empty effect as a conjunction of 0 simple effects

Flat Effect: Example

Example

Consider the effect

$$c \wedge (a \rhd (\neg b \wedge (c \rhd (b \wedge \neg d \wedge \neg a)))) \wedge (\neg b \rhd \neg a)$$

An equivalent flat (and conflict-free) effect is

$$\begin{aligned} & c \wedge \\ & ((a \wedge \neg c) \rhd \neg b) \wedge \\ & ((a \wedge c) \rhd b) \wedge \\ & ((a \wedge c) \rhd \neg d) \wedge \\ & ((\neg b \vee (a \wedge c)) \rhd \neg a) \end{aligned}$$

Note: if we want, we can write c as $(\top \rhd c)$ to make the structure even more uniform, with each simple effect having a condition.

Producing Flat Operators

Theorem

For every operator, an equivalent flat operator and an equivalent flat, conflict-free operator can be computed in polynomial time.

Producing Flat Operators: Proof

Proof Sketch.

Replace the effect e over variables V by

$$\bigwedge_{v \in V}(\mathit{effcond}(v, e) \rhd v) \\ \wedge \bigwedge_{v \in V}(\mathit{effcond}(\neg v, e) \rhd \neg v),$$

which is an equivalent flat effect.

To additionally obtain conflict-freeness, use

$$\bigwedge_{v \in V}(\mathit{effcond}(v, e) \rhd v) \\ \wedge \bigwedge_{v \in V}((\mathit{effcond}(\neg v, e) \wedge \neg \mathit{effcond}(v, e)) \rhd \neg v)$$

instead.

(Conjuncts of the form $(\chi \rhd e)$ where $\chi \equiv \bot$ can be omitted to simplify the effect.)

B4.5 Summary

Summary

- Equivalences can be used to simplify operators and effects.
- In conflict-free operators, the "complicated case" of operator semantics does not arise.
- For flat operators, the only permitted nesting is atomic effects within conditional effects within conjunctive effects, and all atomic effects must be distinct.
- For flat, conflict-free operators, it is easy to determine the condition under which a given literal is made true by applying the operator in a given state.
- Every operator can be transformed into an equivalent flat and conflict-free one in polynomial time.

Planning and Optimization

B5. Positive Normal Form and STRIPS

Malte Helmert and Gabriele Röger

Universität Basel

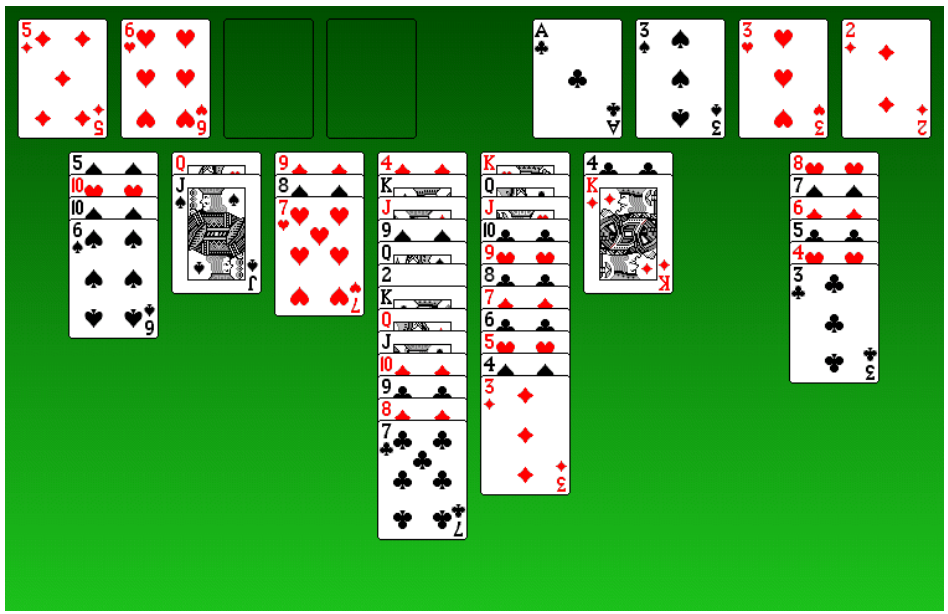
September 29, 2025

Planning and Optimization

September 29, 2025 — B5. Positive Normal Form and STRIPS

B5.1 Motivation

Example: Freecell



Example (Good and Bad Effects)

If we move K♢ to a free tableau position,
the good effect is that 4♣ is now accessible.
The bad effect is that we lose one free tableau position.

What is a Good or Bad Effect?

Question: Which operator effects are good, and which are bad?

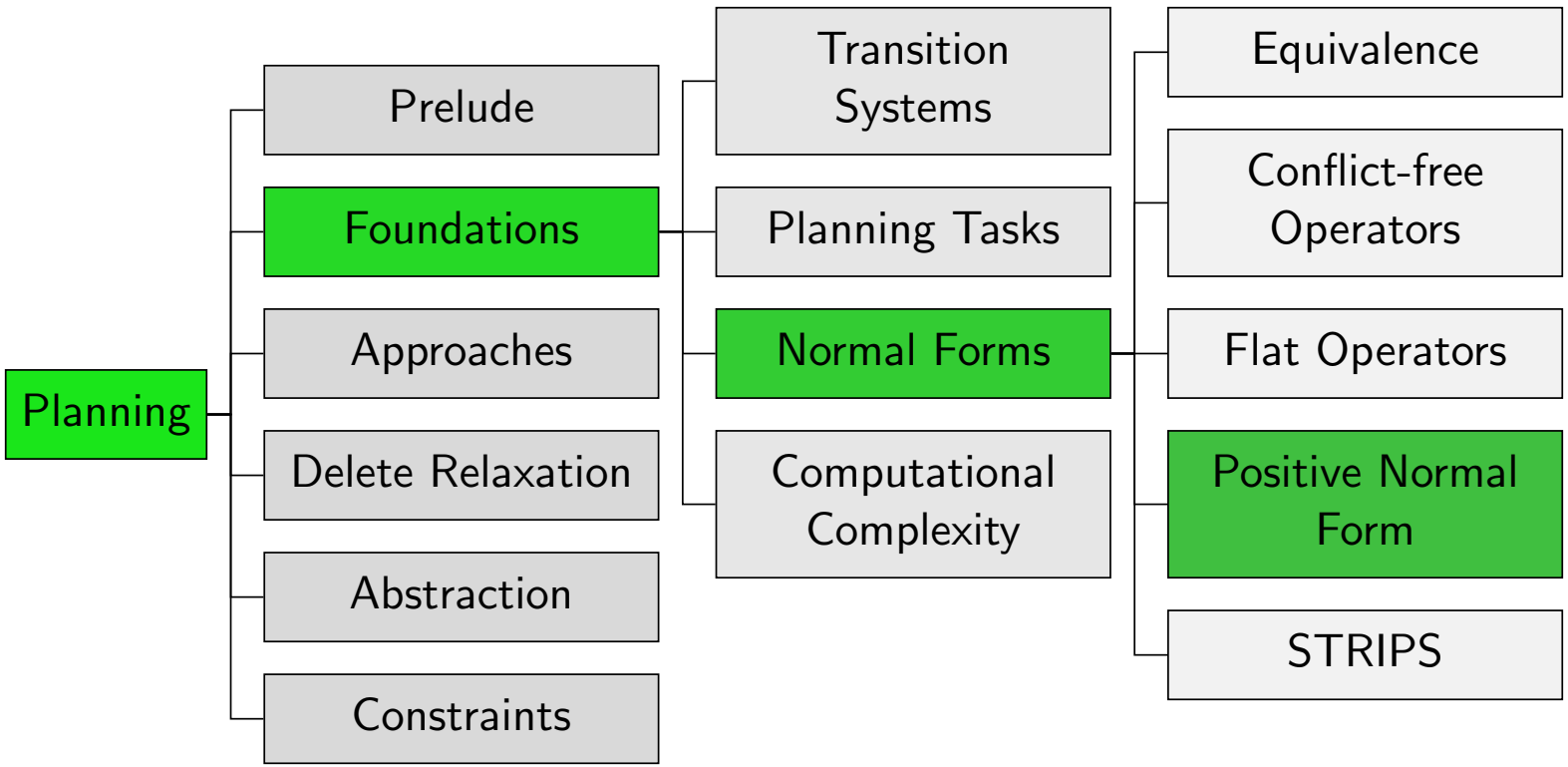
Difficult to answer in general, because it depends on context:

- Locking our door is good if we want to keep burglars out.
- Locking our door is bad if we want to enter.

We now consider a reformulation of propositional planning tasks that makes the distinction between good and bad effects obvious.

B5.2 Positive Normal Form

Content of the Course



Positive Formulas, Operators and Tasks

Definition (Positive Formula)

A logical formula φ is positive if no negation symbols appear in φ.

Note: This includes the negation symbols implied by $\rightarrow$ and $\leftrightarrow$.

Definition (Positive Operator)

An operator o is positive if $\mathit{pre}(o)$ and
all effect conditions in $\mathit{eff}(o)$ are positive.

Definition (Positive Propositional Planning Task)

A propositional planning task $\langle V, I, O, \gamma \rangle$ is positive
if all operators in O and the goal γ are positive.

Positive Normal Form

Definition (Positive Normal Form)

A propositional planning task is in positive normal form
if it is positive and all operator effects are flat.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Identify state variable v occurring negatively in conditions.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Introduce new variable $\hat{v}$ with complementary initial value.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Identify effects on variable v.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \wedge \mathit{bike\text{-}unlocked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \wedge \neg \mathit{bike\text{-}unlocked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Introduce complementary effects for $\hat{v}$.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni} \rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \wedge \mathit{bike\text{-}unlocked} \rangle,$$
$$\langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \wedge \neg \mathit{bike\text{-}unlocked} \rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike}) \rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Identify negative conditions for v.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$
$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$
$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \mathit{bike\text{-}unlocked}, \neg\mathit{home} \wedge \mathit{uni}\rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg\mathit{bike\text{-}locked} \wedge \mathit{bike\text{-}unlocked}\rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}unlocked}, \mathit{bike\text{-}locked} \wedge \neg\mathit{bike\text{-}unlocked}\rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \mathit{bike\text{-}unlocked}) \rhd \neg\mathit{bike})\rangle\}$$
$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Replace by positive condition $\hat{v}$.

Positive Normal Form: Example

Example (Transformation to Positive Normal Form)

$$V = \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\}$$

$$I = \{\mathit{home} \mapsto \mathbf{T}, \mathit{bike} \mapsto \mathbf{T}, \mathit{bike\text{-}locked} \mapsto \mathbf{T},$$
$$\mathit{uni} \mapsto \mathbf{F}, \mathit{lecture} \mapsto \mathbf{F}, \mathit{bike\text{-}unlocked} \mapsto \mathbf{F}\}$$

$$O = \{\langle \mathit{home} \wedge \mathit{bike} \wedge \mathit{bike\text{-}unlocked}, \neg\mathit{home} \wedge \mathit{uni}\rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg\mathit{bike\text{-}locked} \wedge \mathit{bike\text{-}unlocked}\rangle,$$
$$\langle \mathit{bike} \wedge \mathit{bike\text{-}unlocked}, \mathit{bike\text{-}locked} \wedge \neg\mathit{bike\text{-}unlocked}\rangle,$$
$$\langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \mathit{bike\text{-}unlocked}) \rhd \neg\mathit{bike})\rangle\}$$

$$\gamma = \mathit{lecture} \wedge \mathit{bike}$$

Positive Normal Form: Existence

Theorem (Positive Normal Form)

For every propositional planning task Π, there is an equivalent propositional planning task Π' in positive normal form. Moreover, Π' can be computed from Π in polynomial time.

Note: Equivalence here means that the transition systems induced by Π and Π', restricted to the reachable states, are isomorphic.

We prove the theorem by describing a suitable algorithm. (However, we do not prove its correctness or complexity.)

Positive Normal Form: Algorithm

Transformation of $\langle V, I, O, \gamma \rangle$ to Positive Normal Form

Replace all operators with equivalent conflict-free operators.
Convert all conditions to negation normal form (NNF).
while any condition contains a negative literal $\neg v$:
    Let v be a variable which occurs negatively in a condition.
    $V := V \cup \{\hat{v}\}$ for some new propositional state variable $\hat{v}$
    $I(\hat{v}) := \begin{cases} \mathbf{F} & \text{if } I(v) = \mathbf{T} \\ \mathbf{T} & \text{if } I(v) = \mathbf{F} \end{cases}$
    Replace the effect v by $(v \wedge \neg\hat{v})$ in all operators $o \in O$.
    Replace the effect $\neg v$ by $(\neg v \wedge \hat{v})$ in all operators $o \in O$.
    Replace $\neg v$ by $\hat{v}$ in all conditions.
Convert all operators $o \in O$ to flat operators.

Here, all conditions refers to all operator preconditions, operator effect conditions and the goal.

Why Positive Normal Form is Interesting

In the absence of conditional effects, positive normal form allows us to distinguish good and bad effects easily:

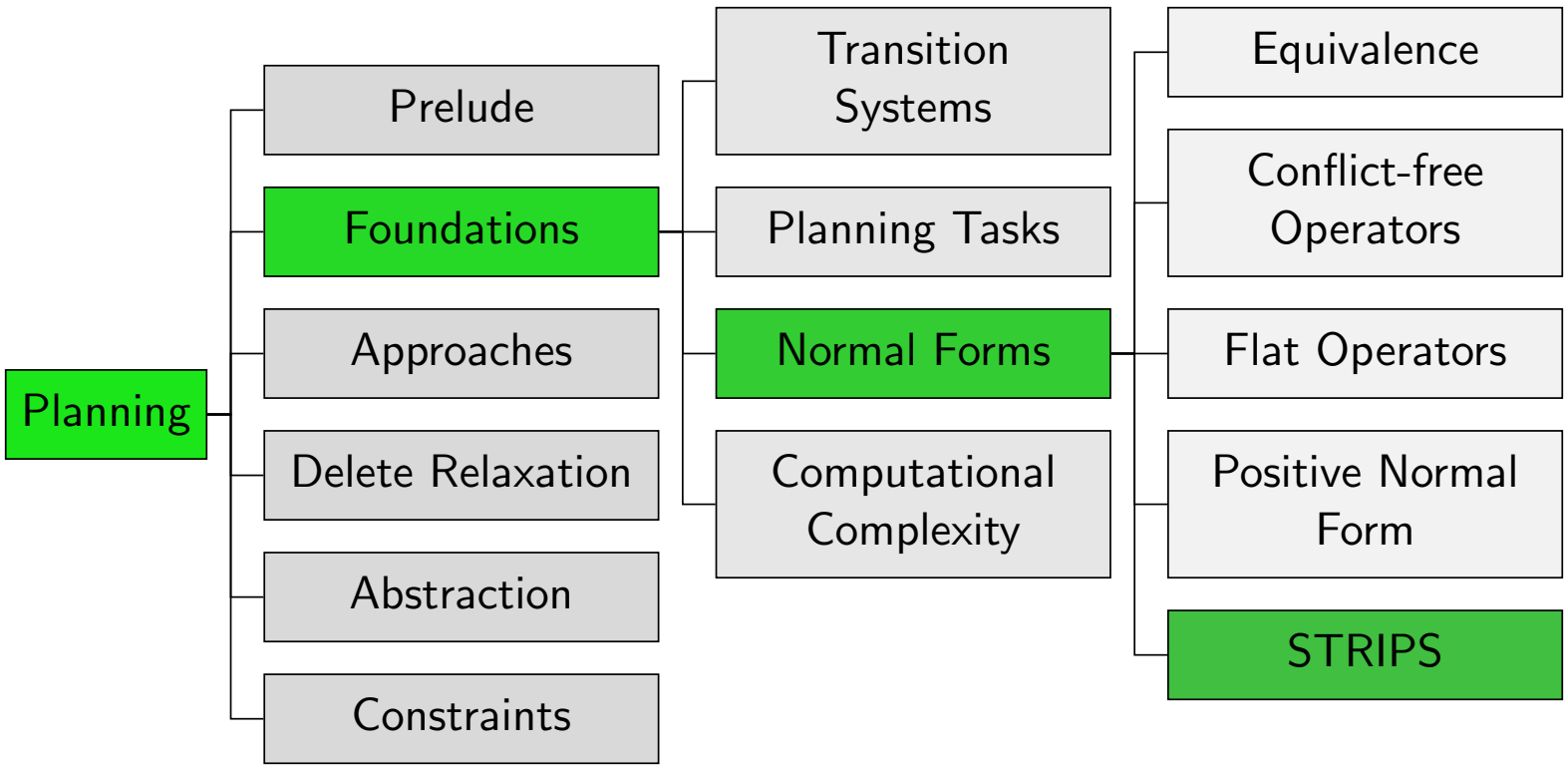
- Effects that make state variables true (add effects) are good.
- Effects that make state variables false (delete effects) are bad.

This is particularly useful for planning algorithms based on delete relaxation, which we will study in Part D.

(Why restriction "in the absence of conditional effects"?)

B5.3 STRIPS

Content of the Course



STRIPS Operators and Planning Tasks

Definition (STRIPS Operator)

An operator o of a prop. planning task is a STRIPS operator if

- $pre(o)$ is a conjunction of state variables, and
- $eff(o)$ is a conflict-free conjunction of atomic effects.

Definition (STRIPS Planning Task)

A propositional planning task $\langle V, I, O, \gamma \rangle$ is a STRIPS planning task if all operators $o \in O$ are STRIPS operators and γ is a conjunction of state variables.

Note: STRIPS operators are conflict-free and flat.
STRIPS is a special case of positive normal form.

STRIPS Operators: Remarks

- Every STRIPS operator is of the form

$$\langle v_1 \wedge \cdots \wedge v_n, \quad \ell_1 \wedge \cdots \wedge \ell_m \rangle$$

  where v_i are state variables and ℓ_j are atomic effects.
- Often, STRIPS operators o are described via three sets of state variables:
  - the preconditions (state variables occurring in $pre(o)$)
  - the add effects (state variables occurring positively in $eff(o)$)
  - the delete effects (state variables occurring negatively in $eff(o)$)
- Definitions of STRIPS in the literature often do not require conflict-freeness. But it is easy to achieve and makes many things simpler.
- There exists a variant called STRIPS with negation where negative literals are also allowed in conditions.

Why STRIPS is Interesting

- STRIPS is particularly simple, yet expressive enough to capture general planning tasks.
- In particular, STRIPS planning is no easier than planning in general (as we will see in Chapters B6–B7).
- Many algorithms in the planning literature are only presented for STRIPS planning tasks (generalization is often, but not always, obvious).

STRIPS

STanford Research Institute Problem Solver
(Fikes & Nilsson, 1971)

Transformation to STRIPS

- Not every operator is equivalent to a STRIPS operator.
- However, each operator can be transformed into a set of STRIPS operators whose "combination" is equivalent to the original operator. (How?)
- However, this transformation may exponentially increase the number of operators. There are planning tasks for which such a blow-up is unavoidable.
- There are polynomial transformations of propositional planning tasks to STRIPS, but these do not lead to isomorphic transition systems (auxiliary states are needed). (They are, however, equivalent in a weaker sense.)

B5.4 Summary

Summary

- A positive task helps distinguish good and bad effects.
  The notion of positive tasks only exists for propositional tasks.
- A positive task with flat operators is in positive normal form.
- STRIPS is even more restrictive than positive normal form,
  forbidding complex preconditions and conditional effects.
- Both forms are expressive enough to capture
  general propositional planning tasks.
- Transformation to positive normal form is possible
  with polynomial size increase.
- Isomorphic transformations of propositional planning tasks to
  STRIPS can increase the number of operators exponentially;
  non-isomorphic polynomial transformations exist.

Planning and Optimization

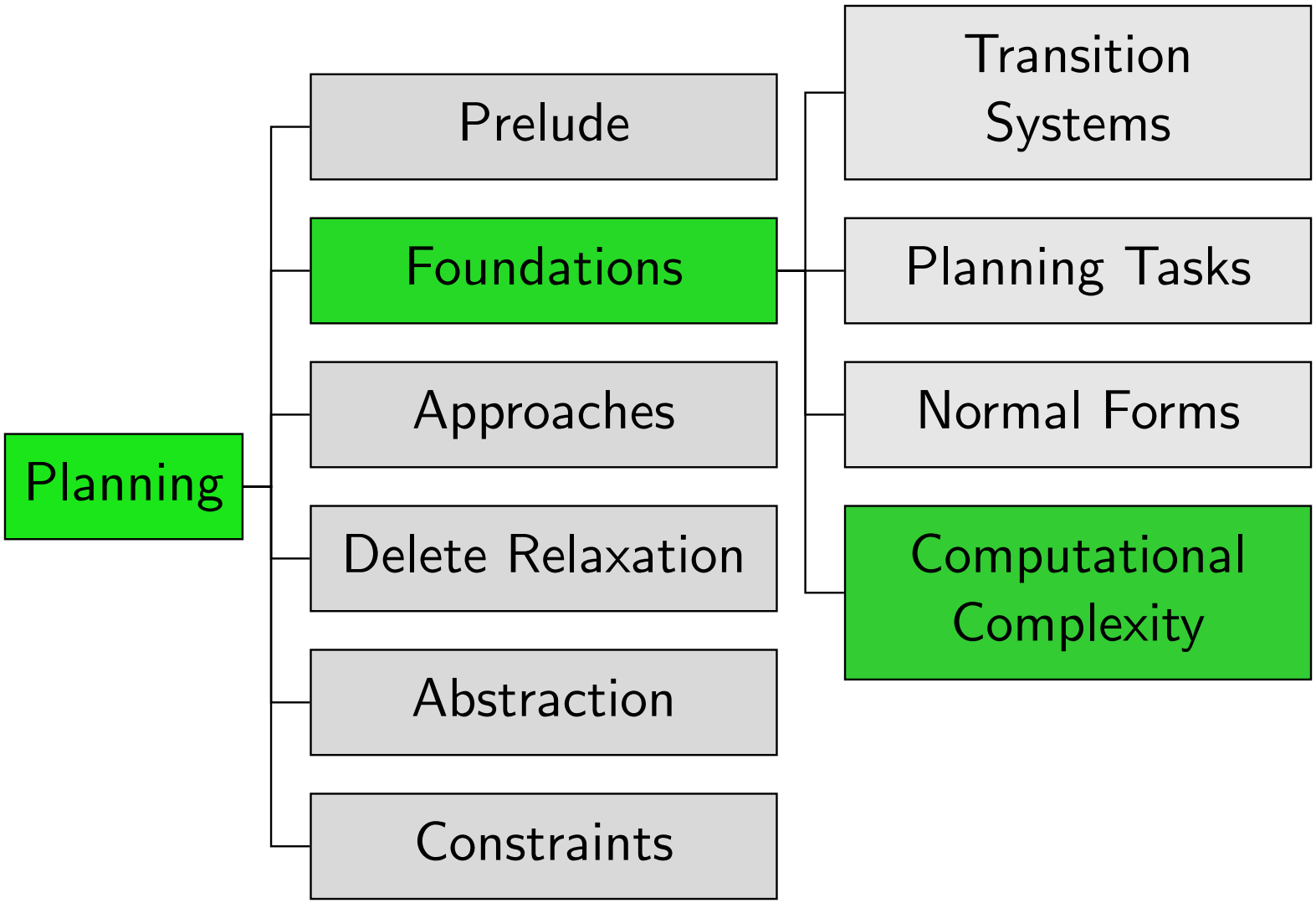
B6. Computational Complexity of Planning: Background

Malte Helmert and Gabriele Röger

Universität Basel

October 1, 2025

Content of the Course



B6.1 Motivation

How Difficult is Planning?

- Using state-space search (e.g., using Dijkstra's algorithm on the transition system), planning can be solved in polynomial time in the number of states.
- However, the number of states is exponential in the number of state variables, and hence in general exponential in the size of the input to the planning algorithm.

⇝ Do non-exponential planning algorithms exist?

⇝ What is the precise computational complexity of planning?

Why Computational Complexity?

- understand the problem
- know what is not possible
- find interesting subproblems that are easier to solve
- distinguish essential features from syntactic sugar
  - Is STRIPS planning easier than general planning?

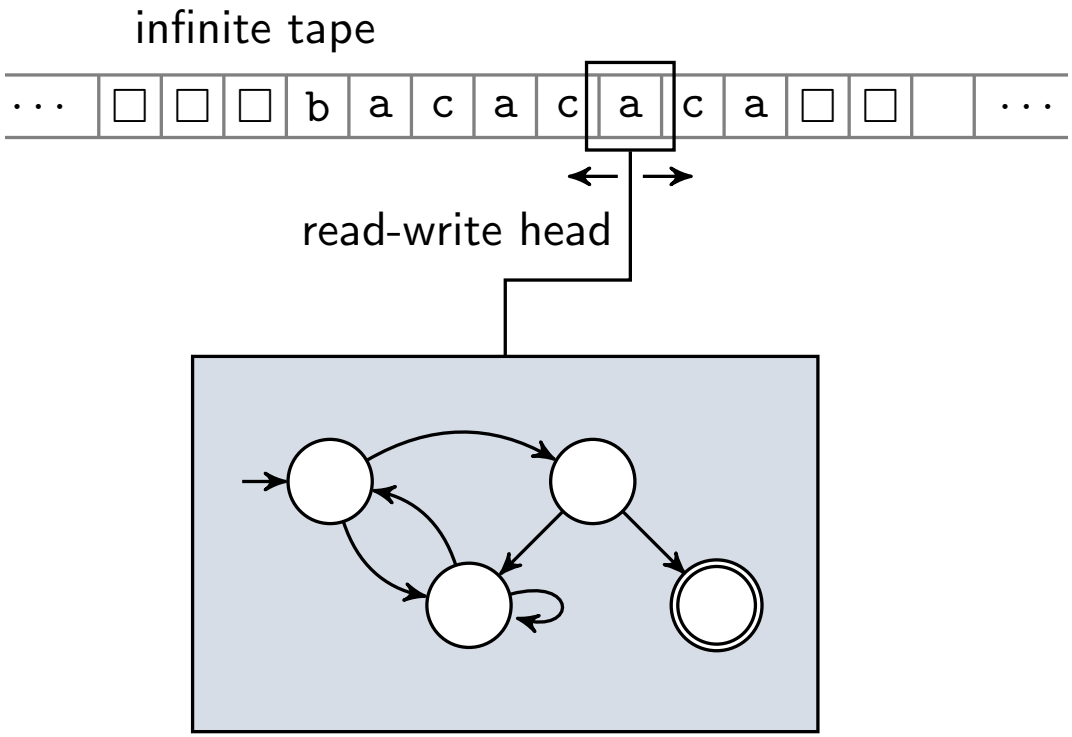
Reminder: Complexity Theory

Need to Catch Up?

- We assume knowledge of complexity theory:
  - languages and decision problems
  - Turing machines: NTMs and DTMs;
    polynomial equivalence with other models of computation
  - complexity classes: P, NP, PSPACE
  - polynomial reductions
- If you are not familiar with these topics, we recommend Chapters B11, D1–D3, D6 of the Theory of Computer Science course at `https://dmi.unibas.ch/en/studium/computer-science-informatik/lehrangebot-fs25/10948-main-lecture-theory-of-computer-science/`

B6.2 Turing Machines

Turing Machines: Conceptually



Turing Machines

Definition (Nondeterministic Turing Machine)

A nondeterministic Turing machine (NTM) is a 6-tuple $\langle \Sigma, \square, Q, q_0, q_Y, \delta \rangle$ with the following components:

- input alphabet Σ and blank symbol $\square \notin \Sigma$
  - alphabets always nonempty and finite
  - tape alphabet $\Sigma_\square = \Sigma \cup \{\square\}$
- finite set Q of internal states with initial state $q_0 \in Q$ and accepting state $q_Y \in Q$
  - nonterminal states $Q' := Q \setminus \{q_Y\}$
- transition relation $\delta : (Q' \times \Sigma_\square) \to 2^{Q \times \Sigma_\square \times \{-1,+1\}}$

Deterministic Turing machine (DTM):
$|\delta(q, s)| = 1$ for all $\langle q, s \rangle \in Q' \times \Sigma_\square$

Turing Machines: Accepted Words

- Initial configuration
  - state q_0
  - input word on tape, all other tape cells contain $\square$
  - head on first symbol of input word
- Step
  - If in state q, reading symbol s, and $\langle q', s', d \rangle \in \delta(q, s)$ then
  - the NTM can transition to state q', replacing s with s' and moving the head one cell to the left/right ($d = -1/+1$).
- Input word ($\in \Sigma^*$) is accepted if some sequence of transitions brings the NTM from the initial configuration into state q_Y.

B6.3 Complexity Classes

Acceptance in Time and Space

Definition (Acceptance of a Language in Time/Space)

Let $f : \mathbb{N}_0 \to \mathbb{N}_0$.

A NTM accepts language $L \subseteq \Sigma^*$ in time f if it accepts each $w \in L$ within $f(|w|)$ steps and does not accept any $w \notin L$ (in any time).

It accepts language $L \subseteq \Sigma^*$ in space f if it accepts each $w \in L$ using at most $f(|w|)$ tape cells and does not accept any $w \notin L$.

Time and Space Complexity Classes

Definition (DTIME, NTIME, DSPACE, NSPACE)

Let $f : \mathbb{N}_0 \to \mathbb{N}_0$.

Complexity class DTIME(f) contains all languages accepted in time f by some DTM.

Complexity class NTIME(f) contains all languages accepted in time f by some NTM.

Complexity class DSPACE(f) contains all languages accepted in space f by some DTM.

Complexity class NSPACE(f) contains all languages accepted in space f by some NTM.

Polynomial Time and Space Classes

Let $\mathcal{P}$ be the set of polynomials $p : \mathbb{N}_0 \to \mathbb{N}_0$ whose coefficients are natural numbers.

Definition (P, NP, PSPACE, NPSPACE)

$$\begin{aligned} \mathsf{P} &= \bigcup_{p \in \mathcal{P}} \mathsf{DTIME}(p) \\ \mathsf{NP} &= \bigcup_{p \in \mathcal{P}} \mathsf{NTIME}(p) \\ \mathsf{PSPACE} &= \bigcup_{p \in \mathcal{P}} \mathsf{DSPACE}(p) \\ \mathsf{NPSPACE} &= \bigcup_{p \in \mathcal{P}} \mathsf{NSPACE}(p) \end{aligned}$$

Polynomial Complexity Class Relationships

Theorem (Complexity Class Hierarchy)

$P \subseteq NP \subseteq PSPACE = NPSPACE$

Proof.

$P \subseteq NP$ and $PSPACE \subseteq NPSPACE$ are obvious because deterministic Turing machines are a special case of nondeterministic ones.

$NP \subseteq NPSPACE$ holds because a Turing machine can only visit polynomially many tape cells within polynomial time.

$PSPACE = NPSPACE$ is a special case of a classical result known as Savitch's theorem (Savitch 1970). □

B6.4 Summary

Summary

- We recalled the definitions of the most important complexity classes from complexity theory:
  - P: decision problems solvable in polynomial time
  - NP: decision problems solvable in polynomial time by nondeterministic algorithms
  - PSPACE: decision problems solvable in polynomial space
  - NPSPACE: decision problems solvable in polynomial space by nondeterministic algorithms
- These classes are related by $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE}$.

Planning and Optimization

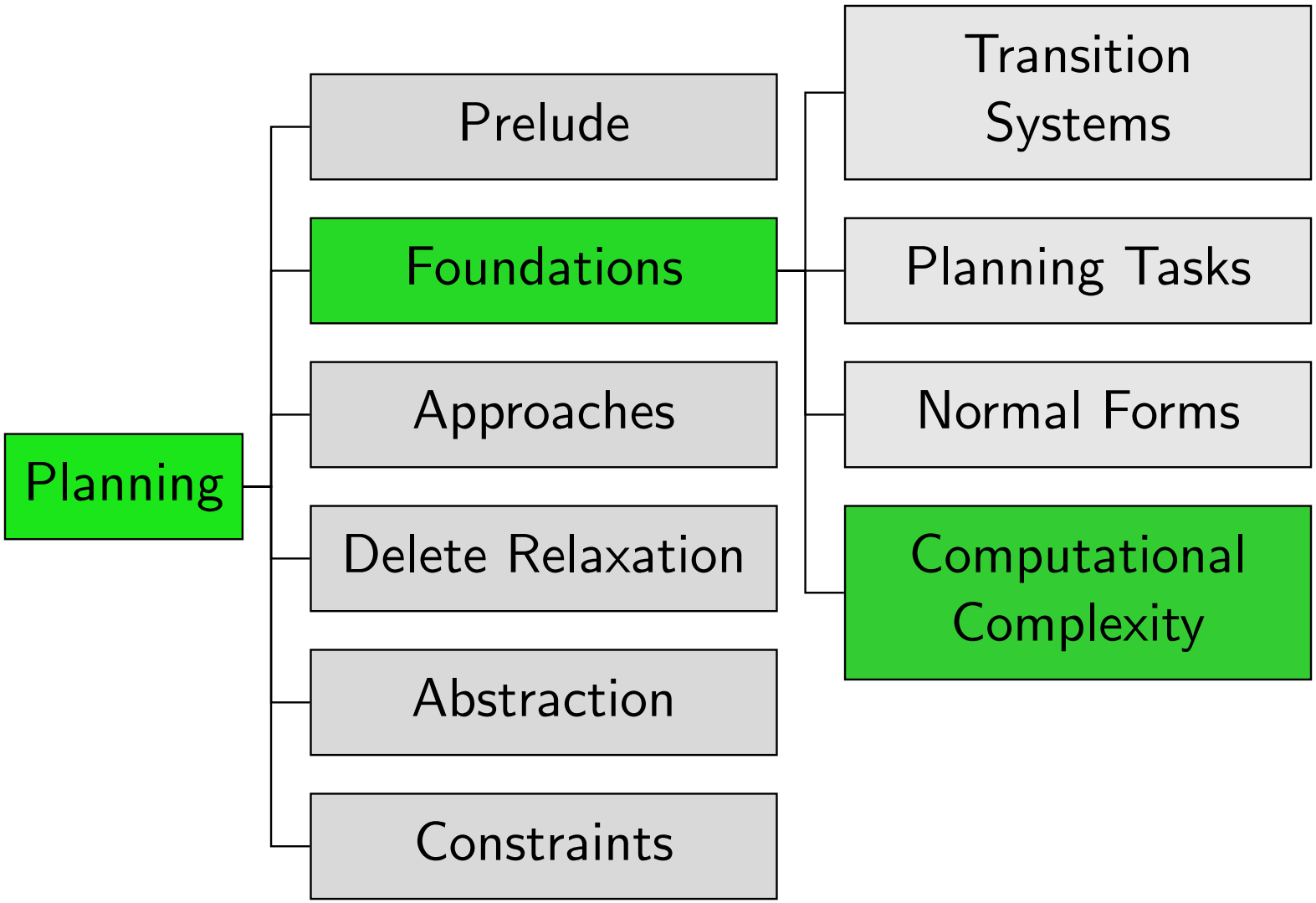
B7. Computational Complexity of Planning: Results

Malte Helmert and Gabriele Röger

Universität Basel

October 1, 2025

Content of the Course



B7.1 (Bounded-Cost) Plan Existence

Decision Problems for Planning

Definition (Plan Existence)

Plan existence (PlanEx) is the following decision problem:

Given: planning task Π

Question: Is there a plan for Π?

$\rightsquigarrow$ decision problem analogue of satisficing planning

Definition (Bounded-Cost Plan Existence)

Bounded-cost plan existence (BCPlanEx)
is the following decision problem:

Given: planning task Π, cost bound $K \in \mathbb{N}_0$

Question: Is there a plan for Π with cost at most K?

$\rightsquigarrow$ decision problem analogue of optimal planning

Plan Existence vs. Bounded-Cost Plan Existence

Theorem (Reduction from PlanEx to BCPlanEx)

$\text{PlanEx} \leq_{\text{p}} \text{BCPlanEx}$

Proof.

Consider a planning task Π with state variables V.

Let $c_{\max}$ be the maximal cost of all operators of Π.

Compute the number of states of Π as $N = 2^{|V|}$.

Π is solvable iff there is solution with cost at most $c_{\max} \cdot (N - 1)$ because a solution need not visit any state twice.

$\rightsquigarrow$ map instance Π of PlanEx to instance $\langle \Pi, c_{\max} \cdot (N - 1) \rangle$ of BCPlanEx

$\rightsquigarrow$ polynomial reduction $\square$

B7.2 PSPACE-Completeness of Planning

Membership in PSPACE

Theorem

$\text{BCPlanEx} \in \text{PSPACE}$

Proof.

Show $\text{BCPlanEx} \in \text{NPSPACE}$ and use Savitch's theorem.

Nondeterministic algorithm:

def plan($\langle V, I, O, \gamma \rangle$, K):
    $s := I$
    $k := K$
    loop forever:
        if $s \models \gamma$: **accept**
        guess $o \in O$
        if o is not applicable in s: **fail**
        if $cost(o) > k$: **fail**
        $s := s[\![o]\!]$
        $k := k - cost(o)$

□

PSPACE-Hardness

Idea: generic reduction

- For an arbitrary fixed DTM M with space bound polynomial p and input w, generate propositional planning task which is solvable iff M accepts w in space $p(|w|)$.
- Without loss of generality, we assume $p(n) \geq n$ for all n.

Reduction: State Variables

Let $M = \langle \Sigma, \square, Q, q_0, q_Y, \delta \rangle$ be the fixed DTM,
and let p be its space-bound polynomial.

Given input $w_1 \dots w_n$, define relevant tape positions
$X := \{-p(n), \dots, p(n)\}$

State Variables

- state_q for all $q \in Q$
- head_i for all $i \in X \cup \{-p(n) - 1, p(n) + 1\}$
- $\text{content}_{i,a}$ for all $i \in X$, $a \in \Sigma_\square$

$\rightsquigarrow$ allows encoding a Turing machine configuration

Reduction: Initial State

Let $M = \langle \Sigma, \square, Q, q_0, q_Y, \delta \rangle$ be the fixed DTM, and let p be its space-bound polynomial.

Given input $w_1 \dots w_n$, define relevant tape positions $X := \{-p(n), \dots, p(n)\}$

Initial State

Initially true:

- state_{q_0}
- head_1
- $\text{content}_{i,w_i}$ for all $i \in \{1, \dots, n\}$
- $\text{content}_{i,\square}$ for all $i \in X \setminus \{1, \dots, n\}$

Initially false:

- all others

Reduction: Operators

Let $M = \langle \Sigma, \square, Q, q_0, q_Y, \delta \rangle$ be the fixed DTM,
and let p be its space-bound polynomial.

Given input $w_1 \dots w_n$, define relevant tape positions
$X := \{-p(n), \dots, p(n)\}$

Operators

One operator for each transition rule $\delta(q, a) = \langle q', a', d \rangle$
and each cell position $i \in X$:

- precondition: $\text{state}_q \wedge \text{head}_i \wedge \text{content}_{i,a}$
- effect: $\neg\text{state}_q \wedge \neg\text{head}_i \wedge \neg\text{content}_{i,a}$
  $\wedge\, \text{state}_{q'} \wedge \text{head}_{i+d} \wedge \text{content}_{i,a'}$

Note that add-after-delete semantics are important here!

Reduction: Goal

Let $M = \langle \Sigma, \square, Q, q_0, q_Y, \delta \rangle$ be the fixed DTM,
and let p be its space-bound polynomial.

Given input $w_1 \dots w_n$, define relevant tape positions
$X := \{-p(n), \dots, p(n)\}$

Goal

state_{q_Y}

PSPACE-Completeness of STRIPS Plan Existence

Theorem (PSPACE-Completeness; Bylander, 1994)

PlanEx *and* BCPlanEx *are* PSPACE*-complete.*
This is true even if only STRIPS tasks are allowed.

Proof.

Membership for BCPlanEx was already shown.

Hardness for PlanEx follows because we just presented a polynomial reduction from an arbitrary problem in PSPACE to PlanEx. (Note that the reduction only generates STRIPS tasks, after trivial cleanup to make them conflict-free.)

Membership for PlanEx and hardness for BCPlanEx follow from the polynomial reduction from PlanEx to BCPlanEx. □

B7.3 More Complexity Results

More Complexity Results

In addition to the basic complexity result presented in this chapter, there are many special cases, generalizations, variations and related problems studied in the literature:

- different planning formalisms
  - e.g., nondeterministic effects, partial observability, schematic operators, numerical state variables
- syntactic restrictions of planning tasks
  - e.g., without preconditions, without conjunctive effects, STRIPS without delete effects
- semantic restrictions of planning task
  - e.g., restricting variable dependencies ("causal graphs")
- particular planning domains
  - e.g., Blocksworld, Logistics, FreeCell

Complexity Results for Different Planning Formalisms

Some results for different planning formalisms:

- nondeterministic effects:
  - fully observable: EXP-complete (Littman, 1997)
  - unobservable: EXPSPACE-complete (Haslum & Jonsson, 1999)
  - partially observable: 2-EXP-complete (Rintanen, 2004)
- schematic operators:
  - usually adds one exponential level to PLANEX complexity
  - e.g., classical case EXPSPACE-complete (Erol et al., 1995)
- numerical state variables:
  - undecidable in most variations (Helmert, 2002)
  - decidable in restricted setting with at most two numeric state variables (Helal and Lakemeyer, 2025)

B7.4 Summary

Summary

- Classical planning is PSPACE-complete.
- This is true both for satisficing and optimal planning (rather, the corresponding decision problems).
- The hardness proof is a polynomial reduction that translates an arbitrary polynomial-space DTM into a STRIPS task:
  - DTM configurations are encoded by state variables.
  - Operators simulate transitions between DTM configurations.
  - The DTM accepts an input iff there is a plan for the corresponding STRIPS task.
- This implies that there is no polynomial algorithm for classical planning unless P = PSPACE.
- It also means that planning is not polynomially reducible to any problem in NP unless NP = PSPACE.

Planning and Optimization

C1. Overview of Classical Planning Algorithms (Part 1)

Malte Helmert and Gabriele Röger

Universität Basel

October 6, 2025

Planning and Optimization

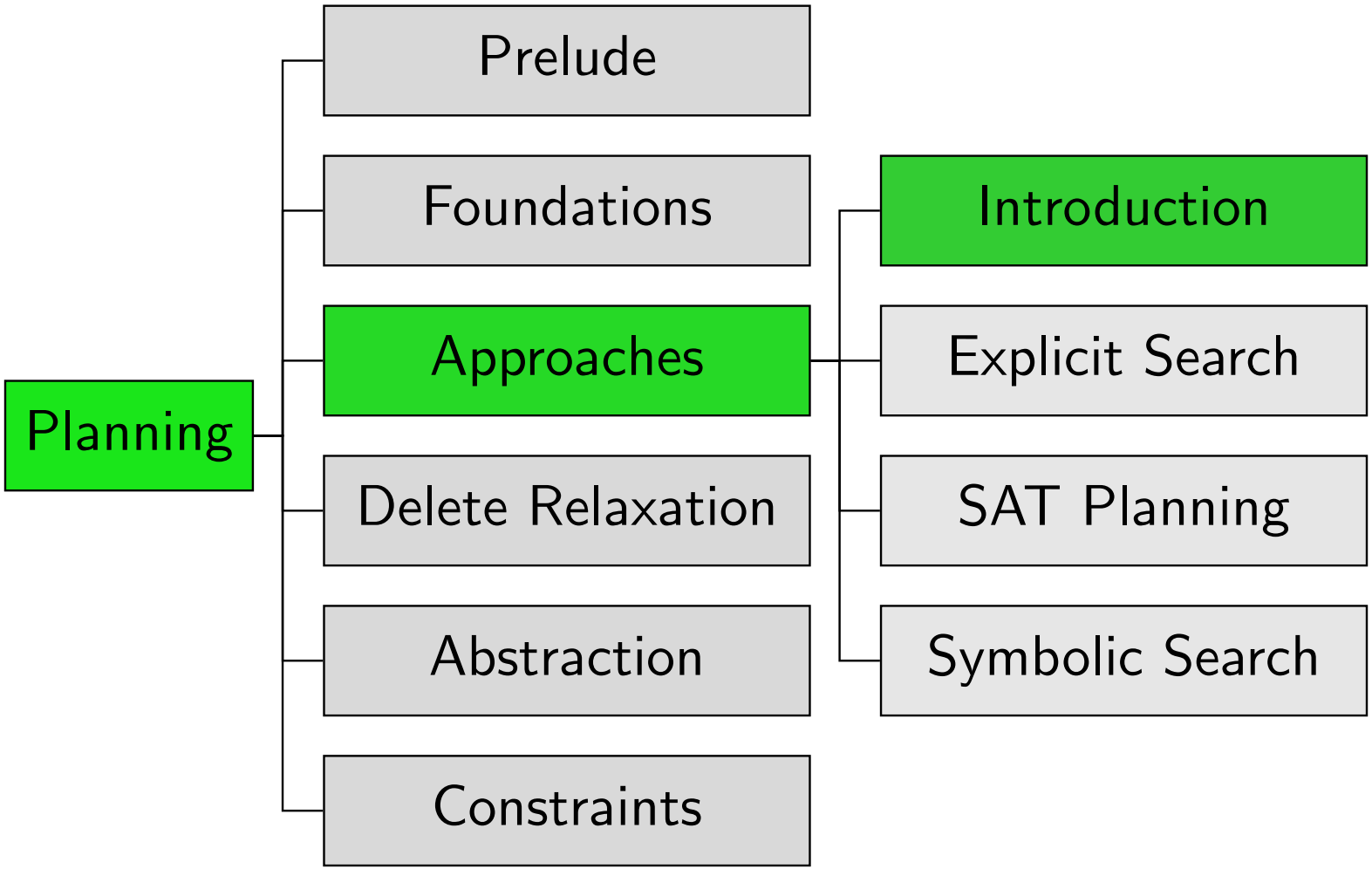
October 6, 2025 — C1. Overview of Classical Planning Algorithms (Part 1)

C1.1 The Big Three

C1.2 Explicit Search

C1.3 Summary

Content of the Course



C1.1 The Big Three

Classical Planning Algorithms

Let's start solving planning tasks!

This Chapter and the Next
very high-level overview of classical planning algorithms

- bird's eye view: no details, just some very brief ideas

The Big Three

Of the many planning approaches, three techniques stand out:

- explicit search $\rightsquigarrow$ Chapters C3–C4, Parts D–F
- SAT planning $\rightsquigarrow$ Chapters C5–C6
- symbolic search $\rightsquigarrow$ Chapters C7–C8

also: many algorithm portfolios

Satisficing or Optimal Planning?

must carefully distinguish:

- satisficing planning: any plan is OK (cheaper ones preferred)
- optimal planning: plans must have minimum cost

solved by similar techniques, but:

- details very different
- almost no overlap between best techniques for satisficing planning and best techniques for optimal planning
- many tasks that are trivial for satisficing planners are impossibly hard for optimal planners

C1.2 Explicit Search

Explicit Search

You know this one already! (Hopefully.)

Reminder: State-Space Search

Need to Catch Up?

- We assume prior knowledge of basic search algorithms:
  - uninformed vs. informed (heuristic)
  - satisficing vs. optimal
  - heuristics and their properties
  - specific algorithms: e.g., breadth-first search, greedy best-first search, A^*
- If you are not familiar with them, we recommend Part B of the Foundations of Artificial Intelligence course:
  `https://dmi.unibas.ch/en/studium/computer-science-informatik/lehrangebot-fs25/13548-lecture-foundations-of-artificial-intelligence/`

Reminder: Interface for Heuristic Search Algorithms

Abstract Interface Needed for Heuristic Search Algorithms

- init() $\rightsquigarrow$ returns initial state
- is_goal(s) $\rightsquigarrow$ tests if s is a goal state
- succ(s) $\rightsquigarrow$ returns all pairs $\langle a, s' \rangle$ with $s \xrightarrow{a} s'$
- cost(a) $\rightsquigarrow$ returns cost of action a
- h(s) $\rightsquigarrow$ returns heuristic value for state s

$\rightsquigarrow$ Foundations of Artificial Intelligence course, Chap. B2 and B9

State Space vs. Search Space

- Planning tasks induce transition systems (a.k.a. state spaces) with an initial state, labeled transitions and goal states.
- State-space search searches state spaces with an initial state, a successor function and goal states.

$\rightsquigarrow$ looks like an obvious correspondence

- However, in planning as search, the state space being searched can be different from the state space of the planning task.
- When we need to make a distinction, we speak of
  - the state space of the planning task whose states are called world states vs.
  - the search space of the search algorithm whose states are called search states.

Design Choice: Search Direction

How to apply explicit search to planning? $\rightsquigarrow$ many design choices!

Design Choice: Search Direction

- progression: forward from initial state to goal
- regression: backward from goal states to initial state
- bidirectional search

$\rightsquigarrow$ Chapters C3–C4

Design Choice: Search Algorithm

How to apply explicit search to planning? $\rightsquigarrow$ many design choices!

Design Choice: Search Algorithm

- uninformed search:
  depth-first, breadth-first, iterative depth-first, ...
- heuristic search (systematic):
  greedy best-first, A*, weighted A*, IDA*, ...
- heuristic search (local):
  hill-climbing, simulated annealing, beam search, ...

Design Choice: Search Control

How to apply explicit search to planning? $\rightsquigarrow$ many design choices!

Design Choice: Search Control

- heuristics for informed search algorithms
- pruning techniques: invariants, symmetry elimination, partial-order reduction, helpful actions pruning, ...

How do we find good heuristics in a domain-independent way?

$\rightsquigarrow$ one of the main focus areas of classical planning research

$\rightsquigarrow$ Parts D–F

C1.3 Summary

Summary

(Joint summary follows after next chapter.)

Planning and Optimization

C2. Overview of Classical Planning Algorithms (Part 2)

Malte Helmert and Gabriele Röger

Universität Basel

October 6, 2025

Planning and Optimization

October 6, 2025 — C2. Overview of Classical Planning Algorithms (Part 2)

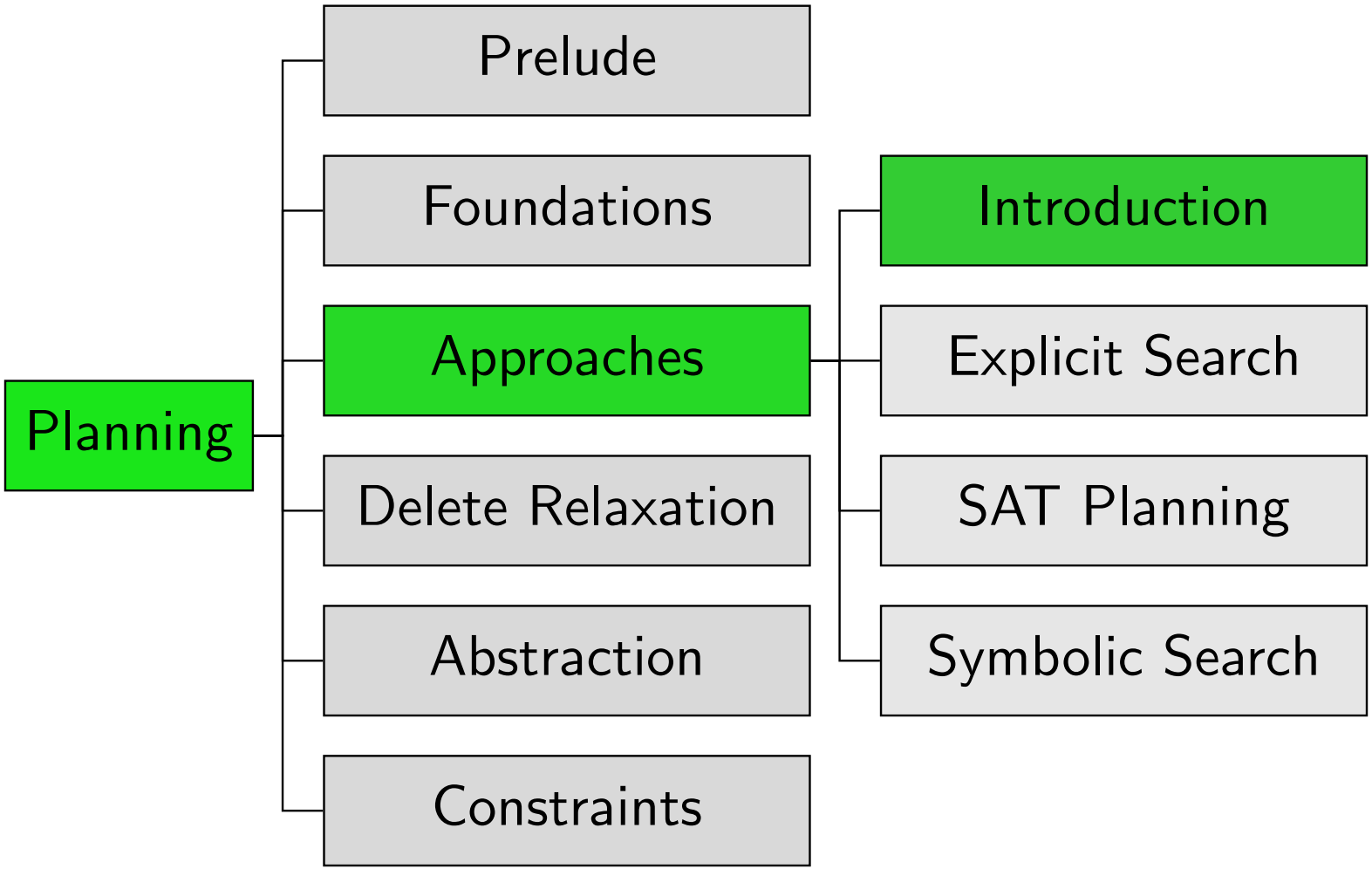
C2.1 SAT Planning

C2.2 Symbolic Search

C2.3 Planning System Examples

C2.4 Summary

Content of the Course



The Big Three (Repeated from Last Chapter)

Of the many planning approaches, three techniques stand out:

- explicit search $\rightsquigarrow$ Chapters C3–C4, Parts D–F
- SAT planning $\rightsquigarrow$ Chapters C5–C6
- symbolic search $\rightsquigarrow$ Chapters C7–C8

also: many algorithm portfolios

C2.1 SAT Planning

SAT Planning: Basic Idea

- formalize problem of finding plan with a given horizon (length bound) as a propositional satisfiability problem and feed it to a generic SAT solver
- to obtain a (semi-) complete algorithm, try with increasing horizons until a plan is found (= the formula is satisfiable)
- important optimization: allow applying several non-conflicting operators "at the same time" so that a shorter horizon suffices

SAT Encodings: Variables

- given propositional planning task $\langle V, I, O, \gamma \rangle$
- given horizon $T \in \mathbb{N}_0$

Variables of SAT Encoding

- propositional variables v^i for all $v \in V$, $0 \leq i \leq T$
  encode state after i steps of the plan
- propositional variables o^i for all $o \in O$, $1 \leq i \leq T$
  encode operator(s) applied in i-th step of the plan

Design Choice: SAT Encoding

Again, there are several important design choices.

Design Choice: SAT Encoding

- sequential or parallel
- many ways of modeling planning semantics in logic

$\rightsquigarrow$ main focus of research on SAT planning

Design Choice: SAT Solver

Again, there are several important design choices.

Design Choice: SAT Solver

- out-of-the-box like Glucose, CaDiCaI, MiniSAT
- planning-specific modifications

Design Choice: Evaluation Strategy

Again, there are several important design choices.

Design Choice: Evaluation Strategy

- always advance horizon by $+1$ or more aggressively
- possibly probe multiple horizons concurrently

C2.2 Symbolic Search

Symbolic Search Planning: Basic Ideas

- search processes sets of states at a time
- operators, goal states, state sets reachable with a given cost etc. represented by binary decision diagrams (BDDs) (or similar data structures)
- hope: exponentially large state sets can be represented as polynomially sized BDDs, which can be efficiently processed
- perform symbolic breadth-first search (or something more sophisticated) on these set representations

Symbolic Breadth-First Progression Search

prototypical algorithm:

Symbolic Breadth-First Progression Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

$\rightsquigarrow$ If we can implement operations *models*, $\{I\}$, $\cap$, $\neq \emptyset$, $\cup$, *apply* and $=$ efficiently, this is a reasonable algorithm.

Design Choice: Symbolic Data Structure

Again, there are several important design choices.

Design Choice: Symbolic Data Structure

- BDDs
- ADDs
- EVMDDs
- SDDs

Other Design Choices

- additionally, same design choices as for explicit search:
  - search direction
  - search algorithm
  - search control (incl. heuristics)
- in practice, hard to make heuristics and other advanced search control efficient for symbolic search
  $\rightsquigarrow$ rarely used

C2.3 Planning System Examples

Planning Systems: FF

FF (Hoffmann & Nebel, 2001)

- problem class: satisficing
- algorithm class: explicit search
- search direction: forward search
- search algorithm: enforced hill-climbing
- heuristic: FF heuristic (inadmissible)
- other aspects: helpful action pruning; goal agenda manager

$\rightsquigarrow$ breakthrough for heuristic search planning;
winner of IPC 2000

Planning Systems: LAMA

LAMA (Richter & Westphal, 2008)

- problem class: satisficing
- algorithm class: explicit search
- search direction: forward search
- search algorithm: restarting Weighted A* (anytime)
- heuristic: FF heuristic and landmark heuristic (inadmissible)
- other aspects: preferred operators; deferred heuristic evaluation; multi-queue search

$\rightsquigarrow$ still one of the leading satisficing planners;
winner of IPC 2008 and IPC 2011 (satisficing tracks)

Planning Systems: Madagascar-pC

Madagascar (Rintanen, 2014)

- problem class: satisficing
- algorithm class: SAT planning
- encoding: parallel $\exists$-step encoding
- SAT solver: using planning-specific action variable selection
- evaluation strategy: exponential horizons, parallelized probing
- other aspects: invariants

$\rightsquigarrow$ second place at IPC 2014 (agile track)

Planning Systems: SymBA*

SymBA* (Torralba, 2015)

- problem class: optimal
- algorithm class: symbolic search
- symbolic data structure: BDDs
- search direction: bidirectional
- search algorithm: mixture of (symbolic) Dijkstra and A*
- heuristic: perimeter abstractions/blind

$\rightsquigarrow$ winner of IPC 2014 (optimal track)

Planning Systems: Scorpion

Scorpion 2023 (Seipp, 2023)

- problem class: optimal
- algorithm class: explicit search
- search direction: forward search
- search algorithm: A^*
- heuristic: abstraction heuristics and cost partitioning

$\rightsquigarrow$ runner-up of IPC 2023 (optimal track)

Planning Systems: Fast Downward Stone Soup

Fast Downward Stone Soup 2023, optimal version (Büchner et al., 2023)

- problem class: optimal
- algorithm class: (portfolio of) explicit search
- search direction: forward search
- search algorithm: A*
- heuristic: all admissible heuristics considered in the course

$\rightsquigarrow$ winner of IPC 2011 (optimal track);
various awards in IPC 2011–2023

Planning Systems: SymK

SymK (Speck et al., 2025)

- problem class: optimal
- algorithm class: symbolic search
- symbolic data structure: BDDs
- search direction: bidirectional
- search algorithm: symbolic Dijkstra algorithm
- heuristic: blind

C2.4 Summary

Summary

big three classes of algorithms for classical planning:

- explicit search
  - design choices: search direction, search algorithm, search control (incl. heuristics)
- SAT planning
  - design choices: SAT encoding, SAT solver, evaluation strategy
- symbolic search
  - design choices: symbolic data structure
    + same ones as for explicit search

Planning and Optimization

C3. Progression and Regression Search

Malte Helmert and Gabriele Röger

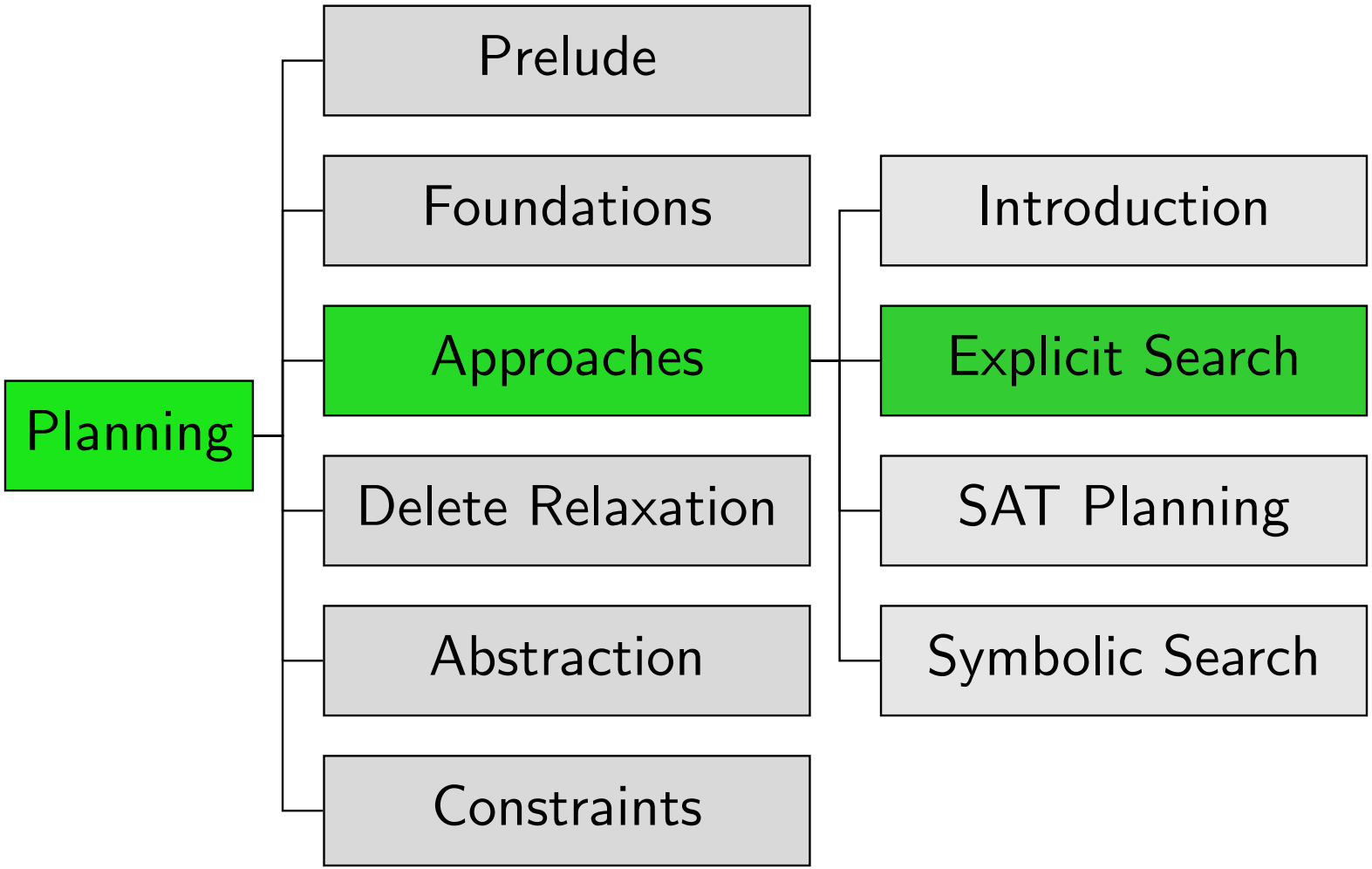
Universität Basel

October 8, 2025

Planning and Optimization

October 8, 2025 — C3. Progression and Regression Search

Content of the Course



C3.1 Introduction

Search Direction

Search direction

- one dimension for classifying search algorithms
- forward search from initial state to goal based on progression
- backward search from goal to initial state based on regression
- bidirectional search

In this chapter we look into progression and regression planning.

Reminder: Interface for Heuristic Search Algorithms

Abstract Interface Needed for Heuristic Search Algorithms

- init() $\rightsquigarrow$ returns initial state
- is_goal(s) $\rightsquigarrow$ tests if s is a goal state
- succ(s) $\rightsquigarrow$ returns all pairs $\langle a, s' \rangle$ with $s \xrightarrow{a} s'$
- cost(a) $\rightsquigarrow$ returns cost of action a
- h(s) $\rightsquigarrow$ returns heuristic value for state s

C3.2 Progression

Planning by Forward Search: Progression

Progression: Computing the successor state $s[\![o]\!]$ of a state s with respect to an operator o.

Progression planners find solutions by forward search:

- start from initial state
- iteratively pick a previously generated state and progress it through an operator, generating a new state
- solution found when a goal state generated

pro: very easy and efficient to implement

Search Space for Progression

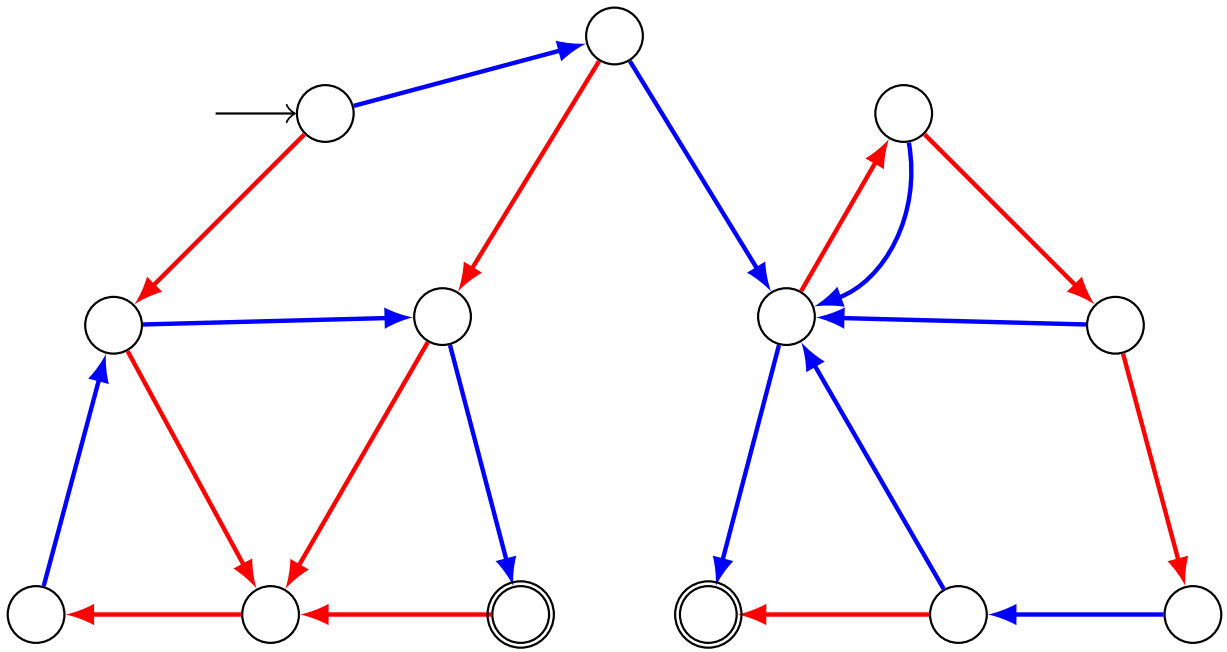
Search Space for Progression

search space for progression in a planning task $\Pi = \langle V, I, O, \gamma \rangle$
(search states are world states s of Π;
actions of search space are operators $o \in O$)

- init() $\rightsquigarrow$ returns I
- is_goal(s) $\rightsquigarrow$ tests if $s \models \gamma$
- succ(s) $\rightsquigarrow$ returns all pairs $\langle o, s[\![o]\!] \rangle$
  where $o \in O$ and o is applicable in s
- cost(o) $\rightsquigarrow$ returns $cost(o)$ as defined in Π
- h(s) $\rightsquigarrow$ estimates cost from s to γ ($\rightsquigarrow$ Parts D–F)

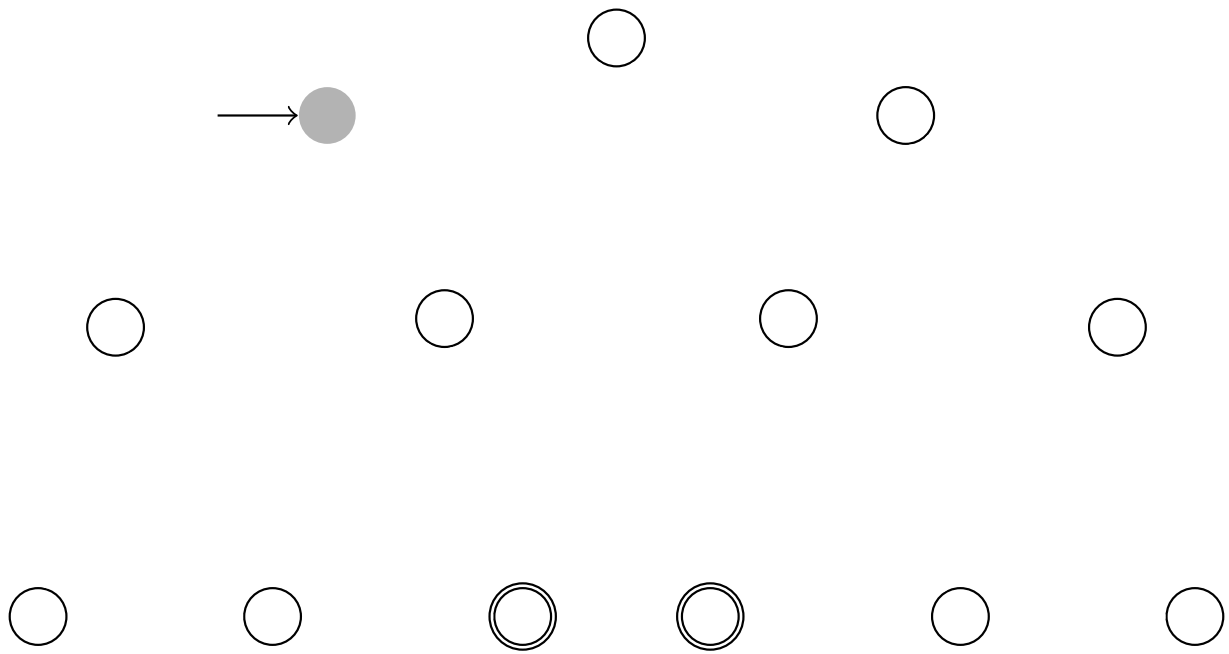
Progression Planning Example

Example of a progression search



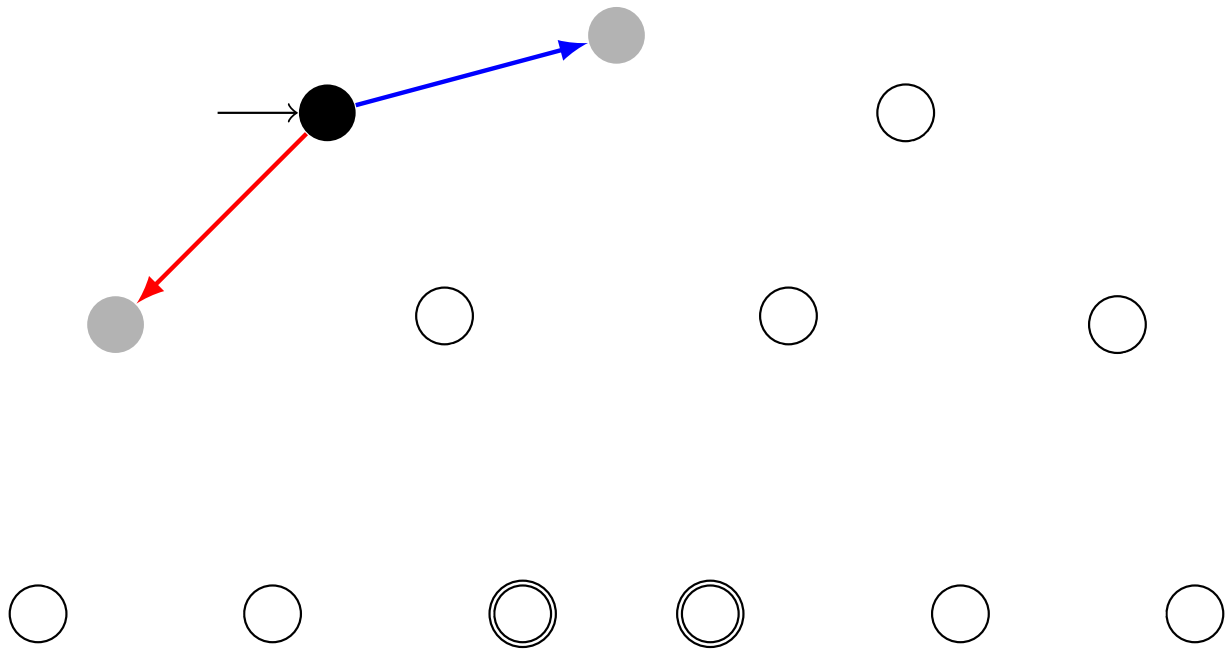
Progression Planning Example

Example of a progression search



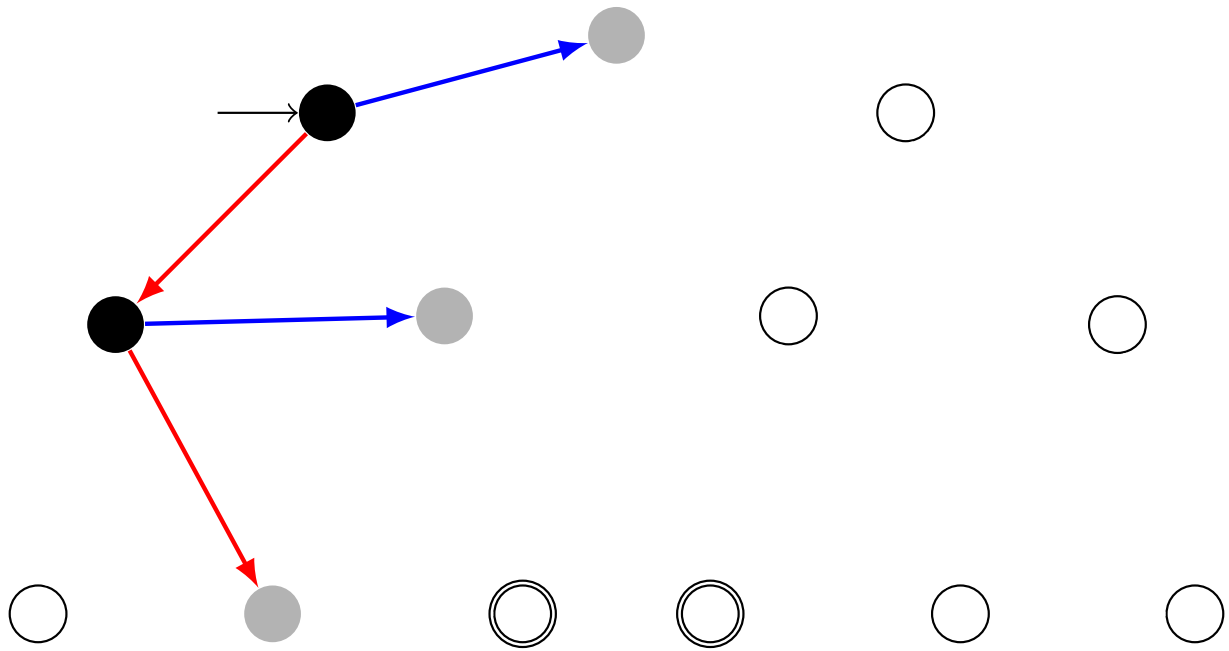
Progression Planning Example

Example of a progression search



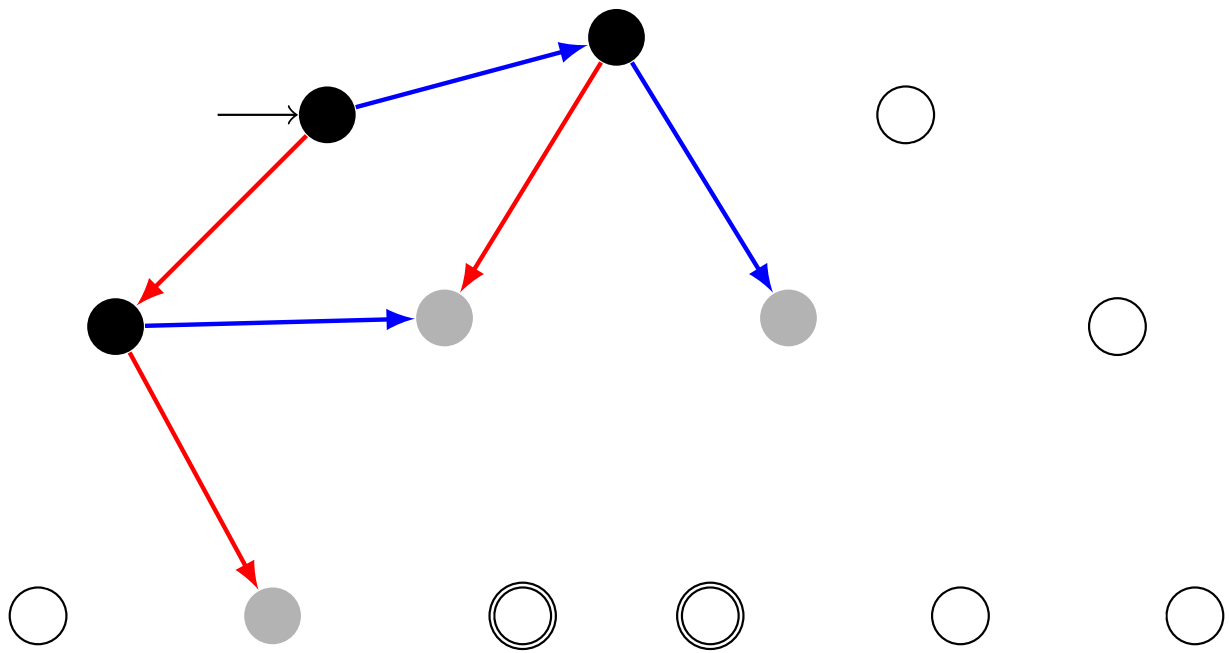
Progression Planning Example

Example of a progression search



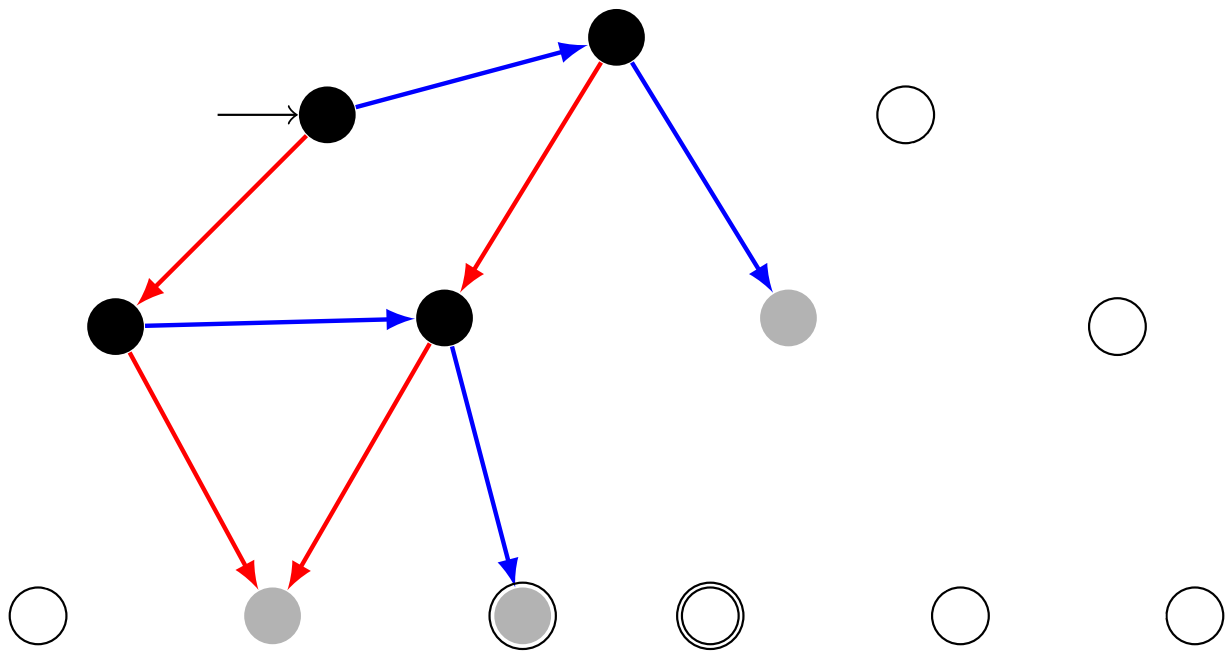
Progression Planning Example

Example of a progression search



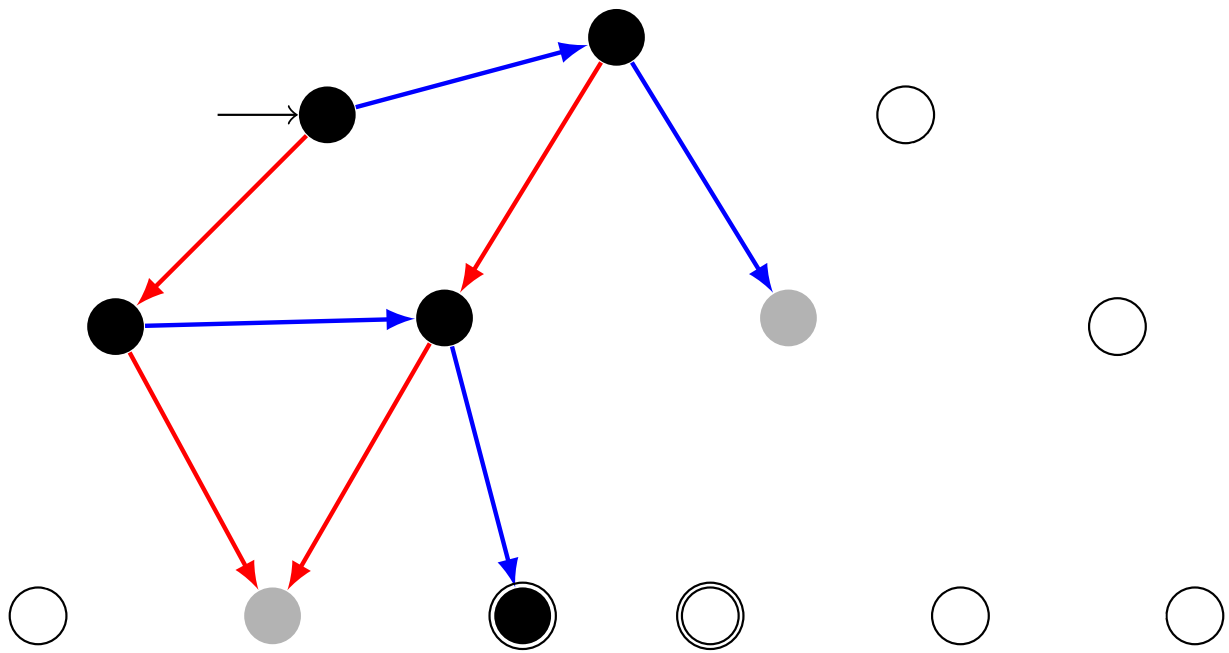
Progression Planning Example

Example of a progression search



Progression Planning Example

Example of a progression search



C3.3 Regression

Forward Search vs. Backward Search

Searching planning tasks in forward vs. backward direction is not symmetric:

- forward search starts from a single initial state;
  backward search starts from a set of goal states
- when applying an operator o in a state s in forward direction,
  there is a unique successor state s';
  if we just applied operator o and ended up in state s',
  there can be several possible predecessor states s

$\rightsquigarrow$ in most natural representation for backward search in planning,
each search state corresponds to a set of world states

Planning by Backward Search: Regression

Regression: Computing the possible predecessor states $regr(S', o)$ of a set of states S' ("subgoal") given the last operator o that was applied.

$\rightsquigarrow$ formal definition in next chapter

Regression planners find solutions by backward search:

- start from set of goal states
- iteratively pick a previously generated subgoal (state set) and regress it through an operator, generating a new subgoal
- solution found when a generated subgoal includes initial state

pro: can handle many states simultaneously
con: basic operations complicated and expensive

Search Space Representation in Regression Planners

identify state sets with logical formulas (again):

- each search state corresponds to a set of world states ("subgoal")
- each search state is represented by a logical formula: φ represents $\{s \in S \mid s \models \varphi\}$
- many basic search operations like detecting duplicates are NP-complete or coNP-complete

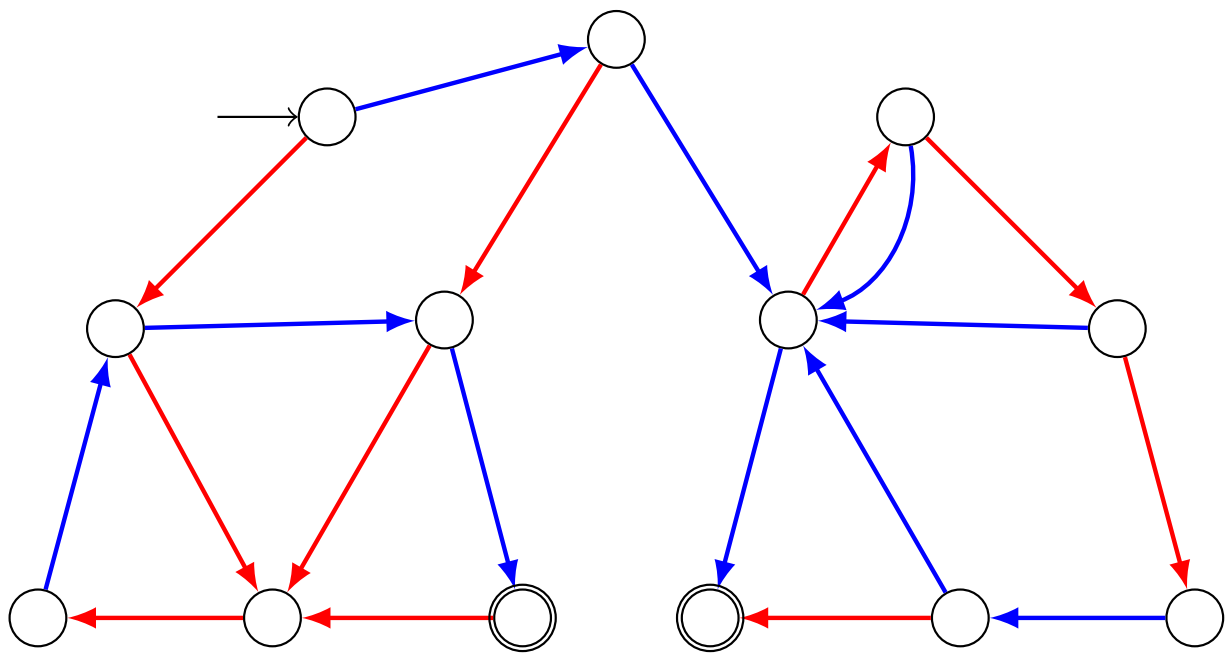
Search Space for Regression

Search Space for Regression

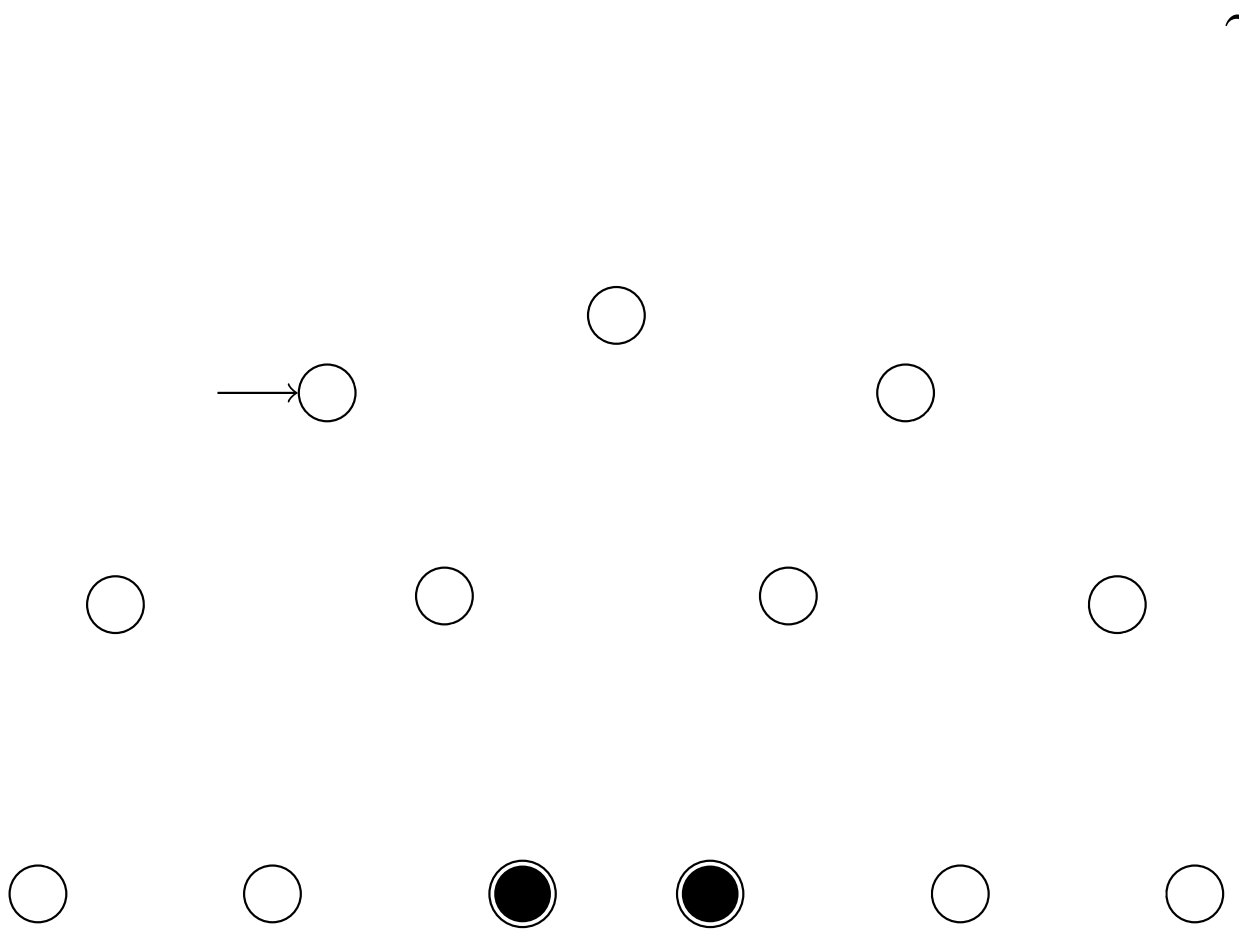
search space for regression in a planning task $\Pi = \langle V, I, O, \gamma \rangle$
(search states are formulas φ describing sets of world states;
actions of search space are operators $o \in O$)

- init() $\rightsquigarrow$ returns γ
- is_goal(φ) $\rightsquigarrow$ tests if $I \models \varphi$
- succ(φ) $\rightsquigarrow$ returns all pairs $\langle o, \mathit{regr}(\varphi, o) \rangle$ where $o \in O$ and $\mathit{regr}(\varphi, o)$ is defined
- cost(o) $\rightsquigarrow$ returns $\mathit{cost}(o)$ as defined in Π
- h(φ) $\rightsquigarrow$ estimates cost from I to φ ($\rightsquigarrow$ Parts D–F)

Regression Planning Example (Depth-first Search)

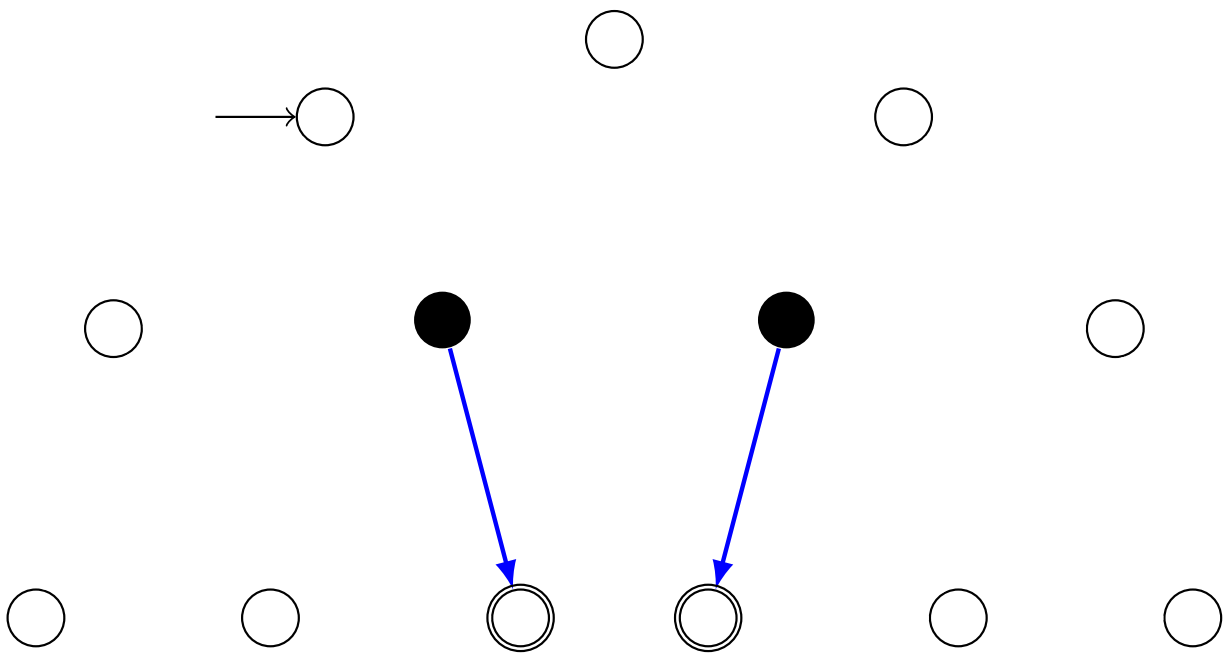


Regression Planning Example (Depth-first Search)



Regression Planning Example (Depth-first Search)

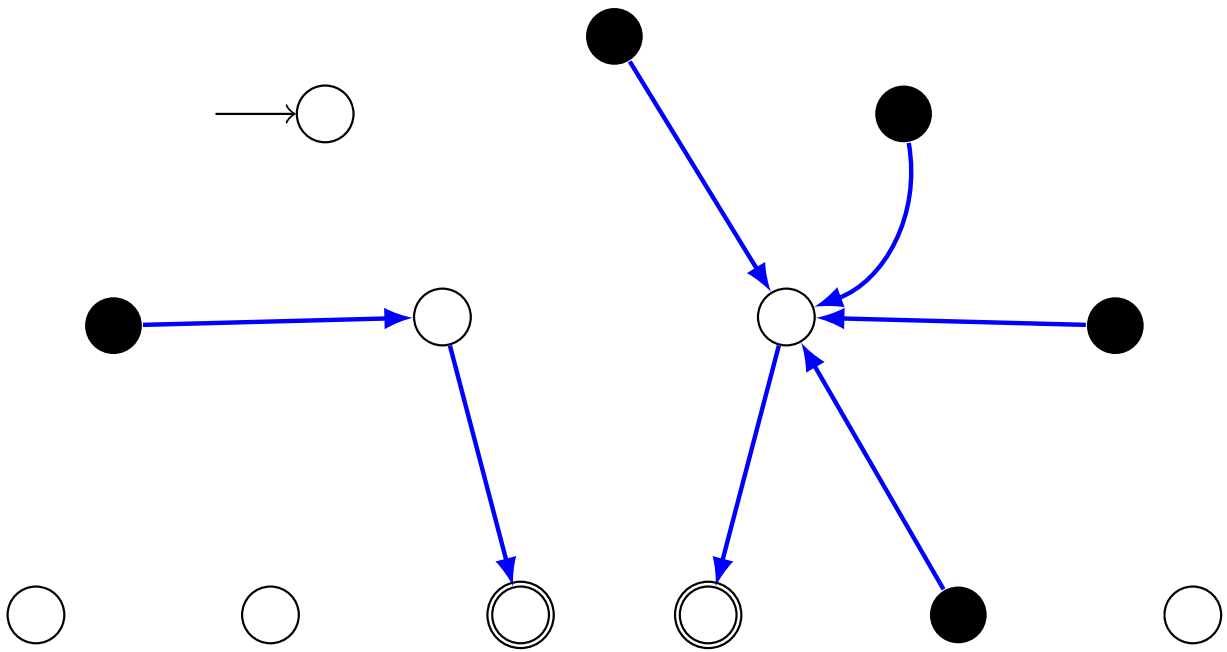
$\varphi_1 = \mathit{regr}(\gamma, \longrightarrow)$ $\varphi_1 \longrightarrow \gamma$



Regression Planning Example (Depth-first Search)

$\varphi_1 = \mathit{regr}(\gamma, \longrightarrow)$
$\varphi_2 = \mathit{regr}(\varphi_1, \longrightarrow)$

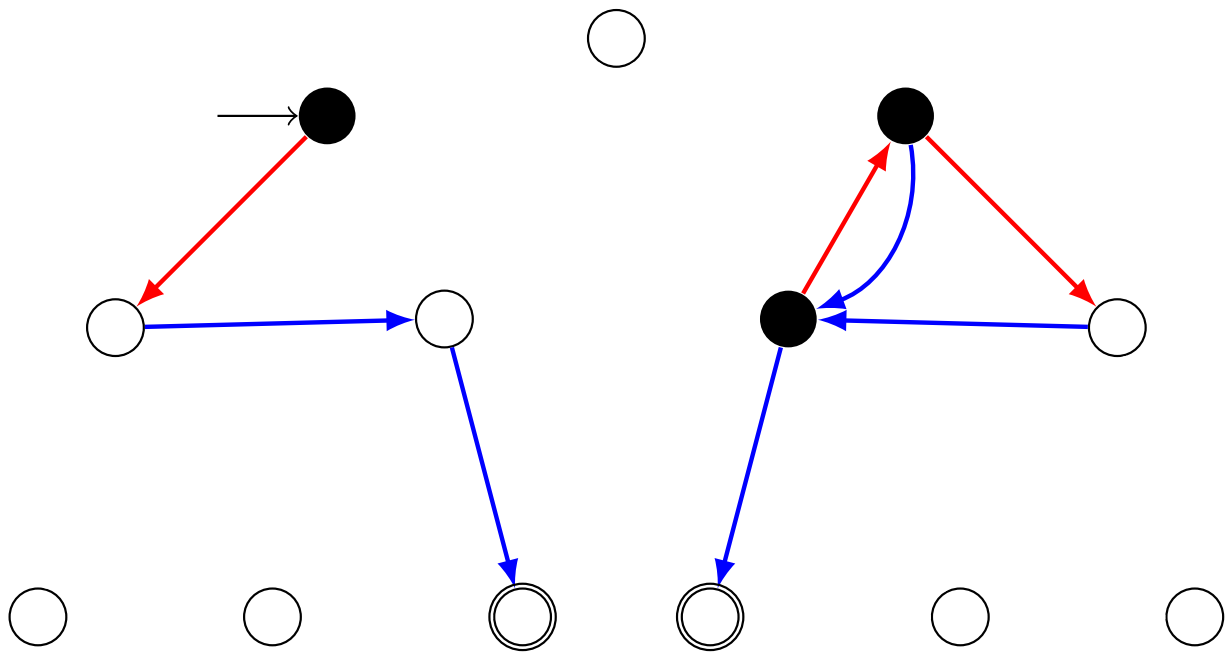
$\varphi_2 \longrightarrow \varphi_1 \longrightarrow \gamma$



Regression Planning Example (Depth-first Search)

$\varphi_1 = \mathit{regr}(\gamma, \longrightarrow)$
$\varphi_2 = \mathit{regr}(\varphi_1, \longrightarrow)$
$\varphi_3 = \mathit{regr}(\varphi_2, \longrightarrow), I \models \varphi_3$

$\varphi_3 \longrightarrow \varphi_2 \longrightarrow \varphi_1 \longrightarrow \gamma$



C3.4 Regression for STRIPS Tasks

Regression for STRIPS Planning Tasks

Regression for STRIPS planning tasks is much simpler than the general case:

- Consider subgoal φ that is conjunction of atoms $a_1 \wedge \cdots \wedge a_n$ (e.g., the original goal γ of the planning task).
- First step: Choose an operator o that deletes no a_i.
- Second step: Remove any atoms added by o from φ.
- Third step: Conjoin $pre(o)$ to φ.

$\rightsquigarrow$ Outcome of this is regression of φ w.r.t. o.
It is again a conjunction of atoms.

optimization: only consider operators adding at least one a_i

STRIPS Regression

Definition (STRIPS Regression)

Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be a conjunction of atoms, and let o be a STRIPS operator which adds the atoms $a_1, \ldots, a_k$ and deletes the atoms $d_1, \ldots, d_l$.

The STRIPS regression of φ with respect to o is

$$sregr(\varphi, o) := \begin{cases} \bot & \text{if } \varphi_i = d_j \text{ for some } i, j \\ pre(o) \wedge \bigwedge(\{\varphi_1, \ldots, \varphi_n\} \setminus \{a_1, \ldots, a_k\}) & \text{else} \end{cases}$$

Note: $sregr(\varphi, o)$ is again a conjunction of atoms, or $\bot$.

Does this Capture the Idea of Regression?

For our definition to capture the concept of regression,
it must have the following property:

Regression Property

For all sets of states described by a conjunction of atoms φ,
all states s and all STRIPS operators o,

$$s \models \mathit{sregr}(\varphi, o) \quad \text{iff} \quad s[\![o]\!] \models \varphi.$$

This is indeed true. We do not prove it now because we prove
this property for general regression (not just STRIPS) later.

C3.5 Summary

Summary

- Progression search proceeds forward from the initial state.
- In progression search, the search space is identical to the state space of the planning task.
- Regression search proceeds backwards from the goal.
- Each search state corresponds to a set of world states, for example represented by a formula.
- Regression is simple for STRIPS operators.
- The theory for general regression is more complex. This is the topic of the following chapter.

Planning and Optimization

C4. General Regression

Malte Helmert and Gabriele Röger

Universität Basel

October 8, 2025

Planning and Optimization
October 8, 2025 — C4. General Regression

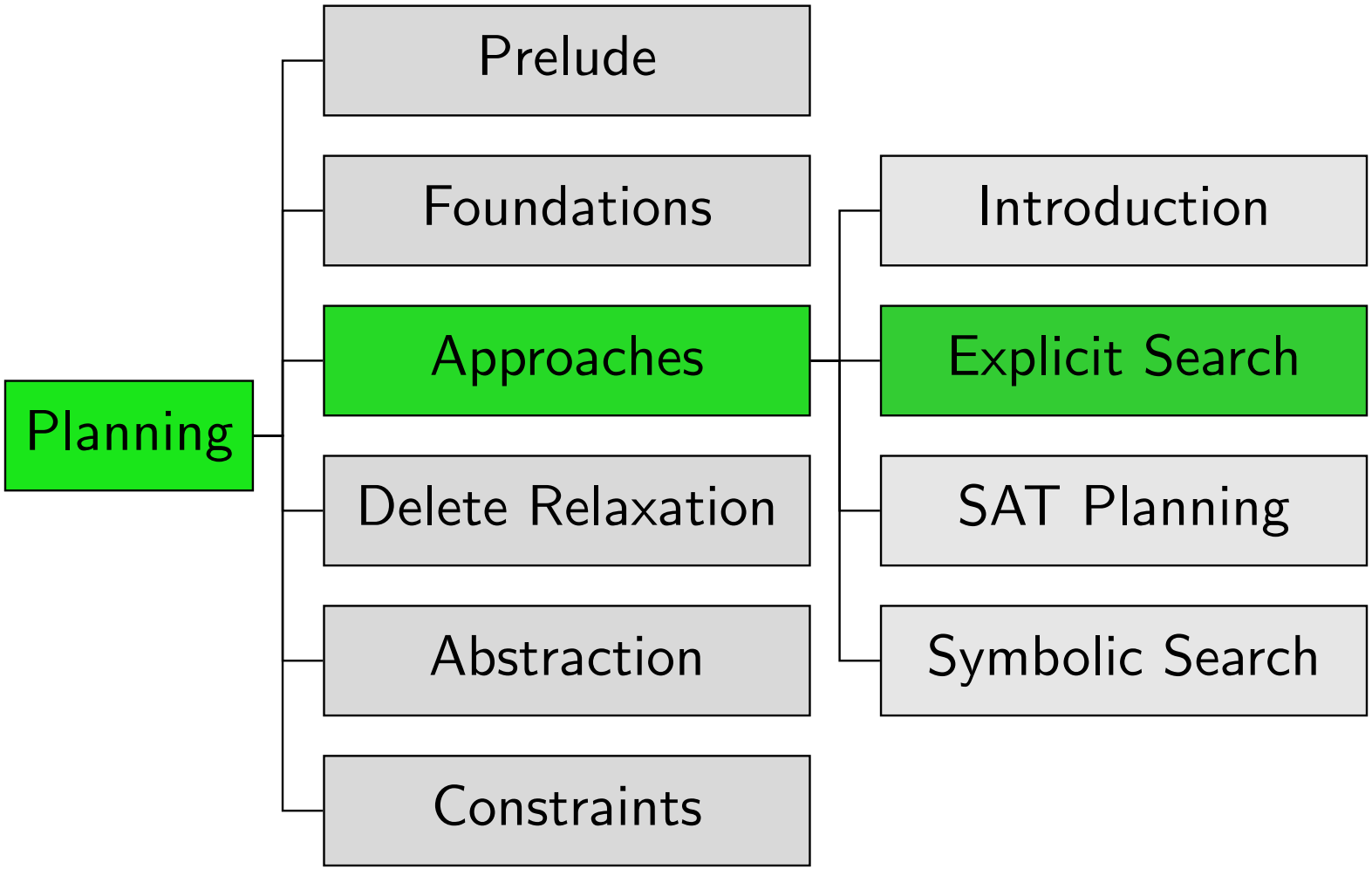
C4.1 Regressing State Variables

C4.2 Regressing Formulas Through Effects

C4.3 Regressing Formulas Through Operators

C4.4 Summary

Content of the Course



Regression for General Planning Tasks

- With disjunctions and conditional effects, things become more tricky. How to regress $a \vee (b \wedge c)$ with respect to $\langle q, d \rhd b \rangle$?
- In this chapter, we show how to regress general sets of states through general operators.
- We extensively use the idea of representing sets of states as formulas.

C4.1 Regressing State Variables

Regressing State Variables: Motivation

Key question for general regression:

- Assume we are applying an operator with effect e.
- What must be true in the predecessor state for propositional state variable v to be true in the successor state?

If we can answer this question, a general definition of regression is only a small additional step.

Regressing State Variables: Key Idea

Assume we are in state s and apply effect e
to obtain successor state s'.

Propositional state variable v is true in s' iff

- effect e makes it true, or
- it remains true, i.e., it is true in s and not made false by e.

Regressing a State Variable Through an Effect

Definition (Regressing a State Variable Through an Effect)

Let e be an effect of a propositional planning task,
and let v be a propositional state variable.

The regression of v through e, written $regr(v, e)$,
is defined as the following logical formula:

$$regr(v, e) = effcond(v, e) \vee (v \wedge \neg effcond(\neg v, e)).$$

Does this capture add-after-delete semantics correctly?

Regressing State Variables: Example

Example

Let $e = (b \rhd a) \wedge (c \rhd \neg a) \wedge b \wedge \neg d$.

v	$\mathit{effcond}(v, e)$	$\mathit{effcond}(\neg v, e)$	$\mathit{regr}(v, e)$
a	b	c	$b \vee (a \wedge \neg c)$
b	$\top$	$\bot$	$\top \vee (b \wedge \neg\bot) \equiv \top$
c	$\bot$	$\bot$	$\bot \vee (c \wedge \neg\bot) \equiv c$
d	$\bot$	$\top$	$\bot \vee (d \wedge \neg\top) \equiv \bot$

Reminder: $\mathit{regr}(v, e) = \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$

Regressing State Variables: Correctness (1)

Lemma (Correctness of $regr(v, e)$)

Let s be a state, e be an effect and v be a state variable of a propositional planning task.

Then $s \models regr(v, e)$ iff $s[\![e]\!] \models v$.

Regressing State Variables: Correctness (2)

Proof.

($\Rightarrow$): We know $s \models \mathit{regr}(v, e)$, and hence
$s \models \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$.

Do a case analysis on the two disjuncts.

Case 1: $s \models \mathit{effcond}(v, e)$.
Then $s[\![e]\!] \models v$ by the first case in the definition of $s[\![e]\!]$ (Ch. B3).

Case 2: $s \models (v \wedge \neg \mathit{effcond}(\neg v, e))$.
Then $s \models v$ and $s \not\models \mathit{effcond}(\neg v, e)$.
We may additionally assume $s \not\models \mathit{effcond}(v, e)$
because otherwise we can apply Case 1 of this proof.
Then $s[\![e]\!] \models v$ by the third case in the definition of $s[\![e]\!]$. ...

Regressing State Variables: Correctness (3)

Proof (continued).

($\Leftarrow$): Proof by contraposition.

We show that if $\mathit{regr}(v, e)$ is false in s, then v is false in $s[\![e]\!]$.

- By prerequisite, $s \not\models \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$.
- Hence $s \models \neg \mathit{effcond}(v, e) \wedge (\neg v \vee \mathit{effcond}(\neg v, e))$.
- From the first conjunct, we get $s \models \neg \mathit{effcond}(v, e)$ and hence $s \not\models \mathit{effcond}(v, e)$.
- From the second conjunct, we get $s \models \neg v \vee \mathit{effcond}(\neg v, e)$.
- Case 1: $s \models \neg v$. Then v is false before applying e and remains false, so $s[\![e]\!] \not\models v$.
- Case 2: $s \models \mathit{effcond}(\neg v, e)$. Then v is deleted by e and not simultaneously added, so $s[\![e]\!] \not\models v$.

C4.2 Regressing Formulas Through Effects

Regressing Formulas Through Effects: Idea

- We can now generalize regression from state variables to general formulas over state variables.
- The basic idea is to replace every occurrence of every state variable v by $regr(v, e)$ as defined in the previous section.
- The following definition makes this more formal.

Regressing Formulas Through Effects: Definition

Definition (Regressing a Formula Through an Effect)

In a propositional planning task, let e be an effect,
and let φ be a formula over propositional state variables.

The regression of φ through e, written $\mathit{regr}(\varphi, e)$,
is defined as the following logical formula:

$$\begin{aligned}
\mathit{regr}(\top, e) &= \top \\
\mathit{regr}(\bot, e) &= \bot \\
\mathit{regr}(v, e) &= \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e)) \\
\mathit{regr}(\neg\psi, e) &= \neg \mathit{regr}(\psi, e) \\
\mathit{regr}(\psi \vee \chi, e) &= \mathit{regr}(\psi, e) \vee \mathit{regr}(\chi, e) \\
\mathit{regr}(\psi \wedge \chi, e) &= \mathit{regr}(\psi, e) \wedge \mathit{regr}(\chi, e).
\end{aligned}$$

Regressing Formulas Through Effects: Example

Example

Let $e = (b \rhd a) \wedge (c \rhd \neg a) \wedge b \wedge \neg d$.

Recall:

- $regr(a, e) \equiv b \vee (a \wedge \neg c)$
- $regr(b, e) \equiv \top$
- $regr(c, e) \equiv c$
- $regr(d, e) \equiv \bot$

We get:

$$\begin{aligned} regr((a \vee d) \wedge (c \vee d), e) &\equiv ((b \vee (a \wedge \neg c)) \vee \bot) \wedge (c \vee \bot) \\ &\equiv (b \vee (a \wedge \neg c)) \wedge c \\ &\equiv b \wedge c \end{aligned}$$

Regressing Formulas Through Effects: Correctness (1)

Lemma (Correctness of $regr(\varphi, e)$)

Let φ be a logical formula, e an effect and s a state of a propositional planning task.

Then $s \models regr(\varphi, e)$ iff $s[\![e]\!] \models \varphi$.

Regressing Formulas Through Effects: Correctness (2)

Proof.

The proof is by structural induction on φ.

Induction hypothesis: $s \models \mathit{regr}(\psi, e)$ iff $s[\![e]\!] \models \psi$
for all proper subformulas ψ of φ.

Base case $\varphi = \top$:

We have $\mathit{regr}(\top, e) = \top$, and $s \models \top$ iff $s[\![e]\!] \models \top$ is correct.

Base case $\varphi = \bot$:

We have $\mathit{regr}(\bot, e) = \bot$, and $s \models \bot$ iff $s[\![e]\!] \models \bot$ is correct.

Base case $\varphi = v$:

We have $s \models \mathit{regr}(v, e)$ iff $s[\![e]\!] \models v$ from the previous lemma. ...

Regressing Formulas Through Effects: Correctness (3)

Proof (continued).

Inductive case $\varphi = \neg\psi$:

$$\begin{aligned} s \models \mathit{regr}(\neg\psi, e) &\text{ iff } s \models \neg\mathit{regr}(\psi, e) \\ &\text{iff } s \not\models \mathit{regr}(\psi, e) \\ &\text{iff } s[\![e]\!] \not\models \psi \\ &\text{iff } s[\![e]\!] \models \neg\psi \end{aligned}$$

Inductive case $\varphi = \psi \vee \chi$:

$$\begin{aligned} s \models \mathit{regr}(\psi \vee \chi, e) &\text{ iff } s \models \mathit{regr}(\psi, e) \vee \mathit{regr}(\chi, e) \\ &\text{iff } s \models \mathit{regr}(\psi, e) \text{ or } s \models \mathit{regr}(\chi, e) \\ &\text{iff } s[\![e]\!] \models \psi \text{ or } s[\![e]\!] \models \chi \\ &\text{iff } s[\![e]\!] \models \psi \vee \chi \end{aligned}$$

Inductive case $\varphi = \psi \wedge \chi$:

Like previous case, replacing "$\vee$" by "$\wedge$"
and replacing "or" by "and".

C4.3 Regressing Formulas Through Operators

Regressing Formulas Through Operators: Idea

- We can now regress arbitrary formulas through arbitrary effects.
- The last missing piece is a definition of regression through operators, describing exactly in which states s applying a given operator o leads to a state satisfying a given formula φ.
- There are two requirements:
  - The operator o must be applicable in the state s.
  - The resulting state $s[\![o]\!]$ must satisfy φ.

Regressing Formulas Through Operators: Definition

Definition (Regressing a Formula Through an Operator)

In a propositional planning task, let o be an operator, and let φ be a formula over state variables.

The regression of φ through o, written $\mathit{regr}(\varphi, o)$, is defined as the following logical formula:

$$\mathit{regr}(\varphi, o) = \mathit{pre}(o) \wedge \mathit{regr}(\varphi, \mathit{eff}(o)).$$

Regressing Formulas Through Operators: Correctness (1)

Theorem (Correctness of $regr(\varphi, o)$)

Let φ be a logical formula, o an operator and s a state of a propositional planning task.

Then $s \models regr(\varphi, o)$ iff o is applicable in s and $s[\![o]\!] \models \varphi$.

Regressing Formulas Through Operators: Correctness (2)

Reminder: $regr(\varphi, o) = pre(o) \wedge regr(\varphi, eff(o))$

Proof.

Case 1: $s \models pre(o)$.

Then o is applicable in s and the statement we must prove simplifies to: $s \models regr(\varphi, e)$ iff $s[\![e]\!] \models \varphi$, where $e = eff(o)$.
This was proved in the previous lemma.

Case 2: $s \not\models pre(o)$.

Then $s \not\models regr(\varphi, o)$ and o is not applicable in s.
Hence both statements are false and therefore equivalent. □

Regression Examples (1)

Examples: compute regression and simplify to DNF

- $regr(b, \langle a, b \rangle)$
  $\equiv a \wedge (\top \vee (b \wedge \neg\bot))$
  $\equiv a$
- $regr(b \wedge c \wedge d, \langle a, b \rangle)$
  $\equiv a \wedge (\top \vee (b \wedge \neg\bot)) \wedge (\bot \vee (c \wedge \neg\bot)) \wedge (\bot \vee (d \wedge \neg\bot))$
  $\equiv a \wedge c \wedge d$
- $regr(b \wedge \neg c, \langle a, b \wedge c \rangle)$
  $\equiv a \wedge (\top \vee (b \wedge \neg\bot)) \wedge \neg(\top \vee (c \wedge \neg\bot))$
  $\equiv a \wedge \top \wedge \bot$
  $\equiv \bot$

Regression Examples (2)

Examples: compute regression and simplify to DNF

- $regr(b, \langle a, c \rhd b \rangle)$
  $\equiv a \wedge (c \vee (b \wedge \neg\bot))$
  $\equiv a \wedge (c \vee b)$
  $\equiv (a \wedge c) \vee (a \wedge b)$
- $regr(b, \langle a, (c \rhd b) \wedge ((d \wedge \neg c) \rhd \neg b) \rangle)$
  $\equiv a \wedge (c \vee (b \wedge \neg(d \wedge \neg c)))$
  $\equiv a \wedge (c \vee (b \wedge (\neg d \vee c)))$
  $\equiv a \wedge (c \vee (b \wedge \neg d) \vee (b \wedge c))$
  $\equiv a \wedge (c \vee (b \wedge \neg d))$
  $\equiv (a \wedge c) \vee (a \wedge b \wedge \neg d)$

C4.4 Summary

Summary

- Regressing a propositional state variable through an (arbitrary) operator must consider two cases:
  - state variables made true (by add effects)
  - state variables remaining true (by absence of delete effects)
- Regression of propositional state variables can be generalized to arbitrary formulas φ by replacing each occurrence of a state variable in φ by its regression.
- Regressing a formula φ through an operator involves regressing φ through the effect and enforcing the precondition.

Planning and Optimization

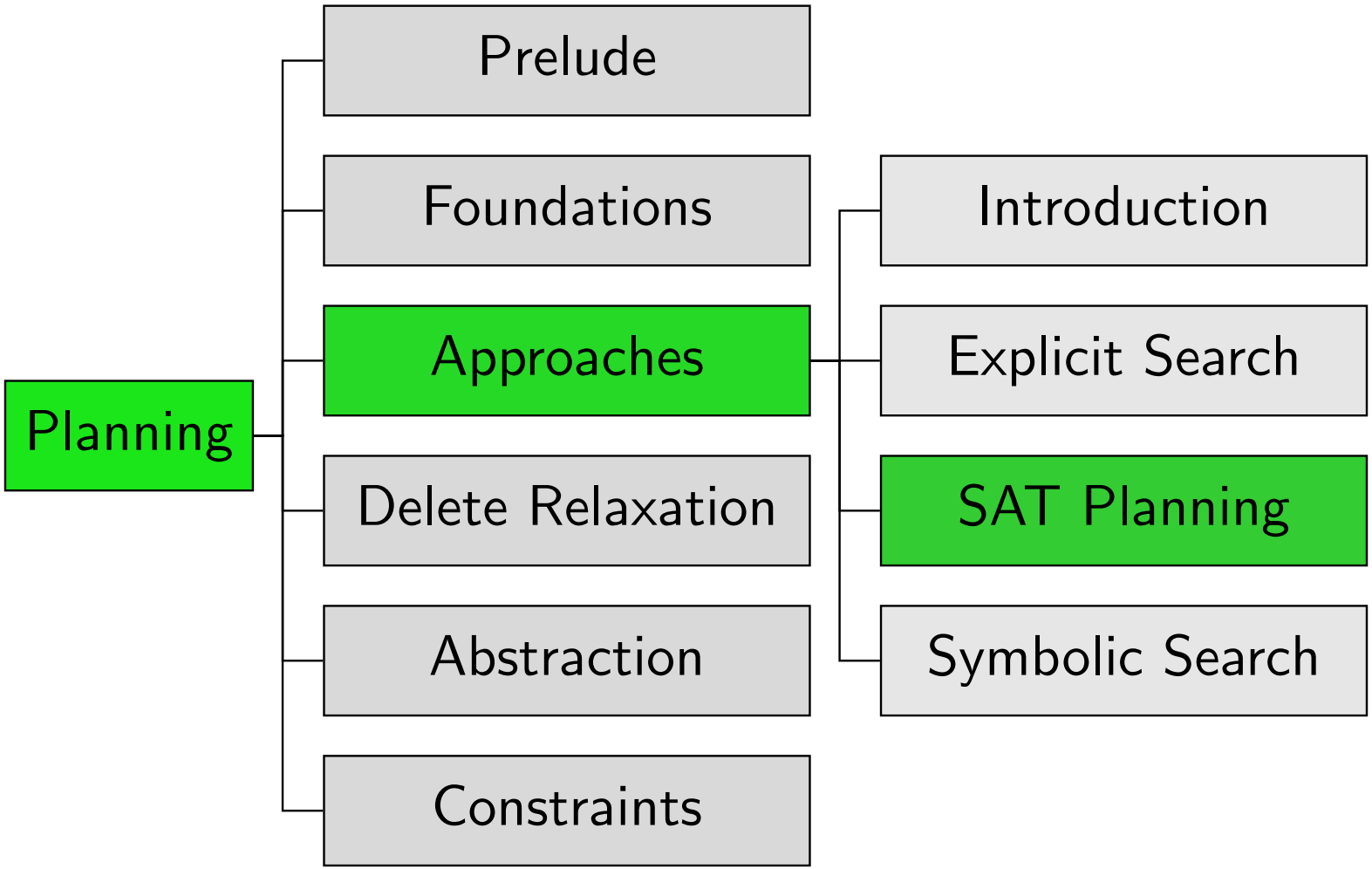
C5. SAT Planning: Core Idea and Sequential Encoding

Malte Helmert and Gabriele Röger

Universität Basel

October 13, 2025

Content of the Course



C5.1 Introduction

SAT Solvers

- SAT solvers (algorithms that find satisfying assignments to CNF formulas) are one of the major success stories in solving hard combinatorial problems.
- Can we leverage them for classical planning?

$\rightsquigarrow$ SAT planning (a.k.a. planning as satisfiability)

background on SAT Solvers:

$\rightsquigarrow$ Foundations of Artificial Intelligence Course, Ch. E4–E5

Complexity Mismatch

- The SAT problem is NP-complete,
  while $\textsc{PlanEx}$ is PSPACE-complete.

$\rightsquigarrow$ one-shot polynomial reduction from $\textsc{PlanEx}$ to SAT not possible (unless NP $=$ PSPACE)

Solution: Iterative Deepening

- We can generate a propositional formula that tests if task Π has a plan with horizon (length bound) T in time $O(\|\Pi\|^k \cdot T)$ ($\rightsquigarrow$ pseudo-polynomial reduction).
- Use as building block of algorithm that probes increasing horizons (a bit like IDA*).
- Can be efficient if there exist plans that are not excessively long.

SAT Planning: Main Loop

basic SAT planning algorithm:

SAT Planning

def satplan(Π):
    for $T \in \{0, 1, 2, \dots\}$:
        $\varphi :=$ build_sat_formula(Π, T)
        $I =$ sat_solver(φ) ▷ returns a model or **none**
        if I **is not none**:
            return extract_plan(Π, T, I)

Termination criterion for unsolvable tasks?

C5.2 Formula Overview

SAT Formula: CNF?

- SAT solvers require conjunctive normal form (CNF), i.e., formulas expressed as collection of clauses.
- We will make sure that our SAT formulas are in CNF when our input is a STRIPS task.
- We do allow fully general propositional tasks, but then the formula may need additional conversion to CNF.

SAT Formula: Variables

- given propositional planning task $\Pi = \langle V, I, O, \gamma \rangle$
- given horizon $T \in \mathbb{N}_0$

Variables of the SAT Formula

- propositional variables v^i for all $v \in V$, $0 \leq i \leq T$
  encode state after i steps of the plan
- propositional variables o^i for all $o \in O$, $1 \leq i \leq T$
  encode operator(s) applied in i-th step of the plan

Formulas with Time Steps

Definition (Time-Stamped Formulas)

Let φ be a propositional logic formula over the variables V.
Let $0 \leq i \leq T$.

We write φ^i for the formula obtained from φ
by replacing each $v \in V$ with v^i.

Example: $((a \wedge b) \vee \neg c)^3 = (a^3 \wedge b^3) \vee \neg c^3$

SAT Formula: Motivation

We want to express a formula whose models
are exactly the plans/traces with T steps.

For this, the formula must express four things:

- The variables v^0 ($v \in V$) define the initial state.
- The variables v^T ($v \in V$) define a goal state.
- We select exactly one operator variable o^i ($o \in O$)
  for each time step $1 \leq i \leq T$.
- If we select o^i, then variables v^{i-1} and v^i ($v \in V$)
  describe a state transition from the $(i-1)$-th state of the plan
  to the i-th state of the plan (that uses operator o).

The final formula is the conjunction of all these parts.

C5.3 Initial State, Goal, Operator Selection

SAT Formula: Initial State

SAT Formula: Initial State

initial state clauses:

- v^0 for all $v \in V$ with $I(v) = \mathbf{T}$
- $\neg v^0$ for all $v \in V$ with $I(v) = \mathbf{F}$

SAT Formula: Goal

SAT Formula: Goal

goal clauses:

- γ^T

For STRIPS, this is a conjunction of unit clauses.
For general goals, this may not be in clause form.

SAT Formula: Operator Selection

Let $O = \{o_1, \ldots, o_n\}$.

SAT Formula: Operator Selection

operator selection clauses:

- $o_1^i \vee \cdots \vee o_n^i$ for all $1 \leq i \leq T$

operator exclusion clauses:

- $\neg o_j^i \vee \neg o_k^i$ for all $1 \leq i \leq T$, $1 \leq j < k \leq n$

C5.4 Transitions

SAT Formula: Transitions

We now get to the interesting/challenging bit:
encoding the transitions.

Key observations: if we apply operator o at time i,

- its precondition must be satisfied at time $i-1$:
  $o^i \rightarrow pre(o)^{i-1}$
- variable v is true at time i iff its regression is true at $i-1$:
  $o^i \rightarrow (v^i \leftrightarrow regr(v, eff(o))^{i-1})$

Question: Why $regr(v, eff(o))$, not $regr(v, o)$?

Simplifications and Abbreviations

- Let us pick the last formula apart to understand it better (and also get a CNF representation along the way).
- Let us call the formula τ ("transition"):
  $\tau = o^i \rightarrow (v^i \leftrightarrow \mathit{regr}(v, \mathit{eff}(o))^{i-1})$.
- First, some abbreviations:
  - Let $e = \mathit{eff}(o)$.
  - Let $\rho = \mathit{regr}(v, e)$ ("regression").
    We have $\rho = \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$.
  - Let $\alpha = \mathit{effcond}(v, e)$ ("added").
  - Let $\delta = \mathit{effcond}(\neg v, e)$ ("deleted").

$\rightsquigarrow$ $\tau = o^i \rightarrow (v^i \leftrightarrow \rho^{i-1})$ with $\rho = \alpha \vee (v \wedge \neg \delta)$

Picking it Apart (1)

Reminder: $\tau = o^i \rightarrow (v^i \leftrightarrow \rho^{i-1})$ with $\rho = \alpha \vee (v \wedge \neg\delta)$

$$
\begin{aligned}
\tau &= o^i \rightarrow (v^i \leftrightarrow \rho^{i-1}) \\
&\equiv o^i \rightarrow ((v^i \rightarrow \rho^{i-1}) \wedge (\rho^{i-1} \rightarrow v^i)) \\
&\equiv \underbrace{(o^i \rightarrow (v^i \rightarrow \rho^{i-1}))}_{\tau_1} \wedge \underbrace{(o^i \rightarrow (\rho^{i-1} \rightarrow v^i))}_{\tau_2}
\end{aligned}
$$

$\rightsquigarrow$ consider this two separate constraints τ_1 and τ_2

Picking it Apart (2)

Reminder: $\tau_1 = o^i \rightarrow (v^i \rightarrow \rho^{i-1})$ with $\rho = \alpha \vee (v \wedge \neg\delta)$

$$
\begin{aligned}
\tau_1 &= o^i \rightarrow (v^i \rightarrow \rho^{i-1}) \\
&\equiv o^i \rightarrow (\neg\rho^{i-1} \rightarrow \neg v^i) \\
&\equiv (o^i \wedge \neg\rho^{i-1}) \rightarrow \neg v^i \\
&\equiv (o^i \wedge \neg(\alpha^{i-1} \vee (v^{i-1} \wedge \neg\delta^{i-1}))) \rightarrow \neg v^i \\
&\equiv (o^i \wedge (\neg\alpha^{i-1} \wedge (\neg v^{i-1} \vee \delta^{i-1}))) \rightarrow \neg v^i \\
&\equiv \underbrace{((o^i \wedge \neg\alpha^{i-1} \wedge \neg v^{i-1}) \rightarrow \neg v^i)}_{\tau_{11}} \wedge \underbrace{((o^i \wedge \neg\alpha^{i-1} \wedge \delta^{i-1}) \rightarrow \neg v^i)}_{\tau_{12}}
\end{aligned}
$$

$\rightsquigarrow$ consider this two separate constraints τ_{11} and τ_{12}

Interpreting the Constraints (1)

Can we give an intuitive description of τ_{11} and τ_{12}?
$\rightsquigarrow$ Yes!

- $\tau_{11} = (o^i \wedge \neg\alpha^{i-1} \wedge \neg v^{i-1}) \rightarrow \neg v^i$

  "When applying o, if v is false and o does not add it, it remains false."
  - called negative frame clause
  - in clause form: $\neg o^i \vee \alpha^{i-1} \vee v^{i-1} \vee \neg v^i$
- $\tau_{12} = (o^i \wedge \neg\alpha^{i-1} \wedge \delta^{i-1}) \rightarrow \neg v^i$

  "When applying o, if o deletes v and does not add it, it is false afterwards." (Note the add-after-delete semantics.)
  - called negative effect clause
  - in clause form: $\neg o^i \vee \alpha^{i-1} \vee \neg\delta^{i-1} \vee \neg v^i$

For STRIPS tasks, these are indeed clauses. (And in general?)

Picking it Apart (3)

Almost done!

Reminder: $\tau_2 = o^i \rightarrow (\rho^{i-1} \rightarrow v^i)$ with $\rho = \alpha \vee (v \wedge \neg\delta)$

$$
\begin{aligned}
\tau_2 &= o^i \rightarrow (\rho^{i-1} \rightarrow v^i) \\
&\equiv (o^i \wedge \rho^{i-1}) \rightarrow v^i \\
&\equiv (o^i \wedge (\alpha^{i-1} \vee (v^{i-1} \wedge \neg\delta^{i-1}))) \rightarrow v^i \\
&\equiv \underbrace{((o^i \wedge \alpha^{i-1}) \rightarrow v^i)}_{\tau_{21}} \wedge \underbrace{((o^i \wedge v^{i-1} \wedge \neg\delta^{i-1}) \rightarrow v^i)}_{\tau_{22}}
\end{aligned}
$$

$\rightsquigarrow$ consider this two separate constraints τ_{21} and τ_{22}

Interpreting the Constraints (2)

How about an intuitive description of τ_{21} and τ_{22}?

- $\tau_{21} = (o^i \wedge \alpha^{i-1}) \rightarrow v^i$

  "When applying o, if o adds v, it is true afterwards."
  - called positive effect clause
  - in clause form: $\neg o^i \vee \neg \alpha^{i-1} \vee v^i$
- $\tau_{22} = (o^i \wedge v^{i-1} \wedge \neg \delta^{i-1}) \rightarrow v^i$

  "When applying o, if v is true and o does not delete it, it remains true."
  - called positive frame clause
  - in clause form: $\neg o^i \vee \neg v^{i-1} \vee \delta^{i-1} \vee v^i$

For STRIPS tasks, these are indeed clauses. (But not in general.)

SAT Formula: Transitions

SAT Formula: Transitions

precondition clauses:

- $\neg o^i \vee pre(o)^{i-1}$ for all $1 \leq i \leq T$, $o \in O$

positive and negative effect clauses:

- $\neg o^i \vee \neg \alpha^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee \neg \delta^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

positive and negative frame clauses:

- $\neg o^i \vee \neg v^{i-1} \vee \delta^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee v^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

where $\alpha = effcond(v, eff(o))$, $\delta = effcond(\neg v, eff(o))$.

For STRIPS, all except the precondition clauses are in clause form.

The precondition clauses are easily convertible to CNF
(one clause $\neg o^i \vee v^{i-1}$ for each precondition atom v of o).

C5.5 Summary

Summary: Sequential SAT Encoding (1)

Sequential SAT Encoding (1)

initial state clauses:

- v^0 for all $v \in V$ with $I(v) = \mathbf{T}$
- $\neg v^0$ for all $v \in V$ with $I(v) = \mathbf{F}$

goal clauses:

- γ^T

operator selection clauses:

- $o_1^i \vee \cdots \vee o_n^i$ for all $1 \leq i \leq T$

operator exclusion clauses:

- $\neg o_j^i \vee \neg o_k^i$ for all $1 \leq i \leq T$, $1 \leq j < k \leq n$

Summary: Sequential SAT Encoding (2)

Sequential SAT Encoding (2)

precondition clauses:

- $\neg o^i \vee pre(o)^{i-1}$ for all $1 \leq i \leq T$, $o \in O$

positive and negative effect clauses:

- $\neg o^i \vee \neg \alpha^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee \neg \delta^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

positive and negative frame clauses:

- $\neg o^i \vee \neg v^{i-1} \vee \delta^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee v^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

where $\alpha = effcond(v, eff(o))$, $\delta = effcond(\neg v, eff(o))$.

Summary

- SAT planning (planning as satisfiability) expresses a sequence of bounded-horizon planning tasks as SAT formulas.
- Plans can be extracted from satisfying assignments; unsolvable tasks are challenging for the algorithm.
- For each time step, there are propositions encoding which state variables are true and which operators are applied.
- We describe a basic sequential encoding where one operator is applied at every time step.
- The encoding produces a CNF formula for STRIPS tasks.
- The encoding follows naturally (with some work) from using regression to link state variables in adjacent time steps.

Planning and Optimization

C6. SAT Planning: Parallel Encoding

Malte Helmert and Gabriele Röger

Universität Basel

October 13, 2025

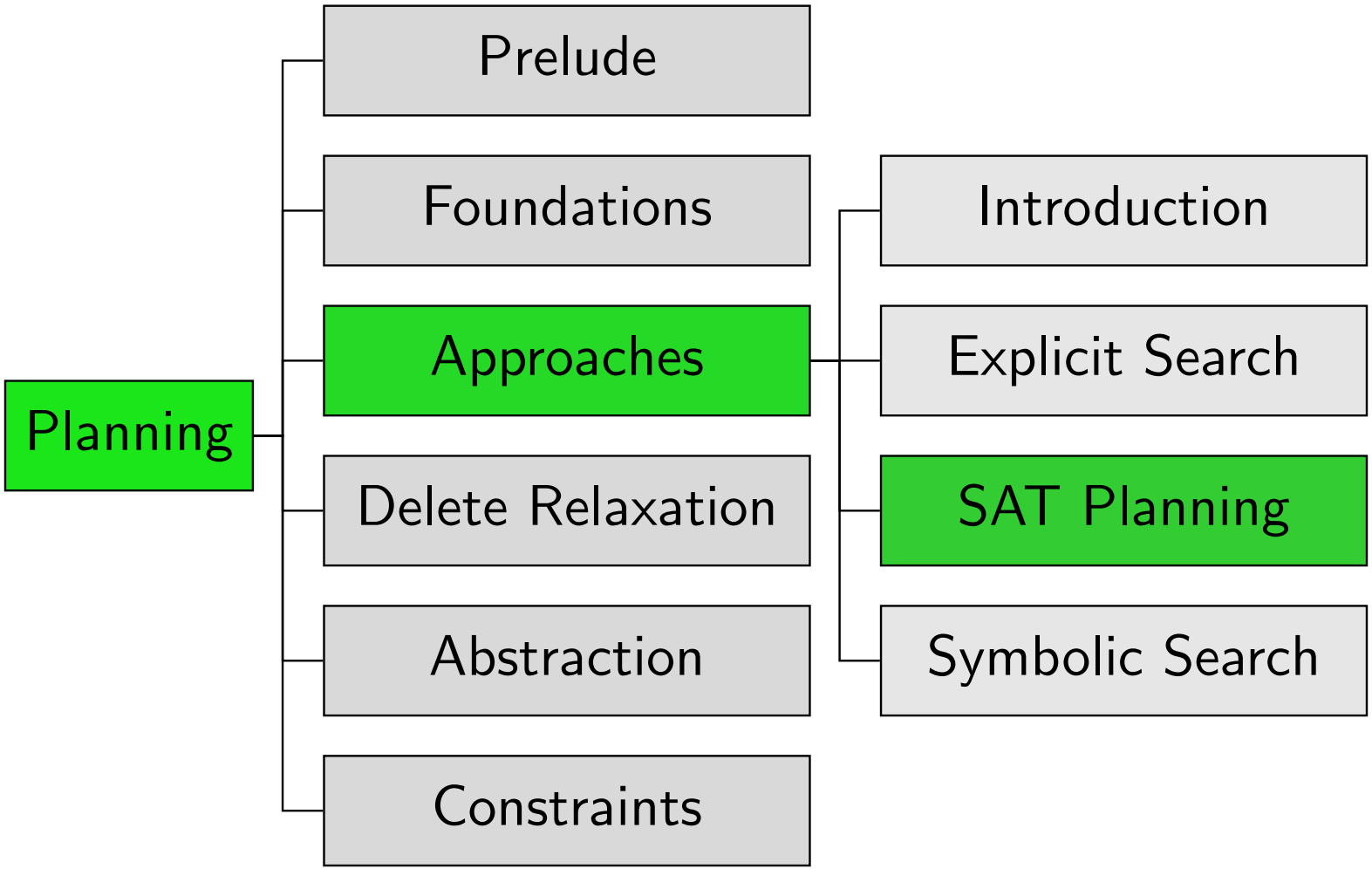
Planning and Optimization
October 13, 2025 — C6. SAT Planning: Parallel Encoding

C6.1 Introduction

C6.2 Adapting the SAT Encoding

C6.3 Summary

Content of the Course



C6.1 Introduction

Efficiency of SAT Planning

- All other things being equal, the most important aspect for efficient SAT solving is the number of propositional variables in the input formula.
- For sufficiently difficult inputs, runtime scales exponentially in the number of variables.

$\rightsquigarrow$ Can we make SAT planning more efficient by using fewer variables?

Number of Variables

Reminder:

- given propositional planning task $\Pi = \langle V, I, O, \gamma \rangle$
- given horizon $T \in \mathbb{N}_0$

Variables of the SAT Formula

- propositional variables v^i for all $v \in V$, $0 \leq i \leq T$
  encode state after i steps of the plan
- propositional variables o^i for all $o \in O$, $1 \leq i \leq T$
  encode operator(s) applied in i-th step of the plan

$\rightsquigarrow$ $|V| \cdot (T + 1) + |O| \cdot T$ variables

$\rightsquigarrow$ SAT solving runtime usually exponential in T

Parallel Plans and Commutativity

Can we get away with shorter horizons?

Idea:

- allow parallel plans in the SAT encoding:
  multiple operators can be applied in the same step
  if they do not interfere

Definition (commutative, interfere)

Let $O' = \{o_1, \ldots, o_n\}$ be a set of operators applicable in state s.

We say that O' is commutative in s if

- for all permutations π of O', $s[\![\pi]\!]$ is defined, and
- for all permutations π, π' of O', $s[\![\pi]\!] = s[\![\pi']\!]$.

We say that the set O' interferes in s if it is not commutative in s.

Parallel Plan Extraction

- If we can guarantee commutativity, we can allow multiple operators at the same time in the SAT encoding.
- A parallel plan (with multiple o^i used for the same i) extracted from the SAT formula can then be converted into a "regular" plan by ordering the operators within each time step arbitrarily.

Challenges for Parallel SAT Encodings

Two challenges remain:

- our current SAT encoding does not allow concurrent operators
- how do we ensure that concurrent operators are commutative?

C6.2 Adapting the SAT Encoding

Reminder: Sequential SAT Encoding (1)

Sequential SAT Encoding (1)

initial state clauses:

- v^0 for all $v \in V$ with $I(v) = \mathbf{T}$
- $\neg v^0$ for all $v \in V$ with $I(v) = \mathbf{F}$

goal clauses:

- γ^T

operator selection clauses:

- $o_1^i \vee \cdots \vee o_n^i$ for all $1 \leq i \leq T$

operator exclusion clauses:

- $\neg o_j^i \vee \neg o_k^i$ for all $1 \leq i \leq T$, $1 \leq j < k \leq n$

$\rightsquigarrow$ operator exclusion clauses must be adapted

Reminder: Sequential SAT Encoding (2)

Sequential SAT Encoding (2)

precondition clauses:

- $\neg o^i \vee \mathit{pre}(o)^{i-1}$ for all $1 \leq i \leq T$, $o \in O$

positive and negative effect clauses:

- $\neg o^i \vee \neg \alpha^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee \neg \delta^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

positive and negative frame clauses:

- $\neg o^i \vee \neg v^{i-1} \vee \delta^{i-1} \vee v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $\neg o^i \vee \alpha^{i-1} \vee v^{i-1} \vee \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

where $\alpha = \mathit{effcond}(v, \mathit{eff}(o))$, $\delta = \mathit{effcond}(\neg v, \mathit{eff}(o))$.

$\rightsquigarrow$ rewrite clauses as implications

Sequential SAT Encoding (2) Rewritten as Implications

Sequential SAT Encoding (2) Rewritten

precondition clauses:

- $o^i \rightarrow pre(o)^{i-1}$ for all $1 \leq i \leq T$, $o \in O$

positive and negative effect clauses:

- $(o^i \wedge \alpha^{i-1}) \rightarrow v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $(o^i \wedge \delta^{i-1} \wedge \neg\alpha^{i-1}) \rightarrow \neg v^i$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

positive and negative frame clauses:

- $(o^i \wedge v^{i-1} \wedge \neg v^i) \rightarrow \delta^{i-1}$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$
- $(o^i \wedge \neg v^{i-1} \wedge v^i) \rightarrow \alpha^{i-1}$ for all $1 \leq i \leq T$, $o \in O$, $v \in V$

where $\alpha = effcond(v, eff(o))$, $\delta = effcond(\neg v, eff(o))$.

$\rightsquigarrow$ frame clauses must be adapted

Adapting the Operator Exclusion Clauses: Idea

Reminder: operator exclusion clauses $\neg o_j^i \vee \neg o_k^i$
for all $1 \leq i \leq T$, $1 \leq j < k \leq n$

- Ideally: replace with clauses that express "for all states s, the operators selected at time i are commutative in s"
- but: testing if a given set of operators interferes in any state is itself an NP-complete problem

$\rightsquigarrow$ use something less heavy: a sufficient condition for commutativity can be expressed at the level of pairs of operators

Conflicting Operators

- Intuitively, two operators conflict if
  - one can disable the precondition of the other,
  - one can override an effect of the other, or
  - one can enable or disable an effect condition of the other.
- If no two operators in a set O' conflict, then O' is commutative in all states.
- This is still difficult to test, so we restrict attention to the STRIPS case in the following.

Definition (Conflicting STRIPS Operator)

Operators o and o' of a STRIPS task Π conflict if

- o deletes a precondition of o' or vice versa, or
- o deletes an add effect of o' or vice versa.

Adapting the Operator Exclusion Clauses: Solution

Reminder: operator exclusion clauses $\neg o_j^i \vee \neg o_k^i$
for all $1 \leq i \leq T$, $1 \leq j < k \leq n$

Solution:

Parallel SAT Formula: Operator Exclusion Clauses

operator exclusion clauses:

- $\neg o_j^i \vee \neg o_k^i$ for all $1 \leq i \leq T$, $1 \leq j < k \leq n$
  such that o_j and o_k conflict

Adapting the Frame Clauses: Idea

Reminder: frame clauses

$$(o^i \wedge v^{i-1} \wedge \neg v^i) \rightarrow \delta^{i-1} \quad \text{for all } 1 \leq i \leq T,\ o \in O,\ v \in V$$
$$(o^i \wedge \neg v^{i-1} \wedge v^i) \rightarrow \alpha^{i-1} \quad \text{for all } 1 \leq i \leq T,\ o \in O,\ v \in V$$

What is the problem?

- These clauses express that if o is applied at time i and the value of v changes, then o caused the change.
- This is no longer true if we want to be able to apply two operators concurrently.

$\rightsquigarrow$ Instead, say "If the value of v changes, then some operator must have caused the change."

Adapting the Frame Clauses: Solution

Reminder: frame clauses

$$(o^i \wedge v^{i-1} \wedge \neg v^i) \rightarrow \delta^{i-1} \quad \text{for all } 1 \leq i \leq T,\ o \in O,\ v \in V$$
$$(o^i \wedge \neg v^{i-1} \wedge v^i) \rightarrow \alpha^{i-1} \quad \text{for all } 1 \leq i \leq T,\ o \in O,\ v \in V$$

Solution:

Parallel SAT Formula: Frame Clauses

positive and negative frame clauses:

- $(v^{i-1} \wedge \neg v^i) \rightarrow ((o_1^i \wedge \delta_{o_1}^{i-1}) \vee \cdots \vee (o_n^i \wedge \delta_{o_n}^{i-1}))$
  for all $1 \leq i \leq T,\ v \in V$
- $(\neg v^{i-1} \wedge v^i) \rightarrow ((o_1^i \wedge \alpha_{o_1}^{i-1}) \vee \cdots \vee (o_n^i \wedge \alpha_{o_n}^{i-1}))$
  for all $1 \leq i \leq T,\ v \in V$

where $\alpha_o = \mathit{effcond}(v, \mathit{eff}(o))$, $\delta_o = \mathit{effcond}(\neg v, \mathit{eff}(o))$,
$O = \{o_1, \ldots, o_n\}$.

For STRIPS, these are in clause form.

C6.3 Summary

Summary

- As a rule of thumb, SAT solvers generally perform better on formulas with fewer variables.
- Parallel encodings reduce the number of variables by shortening the horizon needed to solve a planning task.
- Parallel encodings replace the constraint that operators are not applied concurrently by the constraint that conflicting operators are not applied concurrently.
- To make parallelism possible, the frame clauses also need to be adapted.

Planning and Optimization

C7. Symbolic Search: Binary Decision Diagrams

Malte Helmert and Gabriele Röger

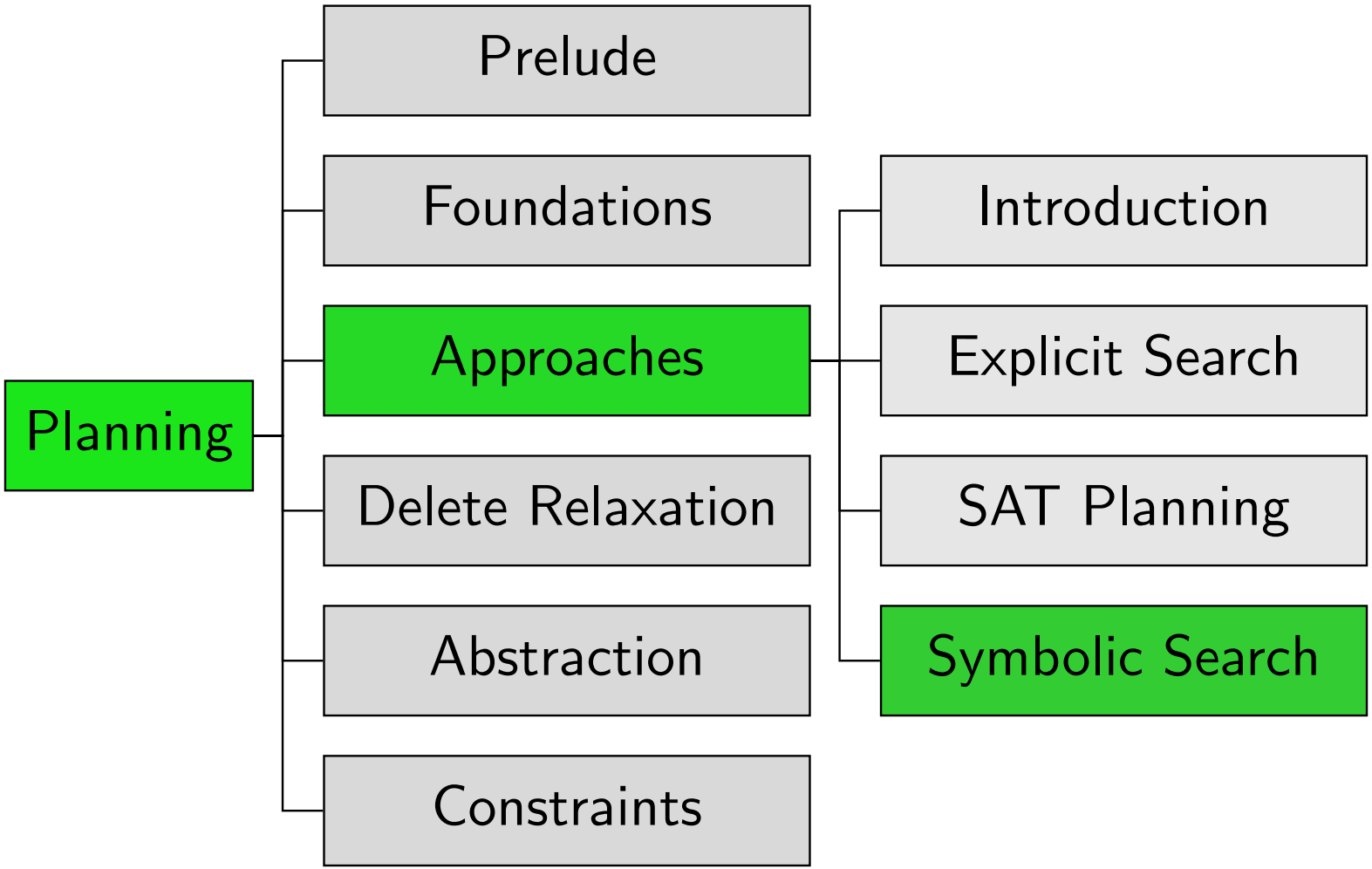
Universität Basel

October 15, 2025

Planning and Optimization

October 15, 2025 — C7. Symbolic Search: Binary Decision Diagrams

Content of the Course



C7.1 Motivation

Symbolic Search Planning: Basic Ideas

- come up with a good data structure for sets of states
- hope: (at least some) exponentially large state sets can be represented as polynomial-size data structures
- simulate a standard search algorithm like breadth-first search using these set representations

Symbolic Breadth-First Progression Search

Symbolic Breadth-First Progression Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

$\rightsquigarrow$ If we can implement operations $models$, $\{I\}$, $\cap$, $\neq \emptyset$, $\cup$, $apply$ and $=$ efficiently, this is a reasonable algorithm.

C7.2 Data Structures for State Sets

Representing State Sets

We need to represent and manipulate state sets (again)!

- How about an explicit representation, like a hash table?
- And how about our good old friend, the formula?

Time Complexity: Explicit Representations vs. Formulas

Let k be the number of state variables,
$|S|$ the number of states in S and
$\|S\|$ the size of the representation of S.

	Hash table	Formula
$s \in S$?	$O(k)$	$O(\|\|S\|\|)$
$S := S \cup \{s\}$	$O(k)$	$O(k)$
$S := S \setminus \{s\}$	$O(k)$	$O(k)$
$S \cup S'$	$O(k\|S\| + k\|S'\|)$	$O(1)$
$S \cap S'$	$O(k\|S\| + k\|S'\|)$	$O(1)$
$S \setminus S'$	$O(k\|S\| + k\|S'\|)$	$O(1)$
$\overline{S}$	$O(k2^k)$	$O(1)$
$\{s \mid s(v) = \mathbf{T}\}$	$O(k2^k)$	$O(1)$
$S = \emptyset$?	$O(1)$	co-NP-complete
$S = S'$?	$O(k\|S\|)$	co-NP-complete
$\|S\|$	$O(1)$	#P-complete

Which Operations are Important?

- Explicit representations such as hash tables are unsuitable because their size grows linearly with the number of represented states.
- Formulas are very efficient for some operations, but not for other important operations needed by the breadth-first search algorithm.
  - Examples: $S \neq \emptyset$?, $S = S'$?

Canonical Representations

- One of the problems with formulas is that they allow many different representations for the same set.
  - For example, all unsatisfiable formulas represent $\emptyset$.

  This makes equality tests expensive.
- We would like data structures with a canonical representation, i.e., with only one possible representation for every state set.
- Reduced ordered binary decision diagrams (BDDs) are an example of such a canonical representation.

Time Complexity: Formulas vs. BDDs

Let k be the number of state variables,
$|S|$ the number of states in S and
$\|S\|$ the size of the representation of S.

	Formula	BDD
$s \in S?$	$O(\|\|S\|\|)$	$O(k)$
$S := S \cup \{s\}$	$O(k)$	$O(k)$
$S := S \setminus \{s\}$	$O(k)$	$O(k)$
$S \cup S'$	$O(1)$	$O(\|\|S\|\|\|\|S'\|\|)$
$S \cap S'$	$O(1)$	$O(\|\|S\|\|\|\|S'\|\|)$
$S \setminus S'$	$O(1)$	$O(\|\|S\|\|\|\|S'\|\|)$
$\overline{S}$	$O(1)$	$O(\|\|S\|\|)$
$\{s \mid s(v) = \mathbf{T}\}$	$O(1)$	$O(1)$
$S = \emptyset?$	co-NP-complete	$O(1)$
$S = S'?$	co-NP-complete	$O(1)$
$\|S\|$	#P-complete	$O(\|\|S\|\|)$

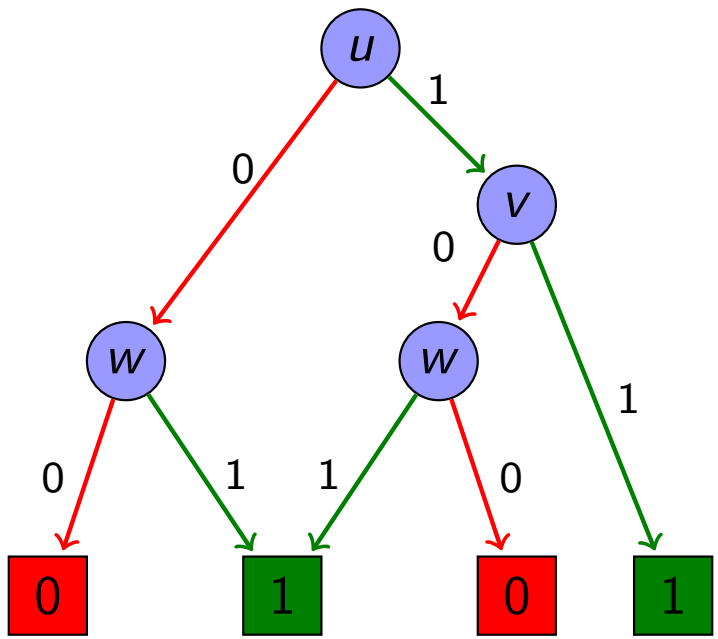
Remark: Optimizations allow BDDs with complementation ($\overline{S}$) in constant time, but we will not discuss this here.

C7.3 Binary Decision Diagrams

BDD Example

Example

Possible BDD for $(u \wedge v) \vee w$



Binary Decision Diagrams: Definition

Definition (BDD)

Let V be a set of propositional variables.

A binary decision diagram (BDD) over V is a directed acyclic graph with labeled arcs and labeled vertices such that:

- There is exactly one node without incoming arcs.
- All sinks (nodes without outgoing arcs) are labeled 0 or 1.
- All other nodes are labeled with a variable $v \in V$ and have exactly two outgoing arcs, labeled 0 and 1.

A note on notation:

- In BDDs, 1 stands for **T** and 0 for **F**.
- We follow this customary notation in BDDs, but stick to **T** and **F** when speaking of logic.

Binary Decision Diagrams: Terminology

BDD Terminology

- The node without incoming arcs is called the root.
- The labeling variable of an internal node is called the decision variable of the node.
- The nodes reached from node n via the arc labeled $i \in \{0, 1\}$ is called the i-successor of n.
- The BDDs which only consist of a single sink are called the zero BDD and one BDD.

Observation: If B is a BDD and n is a node of B, then the subgraph induced by all nodes reachable from n is also a BDD.

- This BDD is called the BDD rooted at n.

BDD Semantics

Testing whether a BDD Includes a Variable Assignment

def bdd-includes(B: BDD, I: variable assignment):
    Set n to the root of B.
    while n is not a sink:
        Set v to the decision variable of n.
        Set n to the 1-successor of n if $I(v) = \mathbf{T}$ and
          to the 0-successor of n if $I(v) = \mathbf{F}$.
    return true if n is labeled 1, **false** if it is labeled 0.

Definition (Set Represented by a BDD)

Let B be a BDD over variables V.

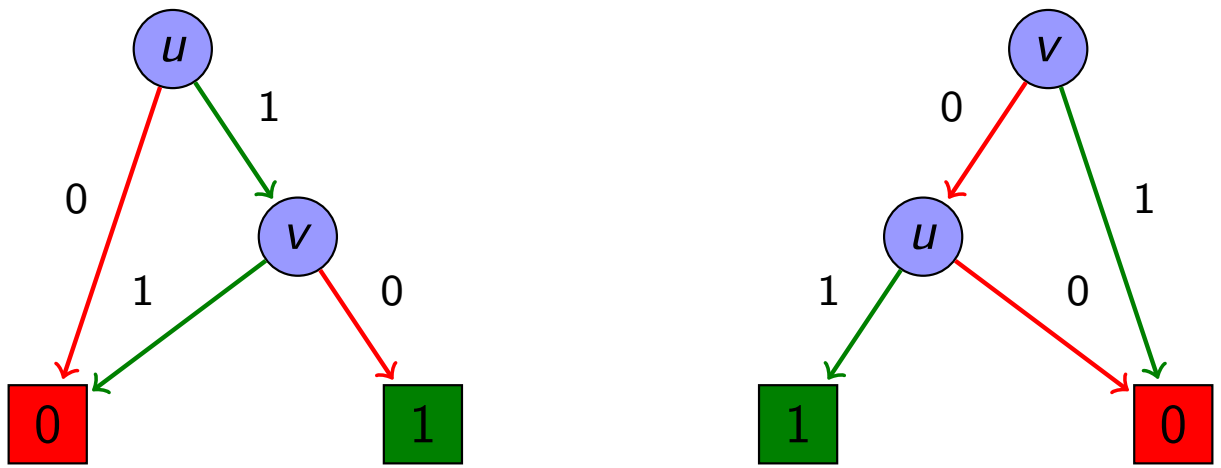
The set represented by B, in symbols $r(B)$,
consists of all variable assignments $I : V \to \{\mathbf{T}, \mathbf{F}\}$
for which *bdd-includes*(B, I) returns true.

C7.4 BDDs as Canonical Representations

Ordered BDDs: Motivation

In general, BDDs are not a canonical representation for sets of interpretations. Here is a simple counter-example ($V = \{u, v\}$):

Example (BDDs for $u \wedge \neg v$ with Different Variable Order)



Both BDDs represent the same state set, namely the singleton set $\{\{u \mapsto \mathbf{T}, v \mapsto \mathbf{F}\}\}$.

Ordered BDDs: Definition

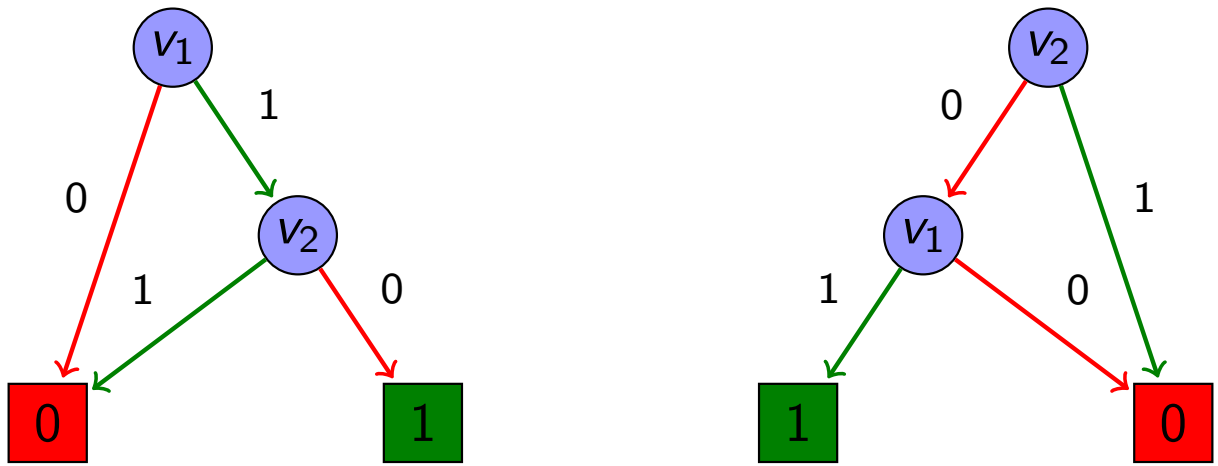
- As a first step towards a canonical representation, we now require that the set of variables is totally ordered by some ordering $\prec$.
- In particular, we will only use variables $v_1, v_2, v_3, \ldots$ and assume the ordering $v_i \prec v_j$ iff $i < j$.

Definition (Ordered BDD)

A BDD is ordered (w.r.t. $\prec$) iff for each arc from a node with decision variable u to a node with decision variable v, we have $u \prec v$.

Ordered BDDs: Example

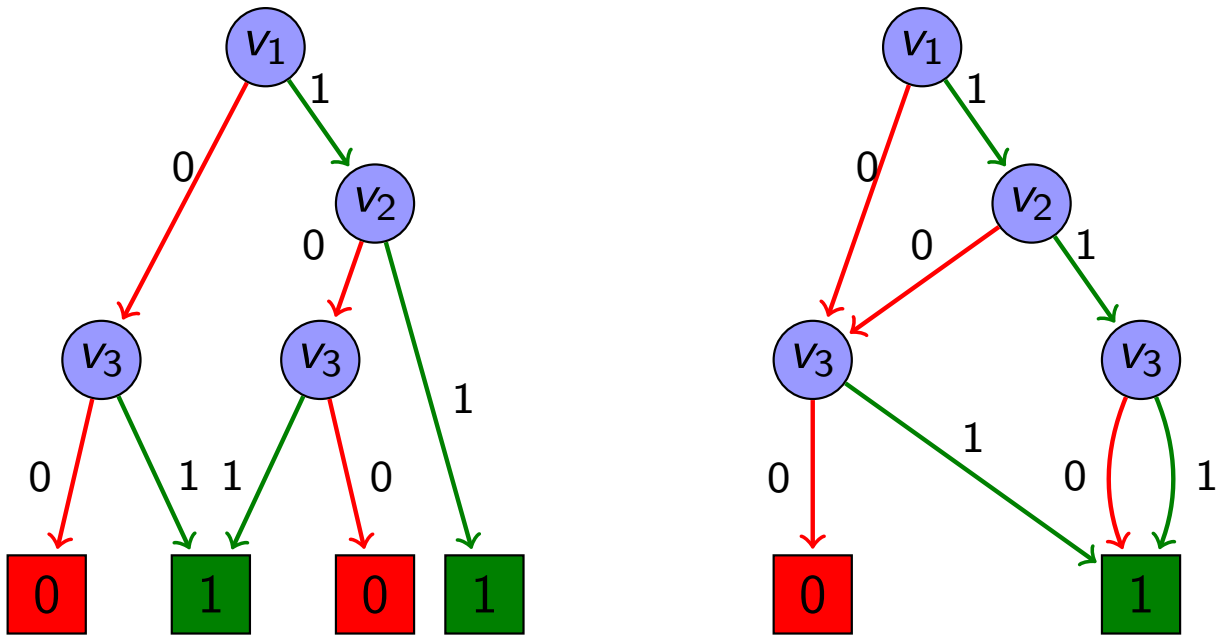
Example (Ordered and Unordered BDD)



The left BDD is ordered w.r.t. the ordering we use in this chapter, the right one is not.

Reduced Ordered BDDs: Are Ordered BDDs Canonical?

Example (Two equivalent BDDs that can be reduced)



- Ordered BDDs are still not canonical:
  both ordered BDDs represent the same set.
- However, ordered BDDs can easily be made canonical.

Reduced Ordered BDDs: Reductions (1)

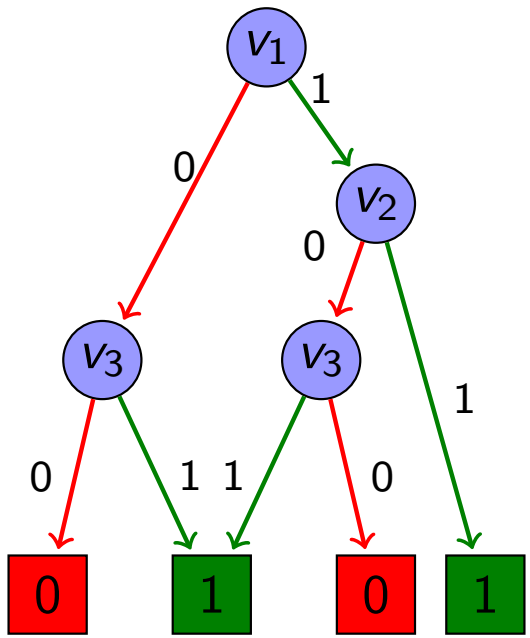
There are two important operations on BDDs
that do not change the set represented by it:

Definition (Isomorphism Reduction)

If the BDDs rooted at two different nodes n and n' are isomorphic,
then all incoming arcs of n' can be redirected to n,
and all BDD nodes unreachable from the root can be removed.

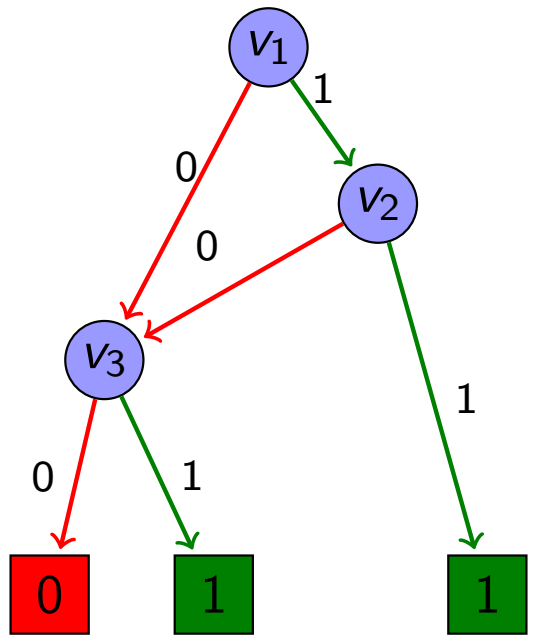
Reduced Ordered BDDs: Reductions (2)

Example (Isomorphism Reduction)



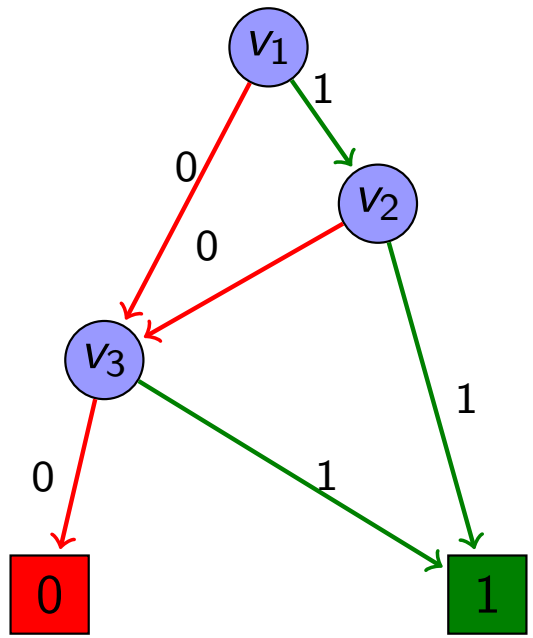
Reduced Ordered BDDs: Reductions (2)

Example (Isomorphism Reduction)



Reduced Ordered BDDs: Reductions (2)

Example (Isomorphism Reduction)



Reduced Ordered BDDs: Reductions (3)

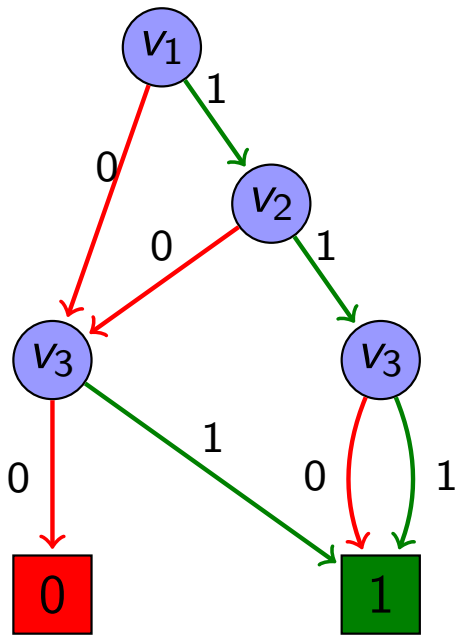
There are two important operations on BDDs
that do not change the set represented by it:

Definition (Shannon Reduction)

If both outgoing arcs of an internal node n of a BDD lead to the same node m, then n can be removed from the BDD, with all incoming arcs of n going to m instead.

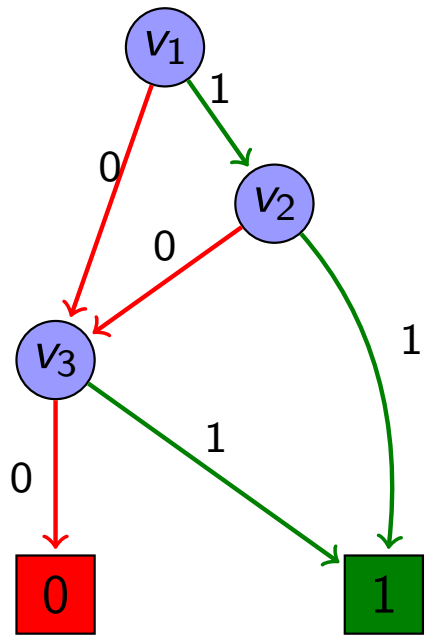
Reduced Ordered BDDs: Reductions (4)

Example (Shannon Reduction)



Reduced Ordered BDDs: Reductions (4)

Example (Shannon Reduction)



Reduced Ordered BDDs: Definition

Definition (Reduced Ordered BDD)

An ordered BDD is reduced iff it does not admit any isomorphism reduction or Shannon reduction.

Theorem (Bryant 1986)

For every state set S and a fixed variable ordering, there exists exactly one reduced ordered BDD representing S.

Moreover, given any ordered BDD B, the equivalent reduced ordered BDD can be computed in linear time in the size of B.

$\rightsquigarrow$ Reduced ordered BDDs are the canonical representation we are looking for.

From now on, we simply say BDD for reduced ordered BDD.

C7.5 Summary

Summary

- Symbolic search is based on the idea of performing a state-space search where many states are considered "at once" by operating on sets of states rather than individual states.
- Binary decision diagrams are a data structure to compactly represent and manipulate sets of variable assignments.
- Reduced ordered BDDs are a canonical representation of such sets.

Planning and Optimization

C8. Symbolic Search: Full Algorithm

Malte Helmert and Gabriele Röger

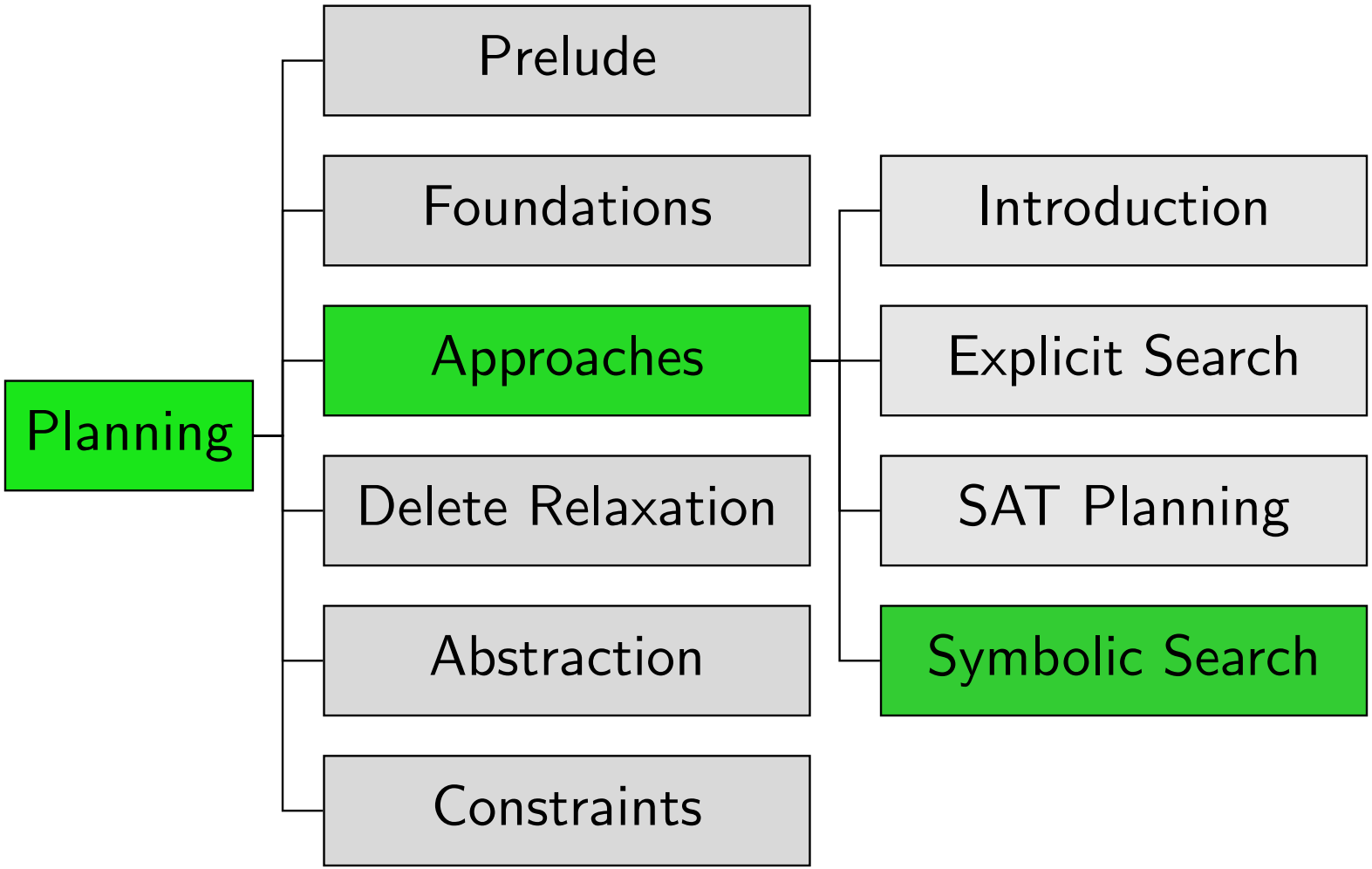
Universität Basel

October 15, 2025

Planning and Optimization

October 15, 2025 — C8. Symbolic Search: Full Algorithm

Content of the Course



Devising a Symbolic Search Algorithm

- We now put the pieces together to build a symbolic search algorithm for propositional planning tasks.
- use BDDs as a black box data structure:
  - care about provided operations and their time complexity
  - do not care about their internal implementation
- Efficient implementations are available as libraries, e.g.:
  - CUDD, a high-performance BDD library
  - libbdd, shipped with Ubuntu Linux

C8.1 Basic BDD Operations

BDD Operations: Preliminaries

- All BDDs work on a fixed and totally ordered set of propositional variables.
- Complexity of operations given in terms of:
  - k, the number of BDD variables
  - $\|B\|$, the number of nodes in the BDD B

BDD Operations (1)

BDD operations: logical/set atoms

- bdd-fullset(): build BDD representing all assignments
  - in logic: $\top$
  - time complexity: $O(1)$
- bdd-emptyset(): build BDD representing $\emptyset$
  - in logic: $\bot$
  - time complexity: $O(1)$
- bdd-atom(v): build BDD representing $\{s \mid s(v) = \mathbf{T}\}$
  - in logic: v
  - time complexity: $O(1)$

BDD Operations (2)

BDD operations: logical/set connectives

- bdd-complement(B): build BDD representing $\overline{r(B)}$
  - in logic: $\neg\varphi$
  - time complexity: $O(\|B\|)$
- bdd-union(B, B'): build BDD representing $r(B) \cup r(B')$
  - in logic: $(\varphi \vee \psi)$
  - time complexity: $O(\|B\| \cdot \|B'\|)$
- bdd-intersection(B, B'): build BDD representing $r(B) \cap r(B')$
  - in logic: $(\varphi \wedge \psi)$
  - time complexity: $O(\|B\| \cdot \|B'\|)$

BDD Operations (3)

BDD operations: Boolean tests

- bdd-includes(B, I): return **true** iff $I \in r(B)$
  - in logic: $I \models \varphi$?
  - time complexity: $O(k)$
- bdd-equals(B, B'): return **true** iff $r(B) = r(B')$
  - in logic: $\varphi \equiv \psi$?
  - time complexity: $O(1)$ (due to canonical representation)

Conditioning: Formulas

The last two basic BDD operations are a bit more unusual and require some preliminary remarks.

Conditioning a variable v in a formula φ to $\mathbf{T}$ or $\mathbf{F}$, written $\varphi[\mathbf{T}/v]$ or $\varphi[\mathbf{F}/v]$, means restricting v to a particular truth value:

Examples:

- $(A \wedge (B \vee \neg C))[\mathbf{T}/B] = (A \wedge (\top \vee \neg C)) \equiv A$
- $(A \wedge (B \vee \neg C))[\mathbf{F}/B] = (A \wedge (\bot \vee \neg C)) \equiv A \wedge \neg C$

Conditioning: Sets of Assignments

We can define the same operation for sets of assignments S:
$S[\mathbf{F}/v]$ and $S[\mathbf{T}/v]$ restrict S to elements with the given value for v and remove v from the domain of definition:

Example:

- $S = \{\{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}, C \mapsto \mathbf{F}\},$
  $\{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{F}\},$
  $\{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{T}\}\}$

$\rightsquigarrow$ $S[\mathbf{T}/B] = \{\{A \mapsto \mathbf{T}, C \mapsto \mathbf{F}\},$
$\{A \mapsto \mathbf{T}, C \mapsto \mathbf{T}\}\}$

Forgetting

Forgetting (a.k.a. existential abstraction) is similar to conditioning: we allow either truth value for v and remove the variable.

We write this as $\exists v\, \varphi$ (for formulas) and $\exists v\, S$ (for sets).

Formally:

- $\exists v\, \varphi = \varphi[\mathbf{T}/v] \vee \varphi[\mathbf{F}/v]$
- $\exists v\, S = S[\mathbf{T}/v] \cup S[\mathbf{F}/v]$

Forgetting: Example

Examples:

- $S = \{\{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}, C \mapsto \mathbf{F}\},$
  $\{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{F}\},$
  $\{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{T}\}\}$

$\rightsquigarrow$ $\exists B\, S = \{\{A \mapsto \mathbf{F}, C \mapsto \mathbf{F}\},$
$\{A \mapsto \mathbf{T}, C \mapsto \mathbf{F}\},$
$\{A \mapsto \mathbf{T}, C \mapsto \mathbf{T}\}\}$

$\rightsquigarrow$ $\exists C\, S = \{\{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}\},$
$\{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}\}\}$

BDD Operations (4)

BDD operations: conditioning and forgetting

- bdd-condition(B, v, t) where $t \in \{\mathbf{T}, \mathbf{F}\}$:
  build BDD representing $r(B)[t/v]$
  - in logic: $\varphi[t/v]$
  - time complexity: $O(\|B\|)$
- bdd-forget(B, v):
  build BDD representing $\exists v\, r(B)$
  - in logic: $\exists v\, \varphi \quad (= \varphi[\mathbf{T}/v] \vee \varphi[\mathbf{F}/v])$
  - time complexity: $O(\|B\|^2)$

C8.2 Formulas and Singletons

Formulas to BDDs

- With the logical/set operations, we can convert propositional formulas φ into BDDs representing the models of φ.
- We denote this computation with bdd-formula(φ).
- Each individual logical connective takes polynomial time, but converting a full formula of length n can take $O(2^n)$ time. (How is this possible?)

Singleton BDDs

- We can convert a single truth assignment I into a BDD representing $\{I\}$ by computing the conjunction of all literals true in I (using bdd-atom, bdd-complement and bdd-intersection).
- We denote this computation with bdd-singleton(I).
- When done in the correct order, this takes time $O(k)$.

C8.3 Renaming

Renaming

We will need to support one final operation on formulas: renaming.

Renaming X to Y in formula φ, written $\varphi[X \to Y]$,
means replacing all occurrences of X by Y in φ.

We require that Y is not present in φ initially.

Example:

- $\varphi = (A \wedge (B \vee \neg C))$

$\rightsquigarrow$ $\varphi[A \to D] = (D \wedge (B \vee \neg C))$

How Hard Can That Be?

- For formulas, renaming is a simple (linear-time) operation.
- For a BDD B, it is equally simple ($O(\|B\|)$) when renaming between variables that are adjacent in the variable order.
- In general, it requires $O(\|B\|^2)$, using the equivalence $\varphi[X \to Y] \equiv \exists X(\varphi \wedge (X \leftrightarrow Y))$

C8.4 Symbolic Breadth-first Search

Planning Task State Variables vs. BDD Variables

Consider propositional planning task $\langle V, I, O, \gamma \rangle$ with states S.

In symbolic planning, we have two BDD variables v and v' for every state variable $v \in V$ of the planning task.

- use unprimed variables v to describe sets of states: $\{s \in S \mid \text{some property}\}$
- use combinations of unprimed and primed variables v, v' to describe sets of state pairs: $\{\langle s, s' \rangle \mid \text{some property}\}$

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Use *bdd-formula*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Use *bdd-singleton*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Use *bdd-intersection*, *bdd-emptyset*, *bdd-equals*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Use *bdd-union*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

Use *bdd-equals*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
    $goal\_states := models(\gamma)$
    $reached_0 := \{I\}$
    $i := 0$
    loop:
        if $reached_i \cap goal\_states \neq \emptyset$:
            return solution found
        $reached_{i+1} := reached_i \cup apply(reached_i, O)$
        if $reached_{i+1} = reached_i$:
            return no solution exists
        $i := i + 1$

How to do this?

The *apply* Function (1)

We need an operation that

- for a set of states *reached* (given as a BDD)
- and a set of operators O
- computes the set of states (as a BDD) that result from applying some operator $o \in O$ in some state $s \in \mathit{reached}$.

We have seen something similar already...

Translating Operators into Formulas

Definition (Operators in Propositional Logic)

Let o be an operator and V a set of state variables.

Define $\tau_V(o) := \mathit{pre}(o) \wedge \bigwedge_{v \in V}(\mathit{regr}(v, \mathit{eff}(o)) \leftrightarrow v')$.

States that o is applicable and describes how

- the new value of v, represented by v',
- must relate to the old state, described by variables V.

The *apply* Function (2)

- The formula $\tau_V(o)$ describes all transitions $s \xrightarrow{o} s'$
  - induced by a single operator o
  - in terms of variables V describing s
  - and variables V' describing s'.
- The formula $\bigvee_{o \in O} \tau_V(o)$ describes state transitions by any operator in O.
- We can translate this formula to a BDD (over variables $V \cup V'$) with *bdd-formula*.
- The resulting BDD is called the transition relation of the planning task, written as $T_V(O)$.

The *apply* Function (3)

Using the transition relation, we can compute $apply(reached, O)$ as follows:

The apply function

def apply($reached$, O):
    $B := T_V(O)$
    $B := bdd\text{-}intersection(B, reached)$
    for each $v \in V$:
        $B := bdd\text{-}forget(B, v)$
    for each $v \in V$:
        $B := bdd\text{-}rename(B, v', v)$
    return B

The *apply* Function (3)

Using the transition relation, we can compute $apply(reached, O)$ as follows:

The apply function

def apply($reached$, O):
    $B := T_V(O)$
    $B := bdd\text{-}intersection(B, reached)$
    for each $v \in V$:
        $B := bdd\text{-}forget(B, v)$
    for each $v \in V$:
        $B := bdd\text{-}rename(B, v', v)$
    return B

This describes the set of state pairs $\langle s, s' \rangle$ where s' is a successor of s in terms of variables $V \cup V'$.

The *apply* Function (3)

Using the transition relation, we can compute $\mathit{apply}(\mathit{reached}, O)$ as follows:

The apply function

def apply($\mathit{reached}$, O):
    $B := T_V(O)$
    $B := \mathit{bdd\text{-}intersection}(B, \mathit{reached})$
    for each $v \in V$:
        $B := \mathit{bdd\text{-}forget}(B, v)$
    for each $v \in V$:
        $B := \mathit{bdd\text{-}rename}(B, v', v)$
    return B

This describes the set of state pairs $\langle s, s' \rangle$ where s' is a successor of s and $s \in \mathit{reached}$ in terms of variables $V \cup V'$.

The *apply* Function (3)

Using the transition relation, we can compute $apply(reached, O)$ as follows:

The apply function

def apply($reached$, O):
    $B := T_V(O)$
    $B := bdd\text{-}intersection(B, reached)$
    for each $v \in V$:
        $B := bdd\text{-}forget(B, v)$
    for each $v \in V$:
        $B := bdd\text{-}rename(B, v', v)$
    return B

This describes the set of states s' which are successors of some state $s \in reached$ in terms of variables V'.

The *apply* Function (3)

Using the transition relation, we can compute $apply(reached, O)$ as follows:

The apply function

def apply($reached$, O):
    $B := T_V(O)$
    $B := bdd\text{-}intersection(B, reached)$
    for each $v \in V$:
        $B := bdd\text{-}forget(B, v)$
    for each $v \in V$:
        $B := bdd\text{-}rename(B, v', v)$
    return B

This describes the set of states s' which are successors of some state $s \in reached$ in terms of variables V.

The *apply* Function (3)

Using the transition relation, we can compute $apply(reached, O)$ as follows:

The apply function

def apply($reached$, O):
    $B := T_V(O)$
    $B := bdd\text{-}intersection(B, reached)$
    for each $v \in V$:
        $B := bdd\text{-}forget(B, v)$
    for each $v \in V$:
        $B := bdd\text{-}rename(B, v', v)$
    return B

Thus, *apply* indeed computes the set of successors of *reached* using operators O.

C8.5 Discussion

Discussion

- This completes the discussion of a (basic) symbolic search algorithm for classical planning.
- We ignored the aspect of solution extraction. This needs some extra work, but is not a major challenge.
- In practice, some steps can be performed slightly more efficiently, but these are comparatively minor details.

Variable Orders

For good performance, we need a good variable ordering.

- Variables that refer to the same state variable before and after operator application (v and v') should be neighbors in the transition relation BDD.

Extensions

Symbolic search can be extended to...

- regression and bidirectional search:
  this is very easy and often effective
- uniform-cost search:
  requires some work, but not too difficult in principle
- heuristic search:
  requires a heuristic representable as a BDD;
  has not really been shown to outperform blind symbolic search

Literature (1)


Randal E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers 35.8, pp. 677–691, 1986.
Reduced ordered BDDs.


Kenneth L. McMillan.
Symbolic Model Checking.
PhD Thesis, 1993.
Symbolic search with BDDs.

Literature (2)


Álvaro Torralba.
Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning.
PhD Thesis, 2015.
State of the art of symbolic search planning.


David Speck, Jendrik Seipp and Álvaro Torralba.
Symbolic Search for Cost-Optimal Planning with Expressive Model Extensions.
Journal of Artificial Intelligence Research 82, pp. 1349–1405, 2025.
More general classes of planning tasks.

C8.6 Summary

Summary

- Symbolic search operates on sets of states
  instead of individual states as in explicit-state search.
- State sets and transition relations can be represented
  as BDDs.
- Based on this, we can implement a blind breadth-first search
  in an efficient way.
- A good variable ordering is crucial for performance.

Planning and Optimization

D1. Delete Relaxation: Relaxed Planning Tasks

Malte Helmert and Gabriele Röger

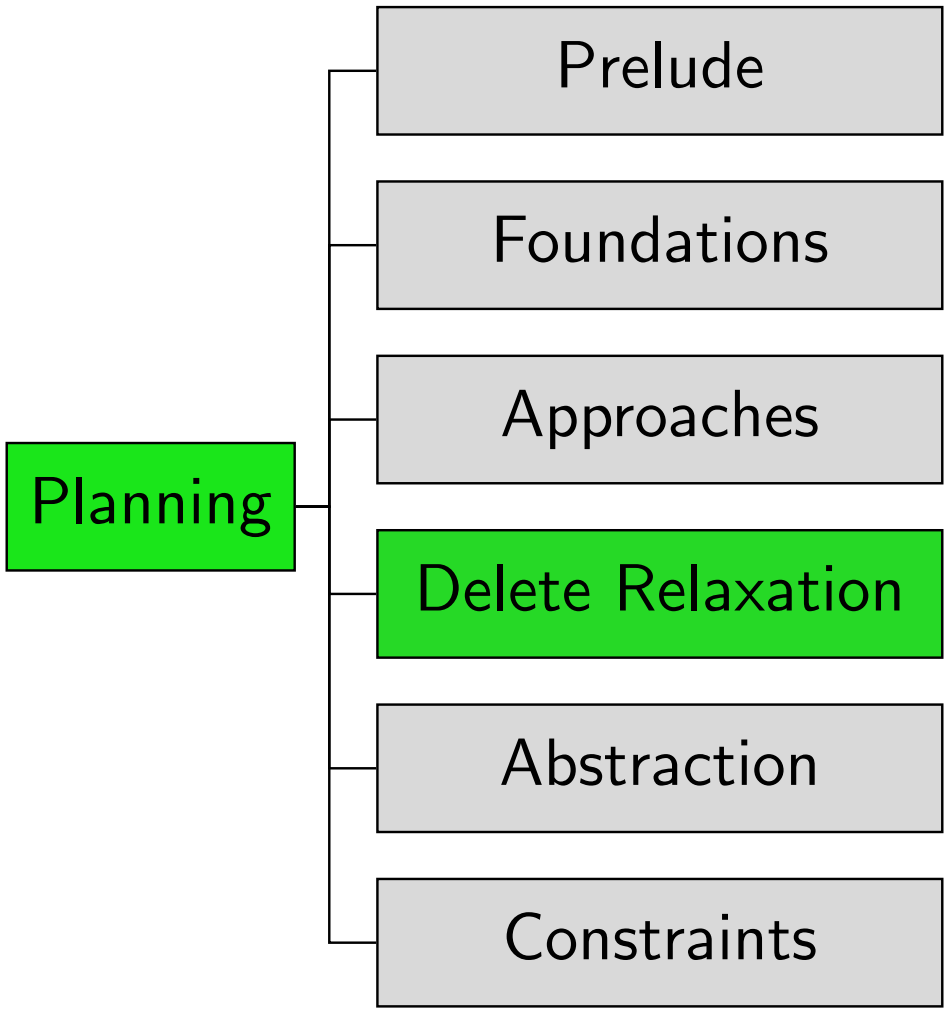
Universität Basel

October 20, 2025

Planning and Optimization

October 20, 2025 — D1. Delete Relaxation: Relaxed Planning Tasks

Content of the Course



D1.1 Heuristics

Planning as Heuristic Search

- Heuristic search is the most common approach to planning.
- ingredients: general search algorithm + heuristic
- heuristic estimates cost from a given state to a given goal
  - progression: from varying states s to fixed goal γ
  - regression: from fixed initial state I to varying subgoals φ
- Over the next weeks, we study the main ideas behind heuristics for planning tasks.

Reminder: Heuristics

Need to Catch Up?

- We assume familiarity with heuristics and their properties:
  - heuristic $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$
  - perfect heuristic h^*: $h^*(s)$ cost of optimal solution from s (∞ if unsolvable)
  - properties of heuristics h:
    - safe: $(h(s) = \infty \Rightarrow h^*(s) = \infty)$ for all states s
    - goal-aware: $h(s) = 0$ for all goal states s
    - admissible: $h(s) \leq h^*(s)$ for all states s
    - consistent: $h(s) \leq cost(o) + h(s')$ for all transitions $s \xrightarrow{o} s'$
  - connections between these properties
- If you are not familiar with these, we recommend Ch. B9–B10 of the Foundations of Artificial Intelligence course:
  `https://dmi.unibas.ch/en/studium/computer-science-informatik/lehrangebot-fs25/13548-lecture-foundations-of-artificial-intelligence/`

D1.2 Coming Up with Heuristics

A Simple Heuristic for Propositional Planning Tasks

STRIPS (Fikes & Nilsson, 1971) used the number of state variables that differ in current state s and a STRIPS goal $v_1 \wedge \cdots \wedge v_n$:

$$h(s) := |\{i \in \{1, \ldots, n\} \mid s \not\models v_i\}|.$$

Intuition: more satisfied goal atoms $\rightsquigarrow$ closer to the goal

$\rightsquigarrow$ STRIPS heuristic (a.k.a. goal-count heuristic)

Criticism of the STRIPS Heuristic

What is wrong with the STRIPS heuristic?

- quite uninformative:
  the range of heuristic values in a given task is small;
  typically, most successors have the same estimate
- very sensitive to reformulation:
  can easily transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states (how?)
- ignores almost all problem structure:
  heuristic value does not depend on the set of operators!

$\rightsquigarrow$ need a better, principled way of coming up with heuristics

Coming Up with Heuristics in a Principled Way

General Procedure for Obtaining a Heuristic

- Simplify the problem, for example by removing problem constraints.
- Solve the simplified problem (ideally optimally).
- Use the solution cost for the simplified problem as a heuristic for the real problem.

As heuristic values are computed for every generated search state, it is important that they can be computed efficiently.

Relaxing a Problem: Example

Example (Route Planning in a Road Network)

The road network is formalized as a weighted graph over points in the Euclidean plane. The weight of an edge is the road distance between two locations.

Example (Relaxation for Route Planning)

Use the Euclidean distance $\sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2}$
as a heuristic for the road distance between $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$
This is a lower bound on the road distance ($\rightsquigarrow$ admissible).

$\rightsquigarrow$ We drop the constraint of having to travel on roads.

Planning Heuristics: Main Concepts

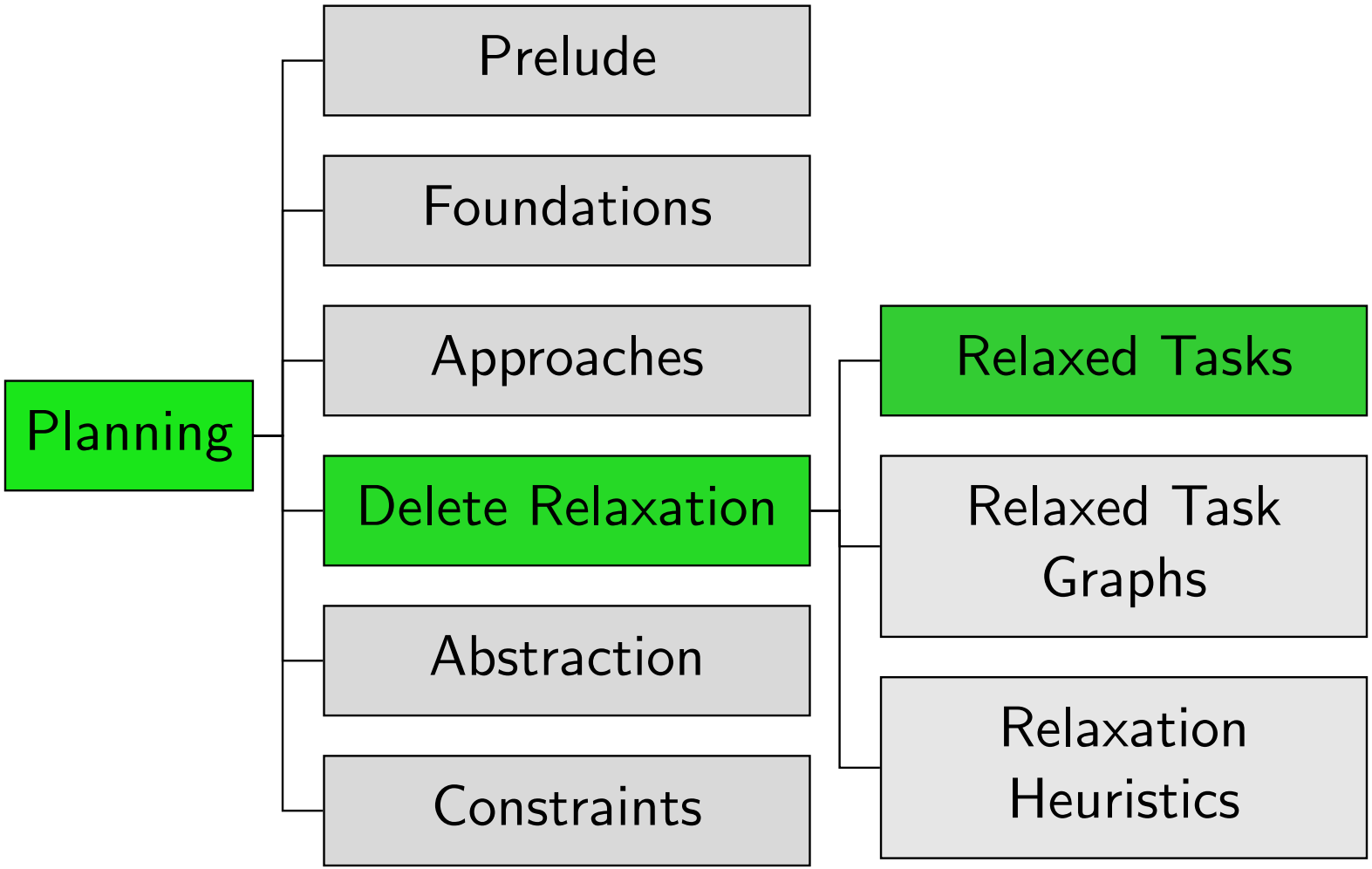
Major ideas for heuristics in the planning literature:

- delete relaxation $\rightsquigarrow$ Part D
- abstraction $\rightsquigarrow$ Part E
- critical paths $\rightsquigarrow$ not considered in this course
- landmarks $\rightsquigarrow$ Part F
- network flows $\rightsquigarrow$ Part F
- potential heuristics $\rightsquigarrow$ Part F

We will consider most of them in this course.

D1.3 Relaxed Planning Tasks

Content of the Course



Delete Relaxation: Idea

In positive normal form (Chapter B5, remember?), good and bad effects are easy to distinguish*:

- Effects that make state variables true are good (add effects).
- Effects that make state variables false are bad (delete effects).

Idea of delete relaxation heuristics: ignore all delete effects.

(*) with a small caveat regarding conditional effects

Delete-Relaxed Planning Tasks

Definition (Delete Relaxation of Operators)

The delete relaxation o^+ of an operator o in positive normal form is the operator obtained by replacing all negative effects $\neg a$ within $\mathit{eff}(o)$ by the do-nothing effect $\top$.

Definition (Delete Relaxation of Propositional Planning Tasks)

The delete relaxation Π^+ of a propositional planning task $\Pi = \langle V, I, O, \gamma \rangle$ in positive normal form is the planning task $\Pi^+ := \langle V, I, \{o^+ \mid o \in O\}, \gamma \rangle$.

Definition (Delete Relaxation of Operator Sequences)

The delete relaxation of an operator sequence $\pi = \langle o_1, \ldots, o_n \rangle$ is the operator sequence $\pi^+ := \langle o_1^+, \ldots, o_n^+ \rangle$.

Note: "delete" is often omitted: relaxation, relaxed

Relaxed Planning Tasks: Terminology

- Planning tasks in positive normal form without delete effects are called relaxed planning tasks.
- Plans for relaxed planning tasks are called relaxed plans.
- If Π is a planning task in positive normal form and π^+ is a plan for Π^+, then π^+ is called a relaxed plan for Π.

D1.4 Summary

Summary

- A general way to come up with heuristics:
  solve a simplified version of the real problem,
  for example by removing problem constraints.
- delete relaxation: given a task in positive normal form,
  discard all delete effects

Planning and Optimization

D2. Delete Relaxation: Properties of Relaxed Planning Tasks

Malte Helmert and Gabriele Röger

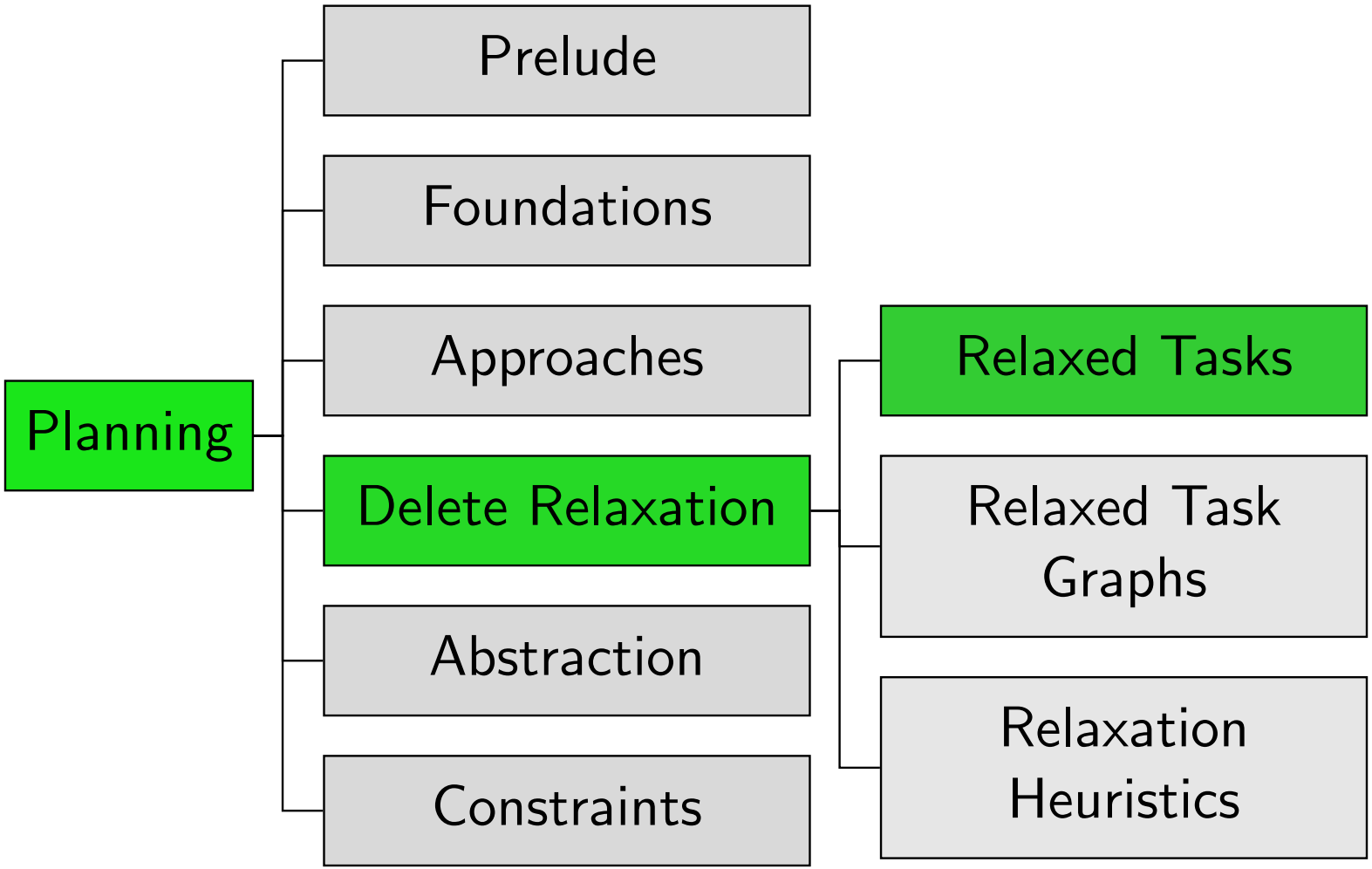
Universität Basel

October 20, 2025

Planning and Optimization

October 20, 2025 — D2. Delete Relaxation: Properties of Relaxed Planning Tasks

Content of the Course



D2.1 The Domination Lemma

On-Set and Dominating States

Definition (On-Set)

The on-set of an interpretation s is the set of propositional variables that are true in s, i.e., $on(s) = s^{-1}(\{\mathbf{T}\})$.

$\rightsquigarrow$ for states of propositional planning tasks:
states can be viewed as sets of (true) state variables

Definition (Dominate)

An interpretation s' dominates an interpretation s if $on(s) \subseteq on(s')$.

$\rightsquigarrow$ all state variables true in s are also true in s'

Domination Lemma (1)

Lemma (Domination)

Let s and s' be interpretations of a set of propositional variables V, and let χ be a propositional formula over V which does not contain negation symbols.

If $s \models \chi$ and s' dominates s, then $s' \models \chi$.

Proof.

Proof by induction over the structure of χ.

- Base case $\chi = \top$: then $s' \models \top$.
- Base case $\chi = \bot$: then $s \not\models \bot$.

...

Domination Lemma (2)

Proof (continued).

- Base case $\chi = v \in V$: if $s \models v$, then $v \in \mathit{on}(s)$.
  With $\mathit{on}(s) \subseteq \mathit{on}(s')$, we get $v \in \mathit{on}(s')$ and hence $s' \models v$.
- Inductive case $\chi = \chi_1 \wedge \chi_2$: by induction hypothesis, our claim holds for the proper subformulas χ_1 and χ_2 of χ.

$$
\begin{aligned}
s \models \chi &\implies s \models \chi_1 \wedge \chi_2 \\
&\implies s \models \chi_1 \text{ and } s \models \chi_2 \\
&\overset{\text{I.H. (twice)}}{\implies} s' \models \chi_1 \text{ and } s' \models \chi_2 \\
&\implies s' \models \chi_1 \wedge \chi_2 \\
&\implies s' \models \chi.
\end{aligned}
$$

- Inductive case $\chi = \chi_1 \vee \chi_2$: analogous

D2.2 The Relaxation Lemma

Add Sets and Delete Sets

Definition (Add Set and Delete Set for an Effect)

Consider a propositional planning task with state variables V.
Let e be an effect over V, and let s be a state over V.
The add set of e in s, written $\mathit{addset}(e, s)$,
and the delete set of e in s, written $\mathit{delset}(e, s)$,
are defined as the following sets of state variables:

$$\mathit{addset}(e, s) = \{v \in V \mid s \models \mathit{effcond}(v, e)\}$$
$$\mathit{delset}(e, s) = \{v \in V \mid s \models \mathit{effcond}(\neg v, e)\}$$

Note: For all states s and operators o applicable in s, we have $\mathit{on}(s[\![o]\!]) = (\mathit{on}(s) \setminus \mathit{delset}(\mathit{eff}(o), s)) \cup \mathit{addset}(\mathit{eff}(o), s)$.

Relaxation Lemma

For this and the following chapters on delete relaxation,
we assume implicitly that we are working with
propositional planning tasks in positive normal form.

Lemma (Relaxation)

Let s be a state, and let s' be a state that dominates s.

1. *If o is an operator applicable in s,*
   then o^+ is applicable in s' and $s'[\![o^+]\!]$ dominates $s[\![o]\!]$.
2. *If π is an operator sequence applicable in s,*
   then π^+ is applicable in s' and $s'[\![\pi^+]\!]$ dominates $s[\![\pi]\!]$.
3. *If additionally π leads to a goal state from state s,*
   then π^+ leads to a goal state from state s'.

Proof of Relaxation Lemma (1)

Proof.

Let V be the set of state variables.

Part 1: Because o is applicable in s, we have $s \models \mathit{pre}(o)$.

Because $\mathit{pre}(o)$ is negation-free and s' dominates s,
we get $s' \models \mathit{pre}(o)$ from the domination lemma.

Because $\mathit{pre}(o^+) = \mathit{pre}(o)$, this shows that o^+ is applicable in s'.

...

Proof of Relaxation Lemma (2)

Proof (continued).

To prove that $s'[\![o^+]\!]$ dominates $s[\![o]\!]$,
we first compare the relevant add sets:

$$\begin{aligned}
addset(eff(o), s) &= \{v \in V \mid s \models effcond(v, eff(o))\} \\
&= \{v \in V \mid s \models effcond(v, eff(o^+))\} \quad (1) \\
&\subseteq \{v \in V \mid s' \models effcond(v, eff(o^+))\} \quad (2) \\
&= addset(eff(o^+), s'),
\end{aligned}$$

where (1) uses $effcond(v, eff(o)) \equiv effcond(v, eff(o^+))$
and (2) uses the dominance lemma (note that effect conditions are negation-free for operators in positive normal form). ...

Proof of Relaxation Lemma (3)

Proof (continued).

We then get:

$$\begin{aligned} on(s[\![o]\!]) &= (on(s) \setminus delset(eff(o), s)) \cup addset(eff(o), s) \\ &\subseteq on(s) \cup addset(eff(o), s) \\ &\subseteq on(s') \cup addset(eff(o^+), s') \\ &= on(s'[\![o^+]\!]), \end{aligned}$$

and thus $s'[\![o^+]\!]$ dominates $s[\![o]\!]$.

This concludes the proof of Part 1. ...

Proof of Relaxation Lemma (4)

Proof (continued).

Part 2: by induction over $n = |\pi|$

Base case: $\pi = \langle\rangle$

The empty plan is trivially applicable in s', and $s'[\![\langle\rangle^+]\!] = s'$ dominates $s[\![\langle\rangle]\!] = s$ by prerequisite.

Inductive case: $\pi = \langle o_1, \ldots, o_{n+1}\rangle$

By the induction hypothesis, $\langle o_1^+, \ldots, o_n^+\rangle$ is applicable in s', and $t' = s'[\![\langle o_1^+, \ldots, o_n^+\rangle]\!]$ dominates $t = s[\![\langle o_1, \ldots, o_n\rangle]\!]$.
Also, o_{n+1} is applicable in t.

Using Part 1, o_{n+1}^+ is applicable in t' and $s'[\![\pi^+]\!] = t'[\![o_{n+1}^+]\!]$ dominates $s[\![\pi]\!] = t[\![o_{n+1}]\!]$.

This concludes the proof of Part 2. ...

Proof of Relaxation Lemma (5)

Proof (continued).

Part 3: Let γ be the goal formula.

From Part 2, we obtain that $t' = s'[\![\pi^+]\!]$ dominates $t = s[\![\pi]\!]$.
By prerequisite, t is a goal state and hence $t \models \gamma$.

Because the task is in positive normal form, γ is negation-free, and hence $t' \models \gamma$ because of the domination lemma.

Therefore, t' is a goal state. □

D2.3 Consequences

Consequences of the Relaxation Lemma

- The relaxation lemma is the main technical result that we will use to study delete relaxation.
- Next, we show two further properties of delete relaxation that will be useful for us.
- They are direct consequences of the relaxation lemma.

Consequences of the Relaxation Lemma (1)

Corollary (Relaxation Preserves Plans and Leads to Dominance)

Let π be an operator sequence that is applicable in state s.
Then π^+ is applicable in s and $s[\![\pi^+]\!]$ dominates $s[\![\pi]\!]$.
If π is a plan for Π, then π^+ is a plan for Π^+.

Proof.

Apply relaxation lemma with $s' = s$. □

- ⇝ Relaxations of plans are relaxed plans.
- ⇝ Delete relaxation is no harder to solve than original task.
- ⇝ Optimal relaxed plans are never more expensive than optimal plans for original tasks.

Consequences of the Relaxation Lemma (2)

Corollary (Relaxation Preserves Dominance)

Let s be a state, let s' be a state that dominates s,
and let π^+ be a relaxed operator sequence applicable in s.

Then π^+ is applicable in s' and $s'[\![\pi^+]\!]$ dominates $s[\![\pi^+]\!]$.

Proof.

Apply relaxation lemma with π^+ for π,
noting that $(\pi^+)^+ = \pi^+$. □

- ⇝ If there is a relaxed plan starting from state s,
  the same plan can be used starting from a dominating state s'.
- ⇝ Dominating states are always "better" in relaxed tasks.

D2.4 Monotonicity

Monotonicity of Relaxed Planning Tasks

Lemma (Monotonicity)

Let s be a state in which relaxed operator o^+ is applicable.
Then $s[\![o^+]\!]$ dominates s.

Proof.

Since relaxed operators only have positive effects,
we have $on(s) \subseteq on(s) \cup addset(eff(o^+), s) = on(s[\![o^+]\!])$. □

$\rightsquigarrow$ Together with our previous results, this means that making a transition in a relaxed planning task never hurts.

Finding Relaxed Plans

Using the theory we developed, we are now ready to study the problem of finding plans for relaxed planning tasks.

$\rightsquigarrow$ next chapter

D2.5 Summary

Summary

- With positive normal form, having more true variables is good.
- We can formalize this as dominance between states.
- It follows that delete relaxation is a simplification:
  it is never harder to solve a relaxed task than the original one.
- In delete-relaxed tasks, applying an operator always takes us
  to a dominating state and therefore never hurts.

Planning and Optimization

D3. Delete Relaxation: Finding Relaxed Plans

Malte Helmert and Gabriele Röger

Universität Basel

October 22, 2025

Planning and Optimization

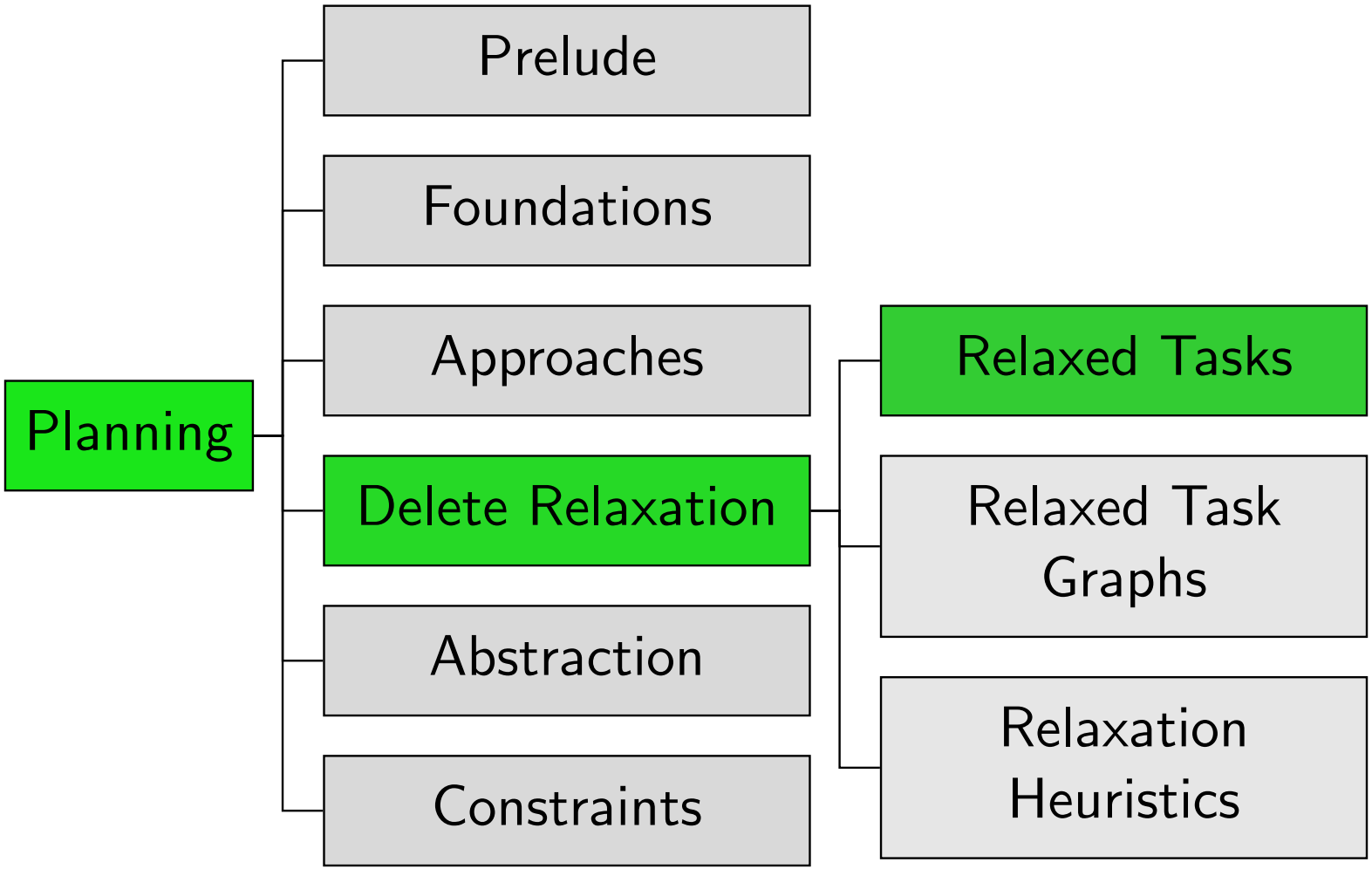
October 22, 2025 — D3. Delete Relaxation: Finding Relaxed Plans

D3.1 Greedy Algorithm

D3.2 Optimal Relaxed Plans

D3.3 Summary

Content of the Course



D3.1 Greedy Algorithm

The Story So Far

- A general way to come up with heuristics is to solve a simplified version of the real problem.
- delete relaxation: given a task in positive normal form, discard all delete effects
  - relaxation lemma: solutions for a state s also work for any dominating state s'
  - monotonicity lemma: $s[\![o]\!]$ dominates s

Greedy Algorithm for Relaxed Planning Tasks

The relaxation and monotonicity lemmas suggest the following algorithm for solving relaxed planning tasks:

Greedy Planning Algorithm for $\langle V, I, O^+, \gamma \rangle$

$s := I$
$\pi^+ := \langle\rangle$
loop forever:
    if $s \models \gamma$:
        return π^+
    else if there is an operator $o^+ \in O^+$ applicable in s
        with $s[\![o^+]\!] \neq s$:
        Append such an operator o^+ to π^+.
        $s := s[\![o^+]\!]$
    else:
        return unsolvable

Correctness of the Greedy Algorithm

The algorithm is sound:

- If it returns a plan, this is indeed a correct solution.
- If it returns "unsolvable", the task is indeed unsolvable
  - Upon termination, there clearly is no relaxed plan from s.
  - By iterated application of the monotonicity lemma, s dominates I.
  - By the relaxation lemma, there is no solution from I.

What about completeness (termination) and runtime?

- Each iteration of the loop adds at least one atom to $on(s)$.
- This guarantees termination after at most $|V|$ iterations.
- Thus, the algorithm can clearly be implemented to run in polynomial time.
  - A good implementation runs in $O(\|\Pi\|)$.

Using the Greedy Algorithm as a Heuristic

We can apply the greedy algorithm within heuristic search for a general (non-relaxed) planning task:

- When evaluating a state s in progression search, solve relaxation of planning task with initial state s.
- When evaluating a subgoal φ in regression search, solve relaxation of planning task with goal φ.
- Set $h(s)$ to the cost of the generated relaxed plan.
  - in general not well-defined: different choices of o^+ in the algorithm lead to different $h(s)$

Is this admissible/safe/goal-aware/consistent?

Properties of the Greedy Algorithm as a Heuristic

Is this an admissible heuristic?

- Yes if the relaxed plans are optimal
  (due to the plan preservation corollary).
- However, usually they are not, because the greedy algorithm can make poor choices of which operators to apply.

How hard is it to find optimal relaxed plans?

D3.2 Optimal Relaxed Plans

Optimal Relaxation Heuristic

Definition (h^+ heuristic)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a planning task in positive normal form with states S.

The optimal delete relaxation heuristic h^+ for Π
is the function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$
where $h(s)$ is the cost of an optimal relaxed plan for s,
i.e., of an optimal plan for $\Pi_s^+ = \langle V, s, O^+, \gamma \rangle$.

(can analogously define a heuristic for regression)

admissible/safe/goal-aware/consistent?

The Set Cover Problem

Can we compute h^+ efficiently?

This question is related to the following problem:

Problem (Set Cover)

Given: *a finite set U, a collection of subsets $C = \{C_1, \ldots, C_n\}$ with $C_i \subseteq U$ for all $i \in \{1, \ldots, n\}$, and a natural number K.*
Question: *Is there a set cover of size at most K, i.e., a subcollection $S = \{S_1, \ldots, S_m\} \subseteq C$ with $S_1 \cup \cdots \cup S_m = U$ and $m \leq K$?*

The following is a classical result from complexity theory:

Theorem (Karp 1972)

The set cover problem is NP-complete.

Complexity of Optimal Relaxed Planning (1)

Theorem (Complexity of Optimal Relaxed Planning)

The BCPlanEx *problem restricted to delete-relaxed planning tasks is NP-complete.*

Proof.

For membership in NP, guess a plan and verify.

It is sufficient to check plans of length at most $|V|$ where V is the set of state variables, so this can be done in nondeterministic polynomial time.

For hardness, we reduce from the set cover problem. ...

Complexity of Optimal Relaxed Planning (2)

Proof (continued).

Given a set cover instance $\langle U, C, K \rangle$, we generate the following relaxed planning task $\Pi^+ = \langle V, I, O^+, \gamma \rangle$:

- $V = U$
- $I = \{v \mapsto \mathbf{F} \mid v \in V\}$
- $O^+ = \{\langle \top, \bigwedge_{v \in C_i} v, 1 \rangle \mid C_i \in C\}$
- $\gamma = \bigwedge_{v \in U} v$

If S is a set cover, the corresponding operators form a plan. Conversely, each plan induces a set cover by taking the subsets corresponding to the operators. There exists a plan of cost at most K iff there exists a set cover of size K.

Moreover, Π^+ can be generated from the set cover instance in polynomial time, so this is a polynomial reduction. □

D3.3 Summary

Summary

- Because of their monotonicity property, delete-relaxed tasks can be solved in polynomial time by a greedy algorithm.
- However, the solution quality of this algorithm is poor.
- For an informative heuristic, we would ideally want to find optimal relaxed plans.
- The solution cost of an optimal relaxed plan is the estimate of the h^+ heuristic.
- However, the bounded-cost plan existence problem for relaxed planning tasks is NP-complete.

Planning and Optimization

D4. Delete Relaxation: AND/OR Graphs

Malte Helmert and Gabriele Röger

Universität Basel

October 22, 2025

Planning and Optimization

October 22, 2025 — D4. Delete Relaxation: AND/OR Graphs

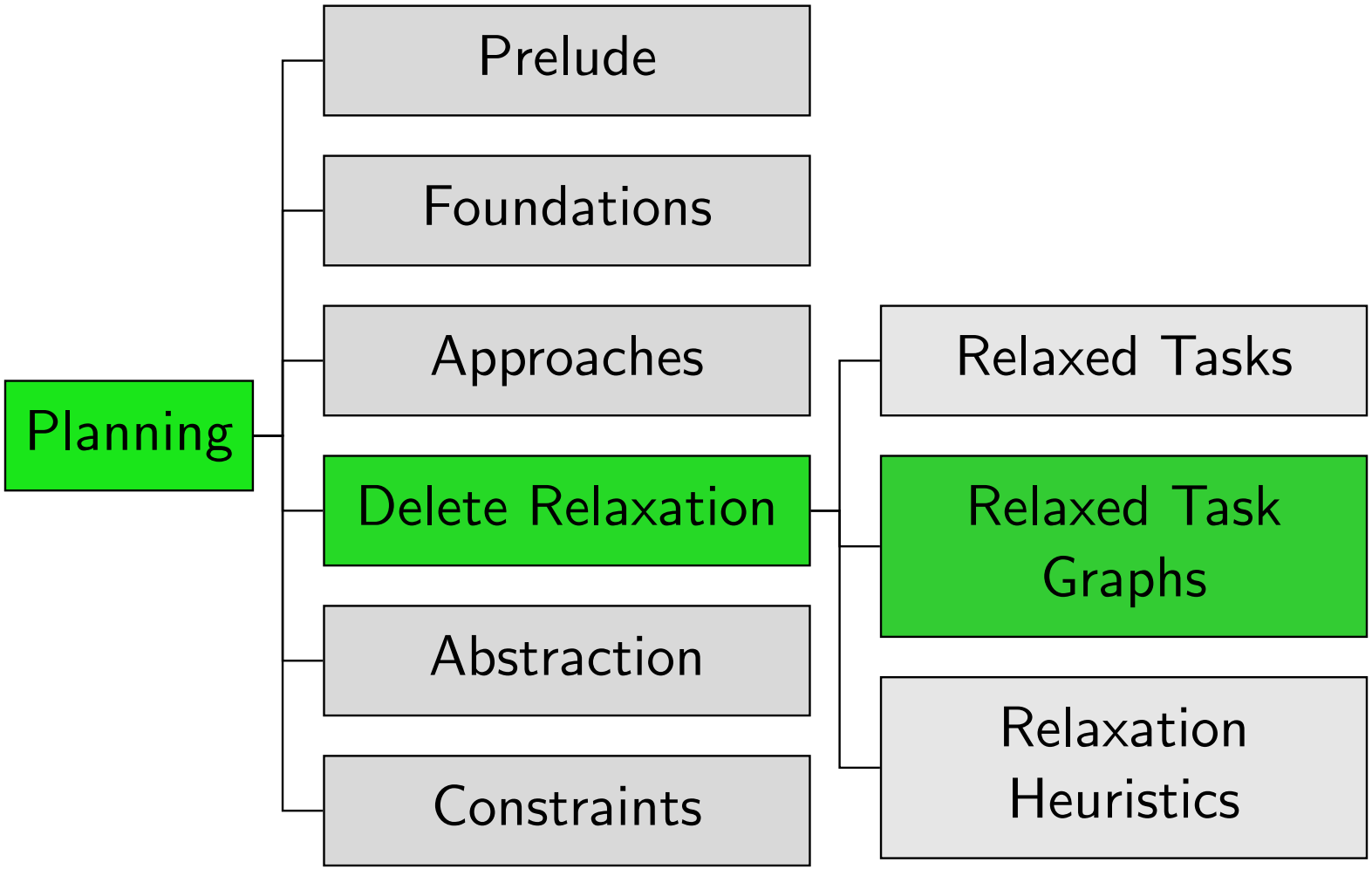
D4.1 AND/OR Graphs

D4.2 Forced Nodes

D4.3 Most/Least Conservative Valuations

D4.4 Summary

Content of the Course



D4.1 AND/OR Graphs

Using Relaxations in Practice

How can we use relaxations for heuristic planning in practice?

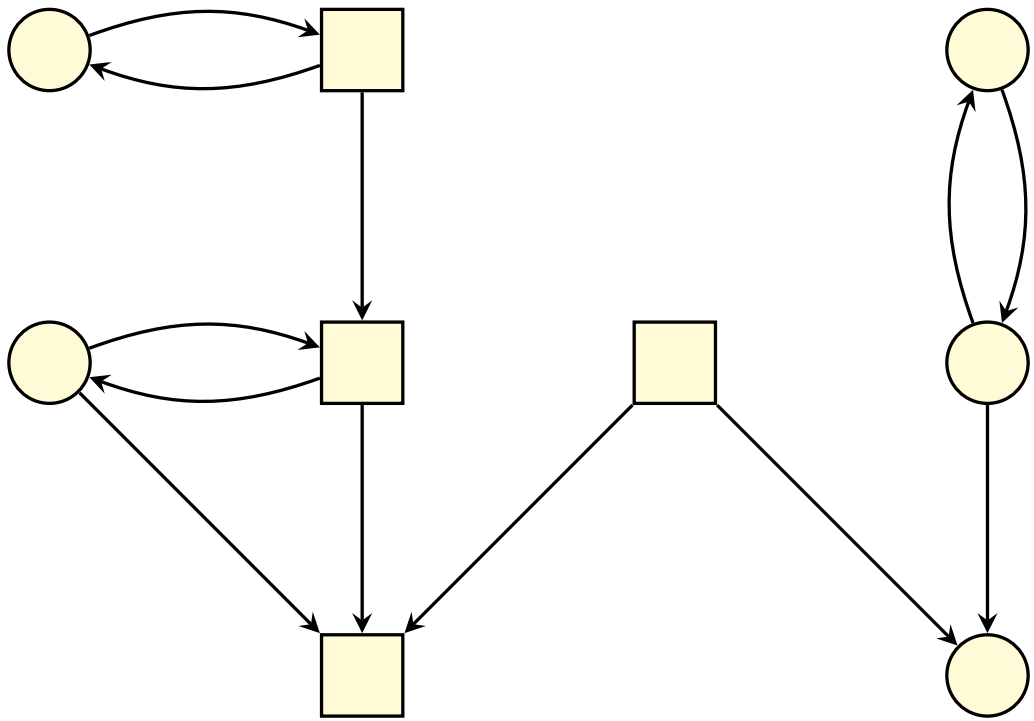
Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution costs as estimates, even though optimal relaxed planning is NP-hard.
  $\rightsquigarrow$ h^+ heuristic
- Do not actually solve the relaxed planning task, but compute an approximation of its solution cost.
  $\rightsquigarrow$ $h^{\max}$ heuristic, h^{add} heuristic, $h^{\text{LM-cut}}$ heuristic
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable".
  $\rightsquigarrow$ h^{FF} heuristic

AND/OR Graphs: Motivation

- Most relaxation heuristics we will consider can be understood in terms of computations on graphical structures called AND/OR graphs.
- We now introduce AND/OR graphs and study some of their major properties.
- In the next chapter, we will relate AND/OR graphs to relaxed planning tasks.

AND/OR Graph Example



AND/OR Graphs

Definition (AND/OR Graph)

An AND/OR graph $\langle N, A, type \rangle$ is a directed graph $\langle N, A \rangle$ with a node label function $type : N \to \{\wedge, \vee\}$ partitioning nodes into

- AND nodes ($type(v) = \wedge$) and
- OR nodes ($type(v) = \vee$).

We write $succ(n)$ for the successors of node $n \in N$, i.e., $succ(n) = \{n' \in N \mid \langle n, n' \rangle \in A\}$.

Note: We draw AND nodes as squares and OR nodes as circles.

AND/OR Graph Valuations

Definition (Consistent Valuations of AND/OR Graphs)

Let G be an AND/OR graph with nodes N.

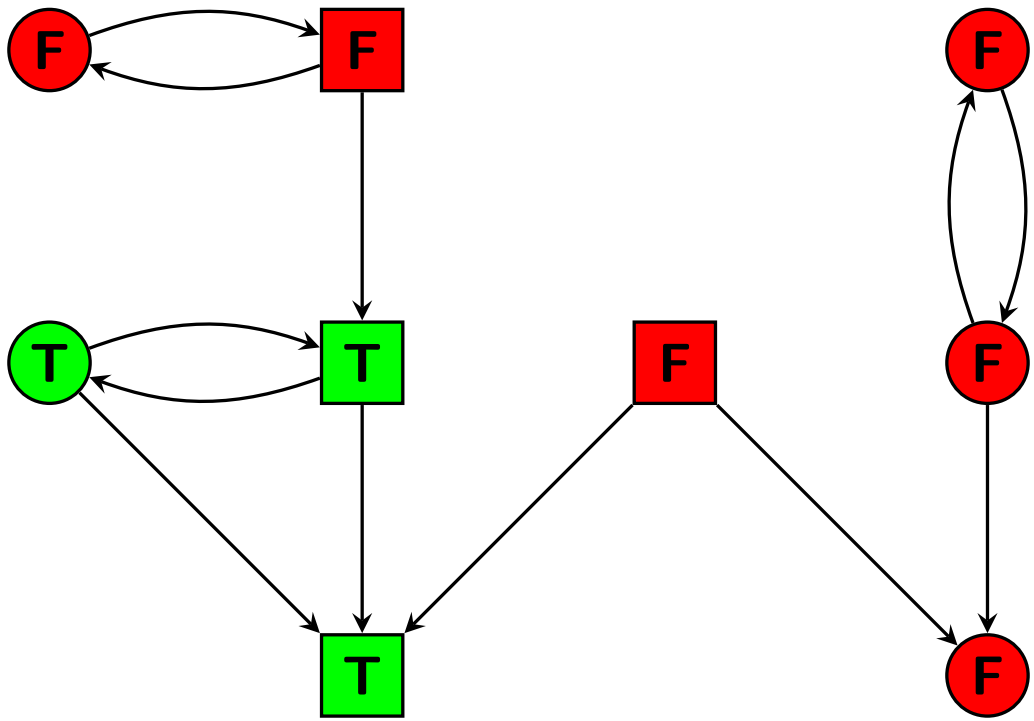
A valuation or truth assignment of G is an interpretation $\alpha : N \to \{\mathbf{T}, \mathbf{F}\}$, treating the nodes as propositional variables.

We say that α is consistent if

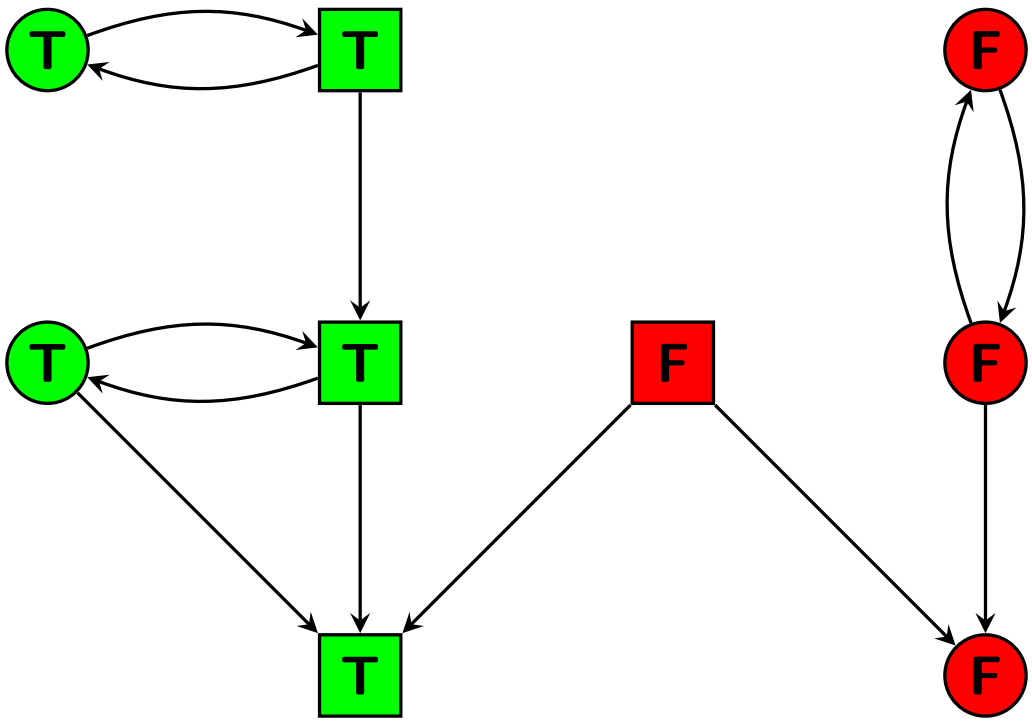
- for all AND nodes $n \in N$: $\alpha \models n$ iff $\alpha \models \bigwedge_{n' \in succ(n)} n'$.
- for all OR nodes $n \in N$: $\alpha \models n$ iff $\alpha \models \bigvee_{n' \in succ(n)} n'$.

Note that $\bigwedge_{n' \in \emptyset} n' = \top$ and $\bigvee_{n' \in \emptyset} n' = \bot$.

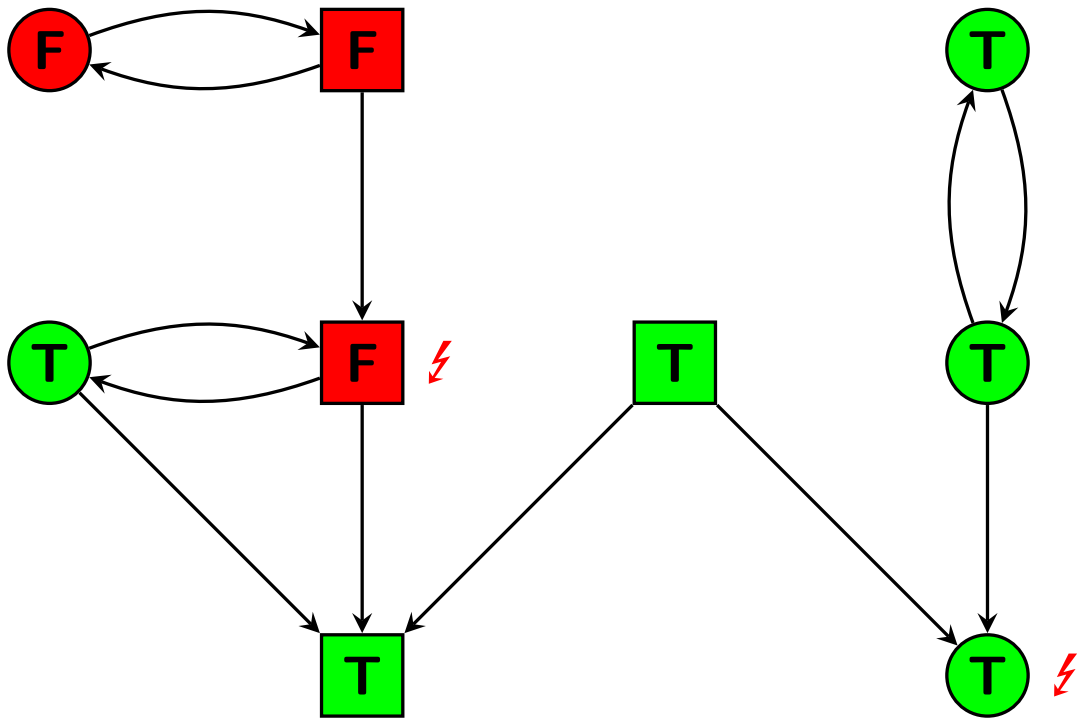
Example: A Consistent Valuation



Example: Another Consistent Valuation



Example: An Inconsistent Valuation



How Do We Find Consistent Valuations?

If we want to use valuations of AND/OR graphs algorithmically, a number of questions arise:

- Do consistent valuations exist for every AND/OR graph?
- Are they unique?
- If not, how are different consistent valuations related?
- Can consistent valuations be computed efficiently?

Our example shows that the answer to the second question is “no”.
In the rest of this chapter, we address the remaining questions.

D4.2 Forced Nodes

Forced Nodes

Definition (Forced True/False Nodes)

Let G be an AND/OR graph.

A node n of G is called forced true
if $\alpha(n) = \mathbf{T}$ for all consistent valuations α of G.

A node n of G is called forced false
if $\alpha(n) = \mathbf{F}$ for all consistent valuations α of G.

How can we efficiently determine that nodes are forced true/false?

$\rightsquigarrow$ We begin by looking at some simple rules.

Rules for Forced True Nodes

Proposition (Rules for Forced True Nodes)

Let n be a node in an AND/OR graph.

Rule **T**-($\wedge$)*: If n is an AND node and all of its successors are forced true, then n is forced true.*

Rule **T**-($\vee$)*: If n is an OR node and at least one of its successors is forced true, then n is forced true.*

Rules for Forced False Nodes

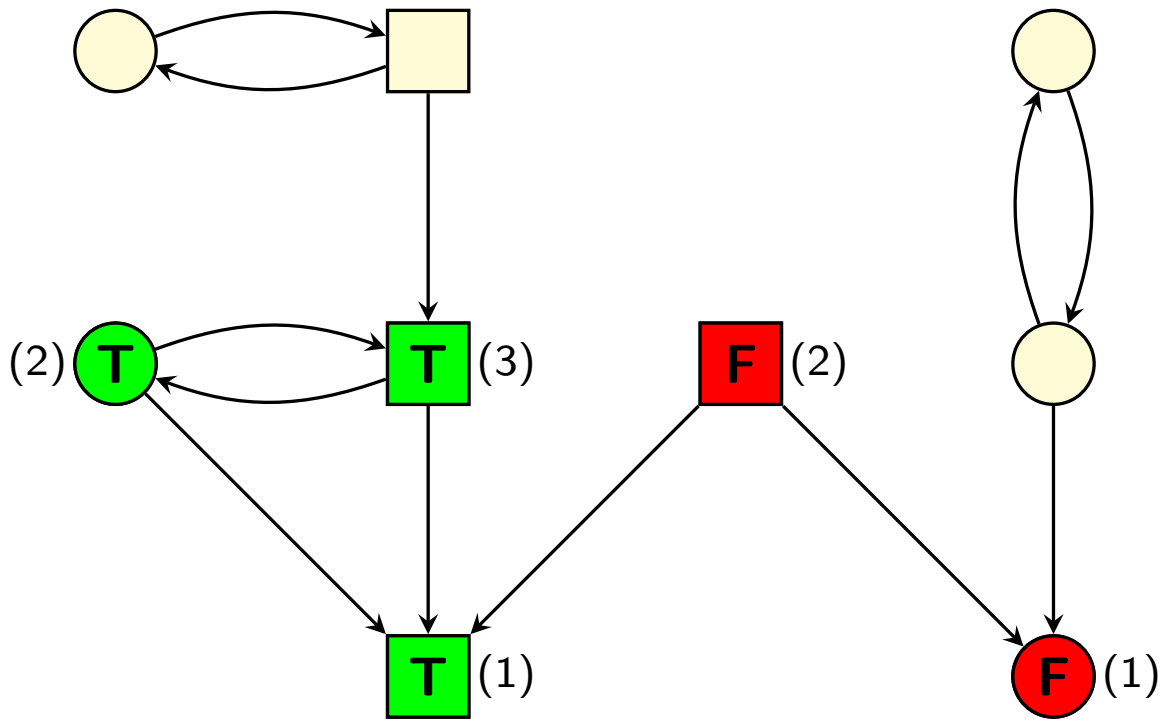
Proposition (Rules for Forced False Nodes)

Let n be a node in an AND/OR graph.

Rule **F**-($\wedge$)*: If n is an AND node and at least one of its successors is forced false, then n is forced false.*

Rule **F**-($\vee$)*: If n is an OR node and all of its successors are forced false, then n is forced false.*

Example: Applying the Rules for Forced Nodes



Completeness of Rules for Forced Nodes

Theorem

If n is a node in an AND/OR graph that is forced true, then this can be derived by a sequence of applications of Rule $\mathbf{T}$*-($\wedge$) and Rule* $\mathbf{T}$*-($\vee$).*

Theorem

If n is a node in an AND/OR graph that is forced false, then this can be derived by a sequence of applications of Rule $\mathbf{F}$*-($\wedge$) and Rule* $\mathbf{F}$*-($\vee$).*

We prove the result for forced true nodes.
The result for forced false nodes can be proved analogously.

Completeness of Rules for Forced Nodes: Proof (1)

Proof.

- Let α be a valuation where $\alpha(n) = \mathbf{T}$ iff there exists a sequence ρ_n of applications of Rules $\mathbf{T}$-$(\wedge)$ and Rule $\mathbf{T}$-$(\vee)$ that derives that n is forced true.
- Because the rules are monotonic, there exists a sequence ρ of rule applications that derives that n is forced true for all $n \in on(\alpha)$. (Just concatenate all ρ_n to form ρ.)
- By the correctness of the rules, we know that all nodes reached by ρ are forced true. It remains to show that none of the nodes not reached by ρ is forced true.
- We prove this by showing that α is consistent, and hence no nodes with $\alpha(n) = \mathbf{F}$ can be forced true.

...

Completeness of Rules for Forced Nodes: Proof (2)

Proof (continued).

Case 1: nodes n with $\alpha(n) = \mathbf{T}$

- In this case, ρ must have reached n in one of the derivation steps. Consider this derivation step.
- If n is an AND node, ρ must have reached all successors of n in previous steps, and hence $\alpha(n') = \mathbf{T}$ for all successors n'.
- If n is an OR node, ρ must have reached at least one successor of n in a previous step, and hence $\alpha(n') = \mathbf{T}$ for at least one successor n'.
- In both cases, α is consistent for node n.

...

Completeness of Rules for Forced Nodes: Proof (3)

Proof (continued).

Case 2: nodes n with $\alpha(n) = \mathbf{F}$

- In this case, by definition of α no sequence of derivation steps reaches n. In particular, ρ does not reach n.
- If n is an AND node, there must exist some $n' \in succ(n)$ which ρ does not reach.
  Otherwise, ρ could be extended using Rule **T**-($\wedge$) to reach n.
  Hence, $\alpha(n') = \mathbf{F}$ for some $n' \in succ(n)$.
- If n is an OR node, there cannot exist any $n' \in succ(n)$ which ρ reaches.
  Otherwise, ρ could be extended using Rule **T**-($\vee$) to reach n.
  Hence, $\alpha(n') = \mathbf{F}$ for all $n' \in succ(n)$.
- In both cases, α is consistent for node n.

Remarks on Forced Nodes

Notes:

- The theorem shows that we can compute all forced nodes by applying the rules repeatedly until a fixed point is reached.
- In particular, this also shows that the order of rule application does not matter: we always end up with the same result.
- In an efficient implementation, the sets of forced nodes can be computed in linear time in the size of the AND/OR graph.
- The proof of the theorem also shows that every AND/OR graph has a consistent valuation, as we explicitly construct one in the proof.

D4.3 Most/Least Conservative Valuations

Most and Least Conservative Valuation

Definition (Most and Least Conservative Valuation)

Let G be an AND/OR graph with nodes N.

The most conservative valuation $\alpha^G_{\text{mcv}} : N \to \{\mathbf{T}, \mathbf{F}\}$ and the least conservative valuation $\alpha^G_{\text{lcv}} : N \to \{\mathbf{T}, \mathbf{F}\}$ of G are defined as:

$$\alpha^G_{\text{mcv}}(n) = \begin{cases} \mathbf{T} & \text{if } n \text{ is forced true} \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$\alpha^G_{\text{lcv}}(n) = \begin{cases} \mathbf{F} & \text{if } n \text{ is forced false} \\ \mathbf{T} & \text{otherwise} \end{cases}$$

Note: α^G_{mcv} is the valuation constructed in the previous proof.

Properties of Most/Least Conservative Valuations

Theorem (Properties of Most/Least Conservative Valuations)

Let G be an AND/OR graph. Then:

1. α^G_{mcv} *is consistent.*
2. α^G_{lcv} *is consistent.*
3. *For all consistent valuations α of G,* $on(\alpha^G_{\text{mcv}}) \subseteq on(\alpha) \subseteq on(\alpha^G_{\text{lcv}})$.

Properties of MCV/LCV: Proof

Proof.

Part 1. was shown in the preceding proof. We showed that the valuation α considered in this proof is consistent and satisfies $\alpha(n) = \mathbf{T}$ iff n is forced true, which implies $\alpha = \alpha^G_{\text{mcv}}$.

The proof of Part 2. is analogous, using the rules for forced false nodes instead of forced true nodes.

Part 3 follows directly from the definitions of forced nodes, α^G_{mcv} and α^G_{lcv}. □

Properties of MCV/LCV: Consequences

This theorem answers our remaining questions about the existence, uniqueness, structure and computation of consistent valuations:

- Consistent valuations always exist
  and can be efficiently computed.
- All consistent valuations lie between
  the most and least conservative one.
- There is a unique consistent valuation iff $\alpha^G_{\mathsf{mcv}} = \alpha^G_{\mathsf{lcv}}$,
  or equivalently iff each node is forced true or forced false.

D4.4 Summary

Summary

- AND/OR graphs are directed graphs with AND nodes and OR nodes.
- We can assign truth values to AND/OR graph nodes.
- Such valuations are called consistent if they match the intuitive meaning of "AND" and "OR".
- Consistent valuations always exist.
- Consistent valuations can be computed efficiently.
- All consistent valuations fall between two extremes:
  - the most conservative valuation, where only nodes that are forced to be true are true
  - the least conservative valuation, where all nodes that are not forced to be false are true

Planning and Optimization

D5. Delete Relaxation: Relaxed Task Graphs

Malte Helmert and Gabriele Röger

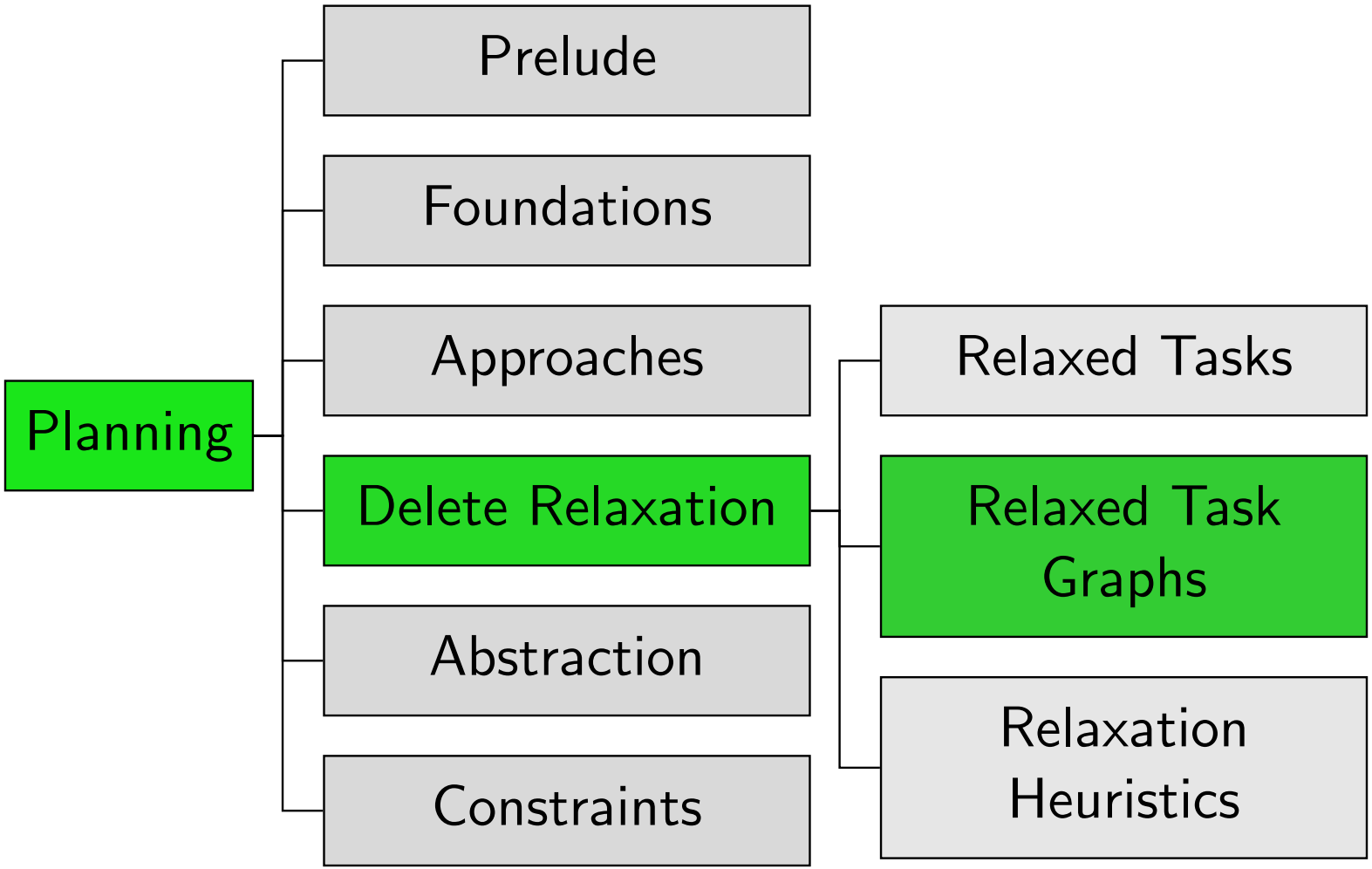
Universität Basel

October 27, 2025

Planning and Optimization

October 27, 2025 — D5. Delete Relaxation: Relaxed Task Graphs

Content of the Course



D5.1 Relaxed Task Graphs

Relaxed Task Graphs

Let Π^+ be a relaxed planning task.

The relaxed task graph of Π^+, in symbols $RTG(\Pi^+)$,
is an AND/OR graph that encodes

- which state variables can become true
  in an applicable operator sequence for Π^+,
- which operators of Π^+ can be included
  in an applicable operator sequence for Π^+,
- if the goal of Π^+ can be reached,
- and how these things can be achieved.

We present its definition in stages.

Note: Throughout this chapter, we assume flat operators.

Running Example

As a running example, consider the relaxed planning task $\langle V, I, \{o_1, o_2, o_3, o_4\}, \gamma \rangle$ with

$$
\begin{aligned}
V &= \{a, b, c, d, e, f, g, h\} \\
I &= \{a \mapsto \mathbf{T}, b \mapsto \mathbf{T}, c \mapsto \mathbf{F}, d \mapsto \mathbf{T}, \\
&\qquad e \mapsto \mathbf{F}, f \mapsto \mathbf{F}, g \mapsto \mathbf{F}, h \mapsto \mathbf{F}\} \\
o_1 &= \langle c \vee (a \wedge b), c \wedge ((c \wedge d) \rhd e), 1 \rangle \\
o_2 &= \langle \top, f, 2 \rangle \\
o_3 &= \langle f, g, 1 \rangle \\
o_4 &= \langle f, h, 1 \rangle \\
\gamma &= e \wedge (g \wedge h)
\end{aligned}
$$

D5.2 Construction

Components of Relaxed Task Graphs

A relaxed task graph has four kinds of components:

- Variable nodes represent the state variables.
- The initial node represent the initial state.
- Operator subgraphs represent the preconditions and effects of operators.
- The goal subgraph represents the goal.

The idea is to construct the graph in such a way that all nodes representing reachable aspects of the task are forced true.

Variable Nodes

Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task.

- For each $v \in V$, $RTG(\Pi^+)$ contains an OR node n_v.
  These nodes are called variable nodes.

Variable Nodes: Example

$$V = \{a, b, c, d, e, f, g, h\}$$

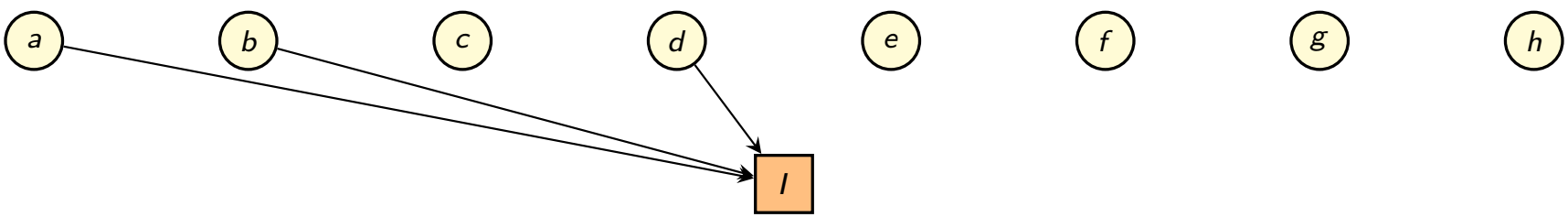
Initial Node

Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task.

- $RTG(\Pi^+)$ contains an AND node n_I.
  This node is called the initial node.
- For all $v \in V$ with $I(v) = \mathbf{T}$, $RTG(\Pi^+)$ has an arc from n_v to n_I. These arcs are called initial state arcs.
- The initial node has no successor nodes.

Initial Node and Initial State Arcs: Example

$$I = \{a \mapsto \mathbf{T}, b \mapsto \mathbf{T}, c \mapsto \mathbf{F}, d \mapsto \mathbf{T}, e \mapsto \mathbf{F}, f \mapsto \mathbf{F}, g \mapsto \mathbf{F}, h \mapsto \mathbf{F}\}$$



Operator Subgraphs

Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task.

For each operator $o^+ \in O^+$, $RTG(\Pi^+)$ contains an operator subgraph with the following parts:

- for each formula φ that occurs as a subformula of the precondition or of some effect condition of o^+, a formula node n_φ (details follow)
- for each conditional effect $(\chi \rhd v)$ that occurs in the effect of o^+, an effect node $n^\chi_{o^+}$ (details follow); unconditional effects are treated as $(\top \rhd v)$

Formula Nodes

Formula nodes n_φ are defined as follows:

- If $\varphi = v$ for some state variable v, n_φ is the variable node n_v (so no new node is introduced).
- If $\varphi = \top$, n_φ is an AND node without outgoing arcs.
- If $\varphi = \bot$, n_φ is an OR node without outgoing arcs.
- If $\varphi = (\varphi_1 \wedge \varphi_2)$, n_φ is an AND node with outgoing arcs to n_{φ_1} and n_{φ_2}.
- If $\varphi = (\varphi_1 \vee \varphi_2)$, n_φ is an OR node with outgoing arcs to n_{φ_1} and n_{φ_2}.

Note: identically named nodes are identical,
so if the same formula occurs multiple times in the task,
the same node is reused.

Effect Nodes

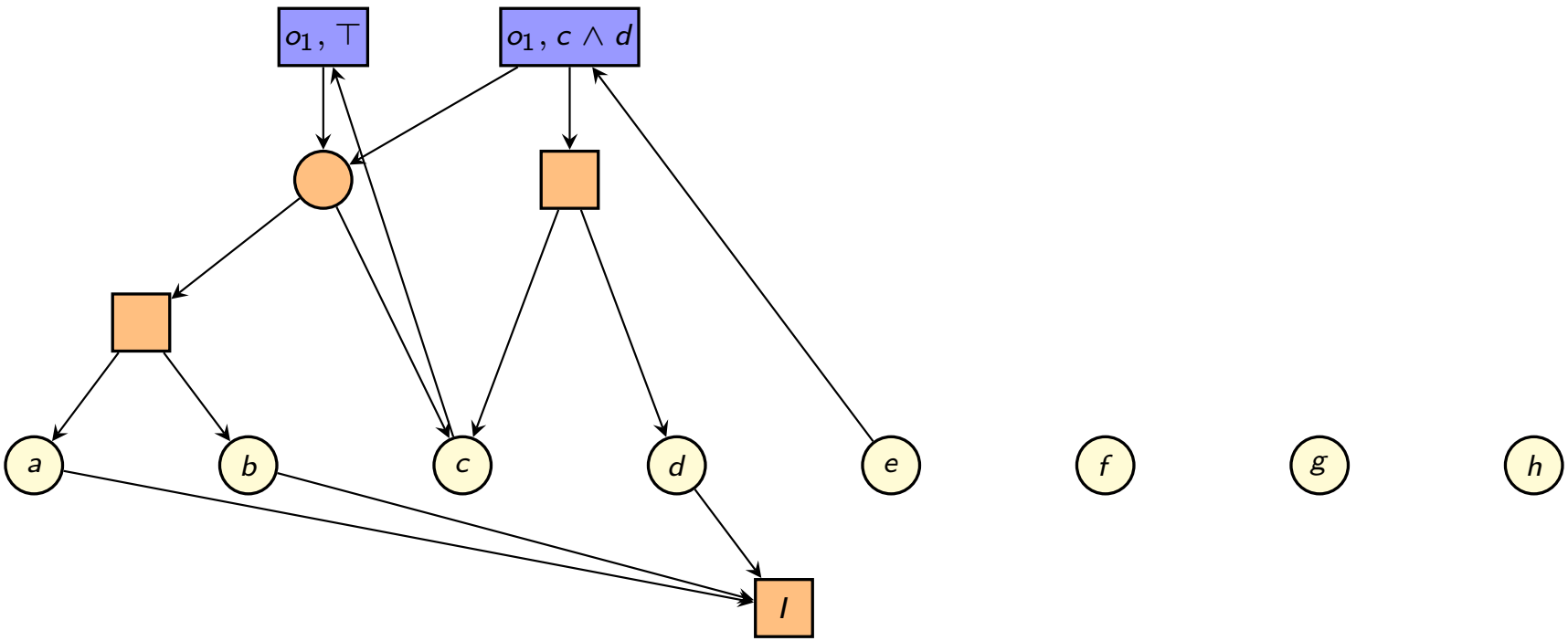
Effect nodes $n^{\chi}_{o^+}$ are defined as follows:

- $n^{\chi}_{o^+}$ is an AND node
- It has an outgoing arc to the formula nodes $n_{pre(o^+)}$ (precondition arcs) and n_{χ} (effect condition arcs).
- Exception: if $\chi = \top$, there is no effect condition arc. (This makes our pictures cleaner.)
- For every conditional effect $(\chi \rhd v)$ in the operator, there is an arc from variable node n_v to $n^{\chi}_{o^+}$ (effect arcs).

Note: identically named nodes are identical,
so if the same effect condition occurs multiple times
in the same operator, this only induces one node.

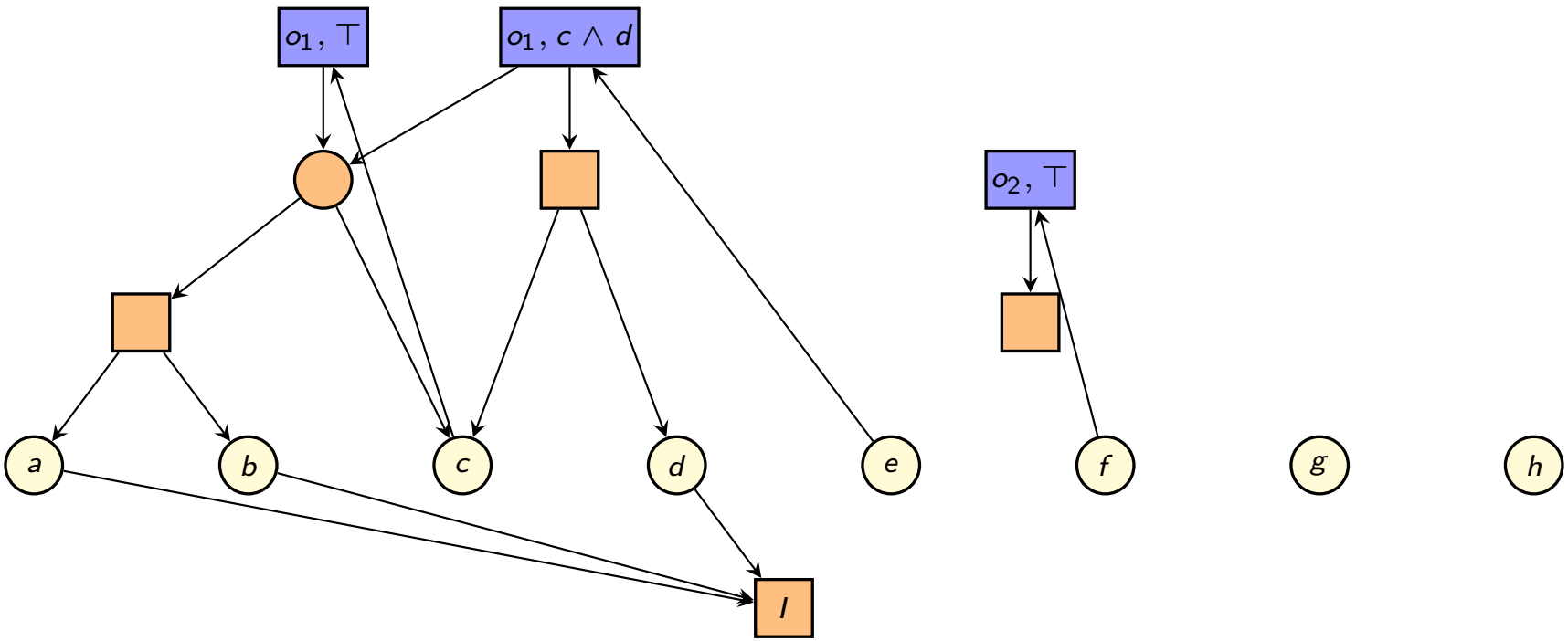
Operator Subgraphs: Example

$$o_1 = \langle c \vee (a \wedge b), c \wedge ((c \wedge d) \rhd e), 1 \rangle$$



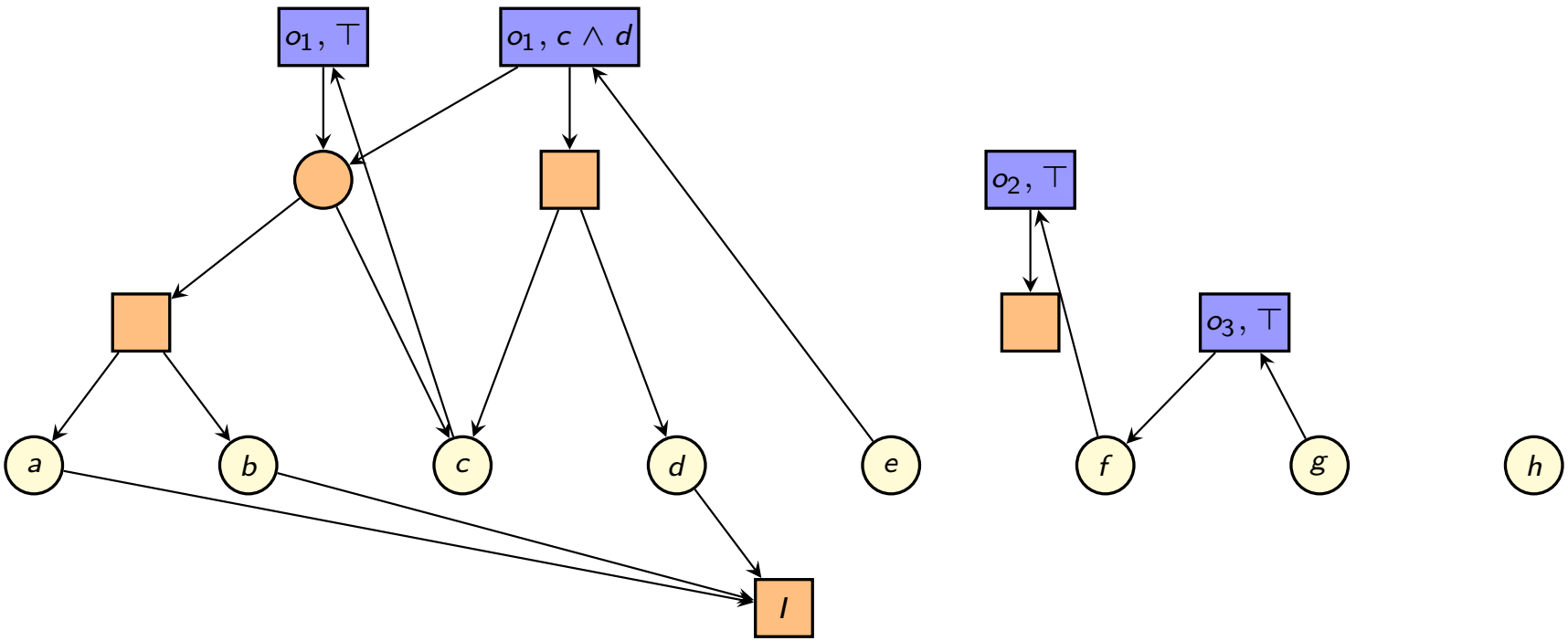
Operator Subgraphs: Example

$$o_2 = \langle \top, f, 2 \rangle$$



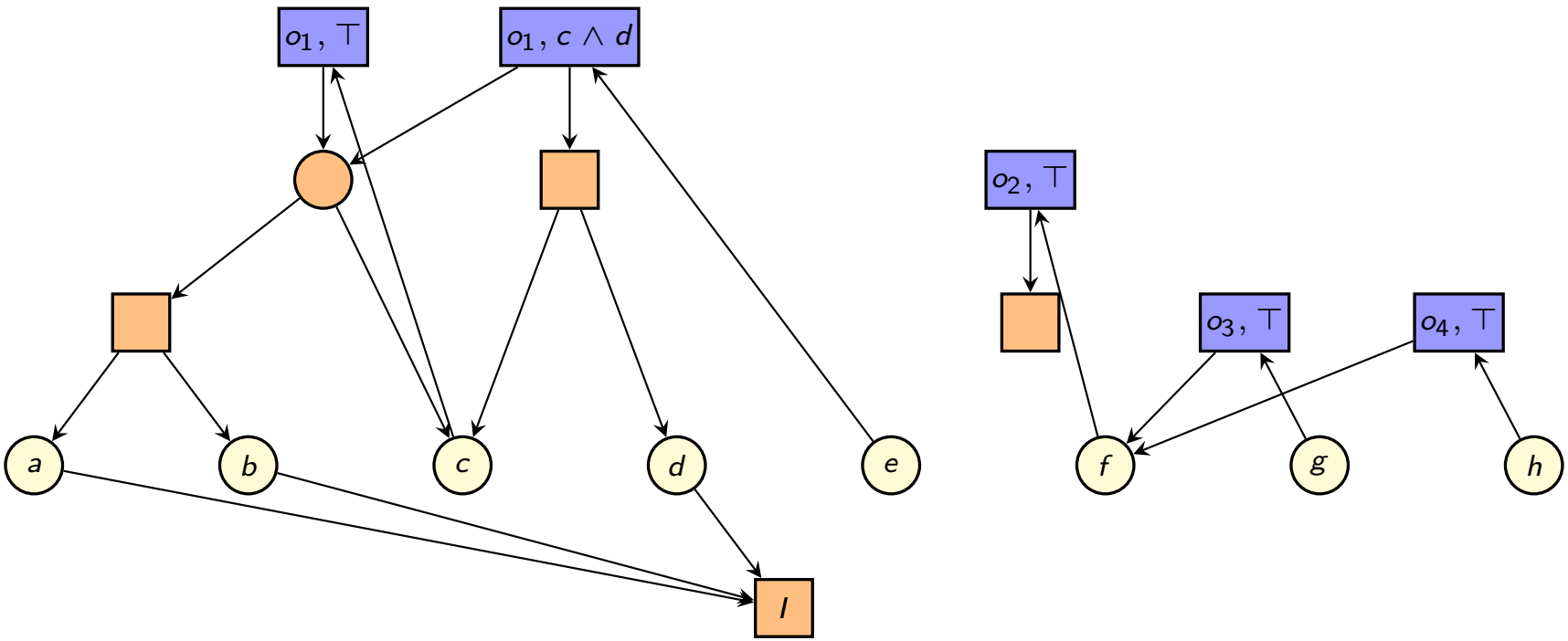
Operator Subgraphs: Example

$$o_3 = \langle f, g, 1 \rangle$$



Operator Subgraphs: Example

$$o_4 = \langle f, h, 1 \rangle$$



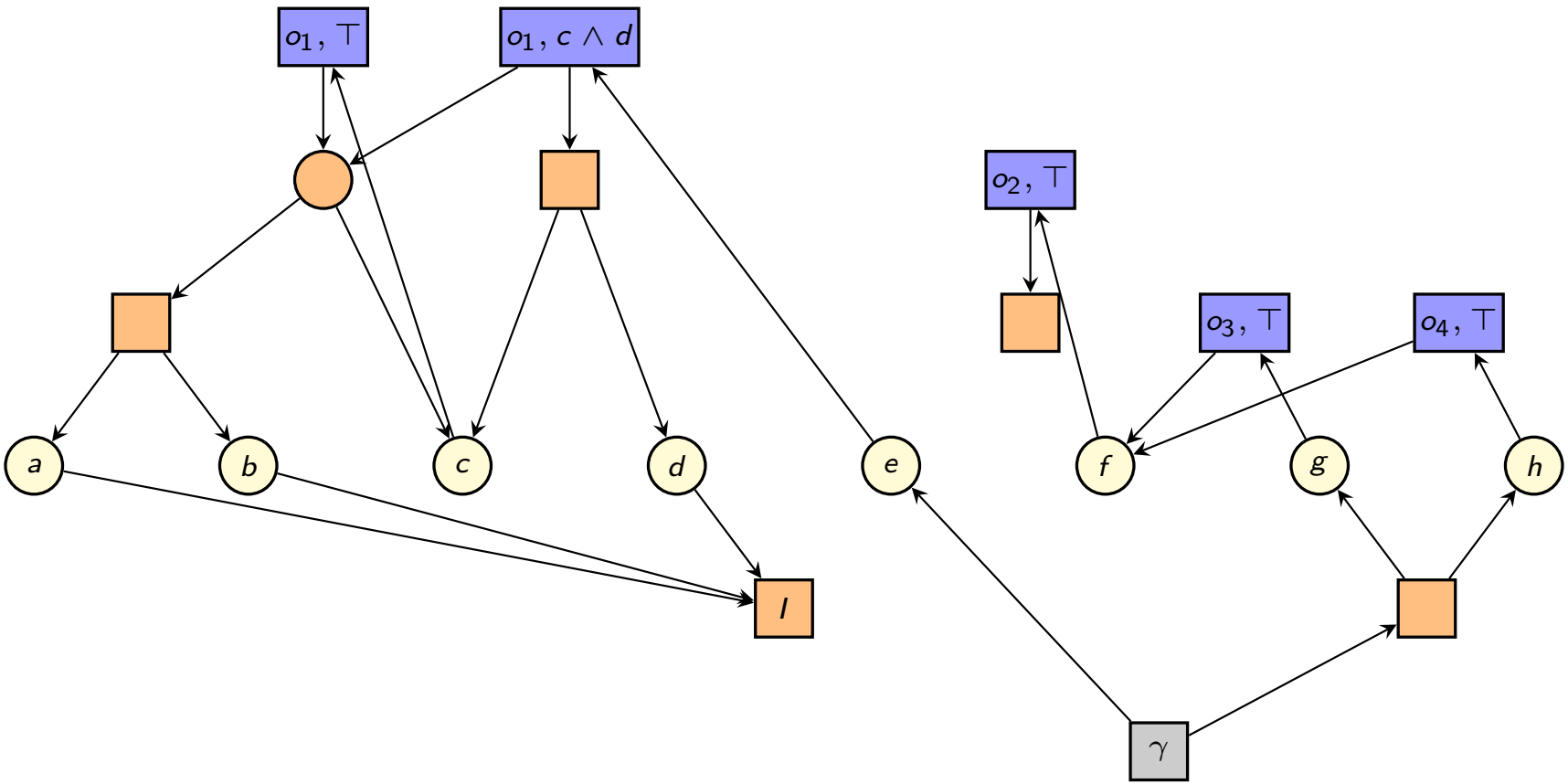
Goal Subgraph

Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task.

$RTG(\Pi^+)$ contains a goal subgraph, consisting of formula nodes for the goal γ and its subformulas, constructed in the same way as formula nodes for preconditions and effect conditions.

Goal Subgraph and Final Relaxed Task Graph: Example

$$\gamma = e \wedge (g \wedge h)$$



D5.3 Reachability Analysis

How Can We Use Relaxed Task Graphs?

- We are now done with the definition of relaxed task graphs.
- Now we want to use them to derive information about planning tasks.
- In the following chapter, we will use them to compute heuristics for delete-relaxed planning tasks.
- Here, we start with something simpler: reachability analysis.

Forced True Nodes and Reachability

Theorem (Forced True Nodes vs. Reachability)

Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task, and let $N_\mathbf{T}$ be the forced true nodes of $RTG(\Pi^+)$.

For all formulas over state variables φ that occur in the definition of Π^+:

φ is true in some reachable state of Π^+ iff $n_\varphi \in N_\mathbf{T}$.

(We omit the proof.)

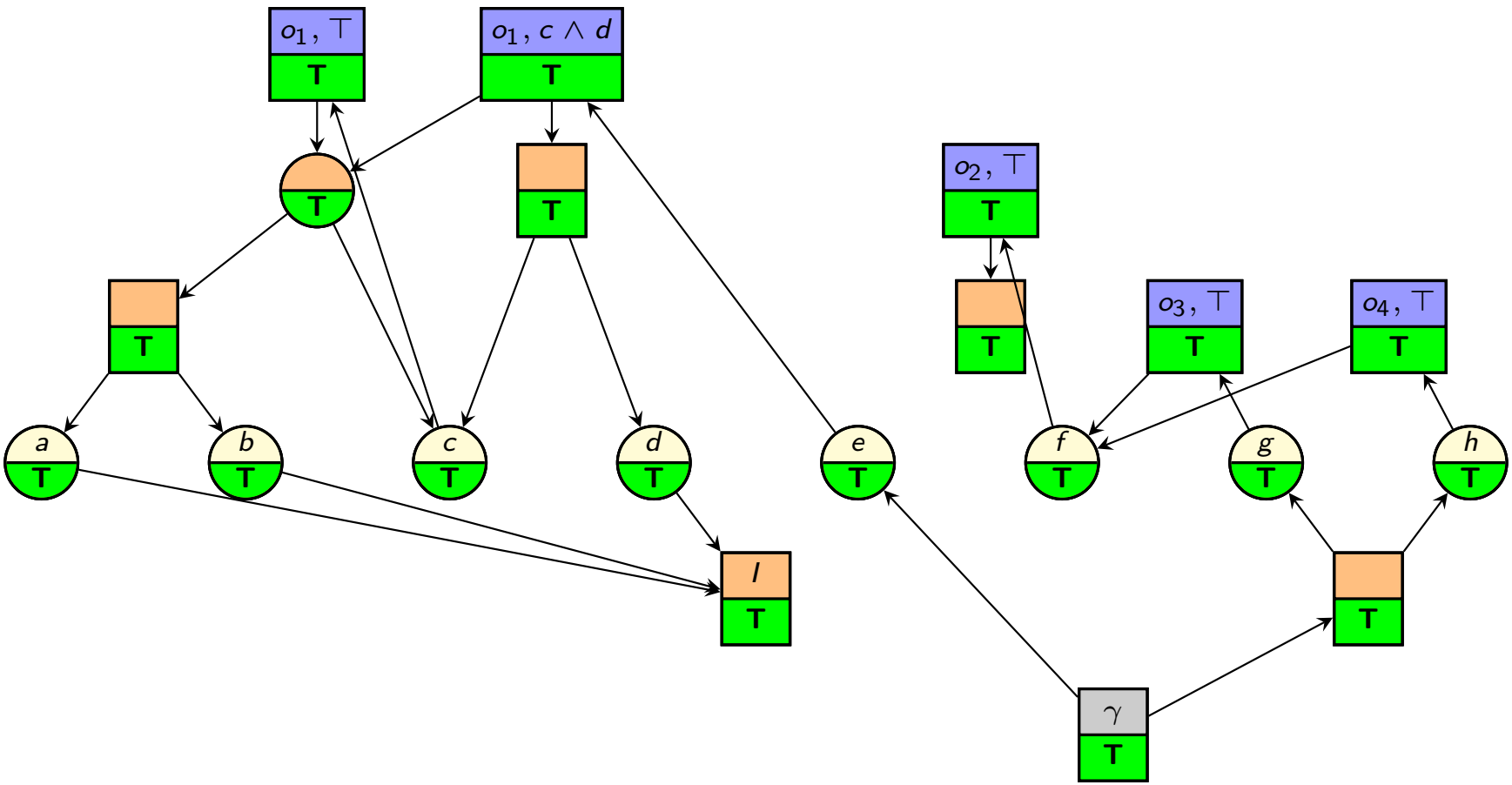
Forced True Nodes and Reachability: Consequences

Corollary

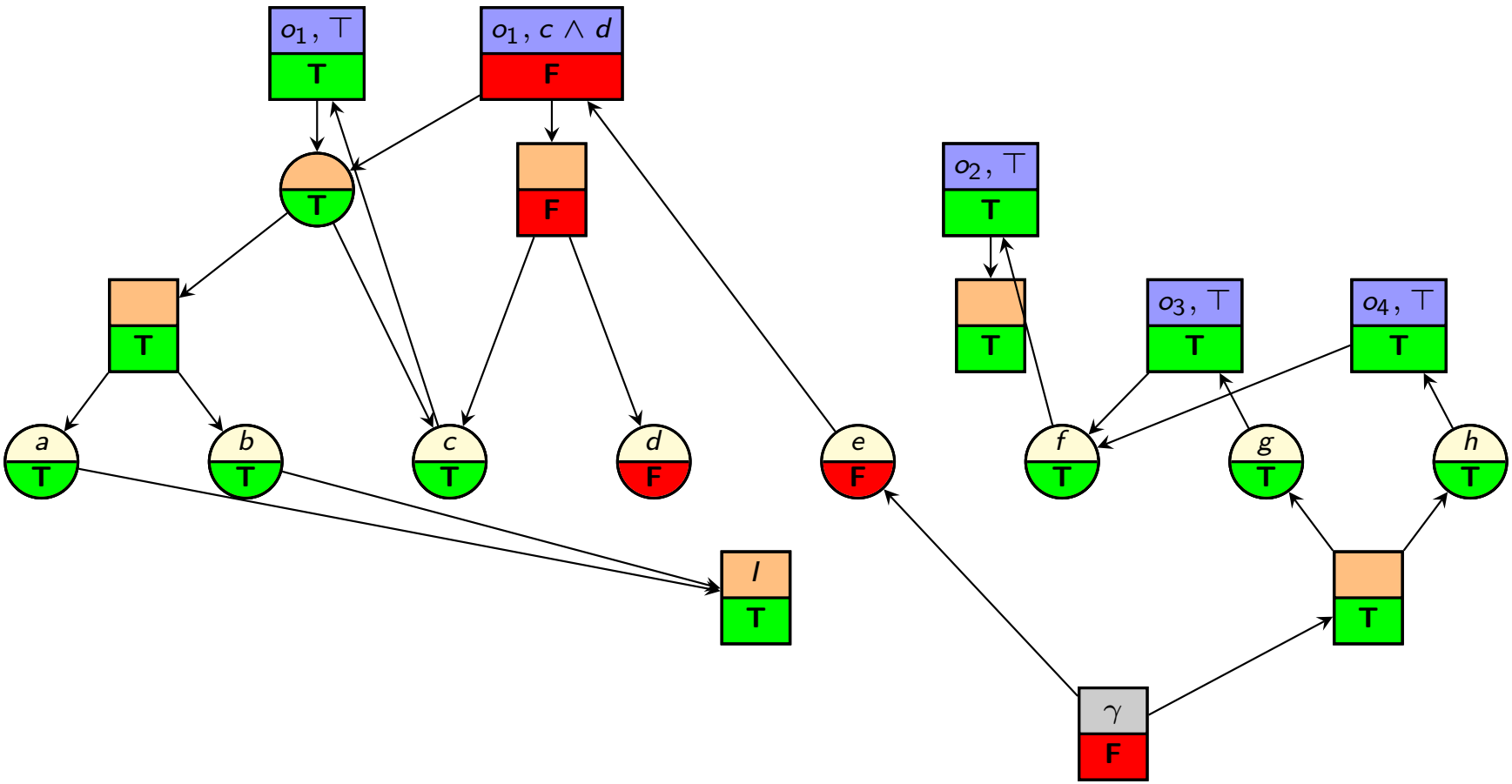
Let $\Pi^+ = \langle V, I, O^+, \gamma \rangle$ be a relaxed planning task, and let $N_{\mathbf{T}}$ be the forced true nodes of $RTG(\Pi^+)$. Then:

- *A state variable $v \in V$ is true in at least one reachable state iff $n_v \in N_{\mathbf{T}}$.*
- *An operator $o^+ \in O^+$ is part of at least one applicable operator sequence iff $n_{pre(o^+)} \in N_{\mathbf{T}}$.*
- *The relaxed task is solvable iff $n_\gamma \in N_{\mathbf{T}}$.*

Reachability Analysis: Example



Reachability Analysis: Example with Different Initial State



D5.4 Remarks

Relaxed Task Graphs in the Literature

Some remarks on the planning literature:

- Usually, only the STRIPS case is studied.

  ⇝ definitions simpler: only variable nodes and operator nodes, no formula nodes or effect nodes

- Usually, so-called relaxed planning graphs (RPGs) are studied instead of RTGs.
- These are temporally unrolled versions of RTGs, i.e., they have multiple layers ("time steps") and are acyclic.

⇝ Foundations of Artificial Intelligence course FS 2025, Ch. F3–F4

D5.5 Summary

Summary

- Relaxed task graphs (RTGs) represent (most of) the information of a relaxed planning task as an AND/OR graph.
- They consist of:
  - variable nodes
  - an initial node
  - operator subgraphs including formula nodes and effect nodes
  - a goal subgraph including formula nodes
- RTGs can be used to analyze reachability in relaxed tasks:
  forced true nodes mean "reachable",
  other nodes mean "unreachable".

Planning and Optimization

D6. Delete Relaxation: $h^{\max}$ and h^{add}

Malte Helmert and Gabriele Röger

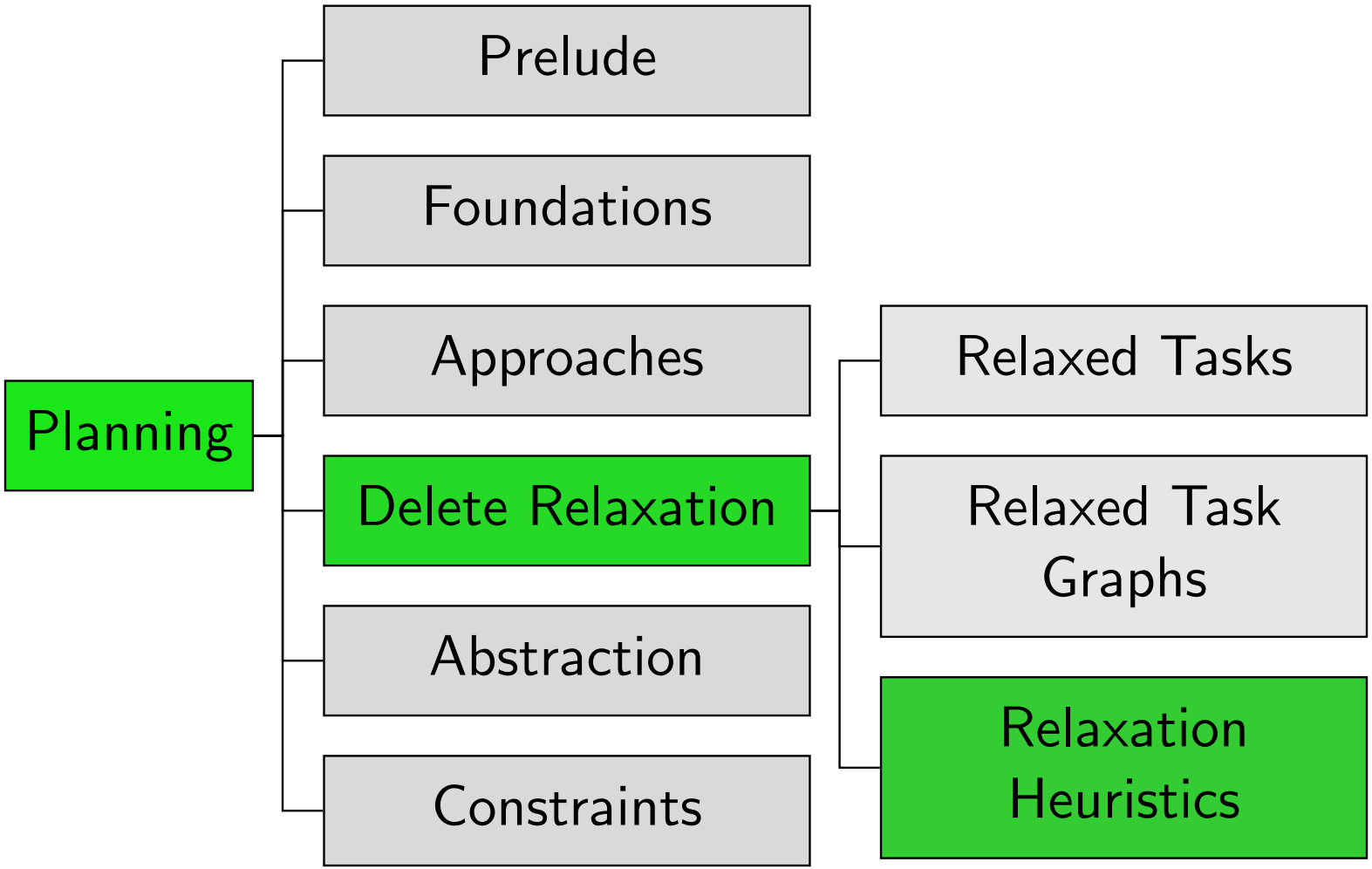
Universität Basel

October 27, 2025

Planning and Optimization

October 27, 2025 — D6. Delete Relaxation: $h^{\max}$ and h^{add}

Content of the Course



D6.1 Introduction

Delete Relaxation Heuristics

- In this chapter, we introduce heuristics based on delete relaxation.
- Their basic idea is to propagate information in relaxed task graphs, similar to the previous chapter.
- Unlike the previous chapter, we do not just propagate information about whether a given node is reachable, but estimates how expensive it is to reach the node.

Reminder: Running Example

We will use the same running example as in the previous chapter:
$\Pi = \langle V, I, \{o_1, o_2, o_3, o_4\}, \gamma \rangle$ with

$$
\begin{aligned}
V &= \{a, b, c, d, e, f, g, h\} \\
I &= \{a \mapsto \mathbf{T}, b \mapsto \mathbf{T}, c \mapsto \mathbf{F}, d \mapsto \mathbf{T}, \\
&\qquad e \mapsto \mathbf{F}, f \mapsto \mathbf{F}, g \mapsto \mathbf{F}, h \mapsto \mathbf{F}\} \\
o_1 &= \langle c \vee (a \wedge b), c \wedge ((c \wedge d) \rhd e), 1 \rangle \\
o_2 &= \langle \top, f, 2 \rangle \\
o_3 &= \langle f, g, 1 \rangle \\
o_4 &= \langle f, h, 1 \rangle \\
\gamma &= e \wedge (g \wedge h)
\end{aligned}
$$

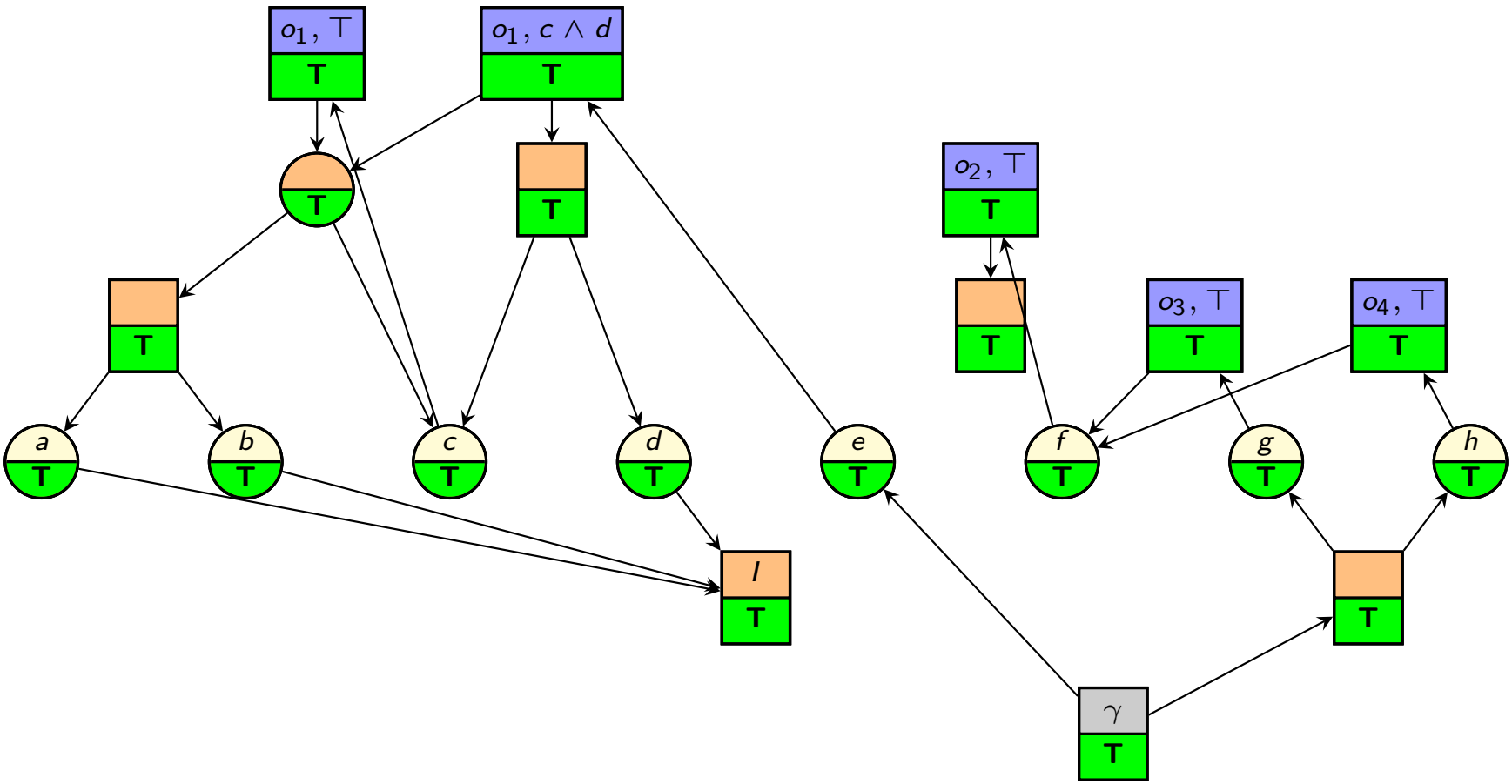
Algorithm for Reachability Analysis (Reminder)

- reachability analysis in RTGs = computing all forced true nodes = computing the most conservative assignment
- Here is an algorithm that achieves this:

Reachability Analysis

Associate a *reachable* attribute with each node.
for all nodes n:
    $n.reachable := \mathbf{false}$
while no fixed point is reached:
    Choose a node n.
    if n is an AND node:
        $n.reachable := \bigwedge_{n' \in succ(n)} n'.reachable$
    if n is an OR node:
        $n.reachable := \bigvee_{n' \in succ(n)} n'.reachable$

Reachability Analysis: Example (Reminder)



D6.2 $h^{\max}$ and h^{add}

Associating Costs with RTG Nodes

Basic intuitions for associating costs with RTG nodes:

- To apply an operator, we must pay its cost.
- To make an OR node true, it is sufficient to make one of its successors true.
  - ⇝ Therefore, we estimate the cost of an OR node as the minimum of the costs of its successors.
- To make an AND node true, all its successors must be made true first.
  - ⇝ We can be optimistic and estimate the cost as the maximum of the successor node costs.
  - ⇝ Or we can be pessimistic and estimate the cost as the sum of the successor node costs.
  - ⇝ We will prove later that this is indeed optimistic/pessimistic.

$h^{\max}$ Algorithm

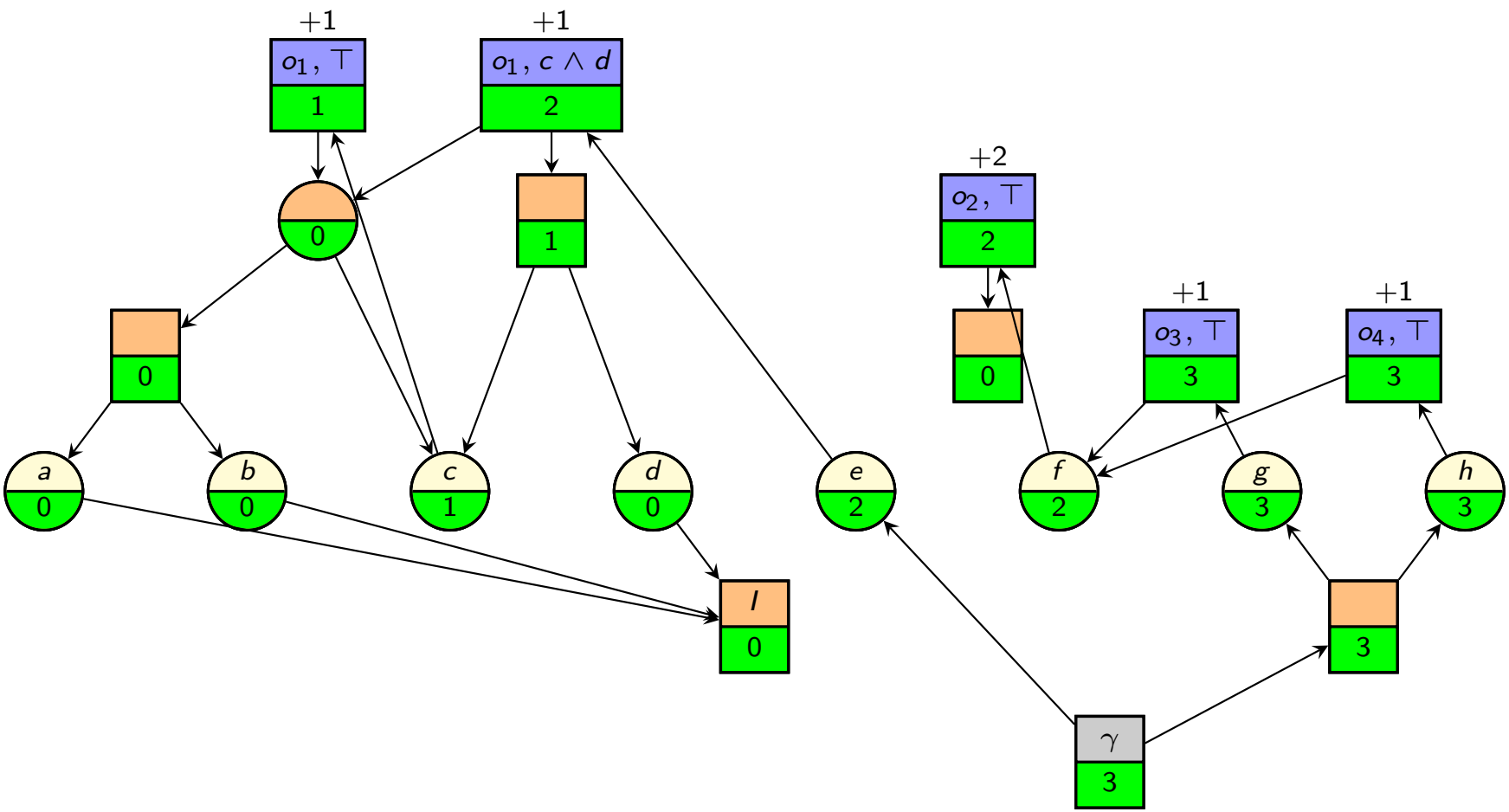
(Differences to reachability analysis algorithm highlighted.)

Computing $h^{\max}$ Values

Associate a *cost* attribute with each node.
for all nodes n:
    $n.cost := \infty$
while no fixed point is reached:
    Choose a node n.
    if n is an AND node that is not an effect node:
        $n.cost := \max_{n' \in succ(n)} n'.cost$
    if n is an effect node for operator o:
        $n.cost := cost(o) + \max_{n' \in succ(n)} n'.cost$
    if n is an OR node:
        $n.cost := \min_{n' \in succ(n)} n'.cost$

The overall heuristic value is the cost of the goal node, $n_\gamma.cost$.

$h^{\max}$: Example



$\rightsquigarrow h^{\max}(I) = 3$

h^{add} Algorithm

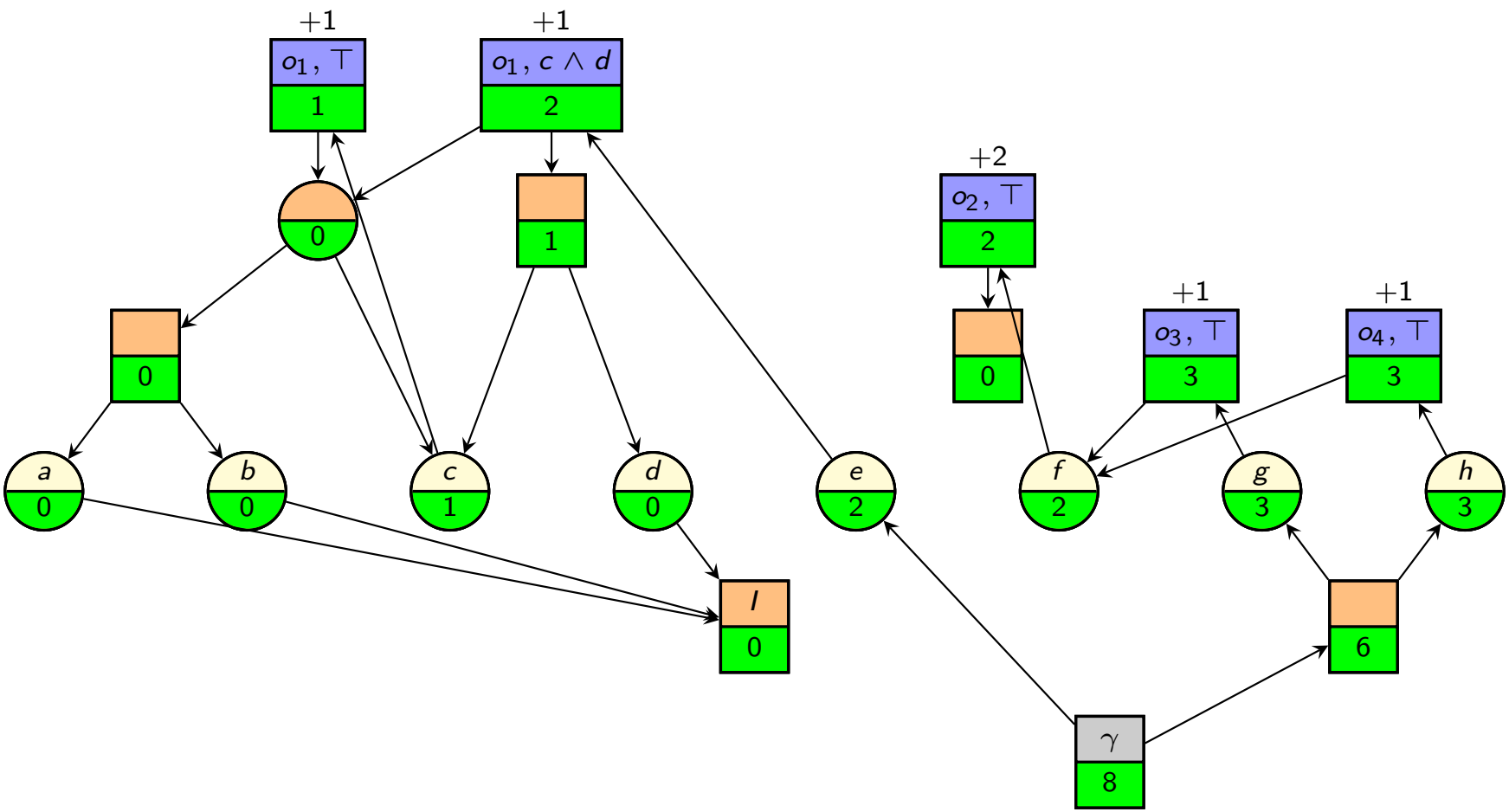
(Differences to $h^{\max}$ algorithm highlighted.)

Computing h^{add} Values

Associate a *cost* attribute with each node.
for all nodes n:
    $n.cost := \infty$
while no fixed point is reached:
    Choose a node n.
    if n is an AND node that is not an effect node:
        $n.cost := \sum_{n' \in succ(n)} n'.cost$
    if n is an effect node for operator o:
        $n.cost := cost(o) + \sum_{n' \in succ(n)} n'.cost$
    if n is an OR node:
        $n.cost := \min_{n' \in succ(n)} n'.cost$

The overall heuristic value is the cost of the goal node, $n_\gamma.cost$.

h^{add}: Example



$$\rightsquigarrow h^{\text{add}}(I) = 8$$

$h^{\max}$ and h^{add}: Definition

We can now define our first non-trivial efficient planning heuristics:

$h^{\max}$ and h^{add} Heuristics

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a propositional planning task in positive normal form.

The $h^{\max}$ heuristic value of a state s, written $h^{\max}(s)$, is obtained by constructing the RTG for $\Pi_s^+ = \langle V, s, O^+, \gamma \rangle$ and then computing $n_\gamma.\mathit{cost}$ using the $h^{\max}$ value algorithm for RTGs.

The h^{add} heuristic value of a state s, written $h^{\text{add}}(s)$, is computed in the same way using the h^{add} value algorithm for RTGs.

Notation: we will use the same notation $h^{\max}(n)$ and $h^{\text{add}}(n)$ for the $h^{\max}/h^{\text{add}}$ values of RTG nodes

D6.3 Properties of $h^{\max}$ and h^{add}

Understanding $h^{\max}$ and h^{add}

We want to understand $h^{\max}$ and h^{add} better:

- Are they well-defined?
- How can they be efficiently computed?
- Are they safe?
- Are they admissible?
- How do they compare to the optimal solution cost for a delete-relaxed task (h^+)?

Well-Definedness of $h^{\max}$ and h^{add} (1)

Are $h^{\max}$ and h^{add} well-defined?

- The algorithms for computing $h^{\max}$ and h^{add} values do not specify in which order the RTG nodes should be selected.
- It turns out that the order does not affect the final result.
  $\rightsquigarrow$ The $h^{\max}$ and h^{add} values are well-defined.
- To show this, we must show
  - that their computation always terminates, and
  - that all executions terminate with the same result.
- For time reasons, we only provide a proof sketch.

Well-Definedness of $h^{\max}$ and h^{add} (2)

Theorem

The fixed point algorithms for computing h^{max} and h^{add} values produce a well-defined result.

Proof Sketch.

Let $V_0, V_1, V_2, \ldots$ be the vectors of cost values during a given execution of the algorithm.

Termination: Note that $V_i \geq V_{i+1}$ for all i.
It is not hard to prove that each node value can only decrease a finite number of times: first from ∞ to some finite value, and then a finite number of additional times. ...

Well-Definedness of $h^{\max}$ and h^{add} (3)

Proof Sketch (continued).

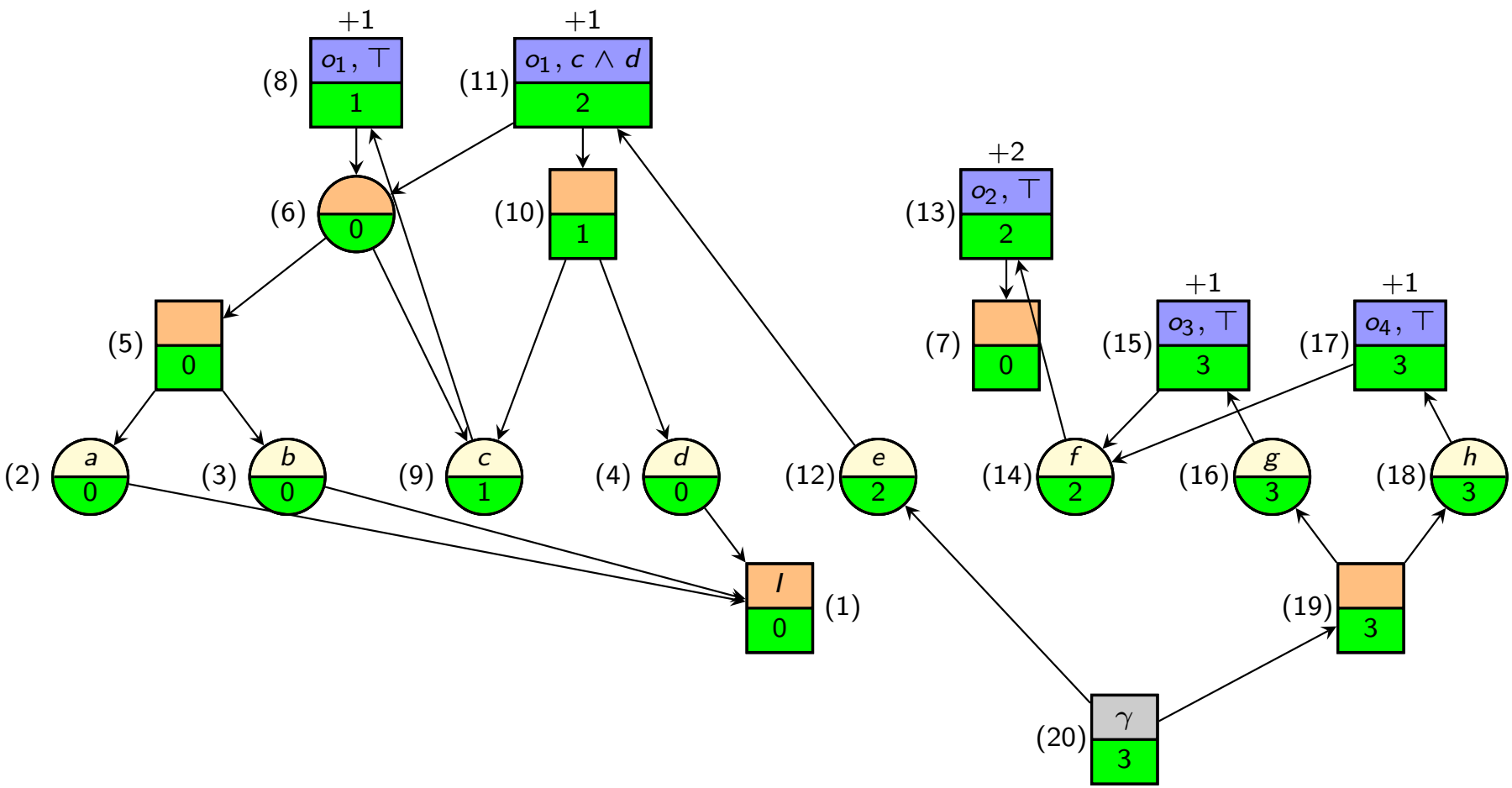
Uniqueness of result: Let $V_0 \geq V_1 \geq V_2 \geq \cdots \geq V_n$ be the finite sequence of cost value vectors until termination during a given execution of the algorithm.

- View the consistency conditions of all nodes (e.g., $n.cost = \min_{n' \in succ(n)} n'.cost$ for all OR nodes n) as a system of equations $\mathbf{E}$.
- V_n must be a solution to $\mathbf{E}$ (otherwise no fixed point is reached with V_n).
- For all $i \in \{0, \ldots, n\}$, show by induction over i that $V_i \geq S$ for all solutions S to $\mathbf{E}$.
- It follows that V_n is the unique maximum solution to $\mathbf{E}$ and hence well-defined.

Efficient Computation of $h^{\max}$ and h^{add}

- If nodes are poorly chosen, the $h^{\max}/h^{\text{add}}$ algorithm can update the same node many times until it reaches its final value.
- However, there is a simple strategy that prevents this: in every iteration, pick a node with minimum new value among all nodes that can be updated to a new value.
- With this strategy, no node is updated more than once. (We omit the proof, which is not complicated.)
- Using a suitable priority queue data structure, this allows computing the $h^{\max}/h^{\text{add}}$ values of an RTG with nodes N and arcs A in time $O(|N| \log |N| + |A|)$.

$h^{\max}$: Example of Efficient Computation



$$\rightsquigarrow h^{\max}(I) = 3$$

Efficient Computation of $h^{\max}$ and h^{add}: Remarks

- In the following chapters, we will always assume that we are using this efficient version of the $h^{\max}$ and h^{add} algorithm.
- In particular, we will assume that all reachable nodes of the relaxed task graph are processed exactly once (and all unreachable nodes not at all), so that it makes sense to speak of certain nodes being processed after others etc.

Heuristic Quality of $h^{\max}$ and h^{add}

This leaves us with the questions about the heuristic quality of $h^{\max}$ and h^{add}:

- Are they safe?
- Are they admissible?
- How do they compare to the optimal solution cost for a delete-relaxed task?

It is easy to see that $h^{\max}$ and h^{add} are safe:
they assign ∞ iff a node is unreachable in the delete relaxation.

In our running example, it seems that $h^{\max}$ is prone to underestimation and h^{add} is prone to overestimation.

We will study this further in the next chapter.

D6.4 Summary

Summary

- $h^{\max}$ and h^{add} values estimate how expensive it is to reach a state variable, operator effect or formula (e.g., the goal).
- They are computed by propagating cost information in relaxed task graphs:
  - At OR nodes, choose the cheapest alternative.
  - At AND nodes, maximize or sum the successor costs.
  - At effect nodes, also add the operator cost.
- $h^{\max}$ and h^{add} values can serve as heuristics.
- They are well-defined and can be computed efficiently by computing them in order of increasing cost along the RTG.

Planning and Optimization

D7. Delete Relaxation: Analysis of $h^{\max}$ and h^{add}

Malte Helmert and Gabriele Röger

Universität Basel

October 29, 2025

Planning and Optimization

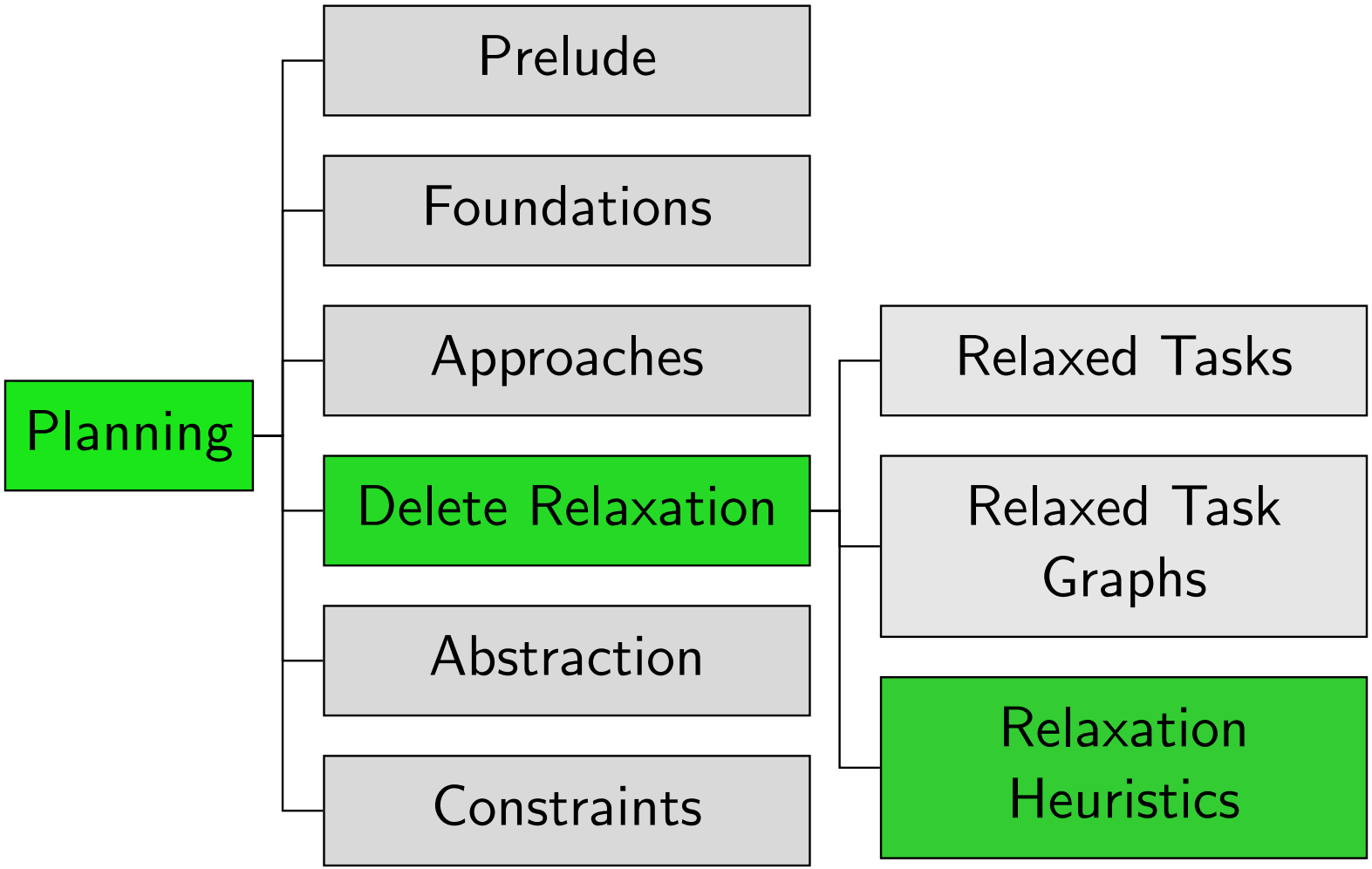
October 29, 2025 — D7. Delete Relaxation: Analysis of $h^{\max}$ and h^{add}

D7.1 Choice Functions

D7.2 Best Achievers

D7.3 Summary

Content of the Course



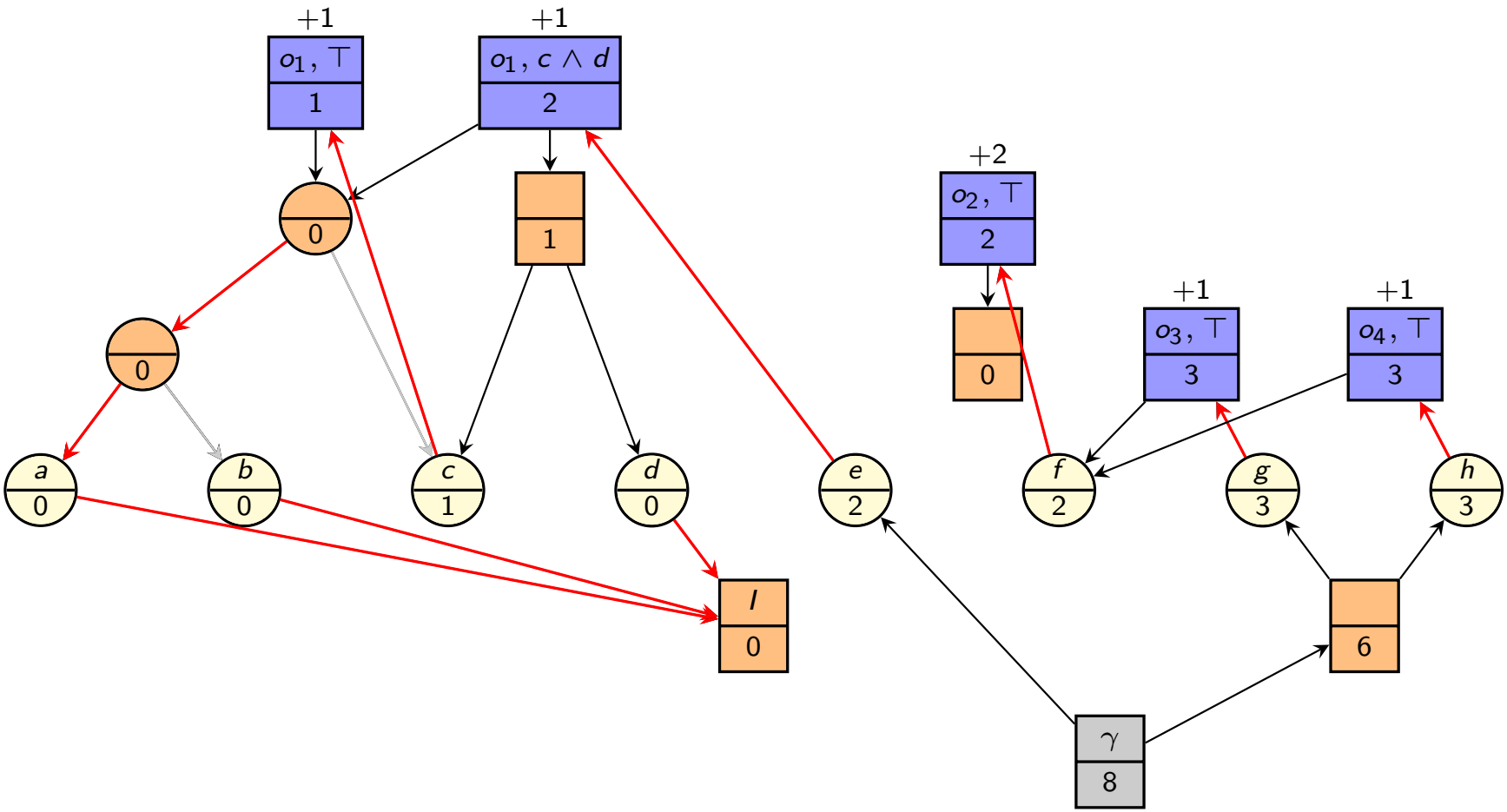
D7.1 Choice Functions

Motivation

- In this chapter, we analyze the behaviour of $h^{\max}$ and h^{add} more deeply.
- Our goal is to understand their shortcomings.
  - In the next chapter we then used this understanding to devise an improved heuristic.
- As a preparation for our analysis, we need some further definitions that concern choices in AND/OR graphs.
- The key observation is that if we want to establish the value of a certain node n, we can to some extent choose how we want to achieve the OR nodes that are relevant to achieving n.

Preview: Choice Function & Best Achievers

Preserve at most one outgoing arc of each OR node,
but node values may not change.



(precondition of o_1 modified to $c \vee (a \vee b)$)

Choice Functions

Definition (Choice Function)

Let G be an AND/OR graph with nodes N and OR nodes $N_\vee$.

A choice function for G is a function $f : N' \to N$ defined on some set $N' \subseteq N_\vee$ such that $f(n) \in succ(n)$ for all $n \in N'$.

- In words, choice functions select (at most) one successor for each OR node of G.
- Intuitively, $f(n)$ selects by which disjunct n is achieved.
- If $f(n)$ is undefined for a given n, the intuition is that n is not achieved.

Reduced Graphs

Once we have decided how to achieve an OR node,
we can remove the other alternatives:

Definition (Reduced Graph)

Let G be an AND/OR graph, and let f be a choice function for G defined on nodes N'.

The reduced graph for f is the subgraph of G where all outgoing arcs of OR nodes are removed except for the chosen arcs $\langle n, f(n) \rangle$ with $n \in N'$.

D7.2 Best Achievers

Choice Functions Induced by $h^{\max}$ and h^{add}

Which choices do $h^{\max}$ and h^{add} make?

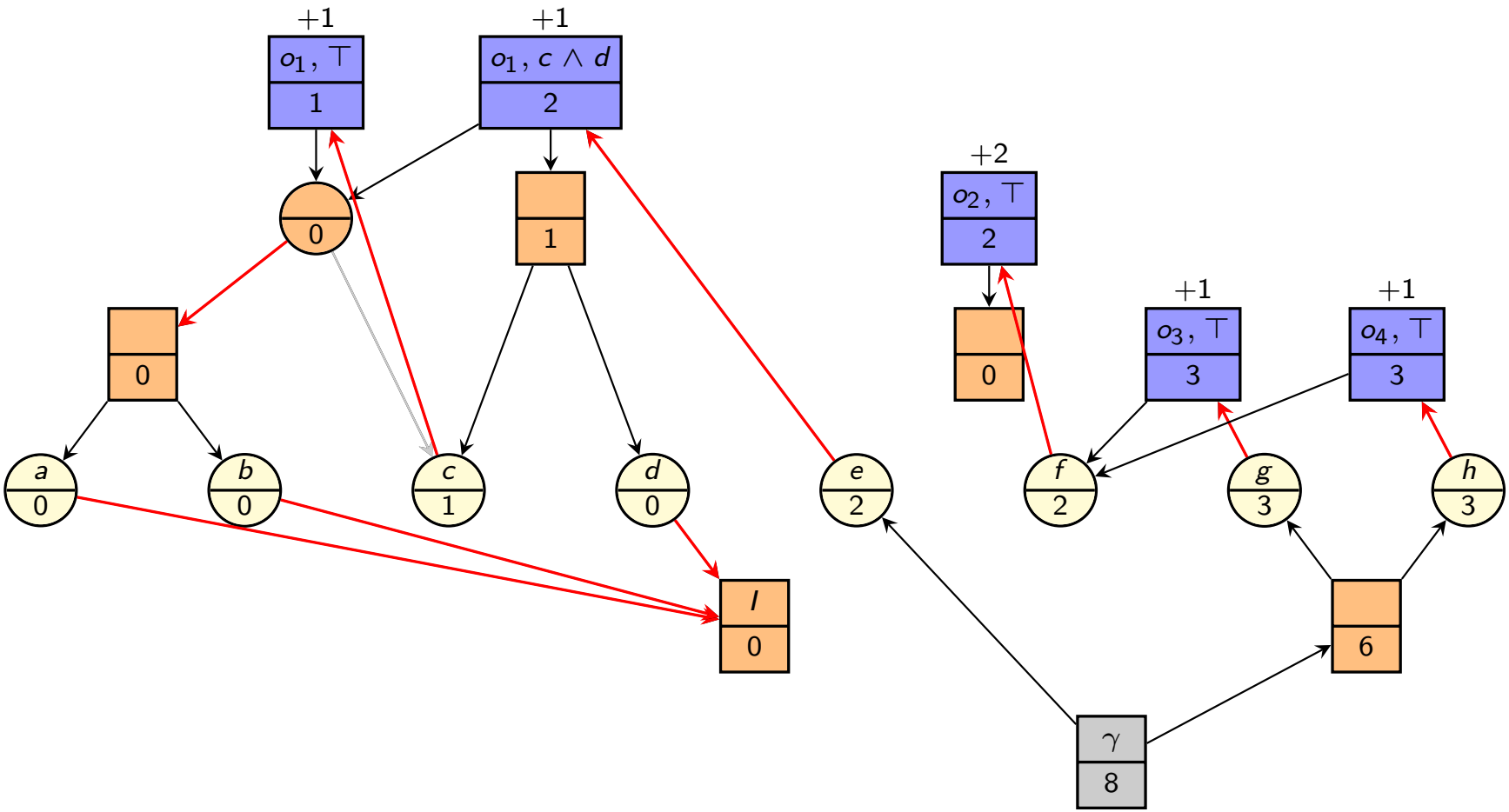
- At every OR node n, we set the cost of n to the minimum of the costs of the successors of n.
- The motivation for this is to achieve n via the successor that can be achieved most cheaply according to our cost estimates.

$\rightsquigarrow$ This corresponds to defining a choice function f with $f(n) \in \arg\min_{n' \in N'} n'.cost$ for all reached OR nodes n, where $N' \subseteq succ(n)$ are all successors of n processed before n.

- The successors chosen by this cost function are called best achievers (according to $h^{\max}$ or h^{add}).
- Note that the best achiever function f is in general not well-defined because there can be multiple minimizers. We assume that ties are broken arbitrarily.

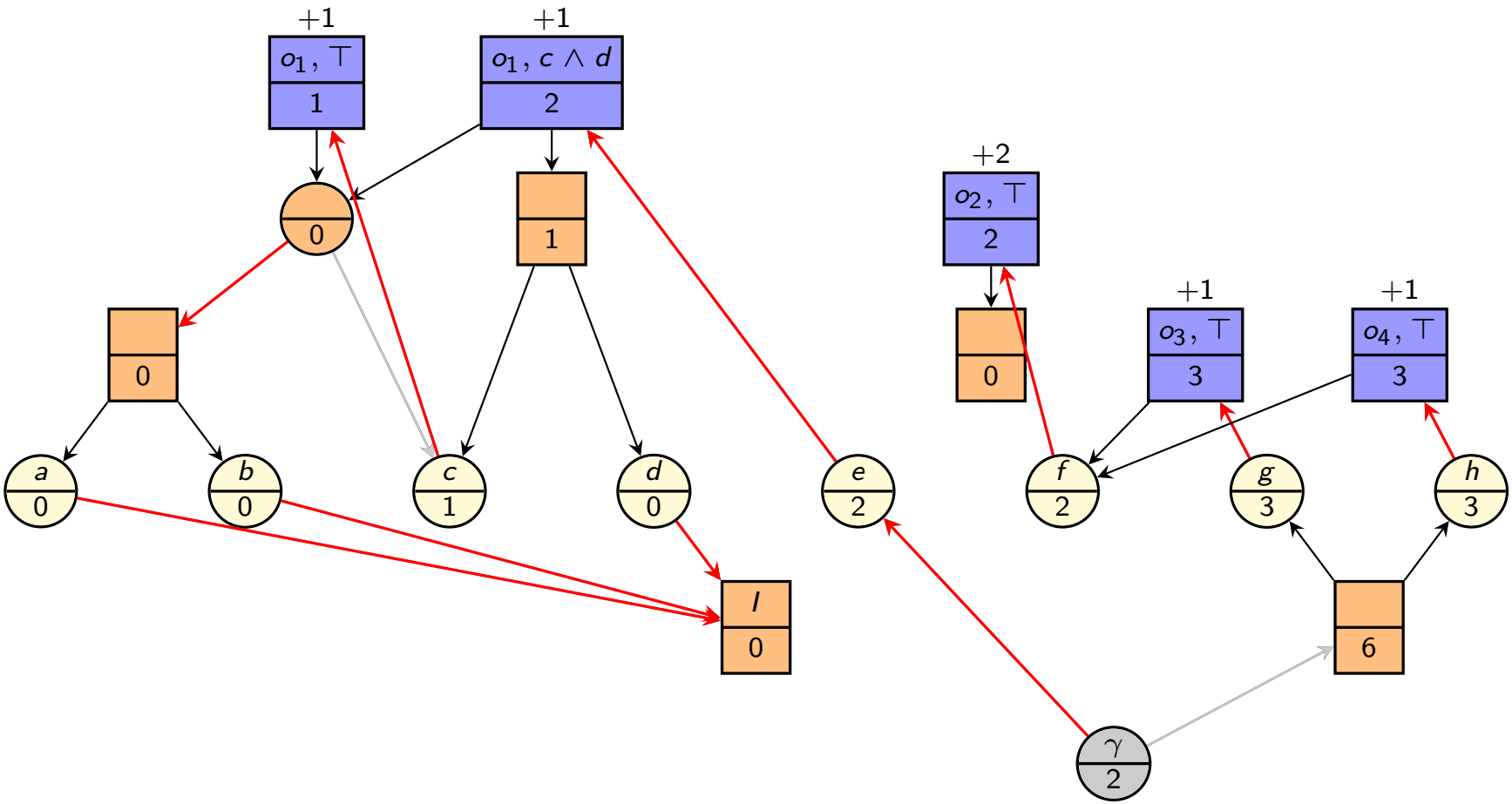
Example: Best Achievers (1)

best achievers for h^{add}



Example: Best Achievers (2)

best achievers for h^{add}; modified goal $e \vee (g \wedge h)$



Best Achiever Graphs

- Observation: The $h^{\max}/h^{\text{add}}$ costs of nodes remain the same if we replace the RTG by the reduced graph for the respective best achiever function.
- The AND/OR graph that is obtained by removing all nodes with infinite cost from this reduced graph is called the best achiever graph for $h^{\max}/h^{\text{add}}$.
  - We write $G^{\max}$ and G^{add} for the best achiever graphs.
- $G^{\max}$ (G^{add}) is always acyclic: for all arcs $\langle n, n' \rangle$ it contains, n is processed by $h^{\max}$ (by h^{add}) after n'.

Paths in Best Achiever Graphs

Let n be a node of the best achiever graph.
Let N_{eff} be the set of effect nodes of the best achiever graph.
The cost of an effect node is the cost of the associated operator.
The cost of a path in the best achiever graph is the sum of costs of all effect nodes on the path.

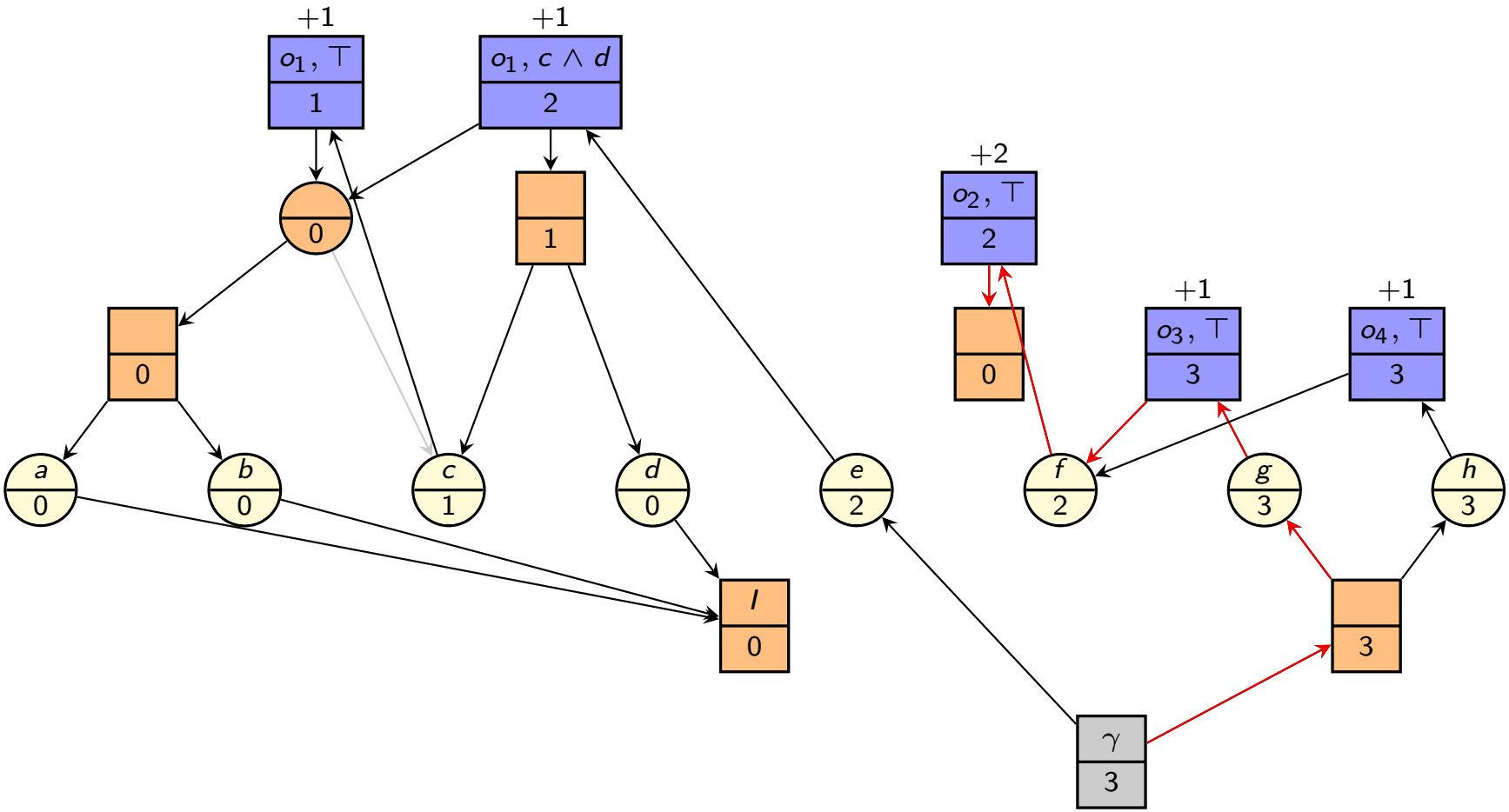
The following properties can be shown by induction:

- $h^{\max}(n)$ is the maximum cost of all paths originating from n in $G^{\max}$. A path achieving this maximum is called a critical path.
- $h^{\text{add}}(n)$ is the sum, over all effect nodes n', of the cost of n' multiplied by the number of paths from n to n' in G^{add}.

In particular, these properties hold for the goal node n_γ if it is reachable.

Example: Undercounting in $h^{\max}$

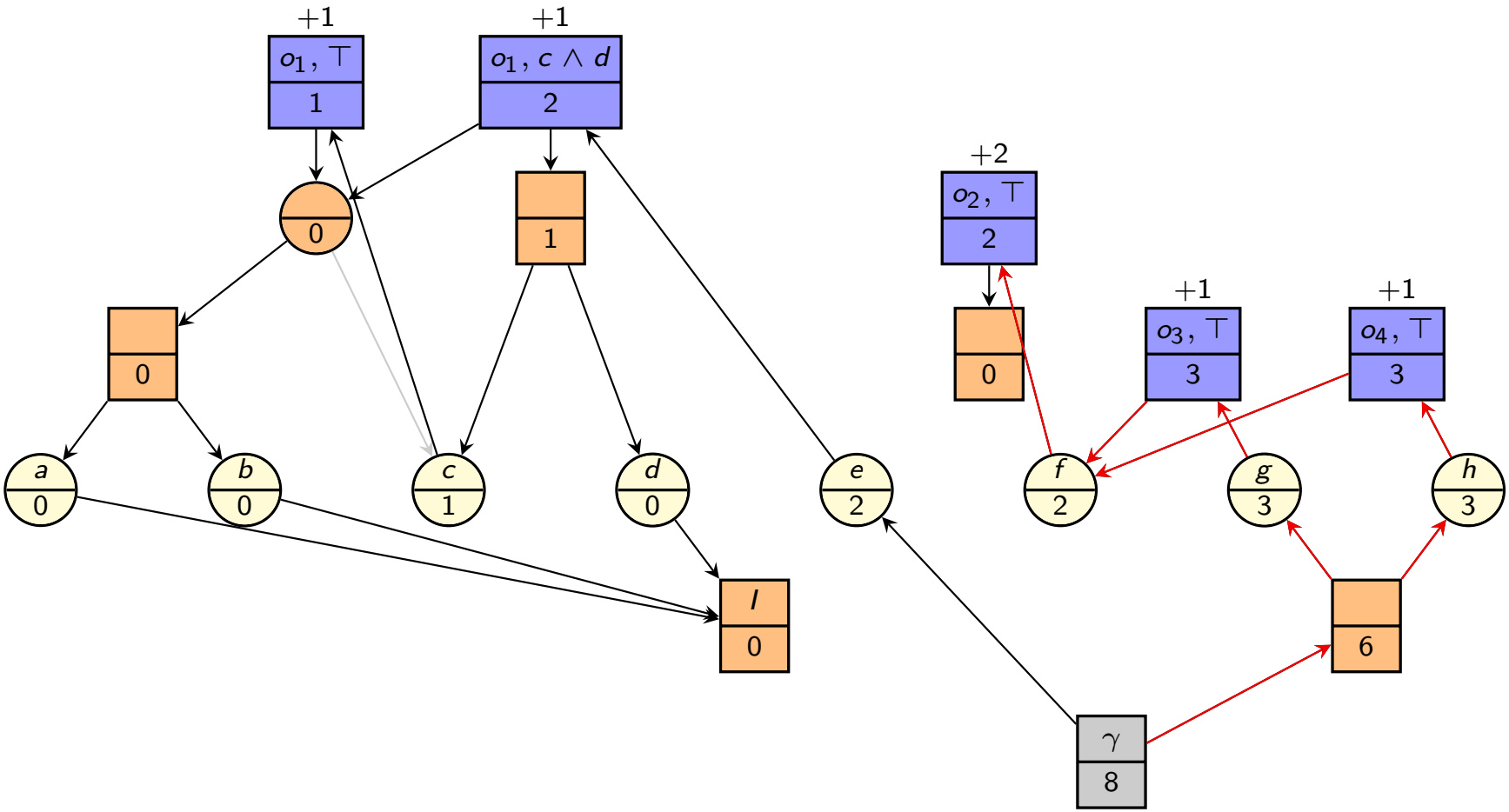
$G^{\max}$: undercounting in $h^{\max}$



$\rightsquigarrow$ o_1 and o_4 not counted because they are off the critical path

Example: Overcounting in h^{add}

G^{add}: overcounting in h^{add}



$\rightsquigarrow$ o_2 counted twice because there are two paths to $n^{\top}_{o_2}$

D7.3 Summary

Summary

- $h^{\max}$ and h^{add} can be used to decide how to achieve OR nodes in a relaxed task graph
  $\rightsquigarrow$ best achievers
- Best achiever graphs help identify shortcomings of $h^{\max}$ and h^{add} compared to the perfect delete relaxation heuristic h^+.
  - $h^{\max}$ underestimates h^+ because it only considers the cost of a critical path for the relaxed planning task.
  - h^{add} overestimates h^+ because it double-counts operators occurring on multiple paths in the best achiever graph.

Planning and Optimization

D8. Delete Relaxation: h^{FF} and Comparison of Heuristics

Malte Helmert and Gabriele Röger

Universität Basel

October 29, 2025

Planning and Optimization

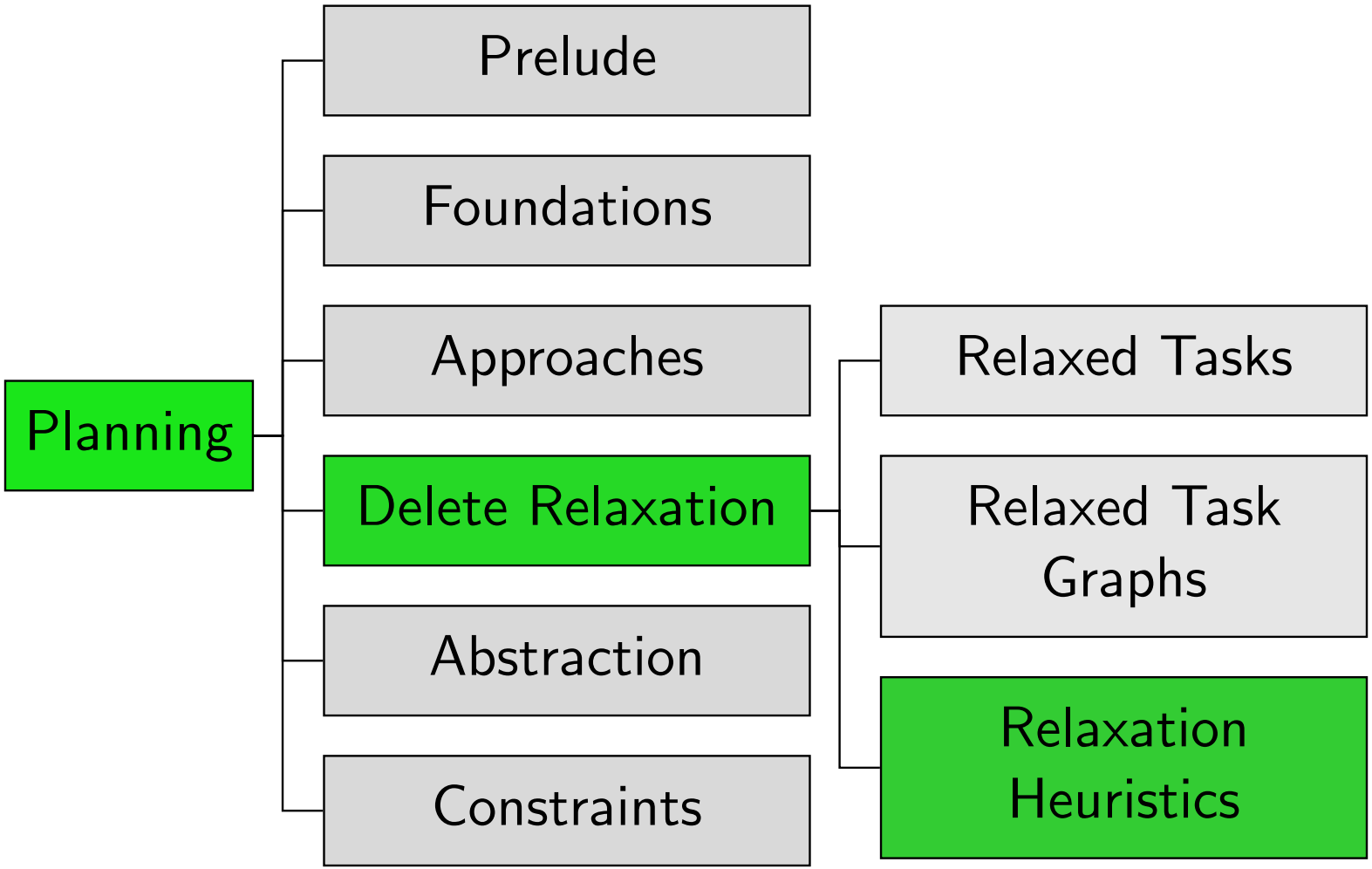
October 29, 2025 — D8. Delete Relaxation: h^{FF} and Comparison of Heuristics

D8.1 The FF Heuristic

D8.2 $h^{\max}$ vs. h^{add} vs. h^{FF} vs. h^{+}

D8.3 Summary

Content of the Course



D8.1 The FF Heuristic

Inaccuracies in h^{max} and h^{add}

- h^{max} is often inaccurate because it undercounts:
  the heuristic estimate only reflects the cost of a critical path, which is often only a small fraction of the overall plan.
- h^{add} is often inaccurate because it overcounts:
  if the same subproblem is reached in many ways, it will be counted many times although it only needs to be solved once.

The FF Heuristic

With best achiever graphs, there is a simple solution to the overcounting of h^{add}: count all effect nodes that h^{add} would count, but only count each of them once.

Definition (FF Heuristic)

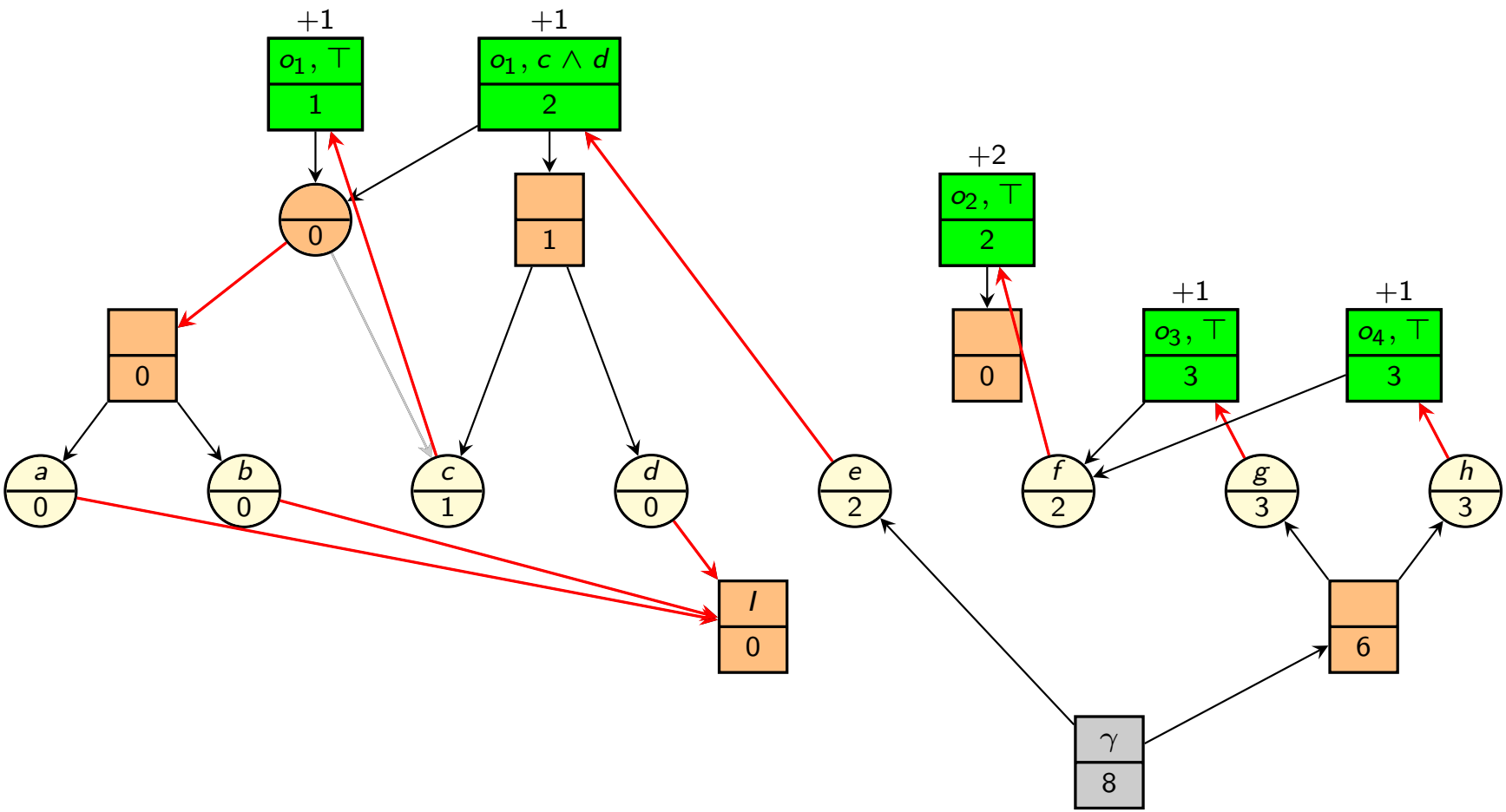
Let $\Pi = \langle V, I, O, \gamma \rangle$ be a propositional planning task in positive normal form. The FF heuristic for a state s of Π, written $h^{\mathrm{FF}}(s)$, is computed as follows:

- Construct the RTG for the task $\langle V, s, O^+, \gamma \rangle$
- Construct the best achiever graph G^{add}.
- Compute the set of effect nodes $\{n_{o_1}^{\chi_1}, \ldots, n_{o_k}^{\chi_k}\}$ reachable from n_γ in G^{add}.
- Return $h^{\mathrm{FF}}(s) = \sum_{i=1}^{k} cost(o_i)$.

Note: h^{FF} is not well-defined; different tie-breaking policies for best achievers can lead to different heuristic values

Example: FF Heuristic (1)

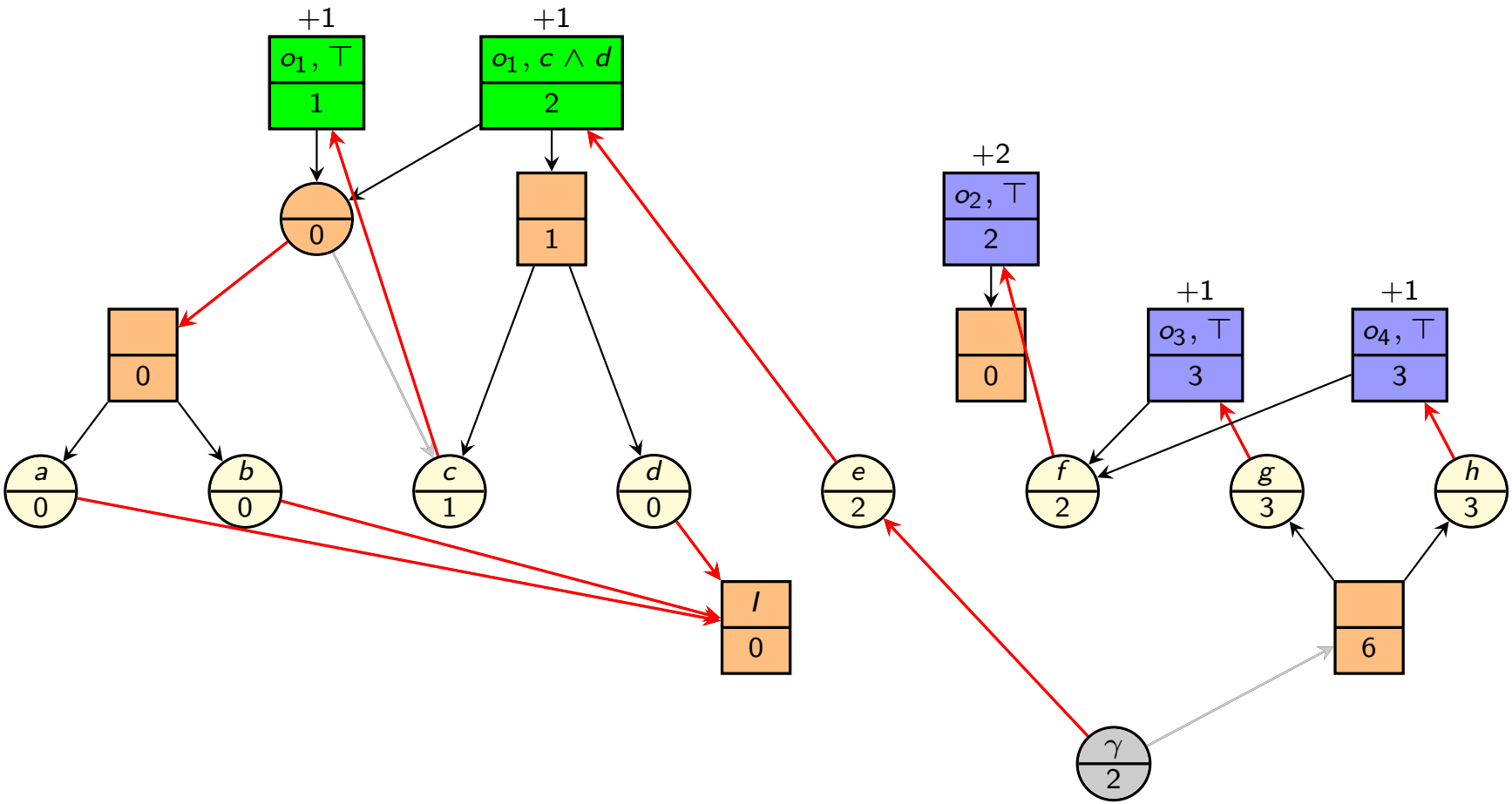
FF heuristic computation



$$h^{\mathrm{FF}}(s) = 1 + 1 + 2 + 1 + 1 = 6$$

Example: FF Heuristic (2)

FF heuristic computation; modified goal $e \vee (g \wedge h)$



$$h^{\mathrm{FF}}(s) = 1 + 1 = 2$$

D8.2 $h^{\max}$ vs. h^{add} vs. h^{FF} vs. h^{+}

Reminder: Optimal Delete Relaxation Heuristic

Definition (h^+ Heuristic)

Let Π be a propositional planning task in positive normal form, and let s be a state of Π.

The optimal delete relaxation heuristic for s, written $h^+(s)$, is the perfect heuristic value $h^*(s)$ of state s in the delete-relaxed task Π^+.

- Reminder: We proved that $h^+(s)$ is hard to compute. ($\mathrm{BCPlanEx}$ is NP-complete for delete-relaxed tasks.)
- The optimal delete relaxation heuristic is often used as a reference point for comparison.

Relationships between Delete Relaxation Heuristics (1)

Theorem

Let Π be a propositional planning task in positive normal form, and let s be a state of Π.

Then:

1. $h^{max}(s) \leq h^{+}(s) \leq h^{FF}(s) \leq h^{add}(s)$
2. $h^{max}(s) = \infty$ *iff* $h^{+}(s) = \infty$ *iff* $h^{FF}(s) = \infty$ *iff* $h^{add}(s) = \infty$
3. h^{max} *and* h^{+} *are admissible and consistent.*
4. h^{FF} *and* h^{add} *are neither admissible nor consistent.*
5. *All four heuristics are safe and goal-aware.*

Relationships between Delete Relaxation Heuristics (2)

Proof Sketch.

for 1:

- To show $h^{max}(s) \leq h^{+}(s)$, show that critical path costs can be defined for arbitrary relaxed plans and that the critical path cost of a plan is never larger than the cost of the plan.
  Then show that $h^{max}(s)$ computes the minimal critical path cost over all delete-relaxed plans.
- To show $h^{+}(s) \leq h^{FF}(s)$, prove that the operators belonging to the effect nodes counted by h^{FF} form a relaxed plan.
  No relaxed plan is cheaper than h^{+} by definition of h^{+}.
- $h^{FF}(s) \leq h^{add}(s)$ is obvious from the description of h^{FF}:
  both heuristics count the same operators,
  but h^{add} may count some of them multiple times.

...

Relationships between Delete Relaxation Heuristics (3)

Proof Sketch (continued).

for 2: all heuristics are infinite iff the task has no relaxed solution

for 3: admissibility follows from $h^{\max}(s) \leq h^+(s)$
because we already know that h^+ is admissible;
we omit the argument for consistency

for 4: construct a counterexample to admissibility for h^{FF}

for 5: goal-awareness is easy to show; safety follows from 2.+3. □

D8.3 Summary

Summary

- The FF heuristic repairs the double-counting of h^{add} and therefore approximates h^+ more closely.
- The key idea is to mark all effect nodes "used" for the h^{add} value of the goal and count each of them once.
- In general, $h^{max}(s) \leq h^+(s) \leq h^{FF}(s) \leq h^{add}(s)$.
- h^{max} and h^+ are admissible; h^{FF} and h^{add} are not.

Literature Pointers

(Some) delete-relaxation heuristics in the planning literature:

- additive heuristic h^{add} (Bonet, Loerincs & Geffner, 1997)
- maximum heuristic h^{max} (Bonet & Geffner, 1999)
- (original) FF heuristic (Hoffmann & Nebel, 2001)
- cost-sharing heuristic h^{cs} (Mirkis & Domshlak, 2007)
- set-additive heuristics h^{sa} (Keyder & Geffner, 2008)
- FF/additive heuristic h^{FF} (Keyder & Geffner, 2008)
- local Steiner tree heuristic h^{lst} (Keyder & Geffner, 2009)

$\rightsquigarrow$ also hybrids such as semi-relaxed heuristics and delete-relaxation landmark heuristics

Planning and Optimization

E1. Planning Tasks in Finite-Domain Representation

Malte Helmert and Gabriele Röger

Universität Basel

November 3, 2025

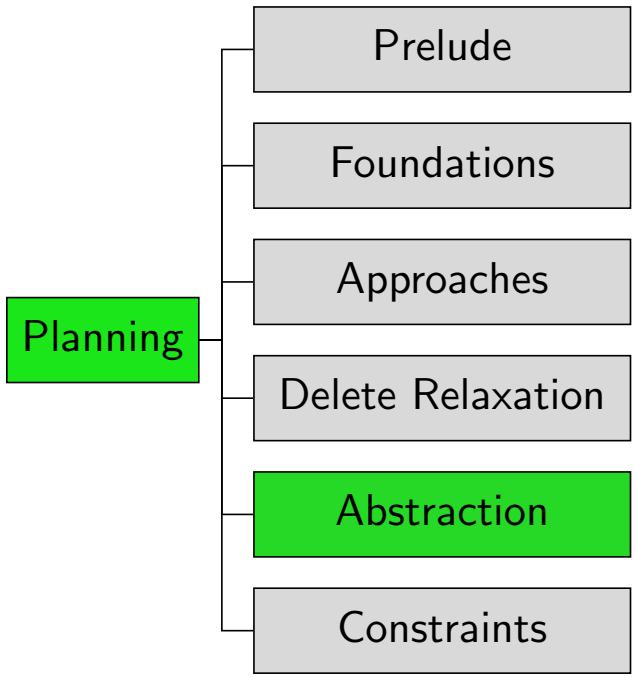
E1.1 Finite-Domain Representation

E1.2 Equivalence and Normal Forms

E1.3 Summary

How We Continue

- The next class of heuristics we will consider are abstraction heuristics.



- However, this requires some preparations.

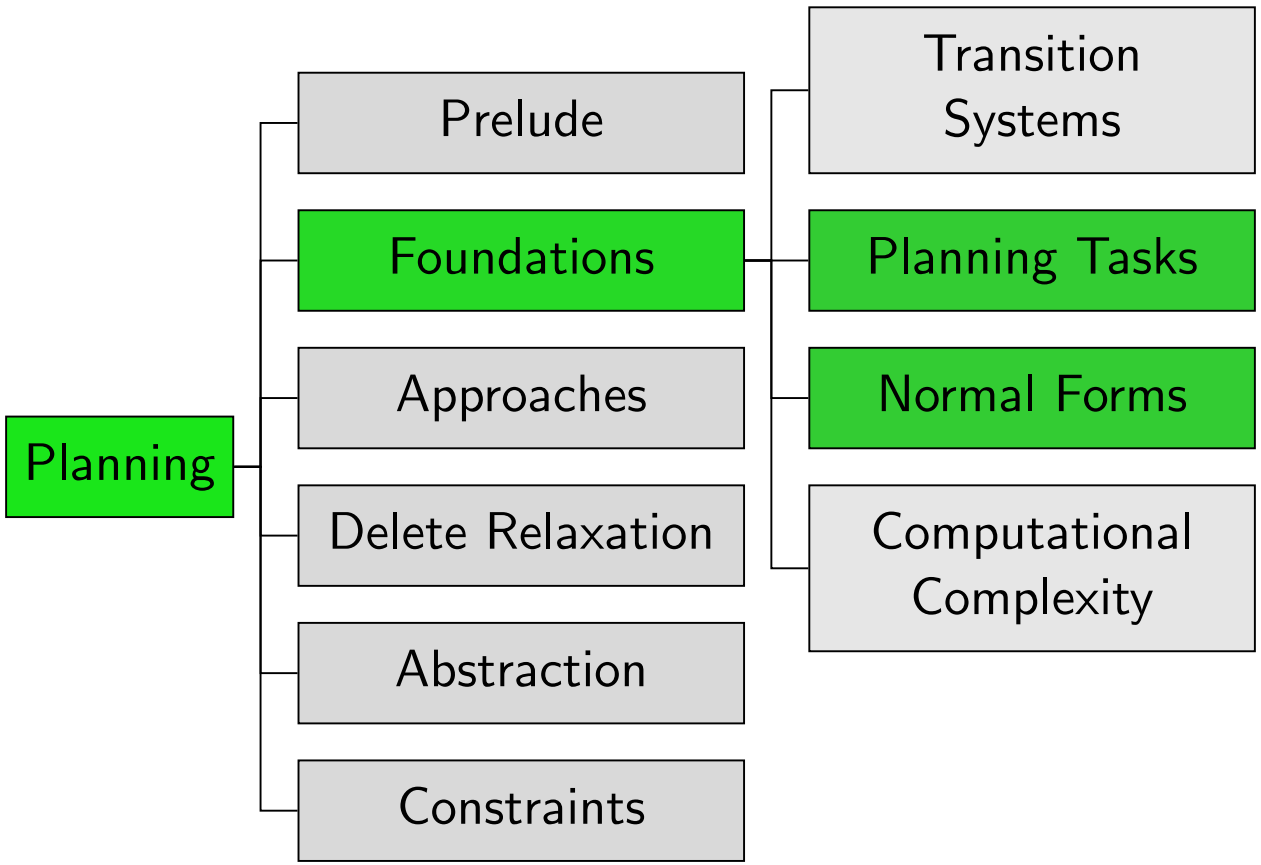
Back to Foundations: Finite-Domain Representation

- Abstraction heuristics benefit from a more compact task representation, called finite-domain representation.
- To understand the relationship to the propositional task representation, we need to know a special kind of invariants, namely mutexes.

$\rightsquigarrow$ We first get to know finite-domain representation (this chapter) and then speak about invariants and transformations between the representations (next chapter).

$\rightsquigarrow$ not specific to abstraction heuristics, but general foundations

Content of the Course



E1.1 Finite-Domain Representation

Finite-Domain State Variables

- So far, we used propositional (Boolean) state variables.
  $\rightsquigarrow$ possible values **T** and **F**
- We now consider finite-domain variables.
  $\rightsquigarrow$ every variable has a finite set of possible values
- A state is still an assignment to the state variables.

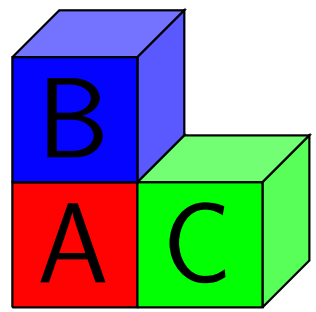
Example: $O(n^2)$ Boolean variables or $O(n)$ finite-domain variables with domain size $O(n)$ suffice for blocks world with n blocks.

Blocks World State with Propositional Variables

Example

$$s(\textit{A-on-B}) = \mathbf{F}$$
$$s(\textit{A-on-C}) = \mathbf{F}$$
$$s(\textit{A-on-table}) = \mathbf{T}$$
$$s(\textit{B-on-A}) = \mathbf{T}$$
$$s(\textit{B-on-C}) = \mathbf{F}$$
$$s(\textit{B-on-table}) = \mathbf{F}$$
$$s(\textit{C-on-A}) = \mathbf{F}$$
$$s(\textit{C-on-B}) = \mathbf{F}$$
$$s(\textit{C-on-table}) = \mathbf{T}$$

$\rightsquigarrow 2^9 = 512$ states



Note: it may be useful to add auxiliary state variables like *A-clear*.

Blocks World State with Finite-Domain Variables

Example

Use three finite-domain state variables:

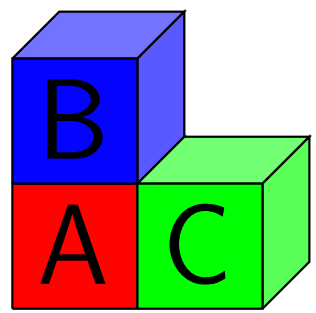
- *below-a*: $\{\mathsf{b}, \mathsf{c}, \mathsf{table}\}$
- *below-b*: $\{\mathsf{a}, \mathsf{c}, \mathsf{table}\}$
- *below-c*: $\{\mathsf{a}, \mathsf{b}, \mathsf{table}\}$

$s(\textit{below-a}) = \mathsf{table}$

$s(\textit{below-b}) = \mathsf{a}$

$s(\textit{below-c}) = \mathsf{table}$

$\rightsquigarrow 3^3 = 27$ states



Note: it may be useful to add auxiliary state variables like *above-a*.

Advantage of Finite-Domain Representation

How many "useless" (physically impossible) states are there with these blocks world state representations?

- There are 13 physically possible states with three blocks:
  - all blocks on table: 1 state
  - all blocks in one stack: $3! = 6$ states
  - two block stacked, the other separate: $\binom{3}{2}2! = 6$
- With propositional variables, $2^9 - 13 = 499$ states are useless.
- With finite-domain variables, only $27 - 13 = 14$ are useless.

Although useless states are unreachable,
they can introduce "shortcuts" in some heuristics
and thus lead to worse heuristic estimates.

Finite-Domain State Variables

Definition (Finite-Domain State Variable)

A finite-domain state variable is a symbol v with an associated domain $\mathrm{dom}(v)$, which is a finite non-empty set of values.

Let V be a finite set of finite-domain state variables.

A state s over V is an assignment $s : V \to \bigcup_{v \in V} \mathrm{dom}(v)$ such that $s(v) \in \mathrm{dom}(v)$ for all $v \in V$.

A formula over V is a propositional logic formula whose atomic propositions are of the form $v = d$ where $v \in V$ and $d \in \mathrm{dom}(v)$.

Slightly extending propositional logic, we treat states s over finite-domain variables as logical interpretations where $s \models v = d$ iff $s(v) = d$.

Example: Finite-Domain State Variables

Example

Consider finite-domain variables $V = \{\mathit{location}, \mathit{bike}\}$ with $\mathrm{dom}(\mathit{location}) = \{\text{at-home}, \text{in-front-of-uni}, \text{in-lecture}\}$ and $\mathrm{dom}(\mathit{bike}) = \{\text{locked}, \text{unlocked}, \text{stolen}\}$.

Consider state $s = \{\mathit{location} \mapsto \text{at-home}, \mathit{bike} \mapsto \mathit{locked}\}$.

Does $s \models (\mathit{location} = \text{at-home} \wedge \neg \mathit{bike} = \text{stolen})$ hold?

Reminder: Syntax of Operators

Definition (Operator)

An operator o over state variables V is an object with three properties:

- a precondition $pre(o)$, a formula over V
- an effect $eff(o)$ over V
- a cost $cost(o) \in \mathbb{R}_0^+$

Only necessary adaptation: What is an effect?

Example

$\langle location = \text{in-front-of-uni},$
$location := \text{in-lecture} \wedge (bike = \text{unlocked} \rhd bike := \text{stolen}), 1\rangle$

Syntax of Effects

Definition (Effect over Finite-Domain State Variables)

Effects over finite-domain state variables V are inductively defined as follows:

- $\top$ is an effect (empty effect).
- If $v \in V$ is a finite-domain state variable and $d \in \mathrm{dom}(v)$, then $v := d$ is an effect (atomic effect).
- If e and e' are effects, then $(e \wedge e')$ is an effect (conjunctive effect).
- If χ is a formula over V and e is an effect, then $(\chi \rhd e)$ is an effect (conditional effect).

Parentheses can be omitted when this does not cause ambiguity.

only change compared to propositional case: atomic effects

Semantics of Effects: Effect Conditions

Definition (Effect Condition with Finite-Domain Representation)

Let $v := d$ be an atomic effect, and let e be an effect.

The effect condition $\mathit{effcond}(v := d, e)$ under which $v := d$ triggers given the effect e is a propositional formula defined as follows:

- $\mathit{effcond}(v := d, \top) = \bot$
- $\mathit{effcond}(v := d, v := d) = \top$
- $\mathit{effcond}(v := d, v' := d') = \bot$
  for atomic effects with $v' \neq v$ or $d' \neq d$
- $\mathit{effcond}(v := d, (e \wedge e')) =$
  $(\mathit{effcond}(v := d, e) \vee \mathit{effcond}(v := d, e'))$
- $\mathit{effcond}(v := d, (\chi \rhd e)) = (\chi \wedge \mathit{effcond}(v := d, e))$

Same definition as for propositional tasks,
we just use the adapted definition of atomic effects.

Conflicting Effects and Consistency Condition

- What should an effect of the form $v := a \wedge v := b$ mean?
- For finite-domain representations, the accepted semantics is to make this illegal, i.e., to make an operator inapplicable if it would lead to conflicting effects.

Definition (Consistency Condition)

Let e be an effect over finite-domain state variables V.

The consistency condition for e, $\mathit{consist}(e)$ is defined as

$$\bigwedge_{v \in V} \bigwedge_{d,d' \in \mathrm{dom}(v), d \neq d'} \neg(\mathit{effcond}(v := d, e) \wedge \mathit{effcond}(v := d', e)).$$

How did we handle conflicting effects in propositional planning tasks?

Semantics of Operators: Finite-Domain Case

Definition (Applicable, Resulting State)

Let V be a set of finite-domain state variables
and e be an effect over V.
If $s \models \mathit{consist}(e)$, the resulting state of applying e in s,
written $s[\![e]\!]$, is the state s' defined as follows for all $v \in V$:

$$s'(v) = \begin{cases} d & \text{if } s \models \mathit{effcond}(v := d, e) \text{ for some } d \in \mathrm{dom}(v) \\ s(v) & \text{otherwise} \end{cases}$$

Let o be an operator over V.
Operator o is applicable in s if $s \models \mathit{pre}(o) \wedge \mathit{consist}(\mathit{eff}(o))$.

If o is applicable in s, the resulting state of applying o in s,
written $s[\![o]\!]$, is the state $s[\![\mathit{eff}(o)]\!]$.

Applying Operators: Example

Example

$V = \{\mathit{location}, \mathit{bike}\}$ with
$\mathrm{dom}(\mathit{location}) = \{\text{at-home}, \text{in-front-of-uni}, \text{in-lecture}\}$ and
$\mathrm{dom}(\mathit{bike}) = \{\text{locked}, \text{unlocked}, \text{stolen}\}$.

State $s = \{\mathit{location} \mapsto \text{in-front-of-uni}, \mathit{bike} \mapsto \text{unlocked}\}$

$o = \langle \mathit{location} = \text{in-front-of-uni}, \mathit{location} := \text{at-home}, 1 \rangle$

$o' = \langle \mathit{location} = \text{in-front-of-uni},$
$\quad \mathit{location} := \text{in-lecture} \wedge (\mathit{bike} = \text{unlocked} \rhd \mathit{bike} := \text{stolen}), 1 \rangle$

What is $s[\![o]\!]$? What is $s[\![o']\!]$?

FDR Planning Tasks

Definition (Planning Task)

An FDR planning task (or planning task in finite-domain representation) is a 4-tuple $\Pi = \langle V, I, O, \gamma \rangle$ where

- V is a finite set of finite-domain state variables,
- I is an assignment for V called the initial state,
- O is a finite set of operators over V, and
- γ is a formula over V called the goal.

Apart from the variables, this is the same definition as for propositional planning tasks, but the underlying concepts have been adapted.

Mapping FDR Planning Tasks to Transition Systems

Definition (Transition System Induced by an FDR Planning Task)

The FDR planning task $\Pi = \langle V, I, O, \gamma \rangle$ induces the transition system $\mathcal{T}(\Pi) = \langle S, L, c, T, s_0, S_\star \rangle$, where

- S is the set of all states over V,
- L is the set of operators O,
- $c(o) = cost(o)$ for all operators $o \in O$,
- $T = \{\langle s, o, s' \rangle \mid s \in S,\ o \text{ applicable in } s,\ s' = s[\![o]\!]\}$,
- $s_0 = I$, and
- $S_\star = \{s \in S \mid s \models \gamma\}$.

Exactly the same definition as for propositional planning tasks, but the underlying concepts have been adapted.

E1.2 Equivalence and Normal Forms

Equivalence and Flat Operators

- The definitions of equivalent effects/operators and flat effects/operators apply equally to finite-domain representation.
- The same is true for the equivalence transformations.

You find the definitions and transformations in Chapter B4.

Conflict-Free Operators

Definition (Conflict-Free)

An effect e over finite-domain state variables V
is called conflict-free if $\mathit{effcond}(v := d, e) \wedge \mathit{effcond}(v := d', e)$
is unsatisfiable for all $v \in V$ and $d, d' \in \mathrm{dom}(v)$ with $d \neq d'$.

An operator o is called conflict-free if $\mathit{eff}(o)$ is conflict-free.

Note: $\mathit{consist}(e) \equiv \top$ for conflict-free e.

Algorithm to make given operator o conflict-free:

- replace $\mathit{pre}(o)$ with $\mathit{pre}(o) \wedge \mathit{consist}(\mathit{eff}(o))$
- replace all atomic effects $v := d$ by $(\mathit{consist}(\mathit{eff}(o)) \rhd v := d)$

The resulting operator o' is conflict-free and $o \equiv o'$.

SAS^+ Operators and Planning Tasks

Definition (SAS^+ Operator)

An operator o of an FDR planning task is a SAS^+ operator if

- $pre(o)$ is a satisfiable conjunction of atoms, and
- $eff(o)$ is a conflict-free conjunction of atomic effects.

Definition (SAS^+ Planning Task)

An FDR planning task $\langle V, O, I, \gamma \rangle$ is a SAS^+ planning task
if all operators $o \in O$ are SAS^+ operators
and γ is a satisfiable conjunction of atoms.

Note: SAS^+ operators are conflict-free and flat.

SAS^+ Operators: Remarks

- Every SAS^+ operator is of the form

$$\langle v_1 = d_1 \wedge \cdots \wedge v_n = d_n, \quad v'_1 := d'_1 \wedge \cdots \wedge v'_m := d'_m \rangle$$

  where all v_i are distinct and all v'_j are distinct.
- Often, SAS^+ operators o are described via two sets of partial assignments:
  - the preconditions $\{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$
  - the effects $\{v'_1 \mapsto d'_1, \ldots, v'_m \mapsto d'_m\}$

SAS^+ vs. STRIPS

- SAS^+ is an analogue of STRIPS planning tasks for FDR, but there is no special role of "positive" conditions.
- Apart from this difference, all comments for STRIPS apply analogously.
- If all variable domains are binary, SAS^+ is essentially STRIPS with negation.

SAS^+

Derives from SAS = Simplified Action Structures
(Bäckström & Klein, 1991)

E1.3 Summary

Summary

- Planning tasks in finite-domain representation (FDR) are an alternative to propositional planning tasks.
- FDR tasks are often more compact (have fewer states).
- This makes many planning algorithms more efficient when working with a finite-domain representation.
- SAS^+ tasks are a restricted form of FDR tasks where only conjunctions of atoms are allowed in the preconditions, effects and goal.
  No conditional effects are allowed.

Planning and Optimization

E2. Invariants and Mutexes

Malte Helmert and Gabriele Röger

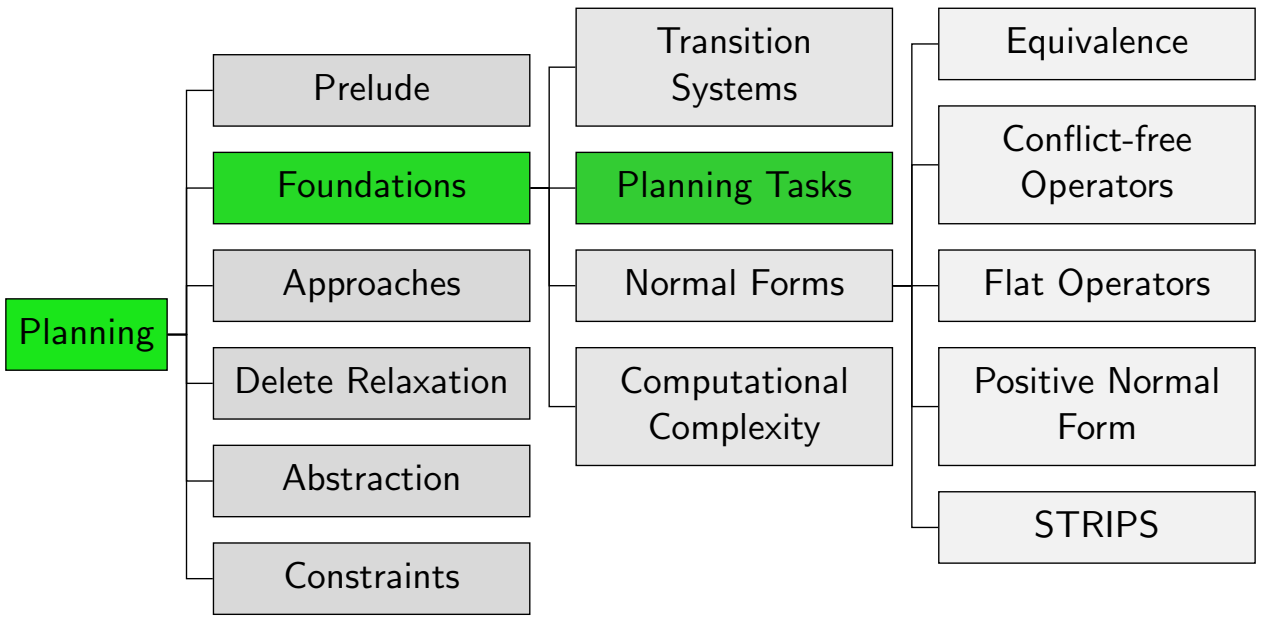
Universität Basel

November 3, 2025

Planning and Optimization

November 3, 2025 — E2. Invariants and Mutexes

Content of the Course



E2.1 Invariants

Invariants

- When we as humans reason about planning tasks, we implicitly make use of "obvious" properties of these tasks.
  - Example: we are never in two places at the same time
- We can represent such properties as logical formulas φ that are true in all reachable states.
  - Example: $\varphi = \neg(\mathit{at\text{-}uni} \wedge \mathit{at\text{-}home})$
- Such formulas are called invariants of the task.

Invariants: Definition

Definition (Invariant)

An invariant of a planning task Π with state variables V
is a logical formula φ over V such that $s \models \varphi$
for all reachable states s of Π.

E2.2 Computing Invariants

Computing Invariants

How does an automated planner come up with invariants?

- Theoretically, testing if a formula φ is an invariant is as hard as planning itself.
  $\rightsquigarrow$ proof idea: a planning task is unsolvable iff the negation of its goal is an invariant
- Still, many practical invariant synthesis algorithms exist.
- To remain efficient (= polynomial-time), these algorithms only compute a subset of all useful invariants.
  $\rightsquigarrow$ sound, but not complete
- Empirically, they tend to at least find the "obvious" invariants of a planning task.

Invariant Synthesis Algorithms

Most algorithms for generating invariants are based on the generate-test-repair approach:

- Generate: Suggest some invariant candidates, e.g., by enumerating all possible formulas φ of a certain size.
- Test: Try to prove that φ is indeed an invariant. Usually done inductively:
  1. Test that initial state satisfies φ.
  2. Test that if φ is true in the current state, it remains true after applying a single operator.
- Repair: If invariant test fails, replace candidate φ by a weaker formula, ideally exploiting why the proof failed.

Invariant Synthesis: References

We will not cover invariant synthesis algorithms in this course.

Literature on invariant synthesis:

- DISCOPLAN (Gerevini & Schubert, 1998)
- TIM (Fox & Long, 1998)
- Edelkamp & Helmert's algorithm (1999)
- Bonet & Geffner's algorithm (2001)
- Rintanen's algorithm (2008)
- Rintanen's algorithm for schematic invariants (2017)

Exploiting Invariants

Invariants have many uses in planning:

- Regression search (C3–C4):
  Prune subgoals that violate (are inconsistent with) invariants.
- Planning as satisfiability (C5–C6):
  Add invariants to a SAT encoding of a planning task
  to get tighter constraints.
- Proving unsolvability:
  If φ is an invariant such that $\varphi \wedge \gamma$ is unsatisfiable,
  the planning task with goal γ is unsolvable.
- Finite-Domain Reformulation:
  Derive a more compact FDR representation (equivalent, but with fewer states) from a given propositional planning task.

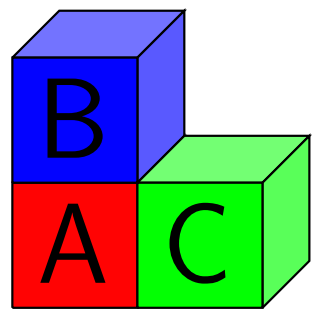
We now discuss the last point because it connects
to our discussion of propositional vs. FDR planning tasks.

E2.3 Mutexes

Reminder: Blocks World (Propositional Variables)

Example

$$
\begin{aligned}
s(\textit{A-on-B}) &= \mathbf{F} \\
s(\textit{A-on-C}) &= \mathbf{F} \\
s(\textit{A-on-table}) &= \mathbf{T} \\
s(\textit{B-on-A}) &= \mathbf{T} \\
s(\textit{B-on-C}) &= \mathbf{F} \\
s(\textit{B-on-table}) &= \mathbf{F} \\
s(\textit{C-on-A}) &= \mathbf{F} \\
s(\textit{C-on-B}) &= \mathbf{F} \\
s(\textit{C-on-table}) &= \mathbf{T}
\end{aligned}
$$



$\rightsquigarrow 2^9 = 512$ states

Reminder: Blocks World (Finite-Domain Variables)

Example

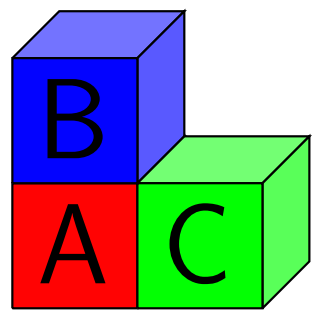
Use three finite-domain state variables:

- *below-a*: $\{\mathsf{b}, \mathsf{c}, \mathsf{table}\}$
- *below-b*: $\{\mathsf{a}, \mathsf{c}, \mathsf{table}\}$
- *below-c*: $\{\mathsf{a}, \mathsf{b}, \mathsf{table}\}$

$s(\textit{below-a}) = \mathsf{table}$

$s(\textit{below-b}) = \mathsf{a}$

$s(\textit{below-c}) = \mathsf{table}$



$\rightsquigarrow 3^3 = 27$ states

Task Reformulation

- Common modeling languages (like PDDL) often give us propositional tasks.
- More compact FDR tasks are often desirable.
- Can we do an automatic reformulation?

Mutexes

Invariants that take the form of binary clauses are called mutexes because they express that certain variable assignments cannot be simultaneously true (are mutually exclusive).

Example (Blocks World)

The invariant $\neg A\text{-}on\text{-}B \vee \neg A\text{-}on\text{-}C$ states that $A\text{-}on\text{-}B$ and $A\text{-}on\text{-}C$ are mutex.

We say that a set of literals is a mutex group if every subset of two literals is a mutex.

Example (Blocks World)

$\{A\text{-}on\text{-}B, A\text{-}on\text{-}C, A\text{-}on\text{-}table\}$ is a mutex group.

Encoding Mutex Groups as Finite-Domain Variables

Let $G = \{\ell_1, \ldots, \ell_n\}$ be a mutex group over n different propositional state variables $V_G = \{v_1, \ldots, v_n\}$.

Then a single finite-domain state variable v_G with $\mathrm{dom}(v_G) = \{\ell_1, \ldots, \ell_n, \text{none}\}$ can replace the n variables V_G:

- $s(v_G) = \ell_i$ represents situations where (exactly) ℓ_i is true
- $s(v_G) = \text{none}$ represents situations where all ℓ_i are false

Note: We can omit the "none" value if $\ell_1 \vee \cdots \vee \ell_n$ is an invariant.

Mutex Covers

Definition (Mutex Cover)

A mutex cover for a propositional planning task Π is a set of mutex groups $\{G_1, \ldots, G_n\}$ where each variable of Π occurs in exactly one group G_i.

A mutex cover is positive if all literals in all groups are positive.

Note: always exists (use trivial group $\{v\}$ if v otherwise uncovered)

Positive Mutex Covers

In the following, we stick to positive mutex covers for simplicity.

If we have $\neg v$ in G for some group G in the cover, we can reformulate the task to use an "opposite" variable $\hat{v}$ instead, as in the conversion to positive normal form (Chapter B5).

E2.4 Reformulation

Mutex-Based Reformulation of Propositional Tasks

Given a conflict-free propositional planning task Π
with positive mutex cover $\{G_1, \ldots, G_n\}$:

- In all conditions where variable $v \in G_i$ occurs,
  replace v with $v_{G_i} = v$.
- In all effects e where variable $v \in G_i$ occurs,
  - Replace all atomic add effects v with $v_{G_i} := v$
  - Replace all atomic delete effects $\neg v$ with
    $(v_{G_i} = v \wedge \neg \bigvee_{v' \in G_i \setminus \{v\}} \mathit{effcond}(v', e)) \rhd v_{G_i} :=$ none

This results in an FDR planning task Π' that is equivalent to Π
(without proof).

Note: the conditional effects encoding delete effects
can often be simplified away to an unconditional or empty effect.

And Back?

- It can also be useful to reformulate an FDR task into a propositional task.
- For example, we might want positive normal form, which requires a propositional task.
- Key idea: each variable/value combination $v = d$ becomes a separate propositional state variable $\langle v, d \rangle$

Converting FDR Tasks into Propositional Tasks

Definition (Induced Propositional Planning Task)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a conflict-free FDR planning task.
The induced propositional planning task Π'
is the propositional planning task $\Pi' = \langle V', I', O', \gamma' \rangle$, where

- $V' = \{\langle v, d \rangle \mid v \in V, d \in \mathrm{dom}(v)\}$
- $I'(\langle v, d \rangle) = \mathbf{T}$ iff $I(v) = d$
- O' and γ' are obtained from O and γ by
  - replacing each atomic formula $v = d$ by the proposition $\langle v, d \rangle$
  - replacing each atomic effect $v := d$ by the effect $\langle v, d \rangle \wedge \bigwedge_{d' \in \mathrm{dom}(v) \setminus \{d\}} \neg \langle v, d' \rangle$.

Notes:

- Again, simplifications are often possible to avoid introducing so many delete effects.
- SAS^+ tasks induce STRIPS tasks.

E2.5 Summary

Summary (1)

- Invariants are common properties of all reachable states, expressed as formulas.
- A number of algorithms for computing invariants exist.
- These algorithms will not find all useful invariants (which is too hard), but try to find some useful subset with reasonable (polynomial) computational effort.

Summary (2)

- Mutexes are invariants that express that certain literals are mutually exclusive.
- Mutex covers provide a way to convert a set of propositional state variables into a potentially much smaller set of finite-domain state variables.
- Using mutex covers, we can reformulate propositional tasks as more compact FDR tasks.
- Conversely, we can reformulate FDR tasks as propositional tasks by introducing propositions for each variable/value pair.

Planning and Optimization

E3. Abstractions: Introduction

Malte Helmert and Gabriele Röger

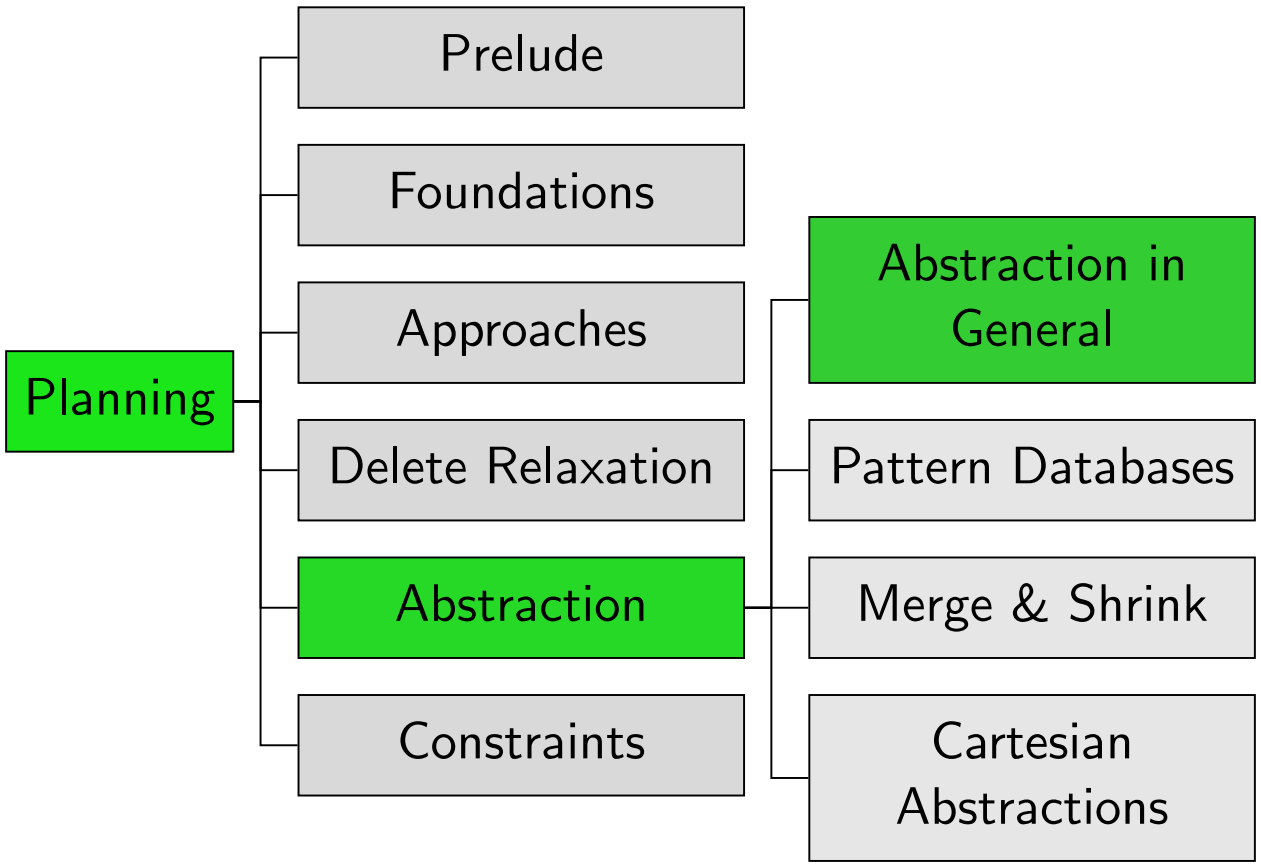
Universität Basel

November 5, 2025

Planning and Optimization

November 5, 2025 — E3. Abstractions: Introduction

Content of the Course



E3.1 Introduction

Coming Up with Heuristics in a Principled Way

General Procedure for Obtaining a Heuristic

Solve a simplified version of the problem.

Major ideas for heuristics in the planning literature:

- delete relaxation
- abstraction
- critical paths
- landmarks
- network flows
- potential heuristics

Heuristics based on abstraction are among the most prominent techniques for optimal planning.

Abstracting a Transition System

Abstracting a transition system means dropping some distinctions between states, while preserving the transition behaviour as much as possible.

- An abstraction of a transition system $\mathcal{T}$ is defined by an abstraction mapping α that defines which states of $\mathcal{T}$ should be distinguished and which ones should not.
- From $\mathcal{T}$ and α, we compute an abstract transition system $\mathcal{T}^\alpha$ which is similar to $\mathcal{T}$, but smaller.
- The abstract goal distances (goal distances in $\mathcal{T}^\alpha$) are used as heuristic estimates for goal distances in $\mathcal{T}$.

Abstracting a Transition System: Example

example from domain-specific heuristic search:

Example (15-Puzzle)

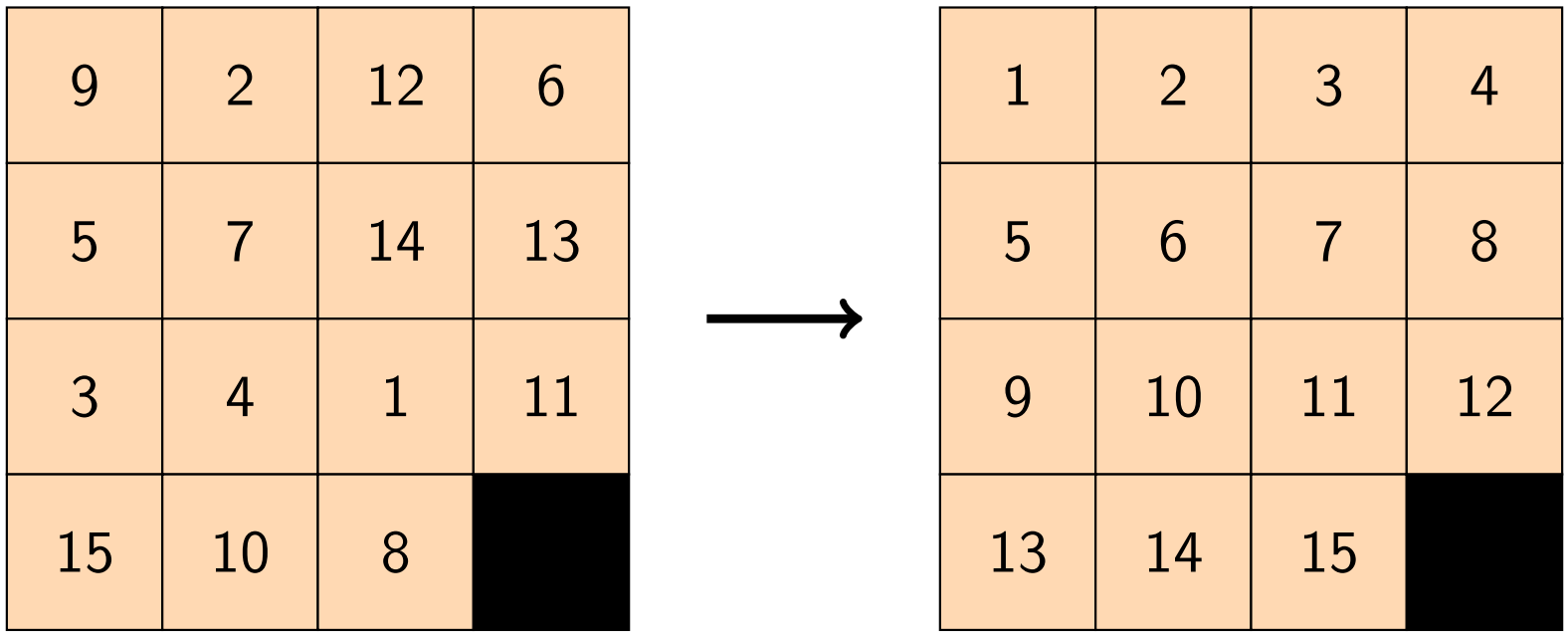
A 15-puzzle state is given by a permutation $\langle b, t_1, \ldots, t_{15} \rangle$ of $\{1, \ldots, 16\}$, where b denotes the blank position and the other components denote the positions of the 15 tiles.

One possible abstraction mapping ignores the precise location of tiles 8–15, i.e., two states are distinguished iff they differ in the position of the blank or one of the tiles 1–7:

$$\alpha(\langle b, t_1, \ldots, t_{15} \rangle) = \langle b, t_1, \ldots, t_7 \rangle$$

The heuristic values for this abstraction correspond to the cost of moving tiles 1–7 to their goal positions.

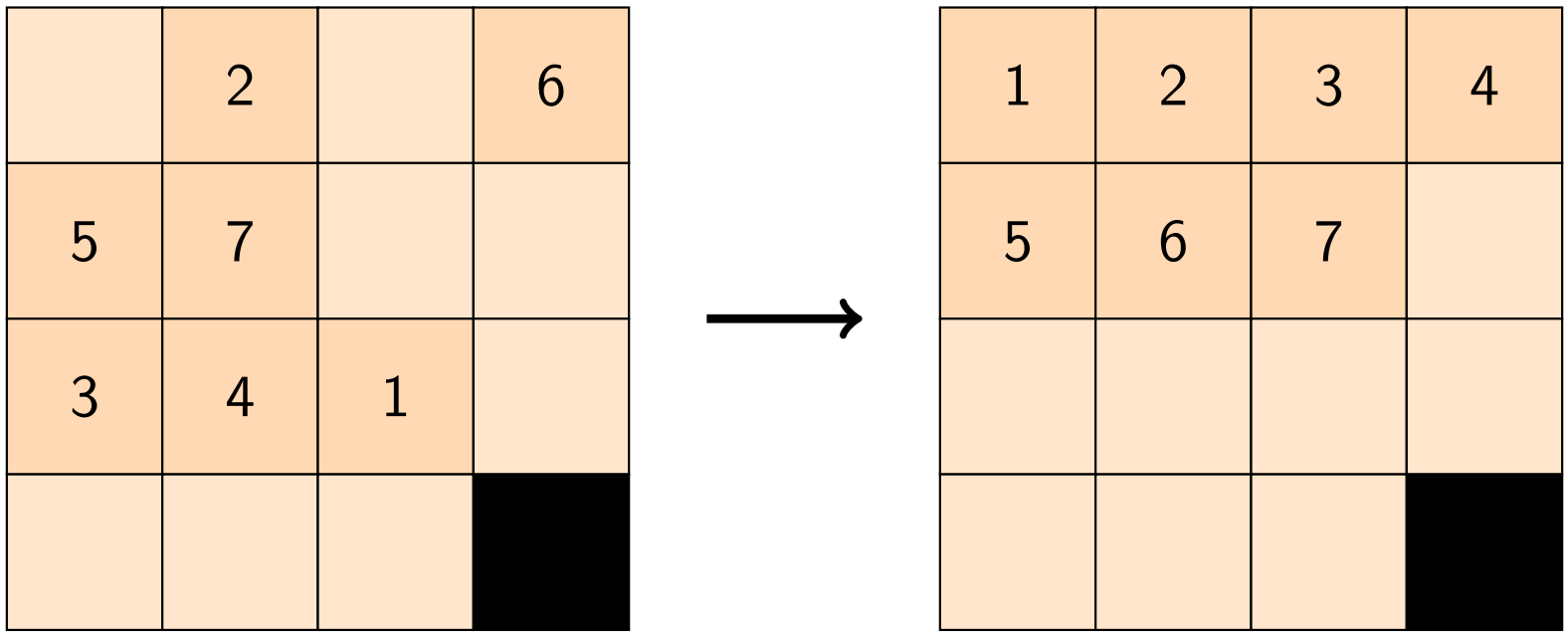
Abstraction Example: 15-Puzzle



real state space:

- $16! = 20922789888000 \approx 2 \cdot 10^{13}$ states
- $\frac{16!}{2} = 10461394944000 \approx 10^{13}$ reachable states

Abstraction Example: 15-Puzzle



abstract state space:

- $16 \cdot 15 \cdot \ldots \cdot 9 = 518918400 \approx 5 \cdot 10^8$ states
- $16 \cdot 15 \cdot \ldots \cdot 9 = 518918400 \approx 5 \cdot 10^8$ reachable states

Computing the Abstract Transition System

Given $\mathcal{T}$ and α, how do we compute $\mathcal{T}^\alpha$?

Requirement

We want to obtain an admissible heuristic.
Hence, $h^*(\alpha(s))$ (in the abstract state space $\mathcal{T}^\alpha$) should never overestimate $h^*(s)$ (in the concrete state space $\mathcal{T}$).

An easy way to achieve this is to ensure that all solutions in $\mathcal{T}$ are also present in $\mathcal{T}^\alpha$:

- If s is a goal state in $\mathcal{T}$, then $\alpha(s)$ is a goal state in $\mathcal{T}^\alpha$.
- If $\mathcal{T}$ has a transition from s to t, then $\mathcal{T}^\alpha$ has a transition from $\alpha(s)$ to $\alpha(t)$ with the same cost.

Computing the Abstract Transition System: Example

Example (15-Puzzle)

In the running example:

- $\mathcal{T}$ has the unique goal state $\langle 16, 1, 2, \ldots, 15 \rangle$.
  - $\rightsquigarrow$ $\mathcal{T}^\alpha$ has the unique goal state $\langle 16, 1, 2, \ldots, 7 \rangle$.
- Let x and y be neighbouring positions in the 4×4 grid.
  $\mathcal{T}$ has a transition from $\langle x, t_1, \ldots, t_{i-1}, y, t_{i+1}, \ldots, t_{15} \rangle$
  to $\langle y, t_1, \ldots, t_{i-1}, x, t_{i+1}, \ldots, t_{15} \rangle$ for all $i \in \{1, \ldots, 15\}$.
  - $\rightsquigarrow$ $\mathcal{T}^\alpha$ has a transition from $\langle x, t_1, \ldots, t_{i-1}, y, t_{i+1}, \ldots, t_7 \rangle$
    to $\langle y, t_1, \ldots, t_{i-1}, x, t_{i+1}, \ldots, t_7 \rangle$ for all $i \in \{1, \ldots, 7\}$.
  - $\rightsquigarrow$ Moreover, $\mathcal{T}^\alpha$ has a transition from $\langle x, t_1, \ldots, t_7 \rangle$
    to $\langle y, t_1, \ldots, t_7 \rangle$ if $y \notin \{t_1, \ldots, t_7\}$.

E3.2 Practical Requirements

Practical Requirements for Abstractions

To be useful in practice, an abstraction heuristic must be efficiently computable. This gives us two requirements for α:

- For a given state s, the abstract state $\alpha(s)$ must be efficiently computable.
- For a given abstract state $\alpha(s)$, the abstract goal distance $h^*(\alpha(s))$ must be efficiently computable.

There are a number of ways of achieving these requirements:

- pattern database heuristics (Culberson & Schaeffer, 1996)
- domain abstractions (Hernádvölgyi and Holte, 2000)
- merge-and-shrink abstractions (Dräger, Finkbeiner & Podelski, 2006)
- Cartesian abstractions (Ball, Podelski & Rajamani, 2001)
- structural patterns (Katz & Domshlak, 2008)

Practical Requirements for Abstractions: Example

Example (15-Puzzle)

In our running example, α can be very efficiently computed: just project the given 16-tuple to its first 8 components.

To compute abstract goal distances efficiently during search, the most common approach is to precompute all abstract goal distances prior to search by performing a backward uniform-cost search from the abstract goal state(s). These distances are then stored in a table (requires $\approx$ 495 MiB RAM).

During search, computing $h^*(\alpha(s))$ is just a table lookup.

This heuristic is an example of a pattern database heuristic.

E3.3 Multiple Abstractions

Multiple Abstractions

- One important practical question is how to come up with a suitable abstraction mapping α.
- Indeed, there is usually a huge number of possibilities, and it is important to pick good abstractions (i.e., ones that lead to informative heuristics).
- However, it is generally not necessary to commit to a single abstraction.

Combining Multiple Abstractions

Maximizing several abstractions:

- Each abstraction mapping gives rise to an admissible heuristic.
- By computing the maximum of several admissible heuristics, we obtain another admissible heuristic which dominates the component heuristics.
- Thus, we can always compute several abstractions and maximize over the individual abstract goal distances.

Adding several abstractions:

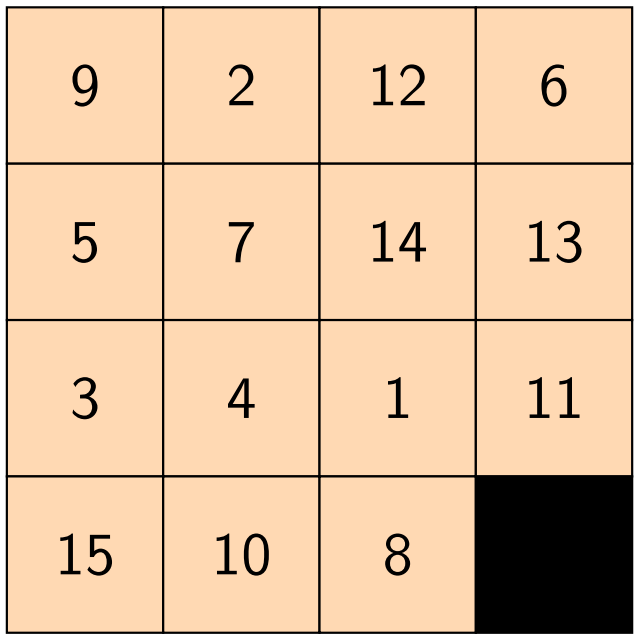
- In some cases, we can even compute the sum of individual estimates and still stay admissible.
- Summation often leads to much higher estimates than maximization, so it is important to understand under which conditions summation of heuristics is admissible.

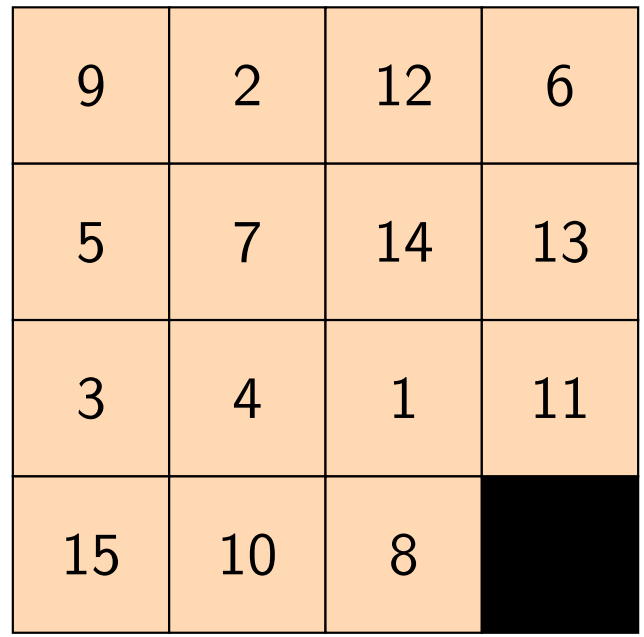
Maximizing Several Abstractions: Example

Example (15-Puzzle)

- mapping to tiles 1–7 was arbitrary
  $\rightsquigarrow$ can use any subset of tiles
- with the same amount of memory required for the tables for the mapping to tiles 1–7, we could store the tables for nine different abstractions to six tiles and the blank
- use maximum of individual estimates

Adding Several Abstractions: Example

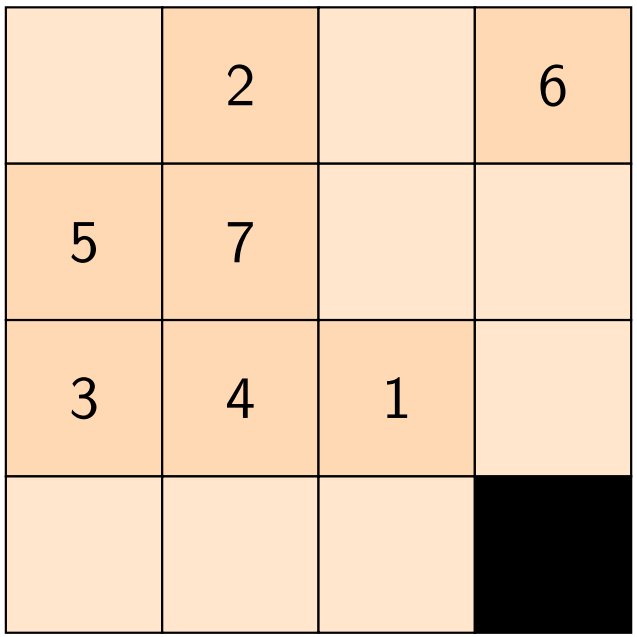


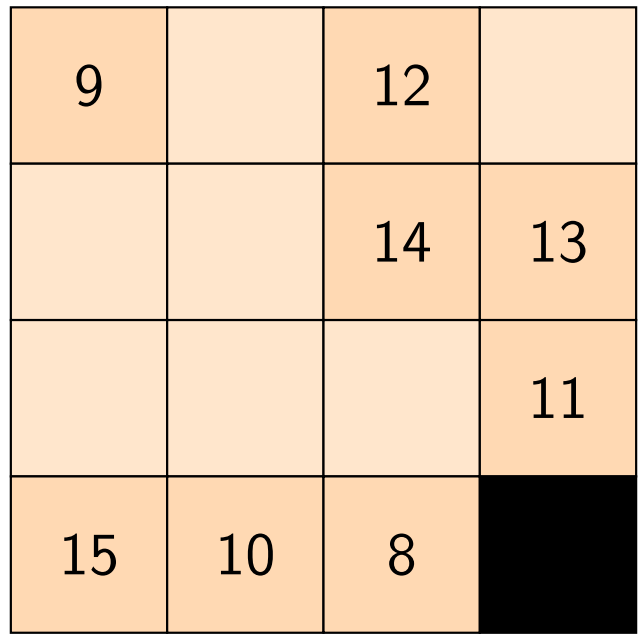


- 1st abstraction: ignore precise location of 8–15
- 2nd abstraction: ignore precise location of 1–7

⇝ Is the sum of the abstraction heuristics admissible?

Adding Several Abstractions: Example

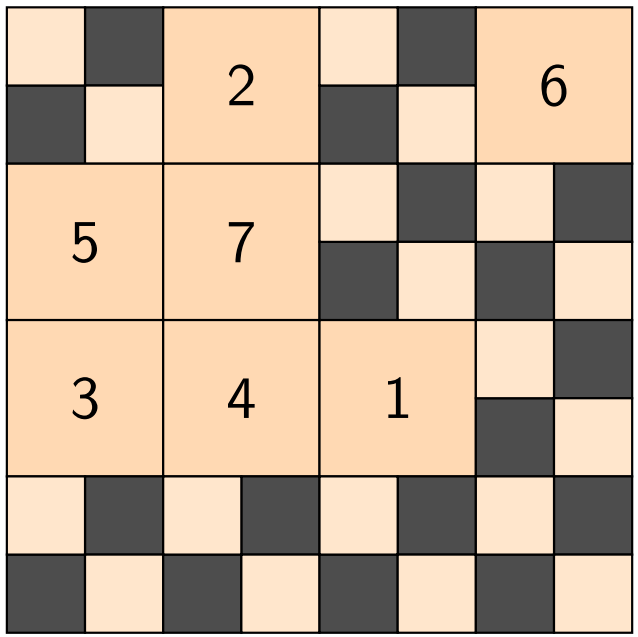


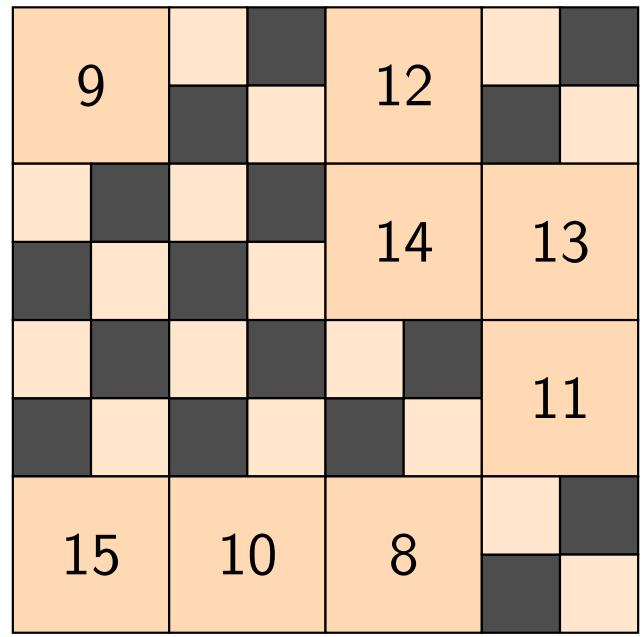


- 1st abstraction: ignore precise location of 8–15
- 2nd abstraction: ignore precise location of 1–7

$\rightsquigarrow$ The sum of the abstraction heuristics is not admissible.

Adding Several Abstractions: Example





- 1st abstraction: ignore precise location of 8–15 and blank
- 2nd abstraction: ignore precise location of 1–7 and blank

$\rightsquigarrow$ The sum of the abstraction heuristics is admissible.

E3.4 Outlook

Our Plan for the Next Lectures

In the following, we take a deeper look at abstractions and their use for admissible heuristics.

In the next two chapters, we formally introduce abstractions and abstraction heuristics and study some of their most important properties.

Afterwards, we discuss some particular classes of abstraction heuristics in detail, namely

- pattern database heuristics,
- merge-and-shrink abstractions and
- Cartesian abstractions.

E3.5 Summary

Summary

- Abstraction is one of the principled ways of deriving heuristics for planning tasks and transition systems in general.
- The key idea is to map states to a smaller abstract transition system $\mathcal{T}^\alpha$ by means of an abstraction function α.
- Goal distances in $\mathcal{T}^\alpha$ are then used as admissible estimates for goal distances in the original transition system.
- To be practical, we must be able to compute abstraction functions and determine abstract goal distances efficiently.
- Often, multiple abstractions are used.
  They can always be maximized admissibly.
- Adding abstraction heuristics is not always admissible.
  When it is, it leads to a stronger heuristic than maximizing.

Planning and Optimization

E4. Abstractions: Formal Definition and Heuristics

Malte Helmert and Gabriele Röger

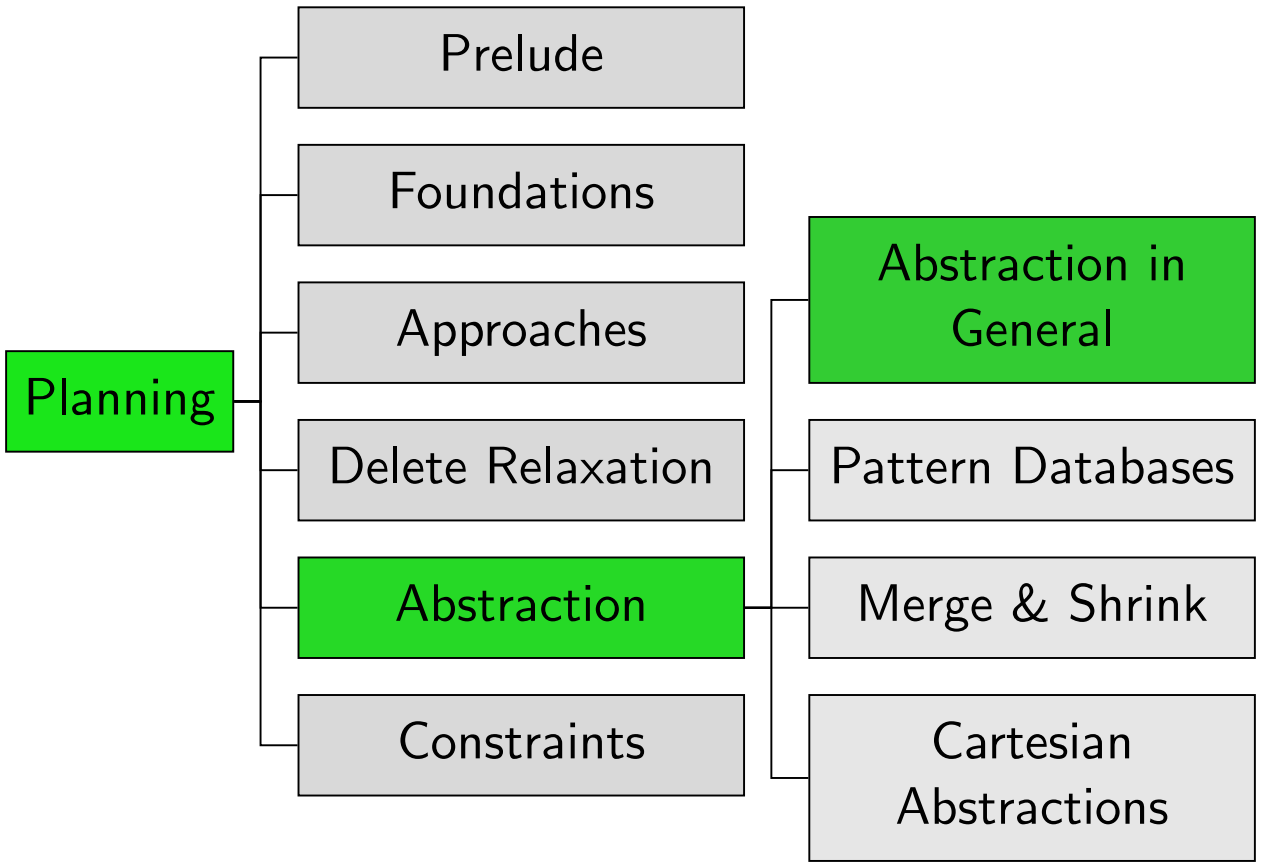
Universität Basel

November 5, 2025

Planning and Optimization

November 5, 2025 — E4. Abstractions: Formal Definition and Heuristics

Content of the Course



E4.1 Reminder: Transition Systems

Transition Systems

Reminder from Chapter B1:

Definition (Transition System)

A transition system is a 6-tuple $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ where

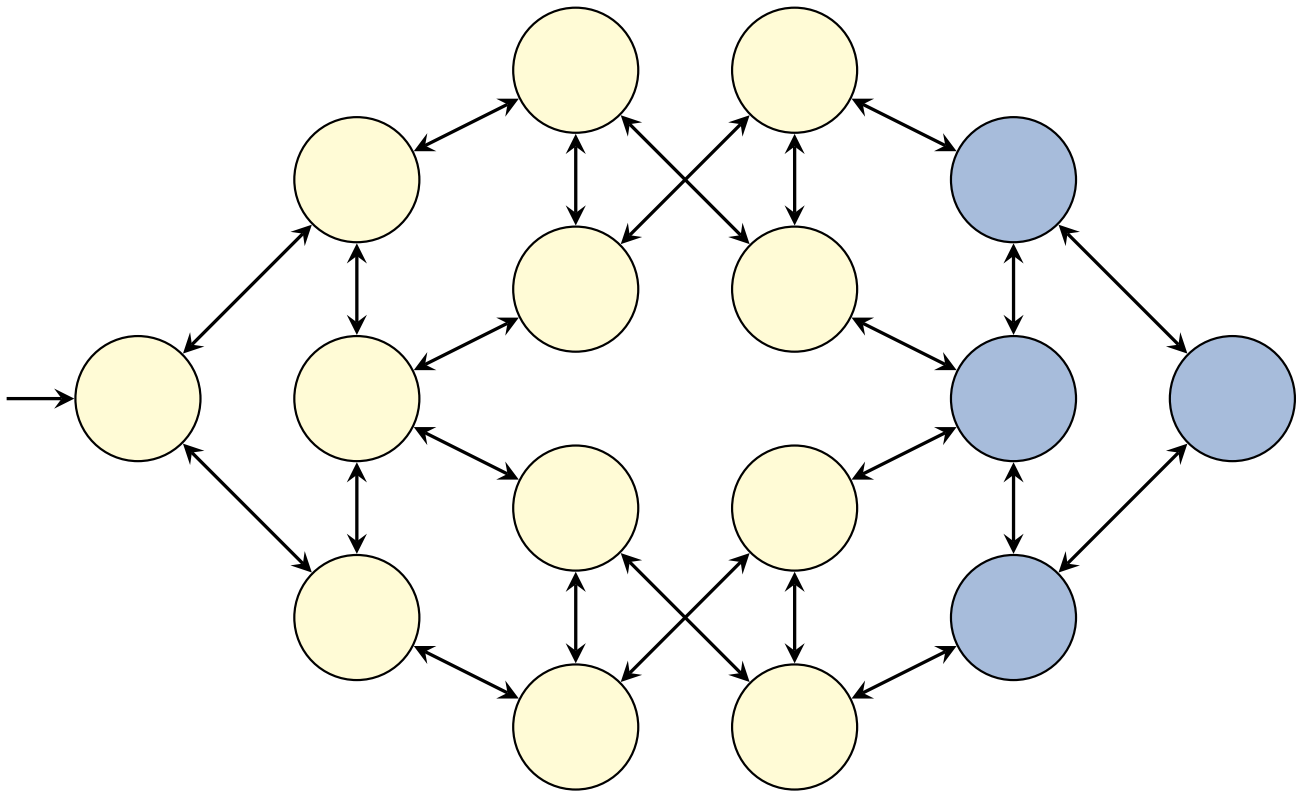
- S is a finite set of states,
- L is a finite set of (transition) labels,
- $c : L \to \mathbb{R}_0^+$ is a label cost function,
- $T \subseteq S \times L \times S$ is the transition relation,
- $s_0 \in S$ is the initial state, and
- $S_\star \subseteq S$ is the set of goal states.

We say that $\mathcal{T}$ has the transition $\langle s, \ell, s' \rangle$ if $\langle s, \ell, s' \rangle \in T$.

We also write this as $s \xrightarrow{\ell} s'$, or $s \to s'$ when not interested in ℓ.

Note: Transition systems are also called state spaces.

Transition Systems: Example



Note: To reduce clutter, our figures often omit arc labels and costs and collapse transitions between identical states. However, these are important for the formal definition of the transition system.

Mapping Planning Tasks to Transition Systems

Reminder from Chapters B3 and E1:

Definition (Transition System Induced by a Planning Task)

The planning task $\Pi = \langle V, I, O, \gamma \rangle$ induces
the transition system $\mathcal{T}(\Pi) = \langle S, L, c, T, s_0, S_\star \rangle$, where

- S is the set of all states over state variables V,
- L is the set of operators O,
- $c(o) = cost(o)$ for all operators $o \in O$,
- $T = \{\langle s, o, s' \rangle \mid s \in S,\ o \text{ applicable in } s,\ s' = s[\![o]\!]\}$,
- $s_0 = I$, and
- $S_\star = \{s \in S \mid s \models \gamma\}$.

(same definition for propositional and finite-domain representation)

Tasks in Finite-Domain Representation

Notes:

- We will focus on planning tasks in finite-domain representation (FDR) while studying abstractions.
- All concepts apply equally to propositional planning tasks.
- However, FDR tasks are almost always used by algorithms in this context because they tend to have fewer useless (physically impossible) states.
- Useless states can hurt the efficiency of abstraction-based algorithms.

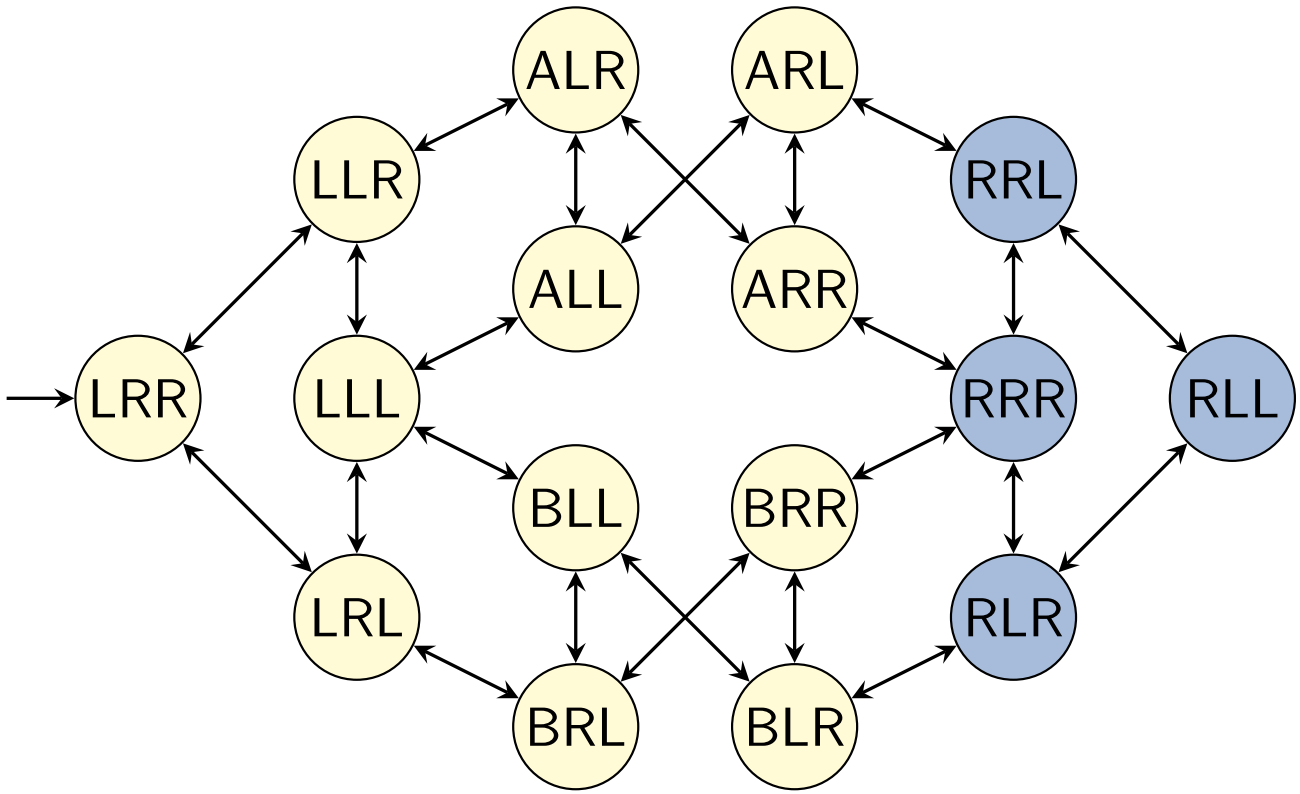
Example Task: One Package, Two Trucks

Example (One Package, Two Trucks)

Consider the following FDR planning task $\langle V, I, O, \gamma \rangle$:

- $V = \{p, t_A, t_B\}$ with
  - $\mathrm{dom}(p) = \{\mathsf{L}, \mathsf{R}, \mathsf{A}, \mathsf{B}\}$
  - $\mathrm{dom}(t_A) = \mathrm{dom}(t_B) = \{\mathsf{L}, \mathsf{R}\}$
- $I = \{p \mapsto \mathsf{L}, t_A \mapsto \mathsf{R}, t_B \mapsto \mathsf{R}\}$
- $O = \{\mathrm{pickup}_{i,j} \mid i \in \{\mathsf{A}, \mathsf{B}\}, j \in \{\mathsf{L}, \mathsf{R}\}\}$
  $\cup \{\mathrm{drop}_{i,j} \mid i \in \{\mathsf{A}, \mathsf{B}\}, j \in \{\mathsf{L}, \mathsf{R}\}\}$
  $\cup \{\mathrm{move}_{i,j,j'} \mid i \in \{\mathsf{A}, \mathsf{B}\}, j, j' \in \{\mathsf{L}, \mathsf{R}\}, j \neq j'\}$, where
  - $\mathrm{pickup}_{i,j} = \langle t_i = j \wedge p = j, p := i, 1 \rangle$
  - $\mathrm{drop}_{i,j} = \langle t_i = j \wedge p = i, p := j, 1 \rangle$
  - $\mathrm{move}_{i,j,j'} = \langle t_i = j, t_i := j', 1 \rangle$
- $\gamma = (p = \mathsf{R})$

Transition System of Example Task



- State $\{p \mapsto i, t_A \mapsto j, t_B \mapsto k\}$ is depicted as ijk.
- Transition labels are again not shown. For example, the transition from LLL to ALL has the label $\text{pickup}_{A,L}$.

E4.2 Abstractions

Abstractions

Definition (Abstraction)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ be a transition system.

An abstraction (also: abstraction function, abstraction mapping) of $\mathcal{T}$ is a function $\alpha : S \to S^\alpha$ defined on the states of $\mathcal{T}$, where S^α is an arbitrary set.

Without loss of generality, we require that α is surjective.

Intuition: α maps the states of $\mathcal{T}$ to another (usually smaller) abstract state space.

Abstract Transition System

Definition (Abstract Transition System)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ be a transition system,
and let $\alpha : S \to S^\alpha$ be an abstraction of $\mathcal{T}$.

The abstract transition system induced by α, in symbols $\mathcal{T}^\alpha$,
is the transition system $\mathcal{T}^\alpha = \langle S^\alpha, L, c, T^\alpha, s_0^\alpha, S_\star^\alpha \rangle$ defined by:

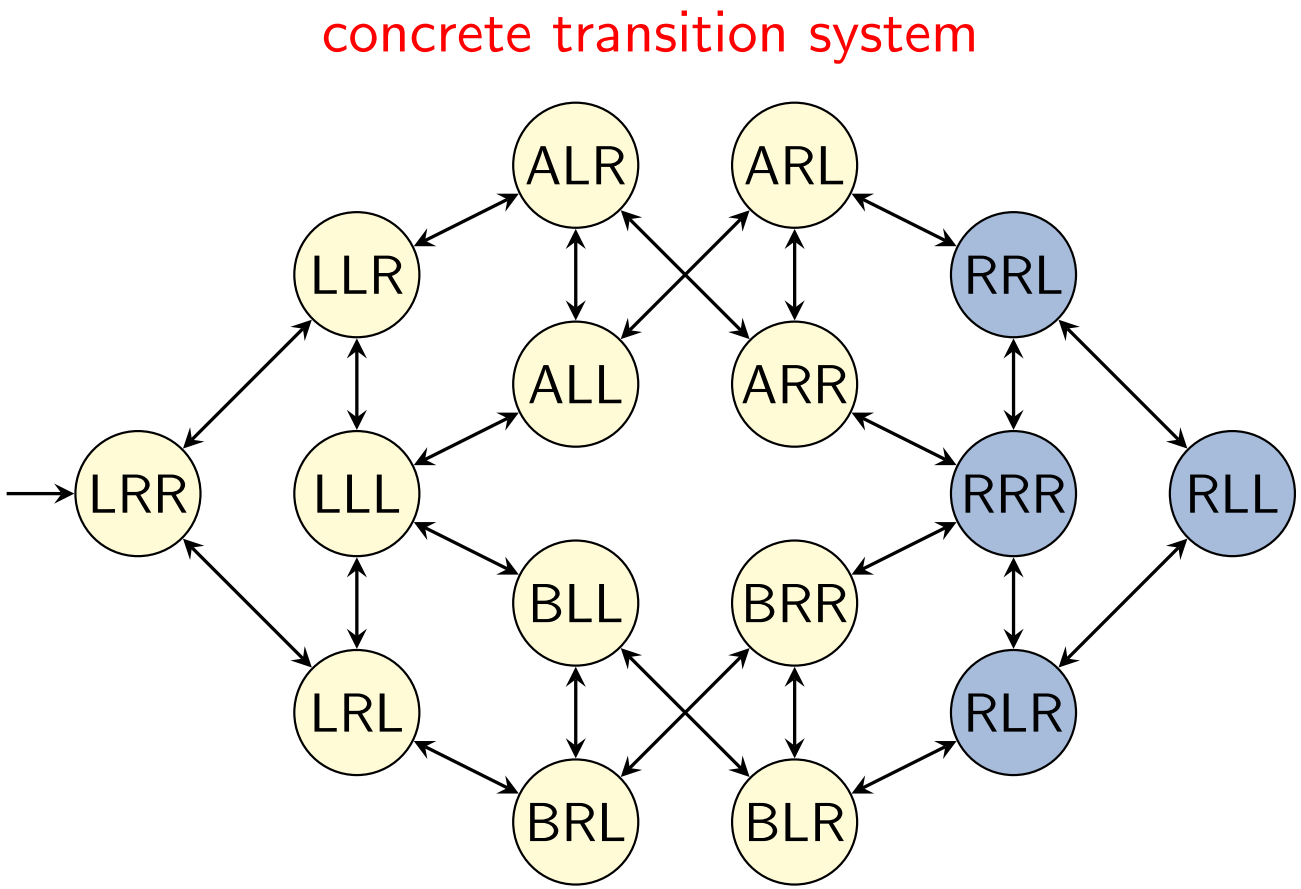
- $T^\alpha = \{\langle \alpha(s), \ell, \alpha(t) \rangle \mid \langle s, \ell, t \rangle \in T\}$
- $s_0^\alpha = \alpha(s_0)$
- $S_\star^\alpha = \{\alpha(s) \mid s \in S_\star\}$

Concrete and Abstract State Space

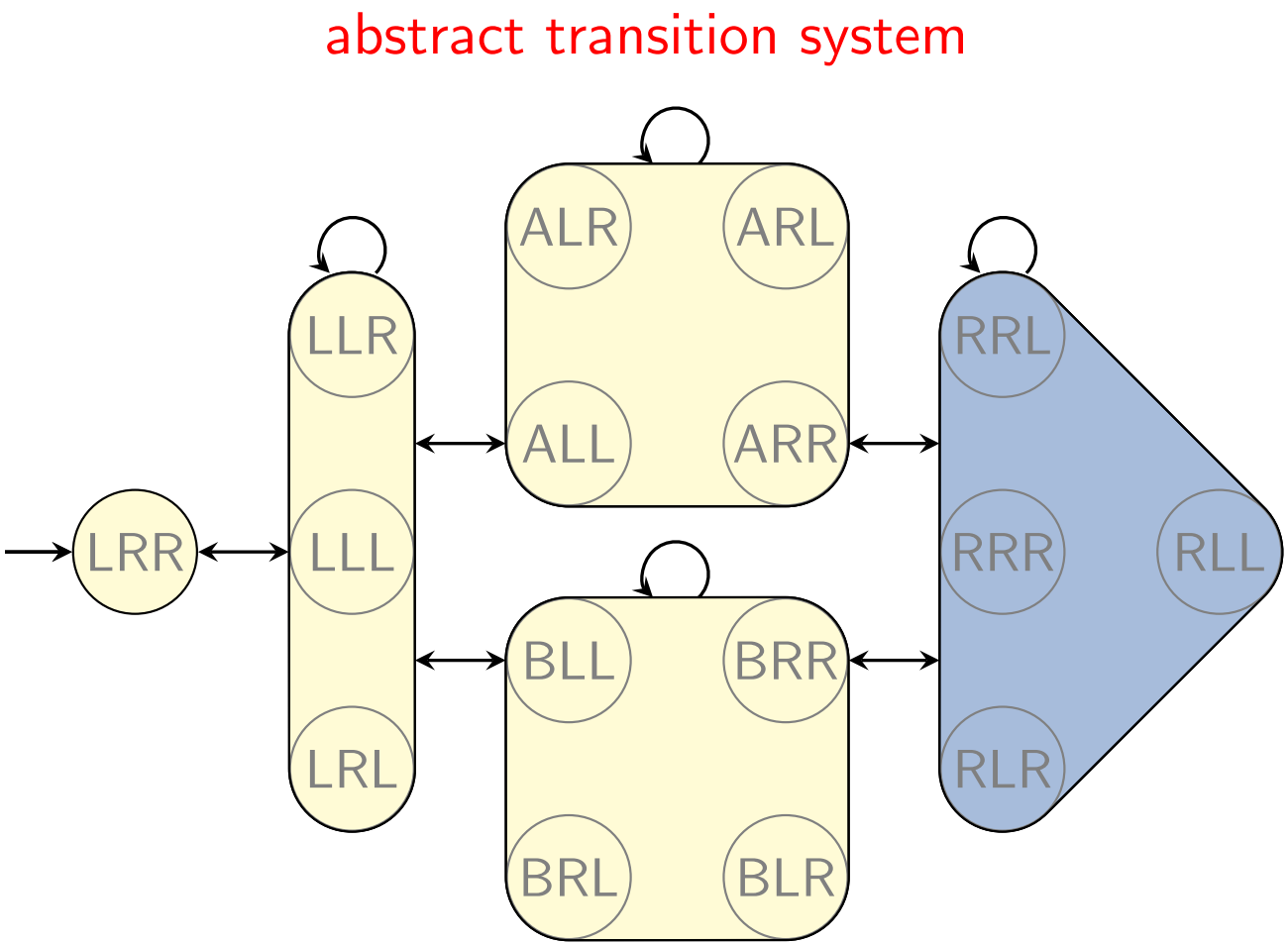
Let $\mathcal{T}$ be a transition system and α be an abstraction of $\mathcal{T}$.

- $\mathcal{T}$ is called the concrete transition system.
- $\mathcal{T}^\alpha$ is called the abstract transition system.
- Similarly: concrete/abstract state space, concrete/abstract transition, etc.

Abstraction: Example



Abstraction: Example



Note: Most arcs represent many parallel transitions.

Strict Homomorphisms

- The abstraction mapping α that transforms $\mathcal{T}$ to $\mathcal{T}^\alpha$ is also called a strict homomorphism from $\mathcal{T}$ to $\mathcal{T}^\alpha$.
- Roughly speaking, in mathematics a homomorphism is a property-preserving mapping between structures.
- A strict homomorphism is one where no additional features are introduced. A non-strict homomorphism in planning would mean that the abstract transition system may include additional transitions and goal states not induced by α.

E4.3 Abstraction Heuristics

Abstraction Heuristics

Definition (Abstraction Heuristic)

Let $\alpha : S \to S^\alpha$ be an abstraction of a transition system $\mathcal{T}$.

The abstraction heuristic induced by α, written h^α,
is the heuristic function $h^\alpha : S \to \mathbb{R}_0^+ \cup \{\infty\}$ defined as

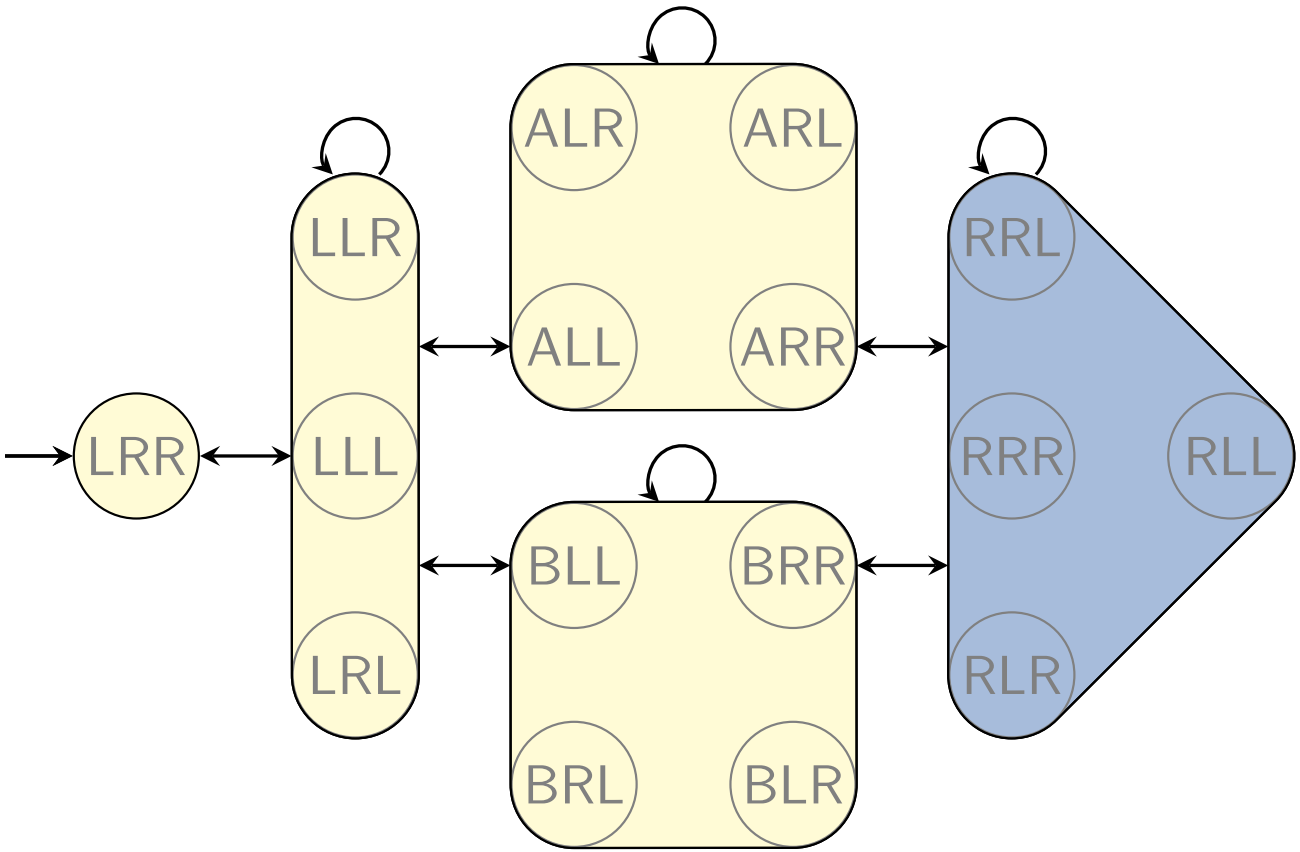
$$h^\alpha(s) = h^*_{\mathcal{T}^\alpha}(\alpha(s)) \quad \text{for all } s \in S,$$

where $h^*_{\mathcal{T}^\alpha}$ denotes the goal distance function in $\mathcal{T}^\alpha$.

Notes:

- $h^\alpha(s) = \infty$ if no goal state of $\mathcal{T}^\alpha$ is reachable from $\alpha(s)$
- We also apply abstraction terminology to planning tasks Π, which stand for their induced transition systems.
  For example, an abstraction of Π is an abstraction of $\mathcal{T}(\Pi)$.

Abstraction Heuristics: Example



$$h^{\alpha}(\{p \mapsto \mathsf{L}, t_{\mathsf{A}} \mapsto \mathsf{R}, t_{\mathsf{B}} \mapsto \mathsf{R}\}) = 3$$

Consistency of Abstraction Heuristics (1)

Theorem (Consistency and Admissibility of h^α)

Let α be an abstraction of a transition system $\mathcal{T}$.
Then h^α is safe, goal-aware, admissible and consistent.

Proof.

We prove goal-awareness and consistency;
the other properties follow from these two.

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$.
Let $\mathcal{T}^\alpha = \langle S^\alpha, L, c, T^\alpha, s_0^\alpha, S_\star^\alpha \rangle$.

Goal-awareness: We need to show that $h^\alpha(s) = 0$ for all $s \in S_\star$, so let $s \in S_\star$. Then $\alpha(s) \in S_\star^\alpha$ by the definition of abstract transition systems, and hence $h^\alpha(s) = h^*_{\mathcal{T}^\alpha}(\alpha(s)) = 0$. ...

Consistency of Abstraction Heuristics (2)

Proof (continued).

Consistency: Consider any state transition $s \xrightarrow{\ell} t$ of $\mathcal{T}$.
We need to show $h^\alpha(s) \leq c(\ell) + h^\alpha(t)$.

By the definition of $\mathcal{T}^\alpha$, we get $\alpha(s) \xrightarrow{\ell} \alpha(t) \in T^\alpha$.
Hence, $\alpha(t)$ is a successor of $\alpha(s)$ in $\mathcal{T}^\alpha$ via the label ℓ.

We get:

$$\begin{aligned} h^\alpha(s) &= h^*_{\mathcal{T}^\alpha}(\alpha(s)) \\ &\leq c(\ell) + h^*_{\mathcal{T}^\alpha}(\alpha(t)) \\ &= c(\ell) + h^\alpha(t), \end{aligned}$$

where the inequality holds because perfect goal distances $h^*_{\mathcal{T}^\alpha}$ are consistent in $\mathcal{T}^\alpha$.
(The shortest path from $\alpha(s)$ to the goal in $\mathcal{T}^\alpha$ cannot be longer than the shortest path from $\alpha(s)$ to the goal via $\alpha(t)$.) □

E4.4 Coarsenings and Refinements

Abstractions of Abstractions

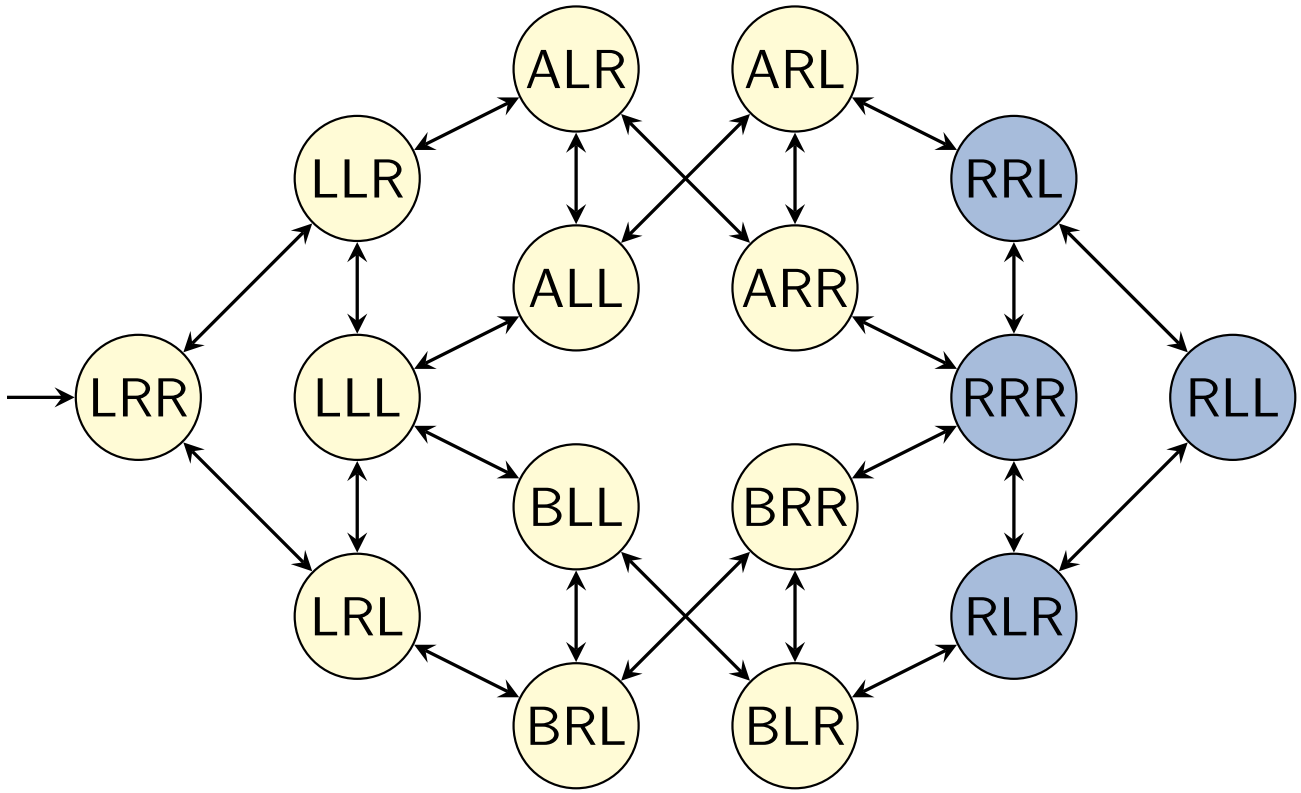
Since abstractions map transition systems to transition systems, they are composable:

- Using a first abstraction $\alpha : S \to S'$, map $\mathcal{T}$ to $\mathcal{T}^\alpha$.
- Using a second abstraction $\beta : S' \to S''$, map $\mathcal{T}^\alpha$ to $(\mathcal{T}^\alpha)^\beta$.

The result is the same as directly using the abstraction $(\beta \circ \alpha)$:

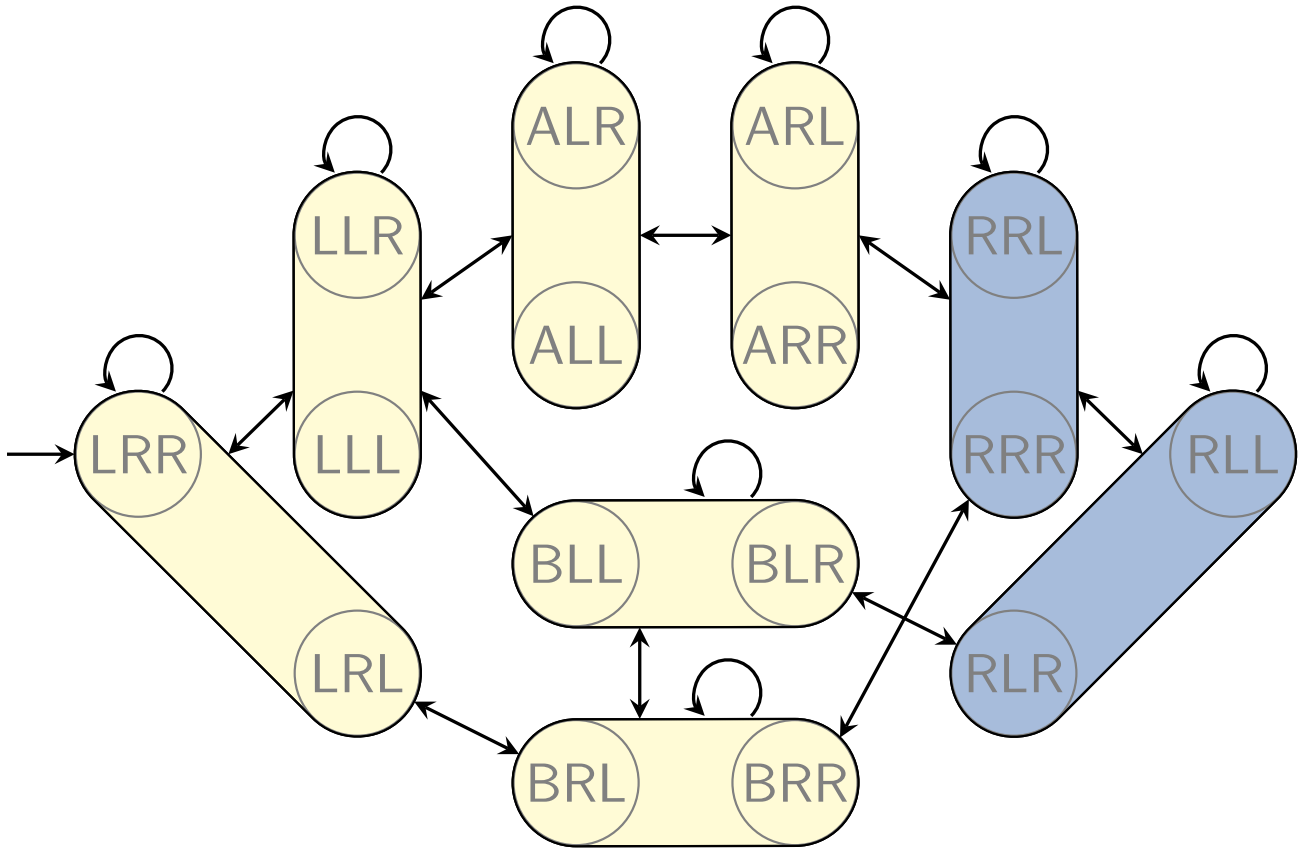
- Let $\gamma : S \to S''$ be defined as $\gamma(s) = (\beta \circ \alpha)(s) = \beta(\alpha(s))$.
- Then $\mathcal{T}^\gamma = (\mathcal{T}^\alpha)^\beta$.

Abstractions of Abstractions: Example (1)



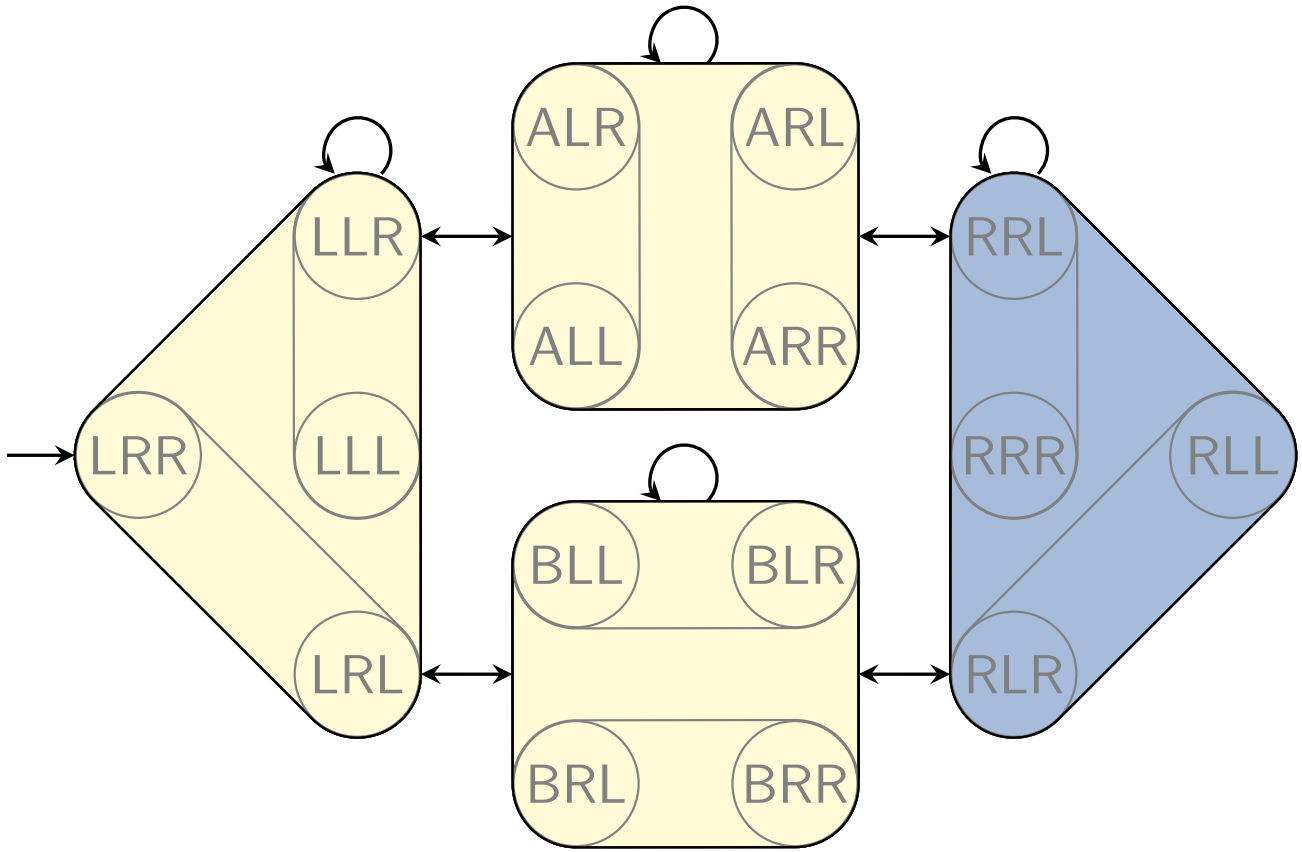
transition system $\mathcal{T}$

Abstractions of Abstractions: Example (2)



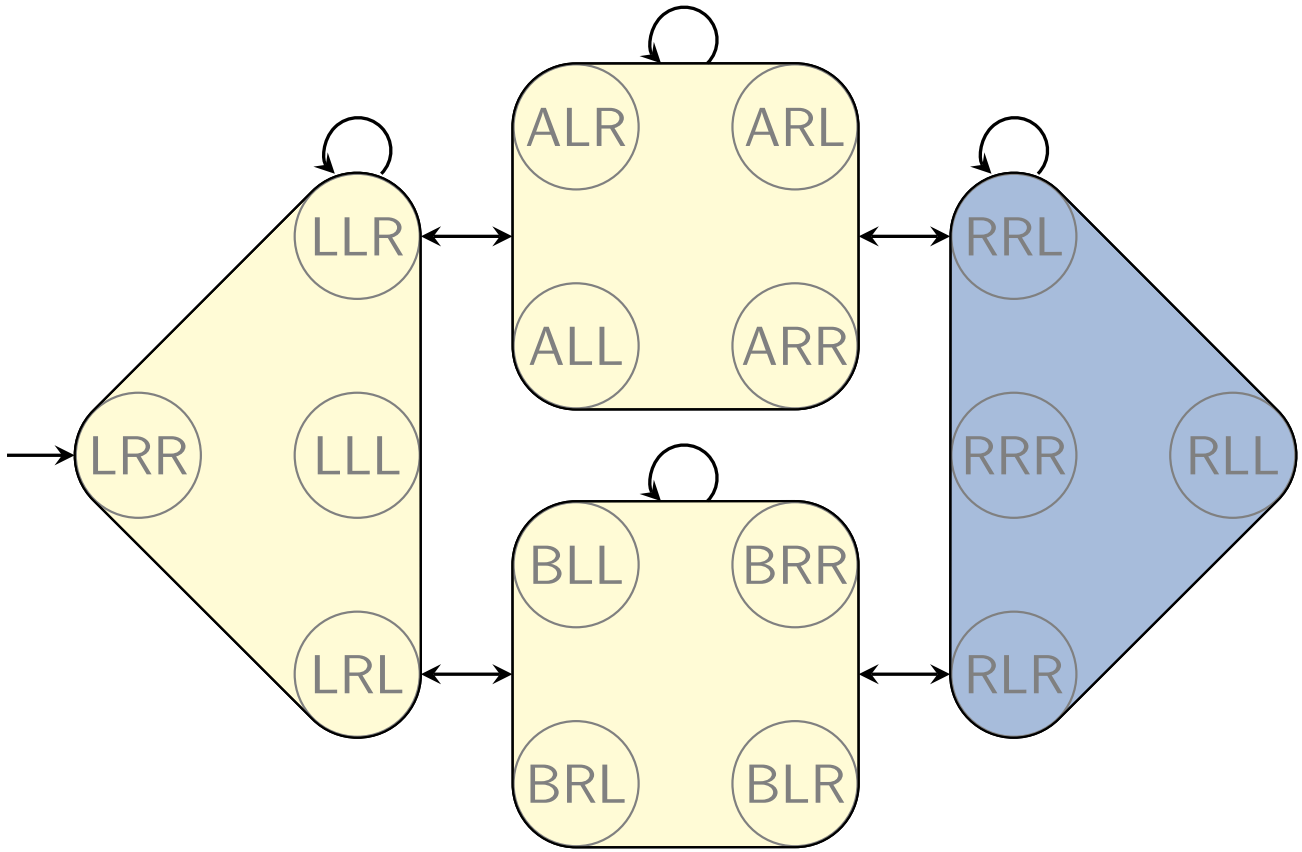
Transition system $\mathcal{T}'$ as an abstraction of $\mathcal{T}$

Abstractions of Abstractions: Example (3)



Transition system $\mathcal{T}''$ as an abstraction of $\mathcal{T}'$

Abstractions of Abstractions: Example (3)



Transition system $\mathcal{T}''$ as an abstraction of $\mathcal{T}$

Coarsenings and Refinements

Definition (Coarsening and Refinement)

Let α and γ be abstractions of the same transition system such that $\gamma = \beta \circ \alpha$ for some function β.

Then γ is called a coarsening of α
and α is called a refinement of γ.

Heuristic Quality of Refinements

Theorem (Heuristic Quality of Refinements)

Let α and γ be abstractions of the same transition system such that α is a refinement of γ.

Then h^α dominates h^γ.

In other words, $h^\gamma(s) \leq h^\alpha(s) \leq h^*(s)$ for all states s.

Heuristic Quality of Refinements: Proof

Proof.

Since α is a refinement of γ,
there exists a function β with $\gamma = \beta \circ \alpha$.

For all states s of Π, we get:

$$\begin{aligned} h^\gamma(s) &= h^*_{\mathcal{T}^\gamma}(\gamma(s)) \\ &= h^*_{\mathcal{T}^\gamma}(\beta(\alpha(s))) \\ &= h^\beta_{\mathcal{T}^\alpha}(\alpha(s)) \\ &\leq h^*_{\mathcal{T}^\alpha}(\alpha(s)) \\ &= h^\alpha(s), \end{aligned}$$

where the inequality holds because $h^\beta_{\mathcal{T}^\alpha}$ is an admissible heuristic in the transition system $\mathcal{T}^\alpha$. □

E4.5 Summary

Summary

- An abstraction is a function α that maps the states S of a transition system to another (usually smaller) set S^α.
- This induces an abstract transition system $\mathcal{T}^\alpha$, which behaves like the original transition system $\mathcal{T}$ except that states mapped to the same abstract state cannot be distinguished.
- Abstractions α induce abstraction heuristics h^α: $h^\alpha(s)$ is the goal distance of $\alpha(s)$ in the abstract transition system.
- Abstraction heuristics are safe, goal-aware, admissible and consistent.
- Abstractions can be composed, leading to coarser vs. finer abstractions. Heuristics for finer abstractions dominate those for coarser ones.

Planning and Optimization

E5. Abstractions: Additive Abstractions

Malte Helmert and Gabriele Röger

Universität Basel

November 10, 2025

Planning and Optimization

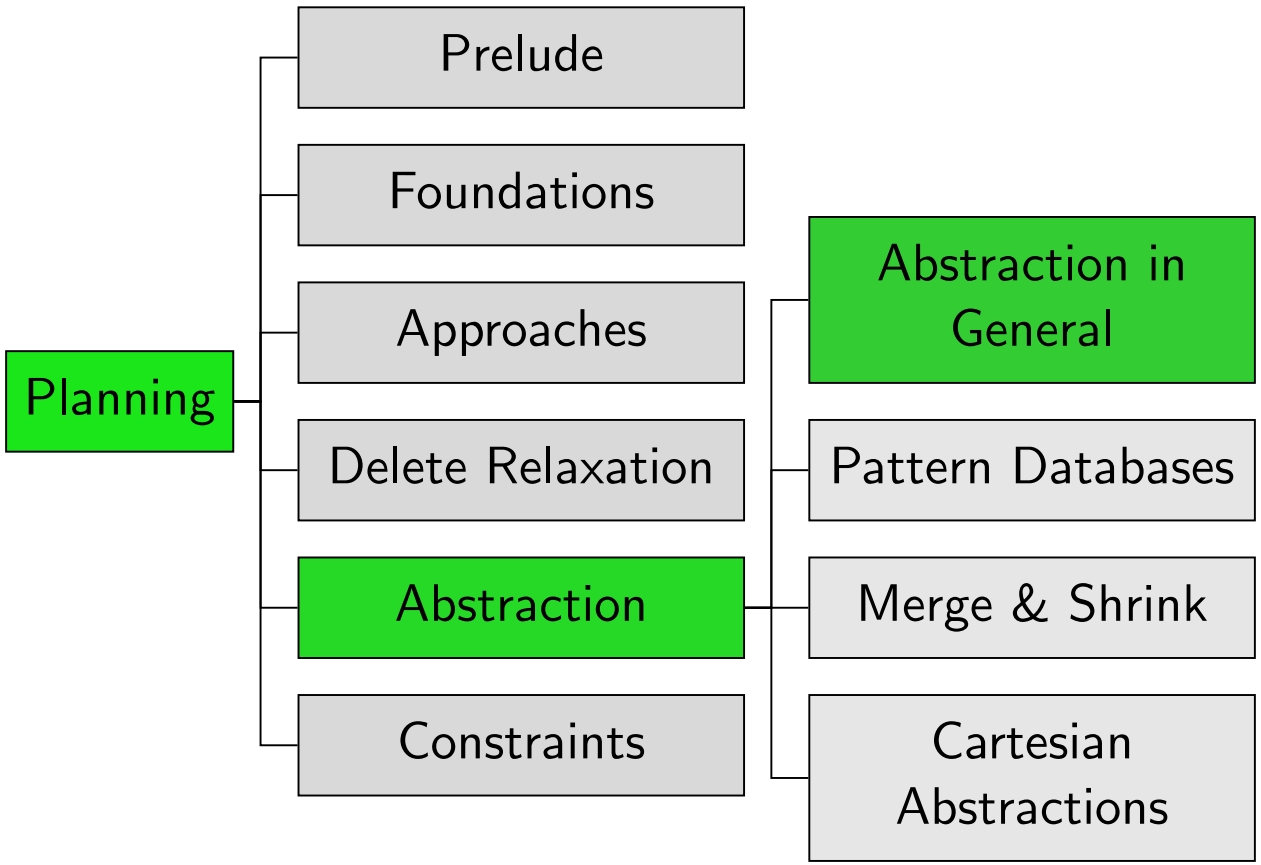
November 10, 2025 — E5. Abstractions: Additive Abstractions

E5.1 Additivity

E5.2 Outlook

E5.3 Summary

Content of the Course



E5.1 Additivity

Orthogonality of Abstractions

Definition (Orthogonal)

Let α_1 and α_2 be abstractions of transition system $\mathcal{T}$.

We say that α_1 and α_2 are orthogonal if for all transitions $s \xrightarrow{\ell} t$ of $\mathcal{T}$, we have $\alpha_1(s) = \alpha_1(t)$ or $\alpha_2(s) = \alpha_2(t)$.

Affecting Transition Labels

Definition (Affecting Transition Labels)

Let $\mathcal{T}$ be a transition system, and let ℓ be one of its labels.

We say that ℓ affects $\mathcal{T}$ if $\mathcal{T}$ has a transition $s \xrightarrow{\ell} t$ with $s \neq t$.

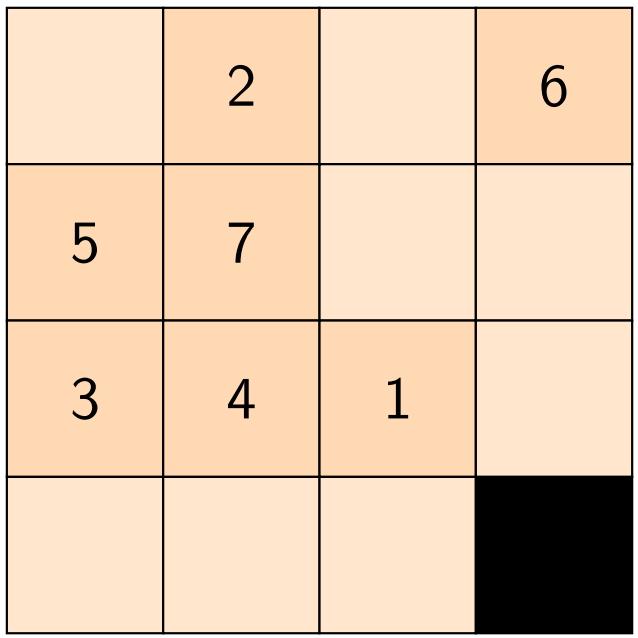
Theorem (Affecting Labels vs. Orthogonality)

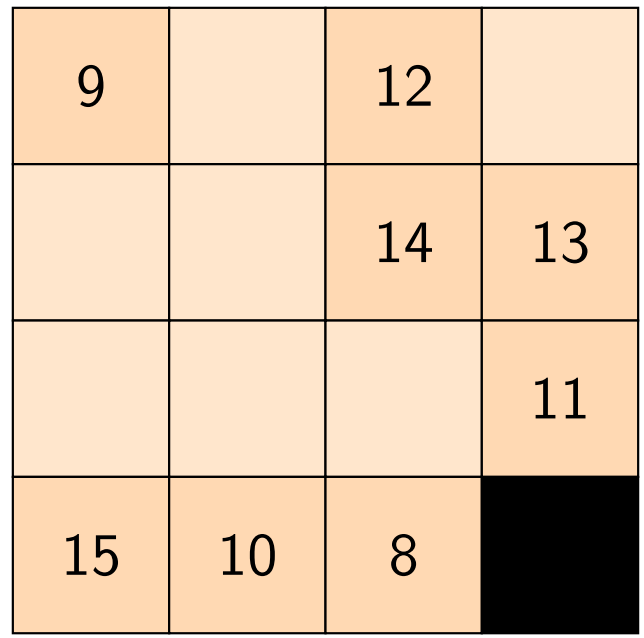
Let α_1 and α_2 be abstractions of transition system $\mathcal{T}$.

If no label of $\mathcal{T}$ affects both $\mathcal{T}^{\alpha_1}$ and $\mathcal{T}^{\alpha_2}$,
then α_1 and α_2 are orthogonal.

(Easy proof omitted.)

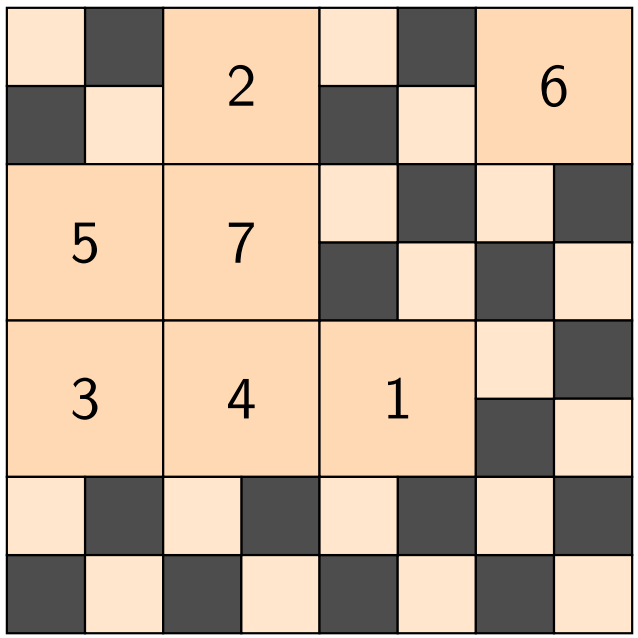
Orthogonal Abstractions: Example

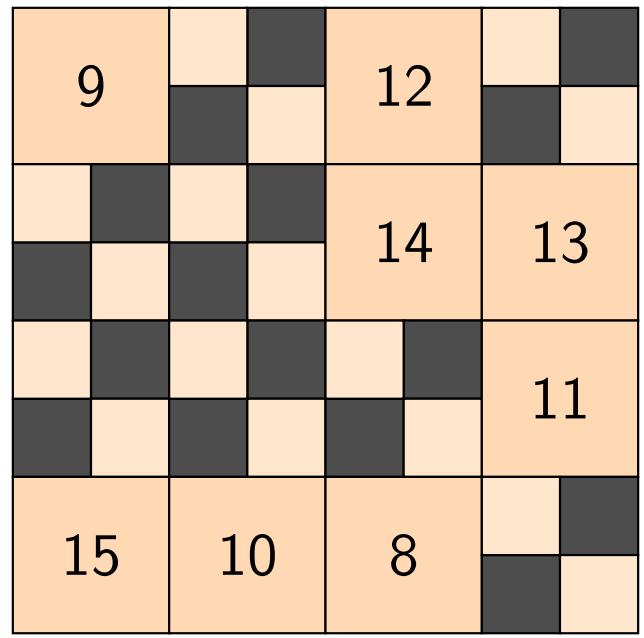




Are the abstractions orthogonal?

Orthogonal Abstractions: Example





Are the abstractions orthogonal?

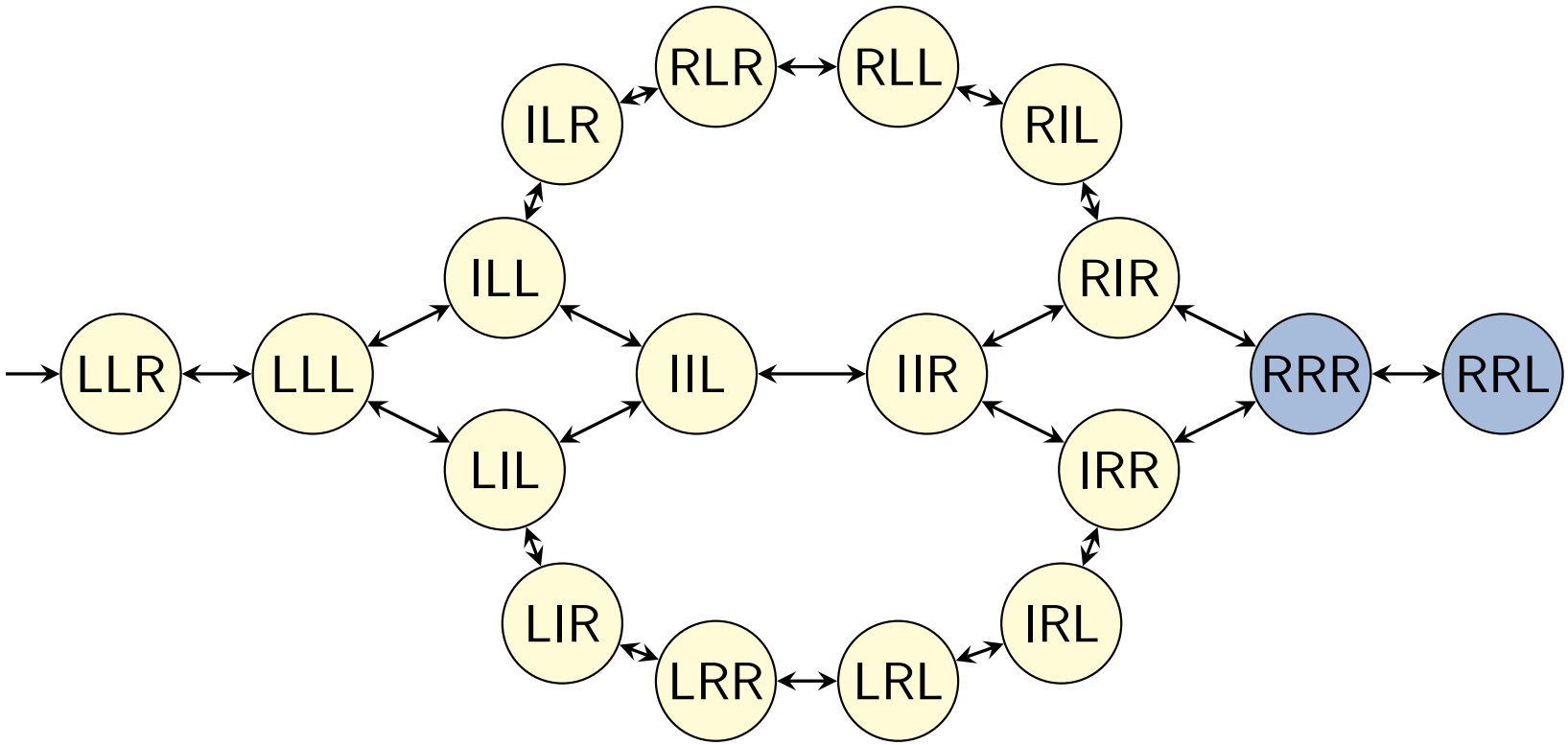
Orthogonality and Additivity

Theorem (Additivity for Orthogonal Abstractions)

Let $h^{\alpha_1}, \ldots, h^{\alpha_n}$ be abstraction heuristics of the same transition system such that α_i and α_j are orthogonal for all $i \neq j$.

Then $\sum_{i=1}^{n} h^{\alpha_i}$ is a safe, goal-aware, admissible and consistent heuristic for Π.

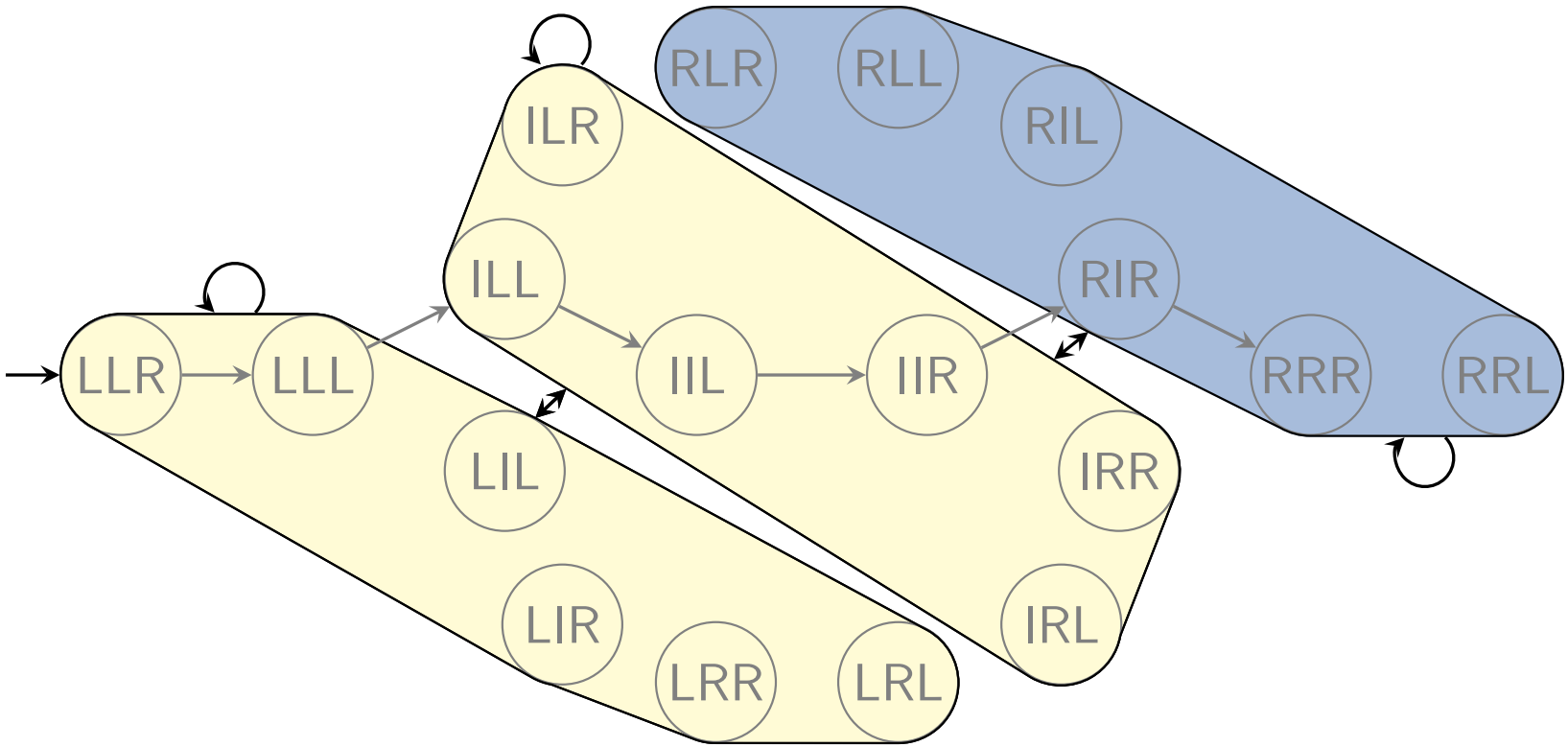
Orthogonality and Additivity: Example



transition system $\mathcal{T}$

state variables: first package, second package, truck

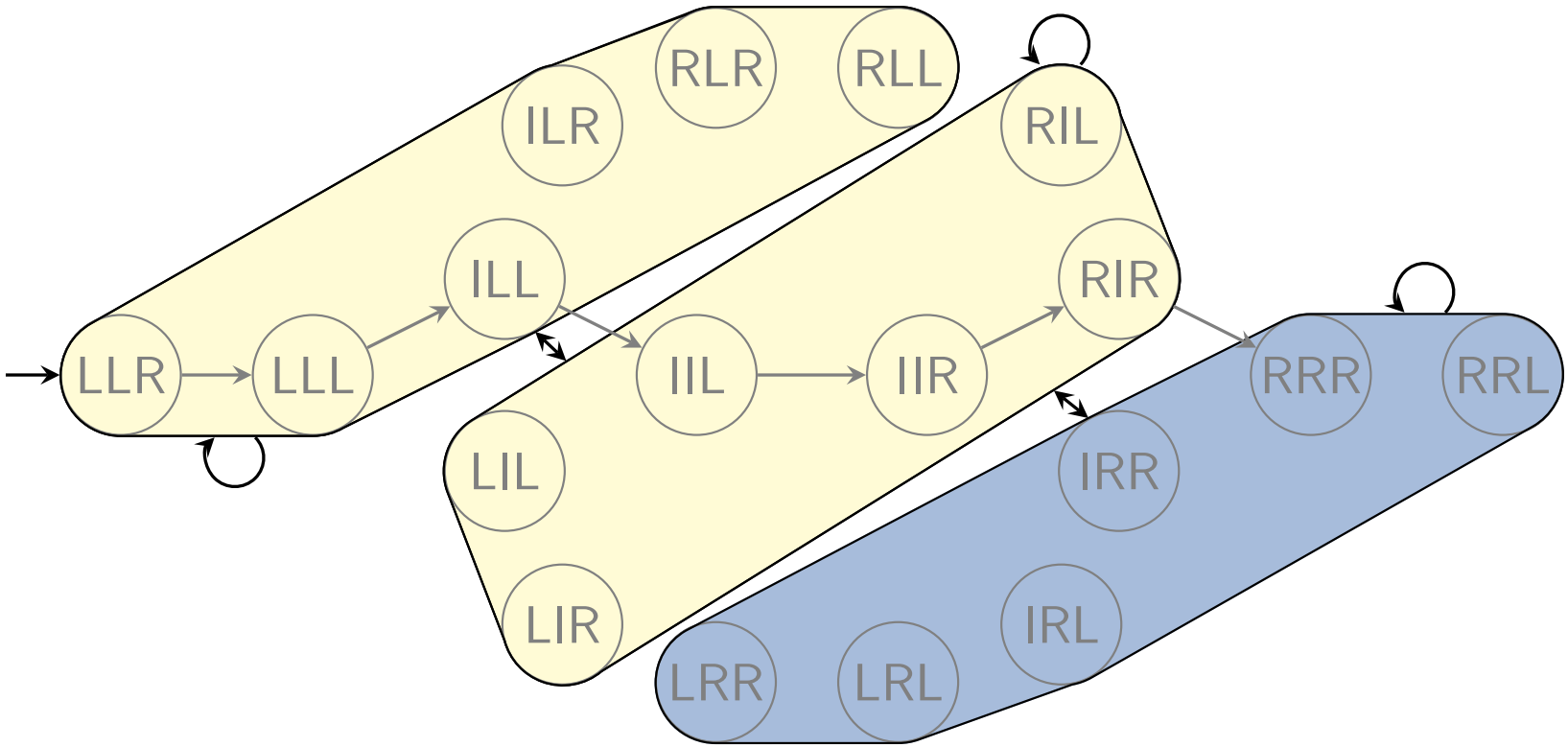
Orthogonality and Additivity: Example



abstraction α_1

abstraction: only consider value of first package

Orthogonality and Additivity: Example



abstraction α_2 (orthogonal to α_1)
abstraction: only consider value of second package

Orthogonality and Additivity: Proof (1)

Proof.

We prove goal-awareness and consistency;
the other properties follow from these two.

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ be the concrete transition system.

Let $h = \sum_{i=1}^{n} h^{\alpha_i}$.

Goal-awareness: For goal states $s \in S_\star$,
$h(s) = \sum_{i=1}^{n} h^{\alpha_i}(s) = \sum_{i=1}^{n} 0 = 0$ because all individual abstraction heuristics are goal-aware. ...

Orthogonality and Additivity: Proof (2)

Proof (continued).

Consistency: Let $s \xrightarrow{o} t \in T$. We must prove $h(s) \leq c(o) + h(t)$.

Because the abstractions are orthogonal, $\alpha_i(s) \neq \alpha_i(t)$ for at most one $i \in \{1, \ldots, n\}$.

Case 1: $\alpha_i(s) = \alpha_i(t)$ for all $i \in \{1, \ldots, n\}$.

$$\begin{aligned}
\text{Then } h(s) &= \sum_{i=1}^{n} h^{\alpha_i}(s) \\
&= \sum_{i=1}^{n} h^*_{\mathcal{T}^{\alpha_i}}(\alpha_i(s)) \\
&= \sum_{i=1}^{n} h^*_{\mathcal{T}^{\alpha_i}}(\alpha_i(t)) \\
&= \sum_{i=1}^{n} h^{\alpha_i}(t) \\
&= h(t) \leq c(o) + h(t).
\end{aligned}$$

...

Orthogonality and Additivity: Proof (3)

Proof (continued).

Case 2: $\alpha_i(s) \neq \alpha_i(t)$ for exactly one $i \in \{1, \ldots, n\}$.

Let $k \in \{1, \ldots, n\}$ such that $\alpha_k(s) \neq \alpha_k(t)$.

Then
$$\begin{aligned} h(s) &= \sum_{i=1}^{n} h^{\alpha_i}(s) \\ &= \sum_{i \in \{1,\ldots,n\} \setminus \{k\}} h^*_{\mathcal{T}^{\alpha_i}}(\alpha_i(s)) + h^{\alpha_k}(s) \\ &\leq \sum_{i \in \{1,\ldots,n\} \setminus \{k\}} h^*_{\mathcal{T}^{\alpha_i}}(\alpha_i(t)) + c(o) + h^{\alpha_k}(t) \\ &= c(o) + \sum_{i=1}^{n} h^{\alpha_i}(t) \\ &= c(o) + h(t), \end{aligned}$$

where the inequality holds because $\alpha_i(s) = \alpha_i(t)$ for all $i \neq k$ and h^{α_k} is consistent. □

E5.2 Outlook

Using Abstraction Heuristics in Practice

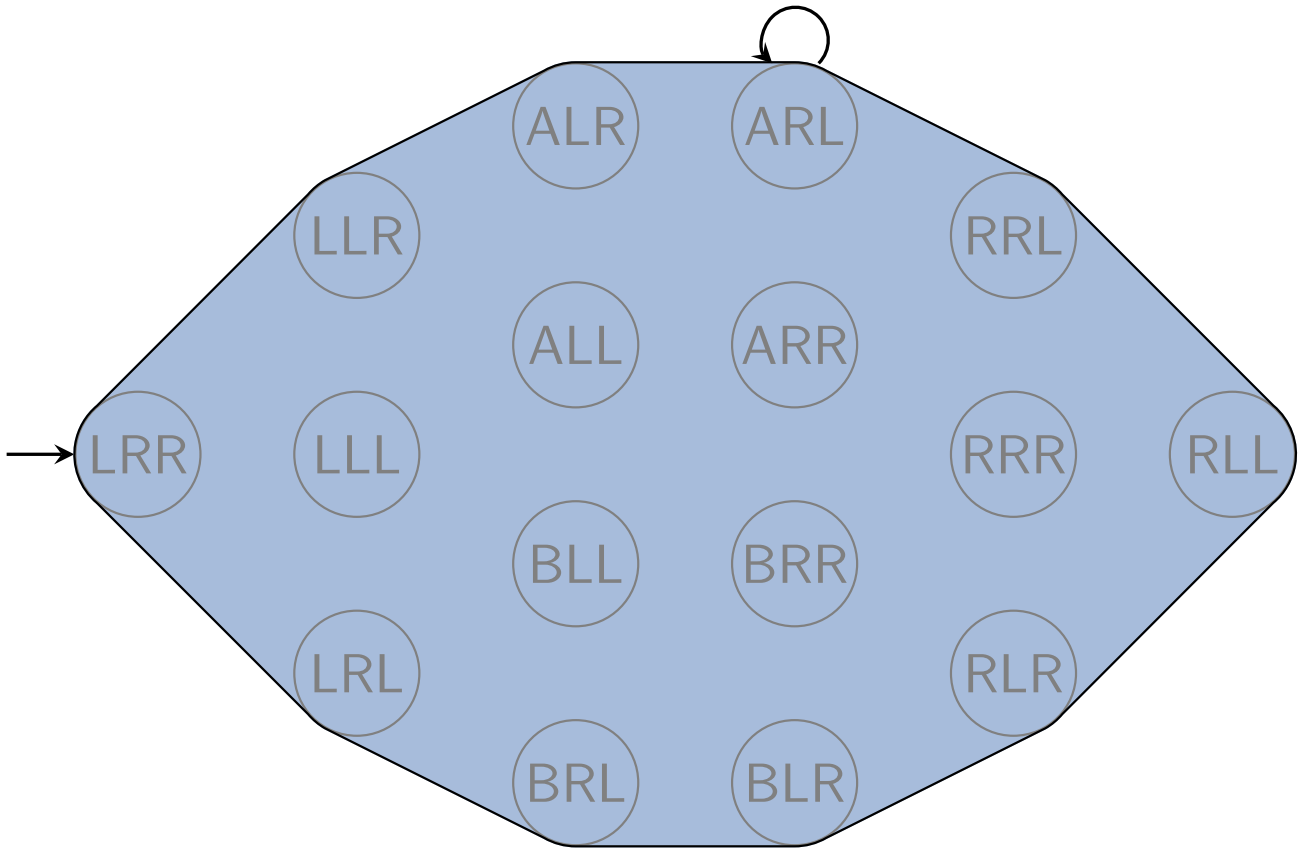
In practice, there are conflicting goals for abstractions:

- we want to obtain an informative heuristic, but
- want to keep its representation small.

Abstractions have small representations if

- there are few abstract states and
- there is a succinct encoding for α.

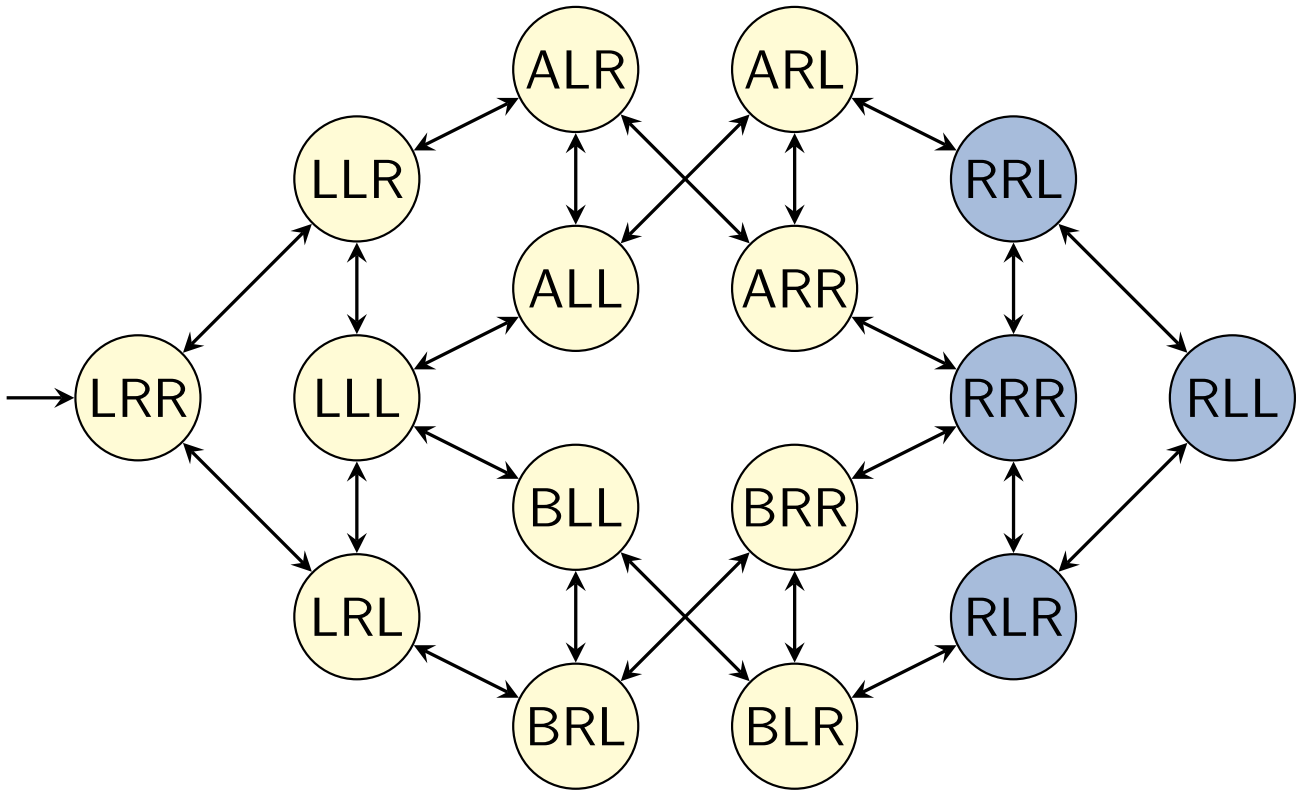
Counterexample: One-State Abstraction



One-state abstraction: $\alpha(s) := \text{const}$.

+ very few abstract states and succinct encoding for α
− completely uninformative heuristic

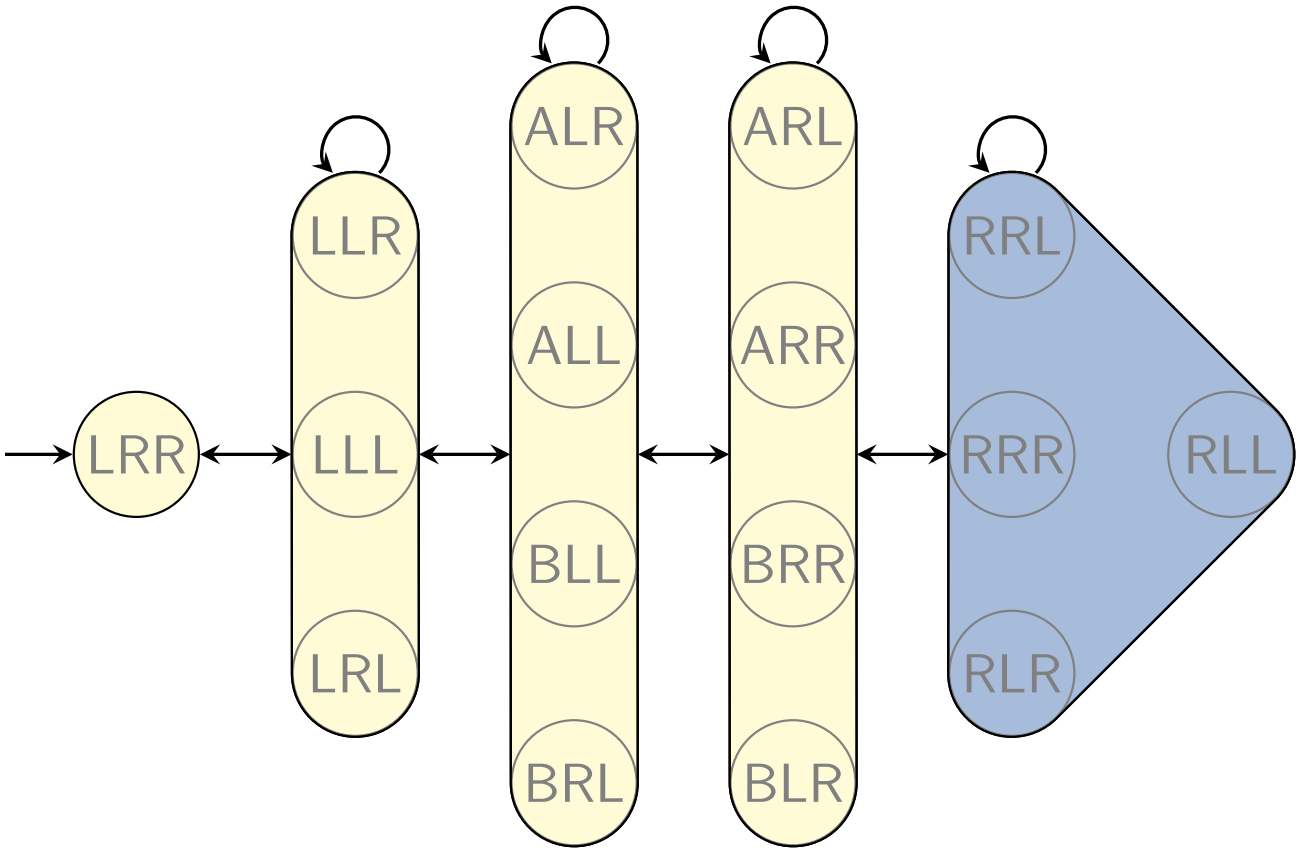
Counterexample: Identity Abstraction



Identity abstraction: $\alpha(s) := s$.

+ perfect heuristic and succinct encoding for α
− too many abstract states

Counterexample: Perfect Abstraction



Perfect abstraction: $\alpha(s) := h^*(s)$.

+ perfect heuristic and usually few abstract states
− usually no succinct encoding for α

Automatically Deriving Good Abstraction Heuristics

Abstraction Heuristics for Planning: Main Research Problem

Automatically derive effective abstraction heuristics
for planning tasks.

$\rightsquigarrow$ we will study three state-of-the-art approaches
in the following chapters

E5.3 Summary

Summary

- Abstraction heuristics from orthogonal abstractions can be added without losing admissibility or consistency.
- One sufficient condition for orthogonality is that all abstractions are affected by disjoint sets of labels.
- Practically useful abstractions are those which give informative heuristics, yet have a small representation.
- Coming up with good abstractions automatically is the main research challenge when applying abstraction heuristics in planning.

Planning and Optimization

E6. Pattern Databases: Introduction

Malte Helmert and Gabriele Röger

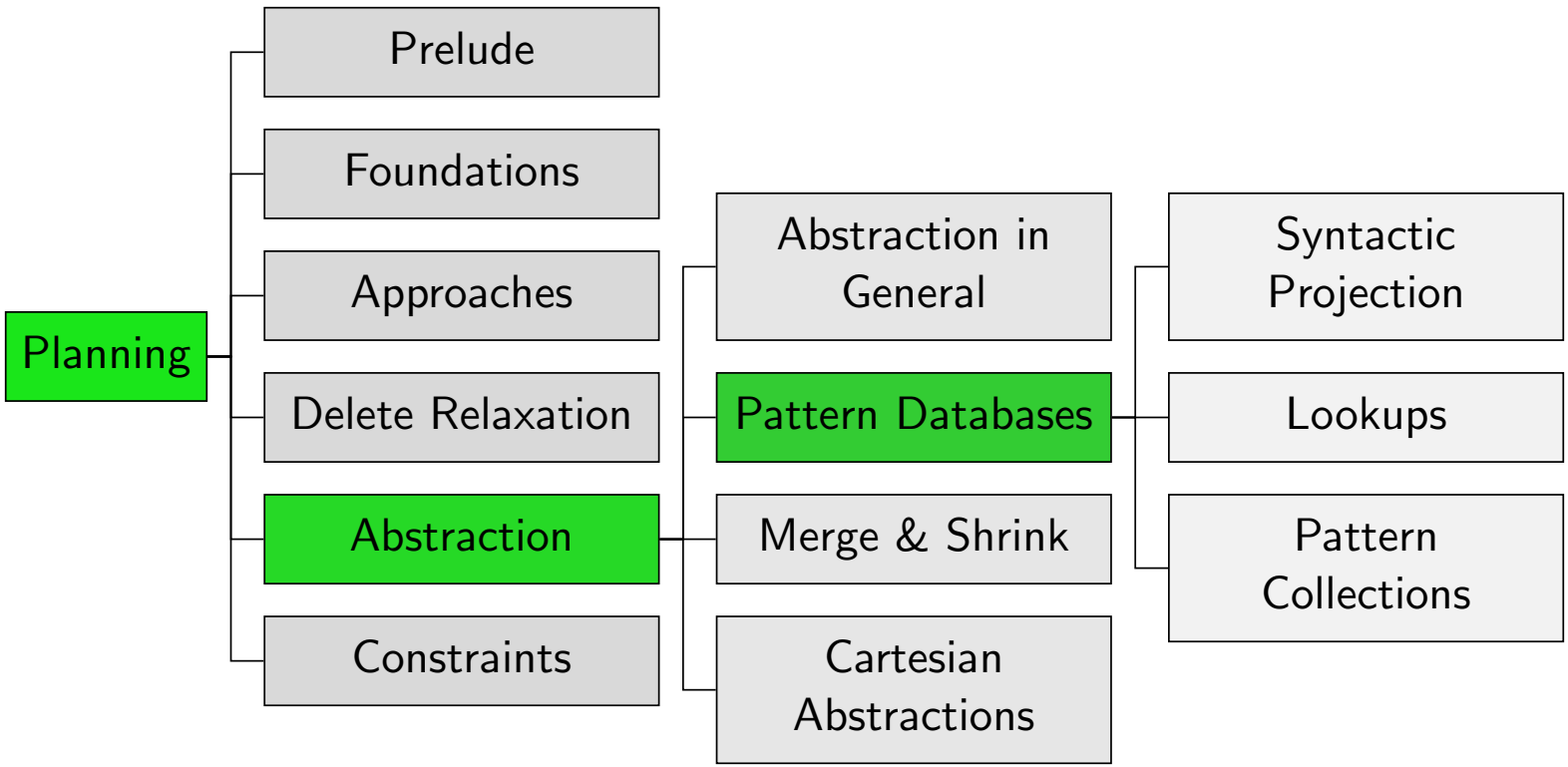
Universität Basel

November 10, 2025

Planning and Optimization

November 10, 2025 — E6. Pattern Databases: Introduction

Content of the Course



E6.1 Projections and Pattern Database Heuristics

Pattern Database Heuristics

- The oldest commonly used abstraction heuristics in search and planning are pattern database (PDB) heuristics.
- PDB heuristics were originally introduced for the 15-puzzle (Culberson & Schaeffer, 1996) and for Rubik's cube (Korf, 1997).
- The first use for domain-independent planning is due to Edelkamp (2001).
- Since then, much research has focused on the theoretical properties of pattern databases, how to use pattern databases more effectively, how to find good patterns, etc.
- Pattern databases are a research area both in planning and in (domain-specific) heuristic search.
- For many search problems, pattern databases are the most effective admissible heuristics currently known.

Pattern Database Heuristics Informally

Pattern Databases: Informally

A pattern database heuristic for a planning task
is an abstraction heuristic where

- some aspects of the task are represented in the abstraction with perfect precision, while
- all other aspects of the task are not represented at all.

This is achieved by projecting the task onto the variables
that describe the aspects that are represented.

Example (15-Puzzle)

- Choose a subset T of tiles (the pattern).
- Faithfully represent the locations of T in the abstraction.
- Assume that all other tiles and the blank can be anywhere in the abstraction.

Projections

Formally, pattern database heuristics are abstraction heuristics induced by a particular class of abstractions called projections.

Definition (Projection)

Let Π be an FDR planning task with variables V and states S.
Let $P \subseteq V$, and let S' be the set of states over P.

The projection $\pi_P : S \to S'$ is defined as $\pi_P(s) := s|_P$,
(where $s|_P(v) := s(v)$ for all $v \in P$).

We call P the pattern of the projection π_P.

In other words, π_P maps two states s_1 and s_2 to the same abstract state iff they agree on all variables in P.

Pattern Database Heuristics

Abstraction heuristics based on projections are called pattern database (PDB) heuristics.

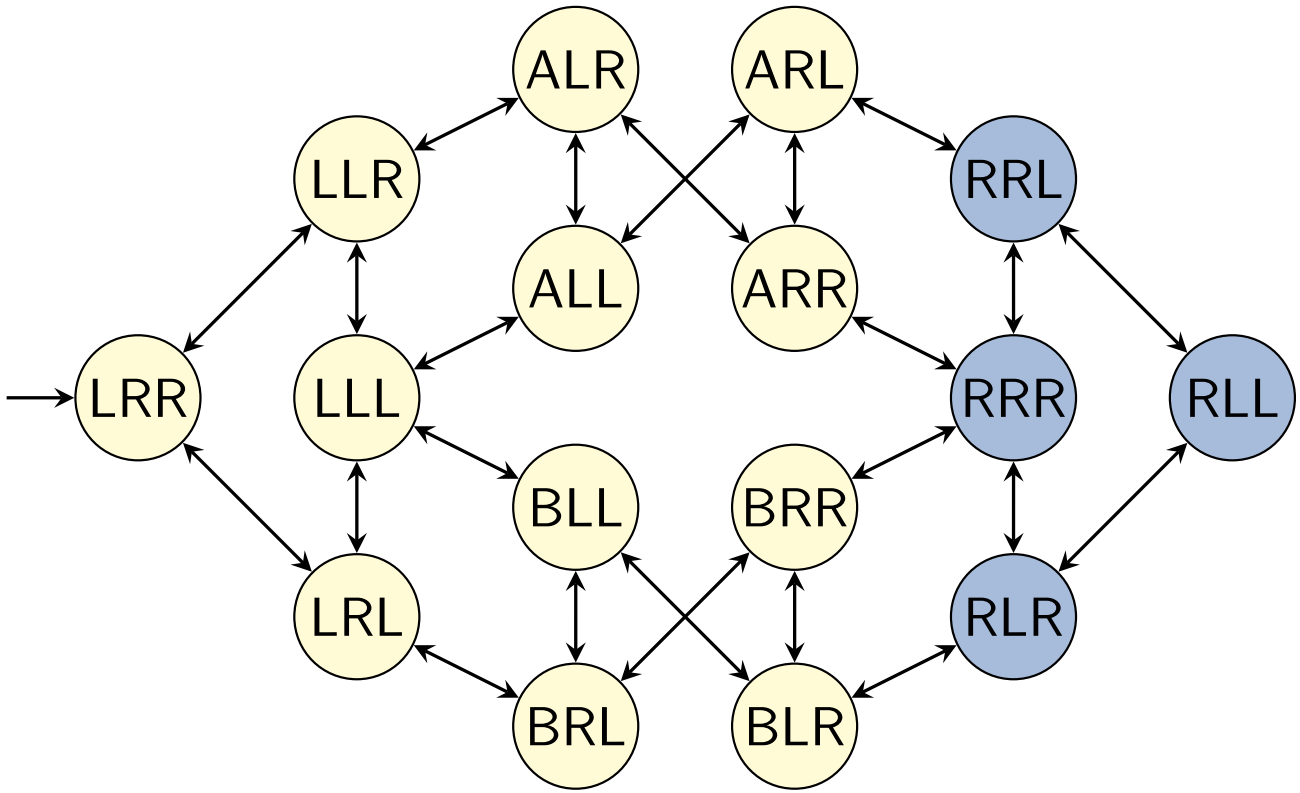
Definition (Pattern Database Heuristic)

The abstraction heuristic induced by π_P is called a pattern database heuristic or PDB heuristic.
We write h^P as a shorthand for h^{π_P}.

Why are they called pattern database heuristics?

- Heuristic values for PDB heuristics are traditionally stored in a 1-dimensional table (array) called a pattern database (PDB). Hence the name "PDB heuristic".
- The word pattern database alludes to endgame databases for 2-player games (in particular chess and checkers).

Example: Transition System

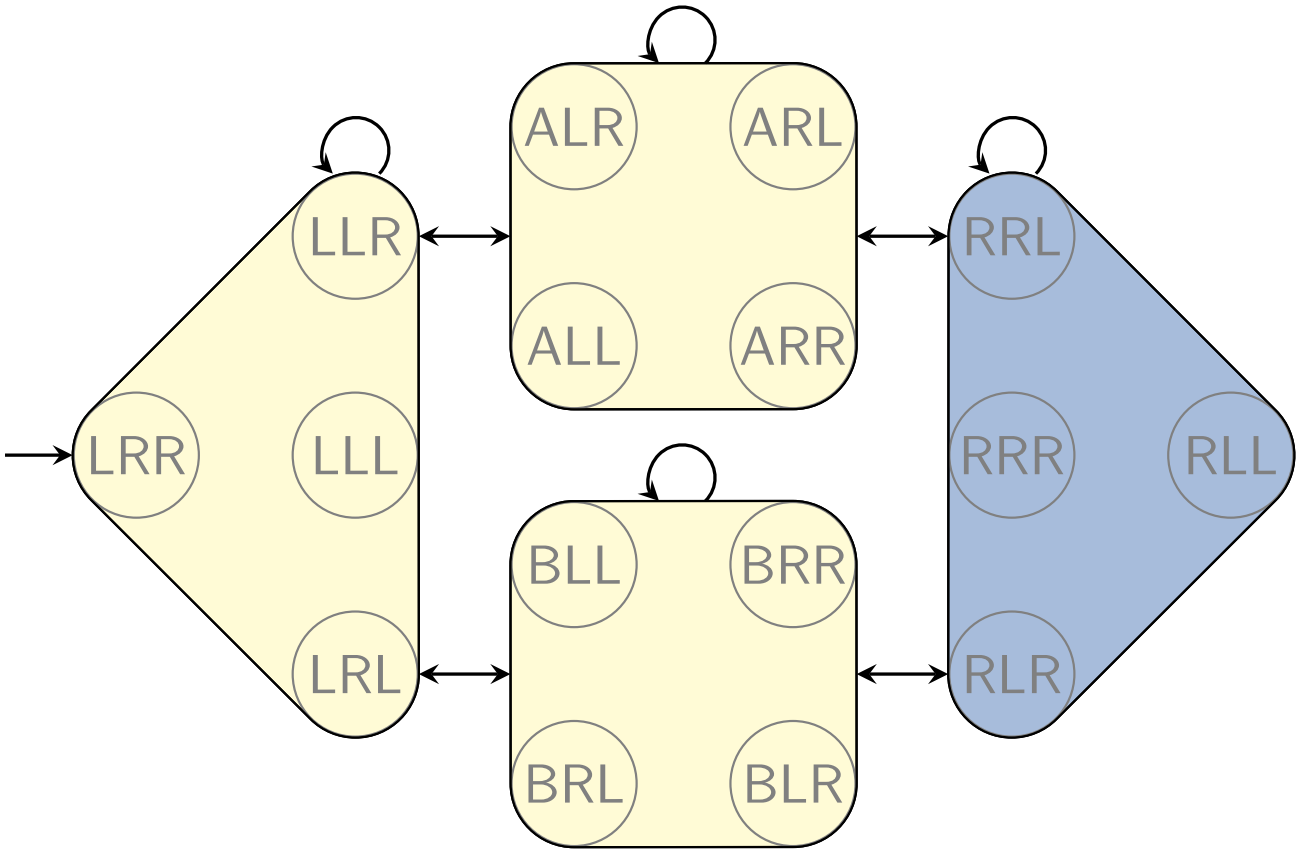


Logistics problem with one package, two trucks, two locations:

- state variable package: $\{L, R, A, B\}$
- state variable truck A: $\{L, R\}$
- state variable truck B: $\{L, R\}$

Example: Projection (1)

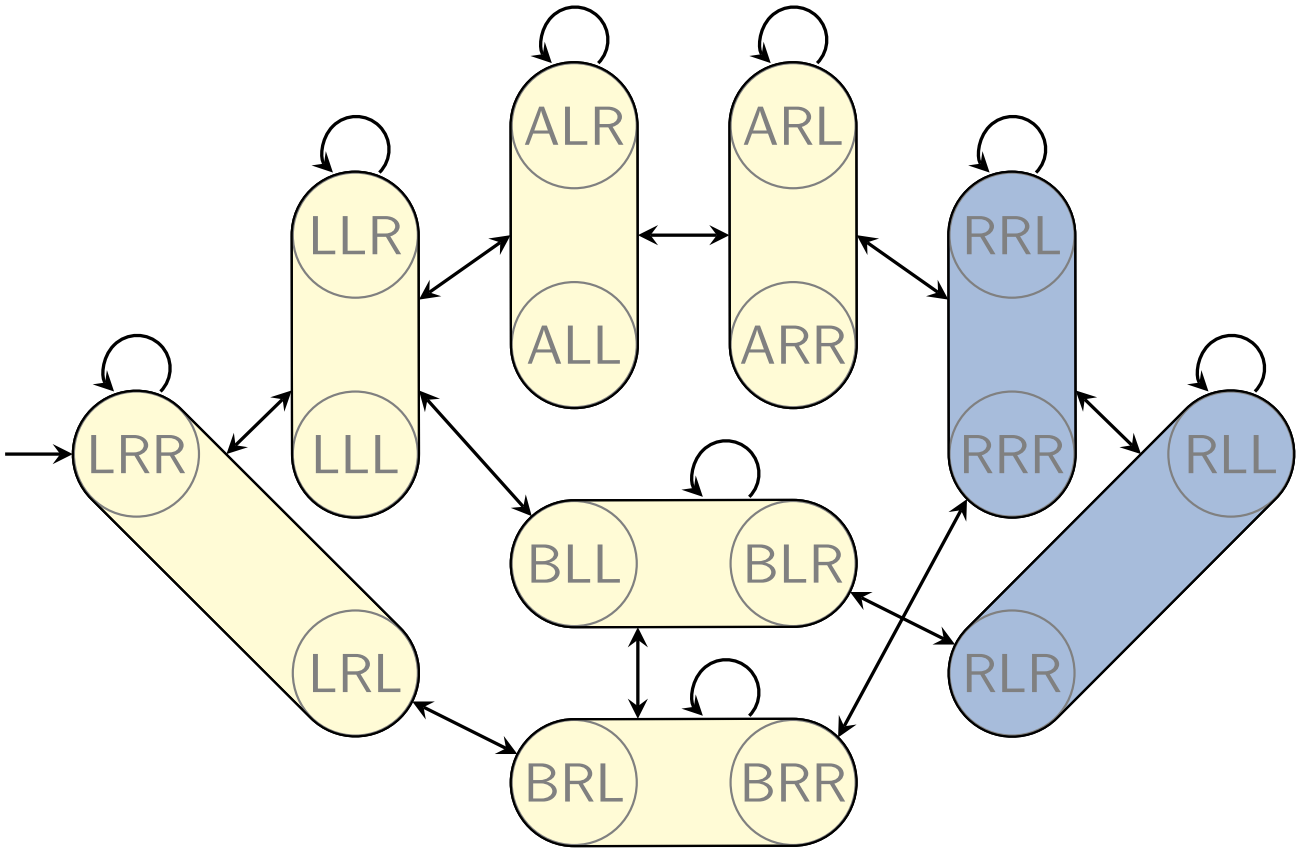
Abstraction induced by $\pi_{\{\text{package}\}}$:



$h^{\{\text{package}\}}(\text{LRR}) = 2$

Example: Projection (2)

Abstraction induced by $\pi_{\{\text{package},\text{truck A}\}}$:



$$h^{\{\text{package},\text{truck A}\}}(\text{LRR}) = 2$$

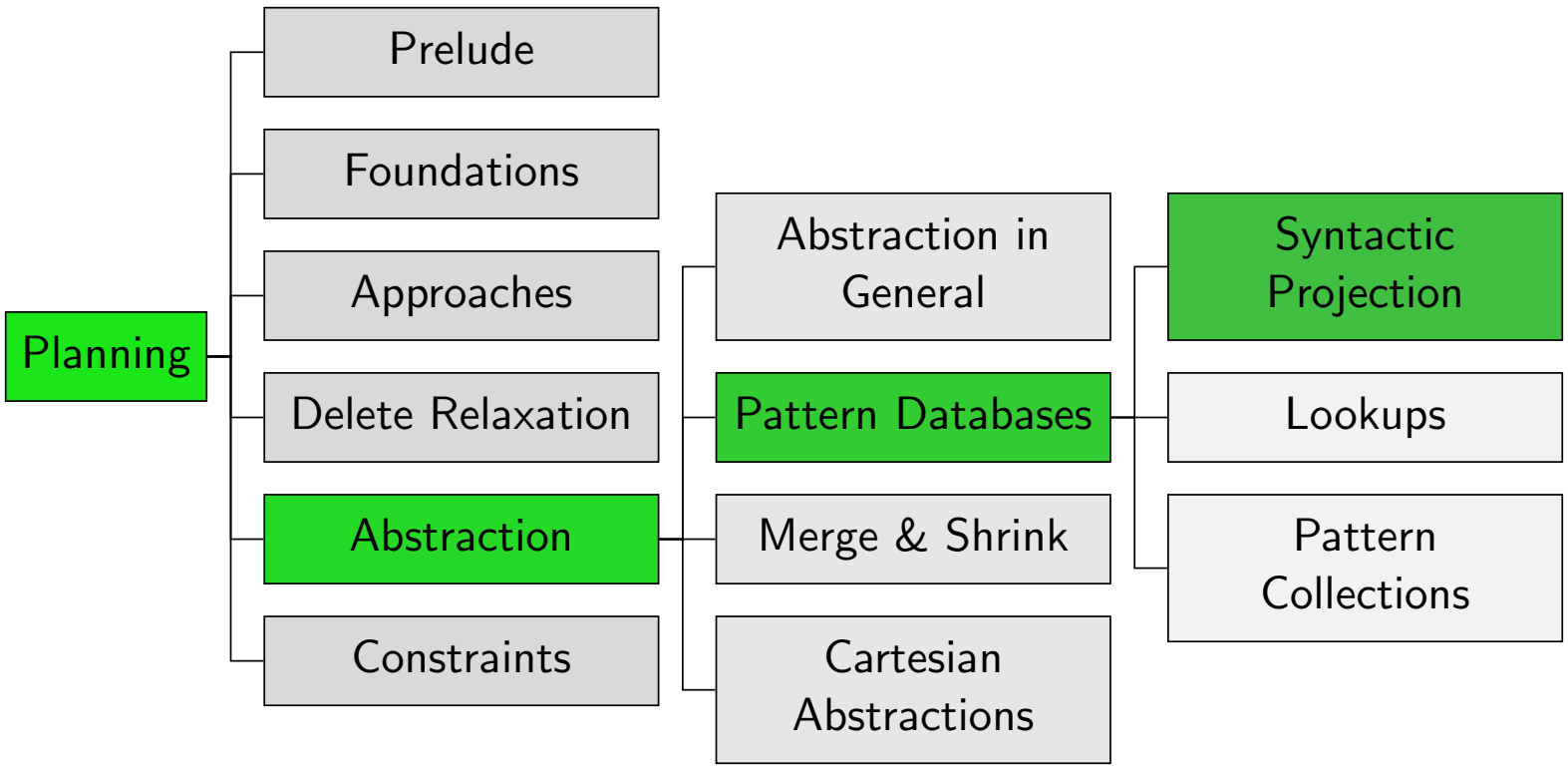
Pattern Databases: Chapter Overview

In the following, we will discuss:

- how to implement PDB heuristics
  $\rightsquigarrow$ this chapter
- how to effectively make use of multiple PDB heuristics
  $\rightsquigarrow$ Chapter E7
- how to find good patterns for PDB heuristics
  $\rightsquigarrow$ Chapter E8

E6.2 Implementing PDBs: Precomputation

Content of the Course



Pattern Database Implementation

Assume we are given a pattern P for a planning task Π.
How do we implement h^P?

1. In a precomputation step, we compute a graph representation for the abstraction $\mathcal{T}(\Pi)^{\pi_P}$ and compute the abstract goal distance for each abstract state.
2. During search, we use the precomputed abstract goal distances in a lookup step.

Precomputation Step

Let Π be a planning task and P a pattern.
Let $\mathcal{T} = \mathcal{T}(\Pi)$ and $\mathcal{T}' = \mathcal{T}^{\pi_P}$.

- We want to compute a graph representation of $\mathcal{T}'$.
- $\mathcal{T}'$ is defined through an abstraction of $\mathcal{T}$.
  - For example, each concrete transition induces an abstract transition.
- However, we cannot compute $\mathcal{T}'$ by iterating over all transitions of $\mathcal{T}$.
  - This would take time $\Omega(\|\mathcal{T}\|)$.
  - This is prohibitively long (or else we could solve the task using uniform-cost search or similar techniques).
- Hence, we need a way of computing $\mathcal{T}'$ in time which is polynomial only in $\|\Pi\|$ and $\|\mathcal{T}'\|$.

Syntactic Projections

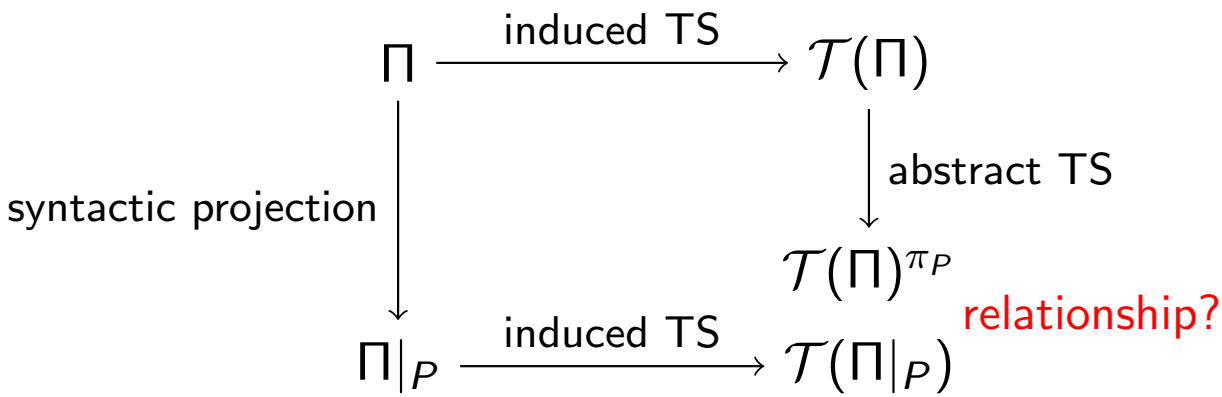
Definition (Syntactic Projection)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be an FDR planning task,
and let $P \subseteq V$ be a subset of its variables.
The syntactic projection $\Pi|_P$ of Π to P is the FDR planning task $\langle P, I|_P, \{o|_P \mid o \in O\}, \gamma|_P \rangle$, where

- $\varphi|_P$ for formula φ is defined as the formula obtained from φ by replacing all atoms $(v = d)$ with $v \notin P$ by $\top$, and
- $o|_P$ for operator o is defined by replacing all formulas φ occurring in the precondition or effect conditions of o with $\varphi|_P$ and all atomic effects $(v := d)$ with $v \notin P$ with the empty effect $\top$.

Put simply, $\Pi|_P$ throws away all information not pertaining to variables in P.

Idea



- $\Pi|_P$ can be computed in linear time in $\|\Pi\|$.
- If $\mathcal{T}(\Pi|_P)$ was "equivalent" to $\mathcal{T}(\Pi)^{\pi_P}$ this would give us an efficient way to compute $\mathcal{T}(\Pi)^{\pi_P}$.
- What do we mean with "equivalent"?
- Is this actually the case?

Isomorphic Transition Systems

Isomorphic = equivalent up to renaming

Definition (Isomorphic Transition Systems)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ and $\mathcal{T}' = \langle S', L', c', T', s'_0, S'_\star \rangle$ be transition systems.

We say that $\mathcal{T}$ is isomorphic to $\mathcal{T}'$, in symbols $\mathcal{T} \sim \mathcal{T}'$, if there exist bijective functions $\varphi : S \to S'$ and $\lambda : L \to L'$ such that:

- $s \xrightarrow{\ell} t \in T$ iff $\varphi(s) \xrightarrow{\lambda(\ell)} \varphi(t) \in T'$,
- $c'(\lambda(\ell)) = c(\ell)$ for all $\ell \in L$,
- $\varphi(s_0) = s'_0$, and
- $s \in S_\star$ iff $\varphi(s) \in S'_\star$.

$(\sim)$ is a an equivalence relation. Two isomorphic transition systems are interchangeable for all practical intents and purposes.

Equivalence Theorem for Syntactic Projections

Theorem (Syntactic Projections vs. Projections)

Let Π *be a* SAS^+ *task, and let* P *be a pattern for* Π*.*
Then $\mathcal{T}(\Pi)^{\pi_P} \sim \mathcal{T}(\Pi|_P)$*.*

Proof.

$\rightsquigarrow$ exercises □

PDB Computation

Using the equivalence theorem, we can compute pattern databases for SAS^+ tasks Π and patterns P:

Computing Pattern Databases

def compute-PDB(Π, P):
    Compute $\Pi' := \Pi|_P$.
    Compute $\mathcal{T}' := \mathcal{T}(\Pi')$.
    Perform a backward uniform-cost search from the goal states of $\mathcal{T}'$ to compute all abstract goal distances.
    $PDB :=$ a table containing all goal distances in $\mathcal{T}'$
    return PDB

The algorithm runs in polynomial time and space in terms of $\|\Pi\| + |PDB|$.

Generalizations of the Equivalence Theorem

- The restriction to SAS^+ tasks is necessary.
- We can slightly generalize the result if we allow general negation-free formulas, but still forbid conditional effects.
  - In that case, the weighted graph of $\mathcal{T}(\Pi)^{\pi_P}$ is isomorphic to a subgraph of the weighted graph of $\mathcal{T}(\Pi|_P)$.
  - This means that we can use $\mathcal{T}(\Pi|_P)$ to derive an admissible estimate of h^P.
- With negations in conditions or with conditional effects, not even this weaker result holds.

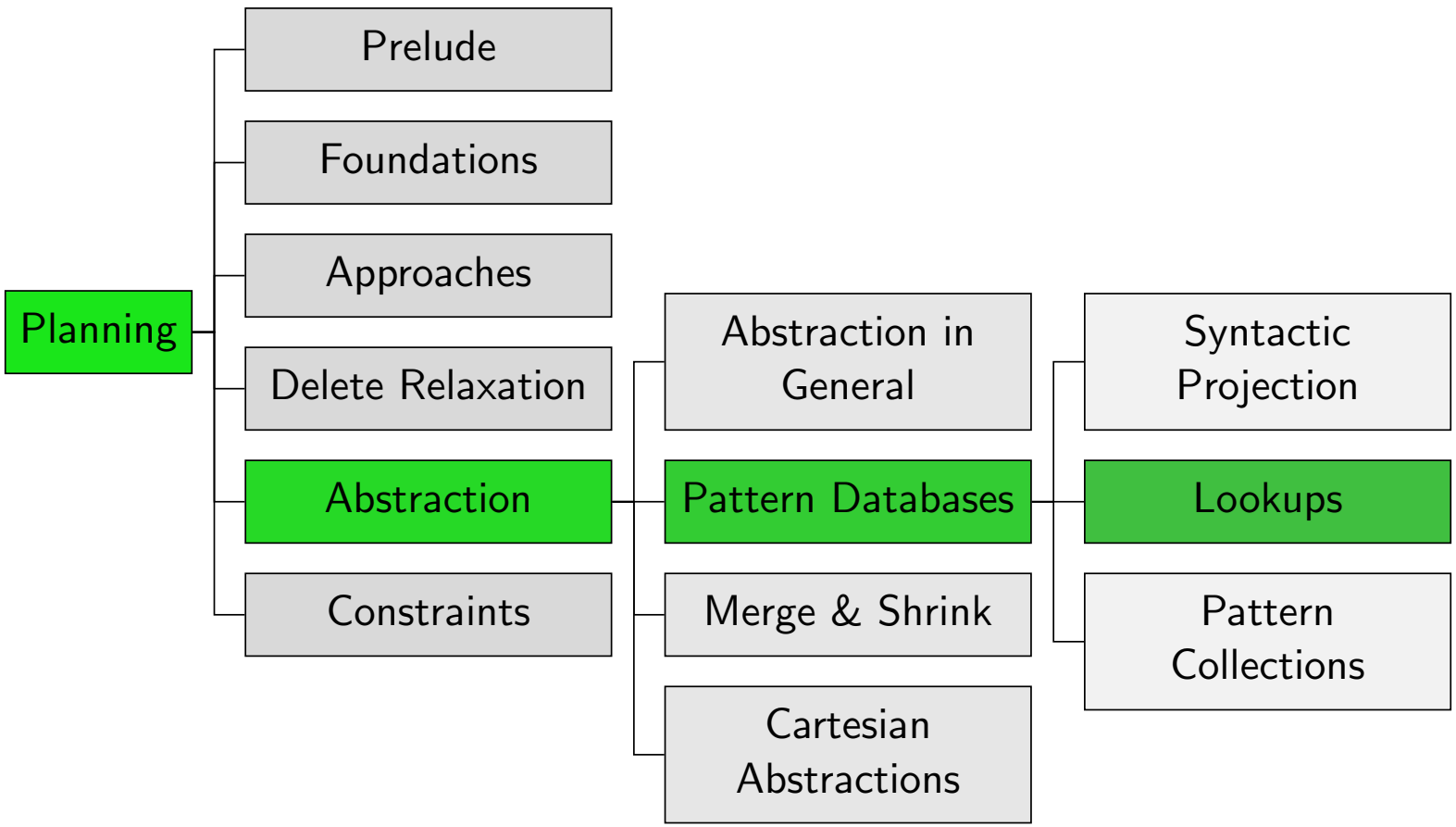
Going Beyond SAS$^+$ Tasks

- Most practical implementations of PDB heuristics are limited to SAS$^+$ tasks (or modest generalizations).
- One way to avoid the issues with general FDR tasks is to convert them to equivalent SAS$^+$ tasks.
- However, most direct conversions can exponentially increase the task size in the worst case.

$\rightsquigarrow$ We will only consider SAS$^+$ tasks in the chapters dealing with pattern databases.

E6.3 Implementing PDBs: Lookup

Content of the Course



Lookup Step: Overview

- During search, the PDB is the only piece of information necessary to represent h^P. (It is not necessary to store the abstract transition system itself at this point.)
- Hence, the space requirements for PDBs during search are linear in the number of abstract states S': there is one table entry for each abstract state.
- During search, $h^P(s)$ is computed by mapping $\pi_P(s)$ to a natural number in the range $\{0, \ldots, |S'| - 1\}$ using a perfect hash function, then looking up the table entry for this number.

Lookup Step: Algorithm

Let $P = \{v_1, \ldots, v_k\}$ be the pattern.

- We assume that all variable domains are natural numbers counted from 0, i.e., $\text{dom}(v) = \{0, 1, \ldots, |\text{dom}(v)| - 1\}$.
- For all $i \in \{1, \ldots, k\}$, we precompute $N_i := \prod_{j=1}^{i-1} |\text{dom}(v_j)|$.

Then we can look up heuristic values as follows:

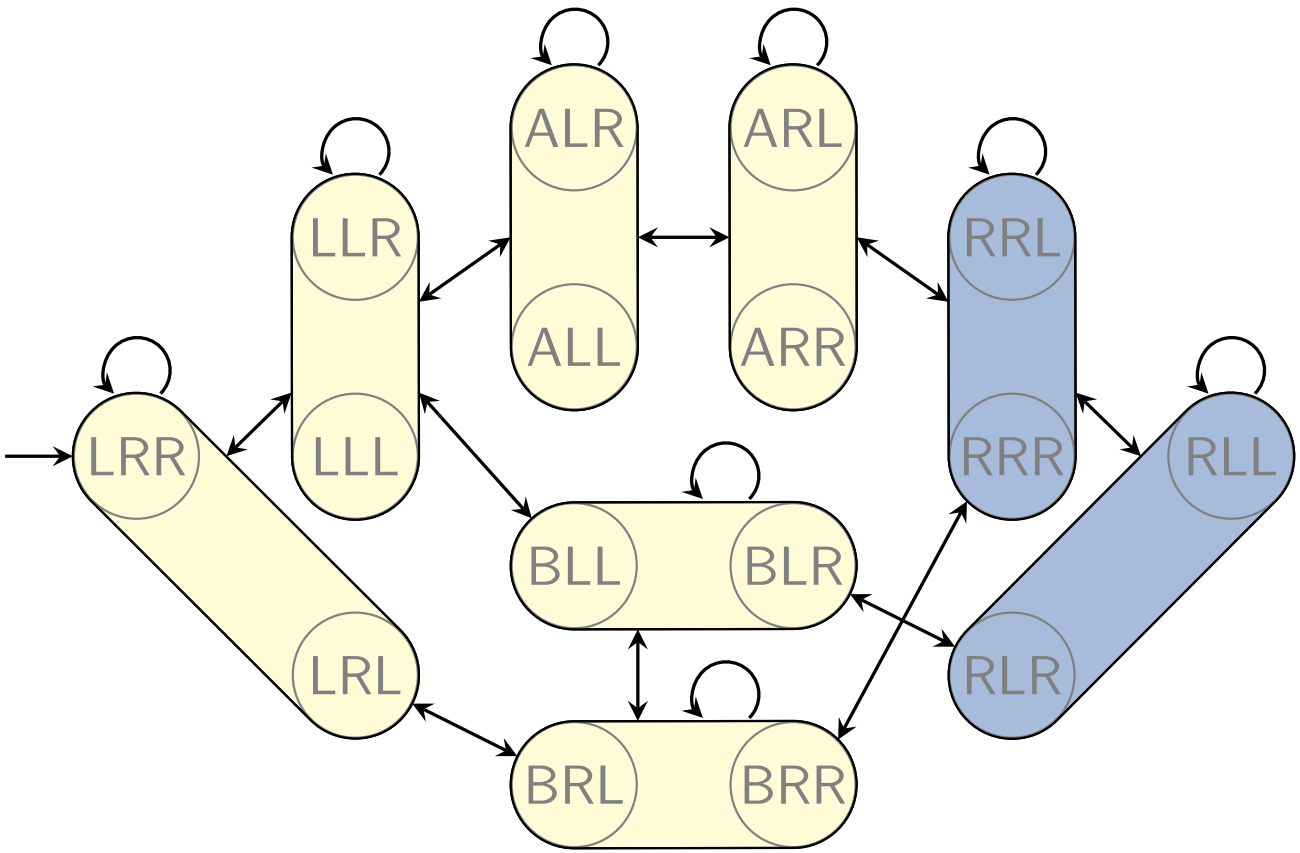
Computing Pattern Database Heuristics

def PDB-heuristic(s):
    $index := \sum_{i=1}^{k} N_i s(v_i)$
    return $PDB[index]$

- This is a very fast operation: it can be performed in $O(k)$.
- For comparison, most relaxation heuristics need time $O(\|\Pi\|)$ per state.

Lookup Step: Example (1)

Abstraction induced by $\pi_{\{\text{package},\text{truck A}\}}$:



Lookup Step: Example (2)

- $P = \{v_1, v_2\}$ with $v_1 =$ package, $v_2 =$ truck A.
- $\text{dom}(v_1) = \{\text{L}, \text{R}, \text{A}, \text{B}\} \approx \{0, 1, 2, 3\}$
- $\text{dom}(v_2) = \{\text{L}, \text{R}\} \approx \{0, 1\}$

$\rightsquigarrow$ $N_1 = \prod_{j=1}^{0} |\text{dom}(v_j)| = 1$, $N_2 = \prod_{j=1}^{1} |\text{dom}(v_j)| = 4$

$\rightsquigarrow$ $\mathit{index}(s) = 1 \cdot s(\text{package}) + 4 \cdot s(\text{truck A})$

Pattern database:

abstract state	LL	RL	AL	BL	LR	RR	AR	BR
index	0	1	2	3	4	5	6	7
value	2	0	2	1	2	0	1	1

E6.4 Summary

Summary

- Pattern database (PDB) heuristics are abstraction heuristics based on projection to a subset of variables.
- For SAS^+ tasks, they can easily be implemented via syntactic projections of the task representation.
- PDBs are lookup tables that store heuristic values, indexed by perfect hash values for projected states.
- PDB values can be looked up very fast, in time $O(k)$ for a projection to k variables.

Planning and Optimization

E7. Pattern Databases: Multiple Patterns

Malte Helmert and Gabriele Röger

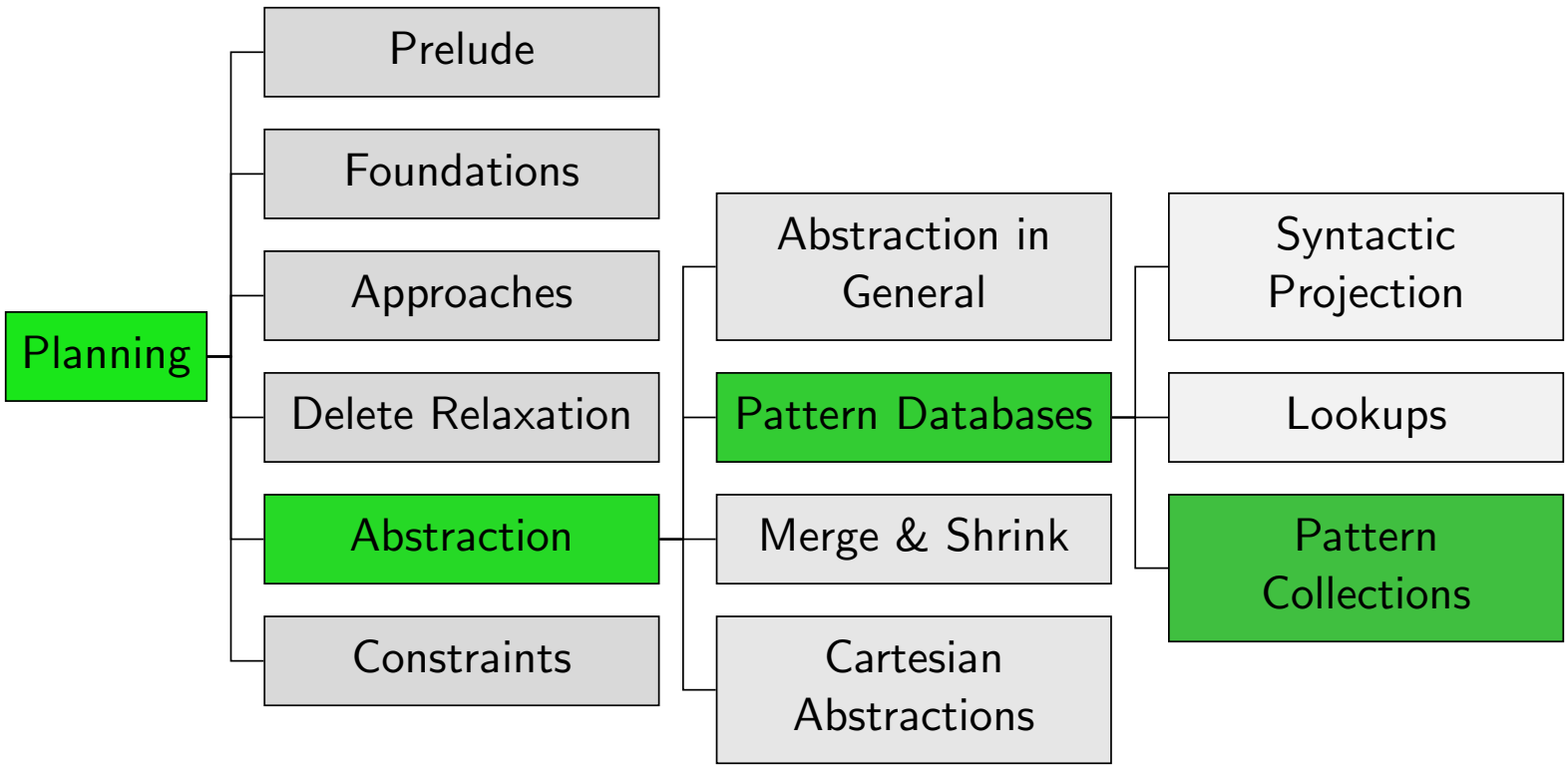
Universität Basel

November 12, 2025

Planning and Optimization

November 12, 2025 — E7. Pattern Databases: Multiple Patterns

Content of the Course



E7.1 Additivity & the Canonical Heuristic

Pattern Collections

- The space requirements for a pattern database grow exponentially with the number of state variables in the pattern.
- This places severe limits on the usefulness of single PDB heuristics h^P for larger planning task.
- To overcome this limitation, planners using pattern databases work with collections of multiple patterns.
- When using two patterns P_1 and P_2, it is always possible to use the maximum of h^{P_1} and h^{P_2} as an admissible and consistent heuristic estimate.
- However, when possible, it is much preferable to use the sum of h^{P_1} and h^{P_2} as a heuristic estimate, since $h^{P_1} + h^{P_2} \geq \max\{h^{P_1}, h^{P_2}\}$.

Criterion for Additive Patterns

Theorem (Additive Pattern Sets)

Let $P_1, \ldots, P_k$ be disjoint patterns for an FDR planning task Π.

If there exists no operator that has an effect on a variable $v_i \in P_i$ and on a variable $v_j \in P_j$ for some $i \neq j$, then $\sum_{i=1}^{k} h^{P_i}$ is an admissible and consistent heuristic for Π.

Proof.

If there exists no such operator, then no label of $\mathcal{T}(\Pi)$ affects both $\mathcal{T}(\Pi)^{\pi_{P_i}}$ and $\mathcal{T}(\Pi)^{\pi_{P_j}}$ for $i \neq j$. By the theorem on affecting transition labels, this means that any two projections π_{P_i} and π_{P_j} are orthogonal. The claim follows with the theorem on additivity for orthogonal abstractions. □

A pattern set $\{P_1, \ldots, P_k\}$ which satisfies the criterion of the theorem is called an additive pattern set or additive set.

Finding Additive Pattern Sets

The theorem on additive pattern sets gives us a simple criterion to decide which pattern heuristics can be admissibly added.

Given a pattern collection $\mathcal{C}$ (i.e., a set of patterns), we can use this information as follows:

1. Build the compatibility graph for $\mathcal{C}$.
   - Vertices correspond to patterns $P \in \mathcal{C}$.
   - There is an edge between two vertices iff no operator affects both incident patterns.
2. Compute all maximal cliques of the graph. These correspond to maximal additive subsets of $\mathcal{C}$.
   - Computing large cliques is an NP-hard problem, and a graph can have exponentially many maximal cliques.
   - However, there are output-polynomial algorithms for finding all maximal cliques (Tomita, Tanaka & Takahashi, 2004) which have led to good results in practice.

Finding Additive Pattern Sets: Example

Example

Consider a planning task with state variables $V = \{v_1, \ldots, v_5\}$ and the pattern collection $\mathcal{C} = \{P_1, \ldots, P_5\}$ with $P_1 = \{v_1, v_2, v_3\}$, $P_2 = \{v_1, v_2\}$, $P_3 = \{v_3\}$, $P_4 = \{v_4\}$ and $P_5 = \{v_5\}$.

There are operators affecting each individual variable, variables v_1 and v_2, variables v_3 and v_4 and variables v_3 and v_5.

What are the maximal cliques in the compatibility graph for $\mathcal{C}$?

Answer: $\{P_1\}$, $\{P_2, P_3\}$, $\{P_2, P_4, P_5\}$

The Canonical Heuristic Function

Definition (Canonical Heuristic Function)

Let $\mathcal{C}$ be a pattern collection for an FDR planning task.

The canonical heuristic $h^{\mathcal{C}}$ for pattern collection $\mathcal{C}$ is defined as

$$h^{\mathcal{C}}(s) = \max_{\mathcal{D} \in \mathit{cliques}(\mathcal{C})} \sum_{P \in \mathcal{D}} h^P(s),$$

where $\mathit{cliques}(\mathcal{C})$ is the set of all maximal cliques in the compatibility graph for $\mathcal{C}$.

For all choices of $\mathcal{C}$, heuristic $h^{\mathcal{C}}$ is admissible and consistent.

Canonical Heuristic Function: Example

Example

Consider a planning task with state variables $V = \{v_1, \ldots, v_5\}$ and the pattern collection $\mathcal{C} = \{P_1, \ldots, P_5\}$ with $P_1 = \{v_1, v_2, v_3\}$, $P_2 = \{v_1, v_2\}$, $P_3 = \{v_3\}$, $P_4 = \{v_4\}$ and $P_5 = \{v_5\}$.

There are operators affecting each individual variable, an operator that affects v_1 and v_2 and an operator that affects v_3 and v_5.

What are the maximal cliques in the compatibility graph for $\mathcal{C}$?

Answer: $\{P_1\}$, $\{P_2, P_3\}$, $\{P_2, P_4, P_5\}$

What is the canonical heuristic function $h^{\mathcal{C}}$?

Answer:

$$\begin{aligned} h^{\mathcal{C}} &= \max\{h^{P_1}, h^{P_2} + h^{P_3}, h^{P_2} + h^{P_4} + h^{P_5}\} \\ &= \max\{h^{\{v_1,v_2,v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_4\}} + h^{\{v_5\}}\} \end{aligned}$$

How Good is the Canonical Heuristic Function?

- The canonical heuristic function is the best possible admissible heuristic we can derive from $\mathcal{C}$ using our additivity criterion.
- Even better heuristic estimates can be obtained from projection heuristics using a more general additivity criterion based on an idea called cost partitioning.

$\rightsquigarrow$ We will return to this topic in Part F.

E7.2 Dominated Additive Sets

Computing $h^{\mathcal{C}}$ Efficiently: Motivation

Consider
$h^{\mathcal{C}} = \max\{h^{\{v_1,v_2,v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_4\}} + h^{\{v_5\}}\}$.

- We need to evaluate this expression for every search node.
- It is thus worth to spend some effort in precomputations to make the evaluation more efficient.

A naive implementation requires 5 PDB lookups
(one for each pattern) and maximizes over 3 additive sets.

Can we do better?

Dominated Sum Theorem

Theorem (Dominated Sum)

Let $\{P_1, \ldots, P_k\}$ be an additive pattern set for an FDR planning task Π, and let P be a pattern with $P_i \subseteq P$ for all $i \in \{1, \ldots, k\}$. Then $\sum_{i=1}^{k} h^{P_i} \leq h^P$.

Proof.

Because $P_i \subseteq P$, all projections π_{P_i} are coarsenings of the projection π_P. Let $\mathcal{T}' := \mathcal{T}(\Pi)^{\pi_P}$.
We can view each h^{P_i} as an abstraction heuristic for solving $\mathcal{T}'$.

By the argumentation of the previous theorem, $\{P_1, \ldots, P_k\}$ is an additive pattern set and hence $\sum_{i=1}^{k} h^{P_i}$ is an admissible heuristic for solving $\mathcal{T}'$. Hence, $\sum_{i=1}^{k} h^{P_i}$ is bounded by the optimal goal distances in $\mathcal{T}'$, which implies $\sum_{i=1}^{k} h^{P_i} \leq h^P$.

Dominated Sum Corollary

Corollary (Dominated Sum)

Let $\{P_1, \ldots, P_n\}$ and $\{Q_1, \ldots, Q_m\}$ be additive pattern sets of an FDR planning task such that each pattern P_i is a subset of some pattern Q_j (not necessarily proper).

Then $\sum_{i=1}^{n} h^{P_i} \leq \sum_{j=1}^{m} h^{Q_j}$.

Proof.

$$\sum_{i=1}^{n} h^{P_i} \overset{(1)}{\leq} \sum_{j=1}^{m} \sum_{P_i \subseteq Q_j} h^{P_i} \overset{(2)}{\leq} \sum_{j=1}^{m} h^{Q_j},$$

where (1) holds because each P_i is contained in some Q_j and (2) follows from the dominated sum theorem. □

Dominance Pruning

- We can use the dominated sum corollary to simplify the representation of $h^{\mathcal{C}}$: sums that are dominated by other sums can be pruned.
- The dominance test can be performed in polynomial time.

Example

$$\begin{aligned} & \max\{h^{\{v_1,v_2,v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_4\}} + h^{\{v_5\}}\} \\ = \; & \max\{h^{\{v_1,v_2,v_3\}}, h^{\{v_1,v_2\}} + h^{\{v_4\}} + h^{\{v_5\}}\} \end{aligned}$$

$\rightsquigarrow$ number of PDB lookups reduced from 5 to 4;
number of additive sets reduced from 3 to 2

E7.3 Redundant Patterns

Redundant Patterns

- The previous example shows that sometimes, not all patterns in a pattern collection are useful.
  - Pattern $\{v_3\}$ could be removed because it does not affect the heuristic value.
- In this section, we will show that certain patterns are never useful and should thus never be considered.
- Knowing about such redundant patterns is useful for algorithms that try to find good patterns automatically.

$\rightsquigarrow$ It allows us to focus on the useful ones.

Non-Goal Patterns

Theorem (Non-Goal Patterns are Trivial)

Let Π be a SAS$^+$ planning task, and let P be a pattern for Π such that no variable in P is mentioned in the goal formula of Π. Then $h^P(s) = 0$ for all states s.

Proof.

All states in the abstraction are goal states. □

$\rightsquigarrow$ Patterns with no goal variables are redundant.
They should not be included in a pattern collection.

Causal Graphs: Motivation

- For more interesting notions of redundancy, we need to introduce causal graphs.
- Causal graphs describe the dependency structure between the state variables of a planning task.
- Causal graphs are a general tool for analyzing planning tasks.
- They are used in many contexts besides abstraction heuristics.

Causal Graphs

Definition (Causal Graph)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be an FDR planning task.

The causal graph of Π, written $CG(\Pi)$, is the directed graph whose vertices are the state variables V and which has an arc $\langle u, v \rangle$ iff $u \neq v$ and there exists an operator $o \in O$ such that:

- u appears anywhere in o (in precondition, effect conditions or atomic effects), and
- v is modified by an effect of o.

Idea: an arc $\langle u, v \rangle$ in the causal graph indicates that variable u is in some way relevant for modifying the value of v

Causally Relevant Variables

Definition (Causally Relevant)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be an FDR planning task,
and let $P \subseteq V$ be a pattern for Π.

We say that $v \in P$ is causally relevant for P if $CG(\Pi)$,
restricted to the variables of P, contains a directed path from v
to a variable $v' \in P$ that is mentioned in the goal formula γ.

Note: The definition implies that variables in P mentioned
in the goal are always causally relevant for P.

Causally Irrelevant Variables are Useless

Theorem (Causally Irrelevant Variables are Useless)

Let $P \subseteq V$ be a pattern for an FDR planning task Π, and let $P' \subseteq P$ consist of all variables that are causally relevant for P.

Then $h^P(s) = h^{P'}(s)$ for all states s.

$\rightsquigarrow$ Patterns P where not all variables are causally relevant are redundant. The smaller subpattern P' should be used instead.

Causally Irrelevant Variables are Useless: Proof

Proof Sketch.

($\geq$): holds because π_P is a refinement of $\pi_{P'}$

($\leq$): Obvious if $h^{P'}(s) = \infty$; else, consider an optimal abstract plan $\langle o_1, \ldots, o_n \rangle$ for $\pi_{P'}(s)$ in $\mathcal{T}(\Pi)^{\pi_{P'}}$.

W.l.o.g., each o_i modifies some variable in P'.
(Other o_i are redundant and can be omitted.)

Because P' includes all variables causally relevant for P,
no variable in $P \setminus P'$ is mentioned in any o_i or in the goal.

Then the same abstract plan also is a solution for $\pi_P(s)$ in $\mathcal{T}(\Pi)^{\pi_P}$.
Hence, the optimal solution cost under abstraction π_P
is no larger than under $\pi_{P'}$.

Causally Connected Patterns

Definition (Causally Connected)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be an FDR planning task,
and let $P \subseteq V$ be a pattern for Π.

We say that P is causally connected if the subgraph of $CG(\Pi)$ induced by P is weakly connected (i.e., contains a path from every vertex to every other vertex, ignoring arc directions).

Disconnected Patterns are Decomposable

Theorem (Causally Disconnected Patterns are Decomposable)

Let $P \subseteq V$ be a pattern for a SAS^+ planning task Π that is not causally connected, and let P_1, P_2 be a partition of P into non-empty subsets such that $CG(\Pi)$ contains no arc between the two sets.

Then $h^P(s) = h^{P_1}(s) + h^{P_2}(s)$ for all states s.

$\rightsquigarrow$ Causally disconnected patterns P are redundant.
The smaller subpatterns P_1 and P_2 should be used instead.

Disconnected Patterns are Decomposable: Proof

Proof Sketch.

($\geq$): There is no arc between P_1 and P_2 in the causal graph, and thus there is no operator that affects both patterns.

Therefore, they are additive, and $h^P \geq h^{P_1} + h^{P_2}$ follows from the dominated sum theorem.

($\leq$): Obvious if $h^{P_1}(s) = \infty$ or $h^{P_2}(s) = \infty$. Else, consider optimal abstract plans ρ_1 for $\mathcal{T}(\Pi)^{\pi_{P_1}}$ and ρ_2 for $\mathcal{T}(\Pi)^{\pi_{P_2}}$.

Because the variables of the two projections do not interact, concatenating the two plans yields an abstract plan for $\mathcal{T}(\Pi)^{\pi_P}$.

Hence, the optimal solution cost under abstraction π_P is at most the sum of costs of ρ_1 and ρ_2, and thus $h^P \leq h^{P_1} + h^{P_2}$.

E7.4 Summary

Summary (1)

- When faced with multiple PDB heuristics (a pattern collection), we want to admissibly add their values where possible, and maximize where addition is inadmissible.
- A set of patterns is additive if each operator affects (i.e., assigns to a variable from) at most one pattern in the set.
- The canonical heuristic function is the best possible additive/maximizing combination for a given pattern collection given this additivity criterion.

Summary (2)

Not all patterns need to be considered, as some are redundant:

- Patterns should include a goal variable (else $h^P = 0$).
- Patterns should only include causally relevant variables (others can be dropped without affecting the heuristic value).
- Patterns should be causally connected (disconnected patterns can be split into smaller subpatterns at no loss).

Planning and Optimization

E8. Pattern Databases: Pattern Selection

Malte Helmert and Gabriele Röger

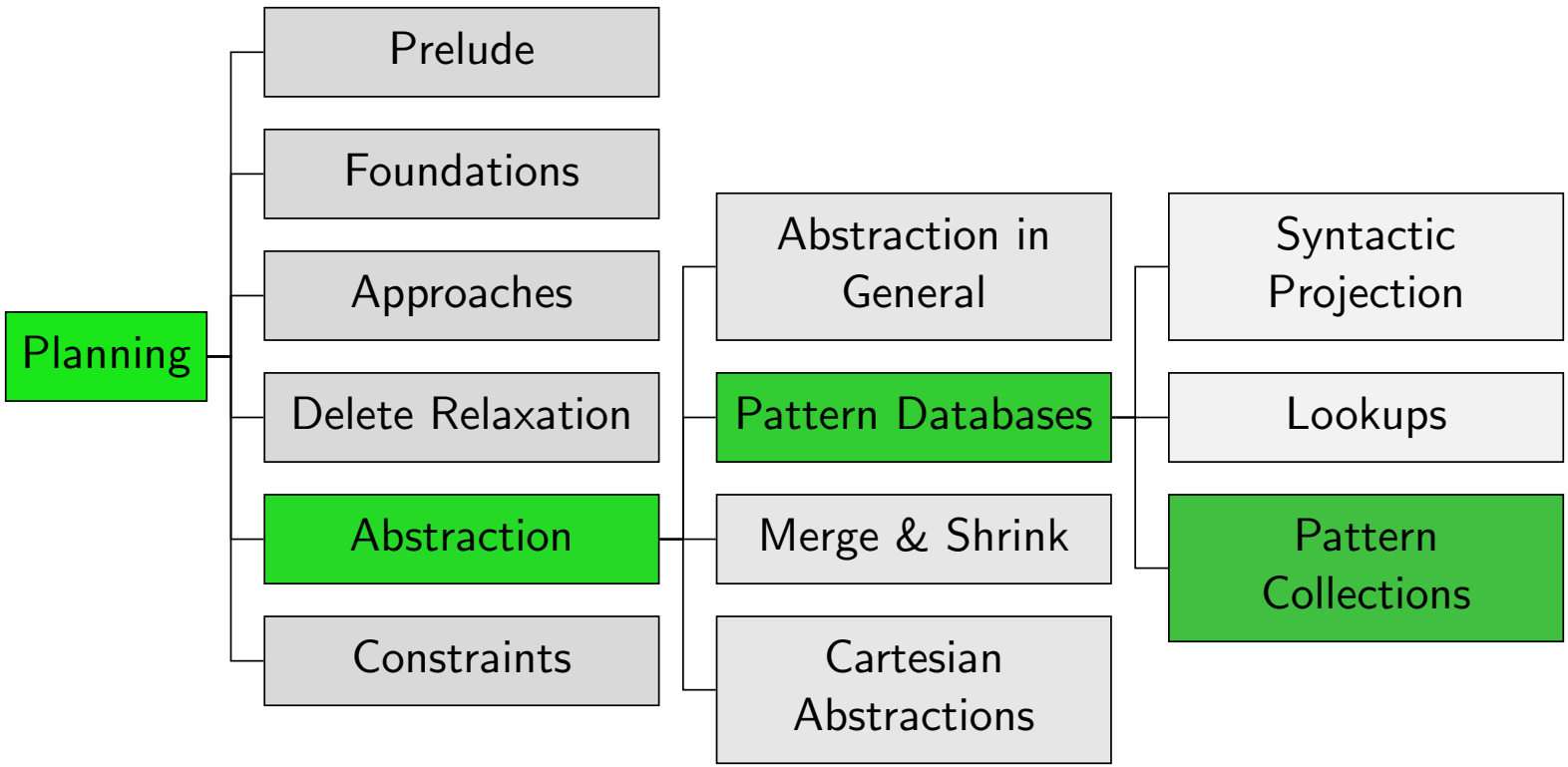
Universität Basel

November 12, 2025

Planning and Optimization

November 12, 2025 — E8. Pattern Databases: Pattern Selection

Content of the Course



E8.1 Pattern Selection as Local Search

Pattern Selection as an Optimization Problem

Only one question remains to be answered now
in order to apply PDBs to planning tasks in practice:

How do we automatically find a good pattern collection?

The Idea

Pattern selection can be cast as an optimization problem:

- Given: a set of candidates
  (= pattern collections which fit into a given memory limit)
- Find: a best possible candidate, or an approximation
  (= pattern collection with high heuristic quality)

Pattern Selection as Local Search

How to solve this optimization problem?

- For problems of interesting size, we cannot hope to find (and prove optimal) a globally optimal pattern collection.
  - Question: How many candidates are there?
- Instead, we try to find good solutions by local search.

Two approaches from the literature:

- Edelkamp (2007): using an evolutionary algorithm
- Haslum et al. (2007): using hill-climbing

$\rightsquigarrow$ in the following: main ideas of the second approach

Pattern Selection as Hill-Climbing

Reminder: Hill Climbing

current := an initial candidate
loop forever:
    next := a neighbour of *current* with maximum *quality*
    if *quality*(*next*) $\leq$ *quality*(*current*):
        return *current*
    current := *next*

more on hill climbing:

$\rightsquigarrow$ Foundations of Artificial Intelligence course FS 2025, Ch. C1–C2

Pattern Selection as Hill-Climbing

Reminder: Hill Climbing

current := an initial candidate
loop forever:
    next := a neighbour of *current* with maximum *quality*
    if *quality*(*next*) $\leq$ *quality*(*current*):
        return *current*
    current := *next*

Three questions to answer to use this for pattern selection:

1. initial candidate: What is the initial pattern collection?
2. neighbourhood: Which pattern collections are considered next starting from a given collection?
3. quality: How do we evaluate the quality of pattern collections?

E8.2 Search Neighbourhood

Search Neighbourhood: Basic Idea

The basic idea is that we

- start from small patterns with only a single variable,
- grow them by adding slightly larger patterns
- and prefer moving to pattern collections that improve the heuristic value of many states.

Initial Pattern Collection

1. Initial Candidate

The initial pattern collection is
$\{\{v\} \mid v \text{ is a state variable mentioned in the goal formula}\}$.

Motivation:

- patterns with one variable are the simplest possible ones and hence a natural starting point
- non-goal patterns are trivial ($\rightsquigarrow$ Chapter E7), so would be useless

Which Pattern Collections to Consider Next

From this initial pattern collection, we incrementally grow larger pattern collections to obtain an improved heuristic.

2. Neighbourhood

The neighbours of $\mathcal{C}$ are all pattern collections $\mathcal{C} \cup \{P'\}$ where

- $P' = P \cup \{v\}$ for some $P \in \mathcal{C}$,
- $P' \notin \mathcal{C}$,
- all variables of P' are causally relevant for P',
- P' is causally connected, and
- all pattern databases in $\mathcal{C} \cup \{P'\}$ can be represented within some prespecified space limit.

$\rightsquigarrow$ add one pattern with one additional variable at a time

$\rightsquigarrow$ use criteria for redundant patterns ($\rightsquigarrow$ Chapter E7) to avoid neighbours that cannot improve the heuristic

Checking Causal Relevance and Connectivity

Remark: For causal relevance and connectivity, there is a sufficient and necessary criterion which is easy to check:

- v is a predecessor of some $u \in P$ in the causal graph, or
- v is a successor of some $u \in P$ in the causal graph and is mentioned in the goal formula.

Evaluating the Quality of Pattern Collections

- The last question we need to answer is how to evaluate the quality of pattern collections.
- This is perhaps the most critical point: without a good evaluation criterion, pattern collections are chosen blindly.

Approaches for Evaluating Heuristic Quality

Three approaches have been suggested:

- estimating the mean heuristic value of the resulting heuristic (Edelkamp, 2007)
- estimating search effort under the resulting heuristic using a model for predicting search effort (Haslum et al., 2007; Franco et al., 2017)
- sampling states in the state space and counting how many of them have improved heuristic values compared to the current pattern collection (Haslum et al., 2007)

The last approach is most commonly used and has been shown to work well experimentally.

Heuristic Quality by Improved Sample States

3. Quality

- Generate M states $s_1, \ldots, s_M$ through random walks in the state space from the initial state (according to certain parameters not discussed in detail).
- The degree of improvement of a pattern collection $\mathcal{C}'$ which is generated as a successor of collection $\mathcal{C}$ is the number of sample states s_i for which $h^{\mathcal{C}'}(s_i) > h^{\mathcal{C}}(s_i)$.
- Use the degree of improvement as the quality measure for $\mathcal{C}'$.

Computing $h^{\mathcal{C}'}(s)$

- So we need to compute $h^{\mathcal{C}'}(s)$ for some states s and each candidate successor collection $\mathcal{C}'$.
- We have PDBs for all patterns in $\mathcal{C}$, but not for the new pattern $P' \in \mathcal{C}'$ (of the form $P \cup \{v\}$ for some $P \in \mathcal{C}$).
- If possible, we want to avoid fully computing all PDBs for all neighbours.

Idea:

- For SAS$^+$ tasks Π, $h^{P'}(s)$ is identical to the optimal solution cost for the syntactic projection $\Pi|_{P'}$.
- We can use any optimal planning algorithm for this.
- In particular, we can use A* search using h^P as a heuristic.

E8.3 Literature

References (1)

References on planning with pattern databases:


Stefan Edelkamp.
Planning with Pattern Databases.
Proc. ECP 2001, pp. 13–24, 2001.
First paper on planning with pattern databases.


Stefan Edelkamp.
Symbolic Pattern Databases in Heuristic Search Planning.
Proc. AIPS 2002, pp. 274–283, 2002.
Uses BDDs to store pattern databases more compactly.

References (2)

References on planning with pattern databases:


Patrik Haslum, Blai Bonet and Héctor Geffner.
New Admissible Heuristics for Domain-Independent Planning.
Proc. AAAI 2005, pp. 1164–1168, 2005.
Introduces constrained PDBs.
First pattern selection methods based on heuristic quality.

References (3)

References on planning with pattern databases:


Stefan Edelkamp.
Automated Creation of Pattern Database Search Heuristics.
Proc. MoChArt 2006, pp. 121–135, 2007.
First search-based pattern selection method.


Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet and Sven Koenig.
Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning.
Proc. AAAI 2007, pp. 1007–1012, 2007.
Introduces canonical heuristic for pattern collections.
Search-based pattern selection based on Korf, Reid & Edelkamp's theory for search effort estimation.

References (4)

References on planning with pattern databases:


Santiago Franco, Álvaro Torralba, Levi H. S. Lelis and Mike Barley.
On Creating Complementary Pattern Databases
Proc. IJCAI 2017, pp. 4302–4309, 2017.
Improved version of Edelkamp's pattern collection selection approach evaluating pattern collections based on a prediction of A* search effort.

E8.4 Summary

Summary

- One way to automatically find a good pattern collection is by searching in the space of pattern collections.
- One such approach uses hill-climbing search
  - starting from single-variable patterns
  - adding patterns with one additional variable at a time
  - evaluating patterns by the number of improved sample states
- By exploiting what we know about redundant patterns, the hill-climbing search space can be reduced significantly.

Planning and Optimization

E9. Merge-and-Shrink: Factored Transition Systems

Malte Helmert and Gabriele Röger

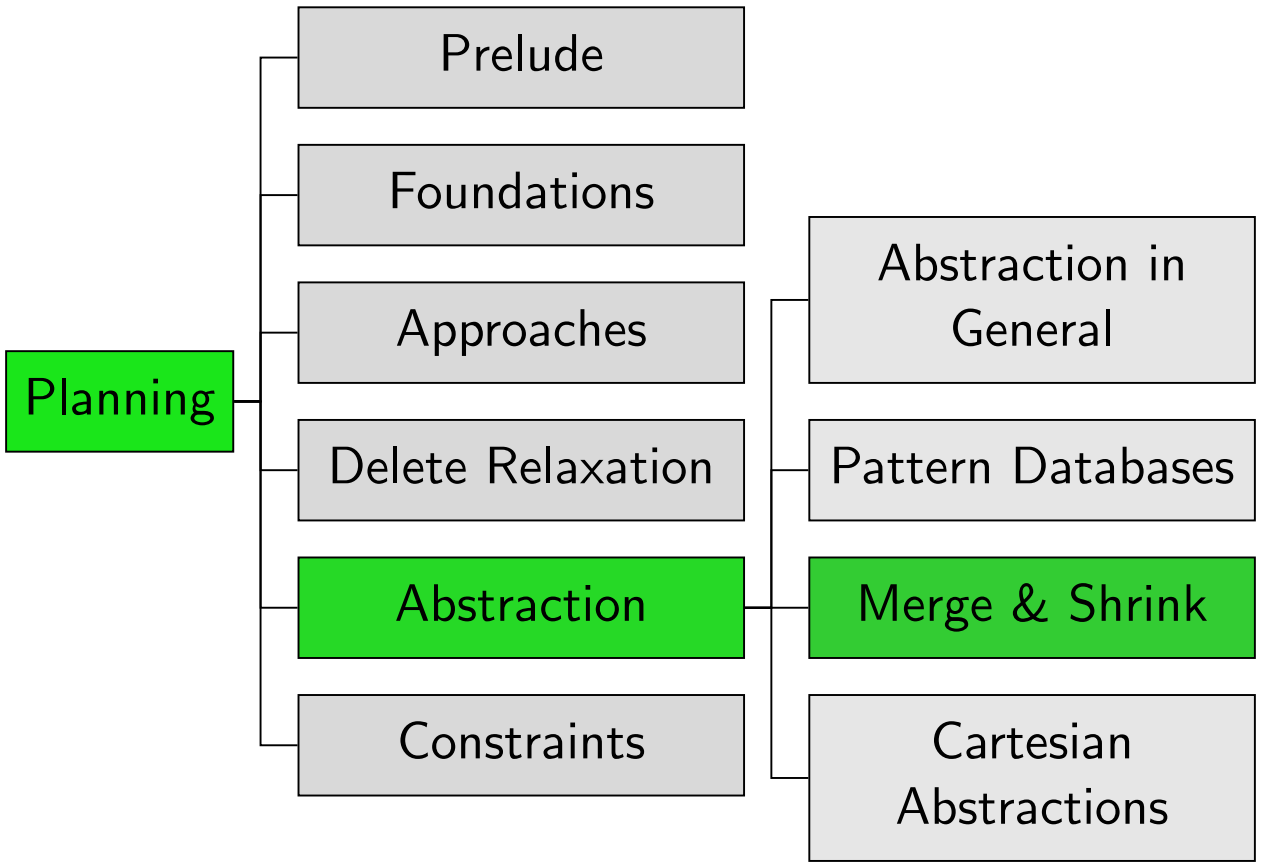
Universität Basel

November 17, 2025

Planning and Optimization

November 17, 2025 — E9. Merge-and-Shrink: Factored Transition Systems

Content of the Course

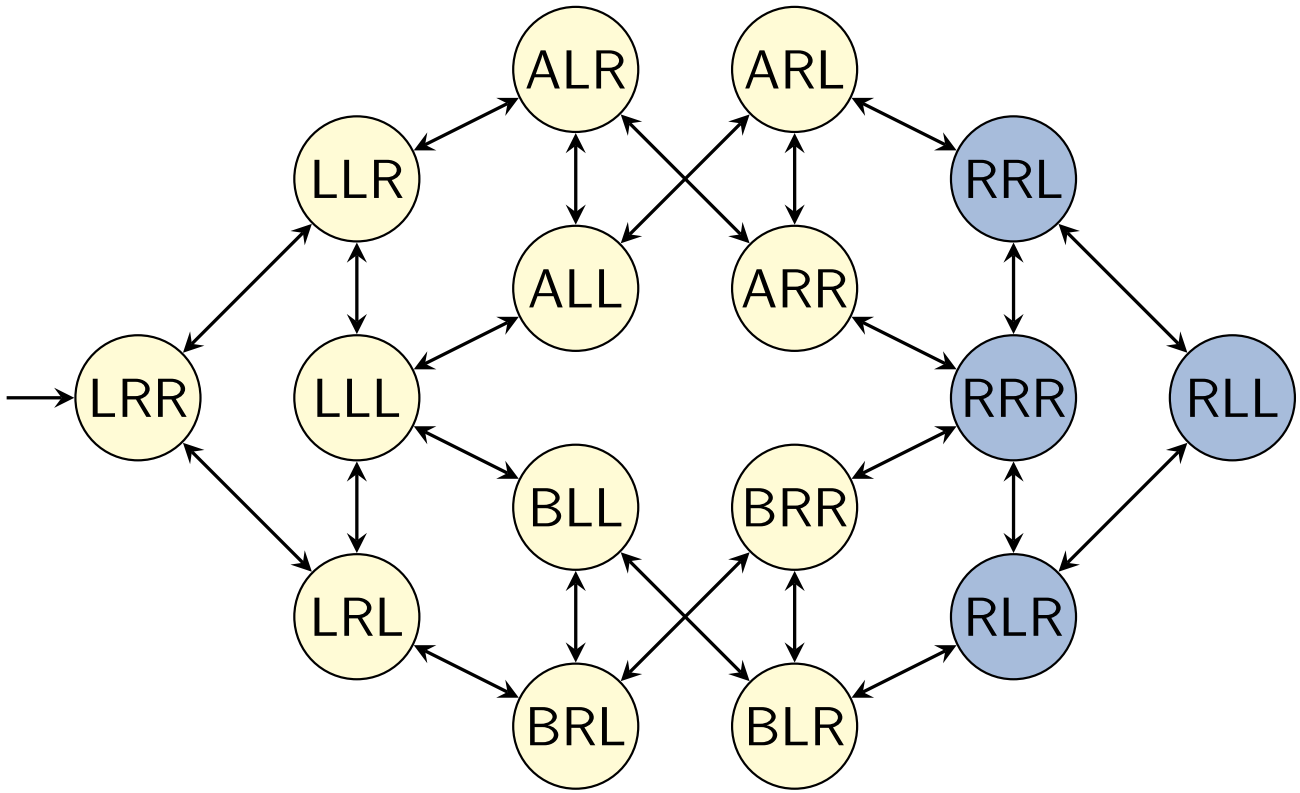


E9.1 Motivation

Beyond Pattern Databases

- Despite their popularity, pattern databases have some fundamental limitations ($\rightsquigarrow$ example on next slides).
- Today and next time, we study a class of abstractions called merge-and-shrink abstractions.
- Merge-and-shrink abstractions can be seen as a proper generalization of pattern databases.
  - They can do everything that pattern databases can do (modulo polynomial extra effort).
  - They can do some things that pattern databases cannot.

Back to the Running Example

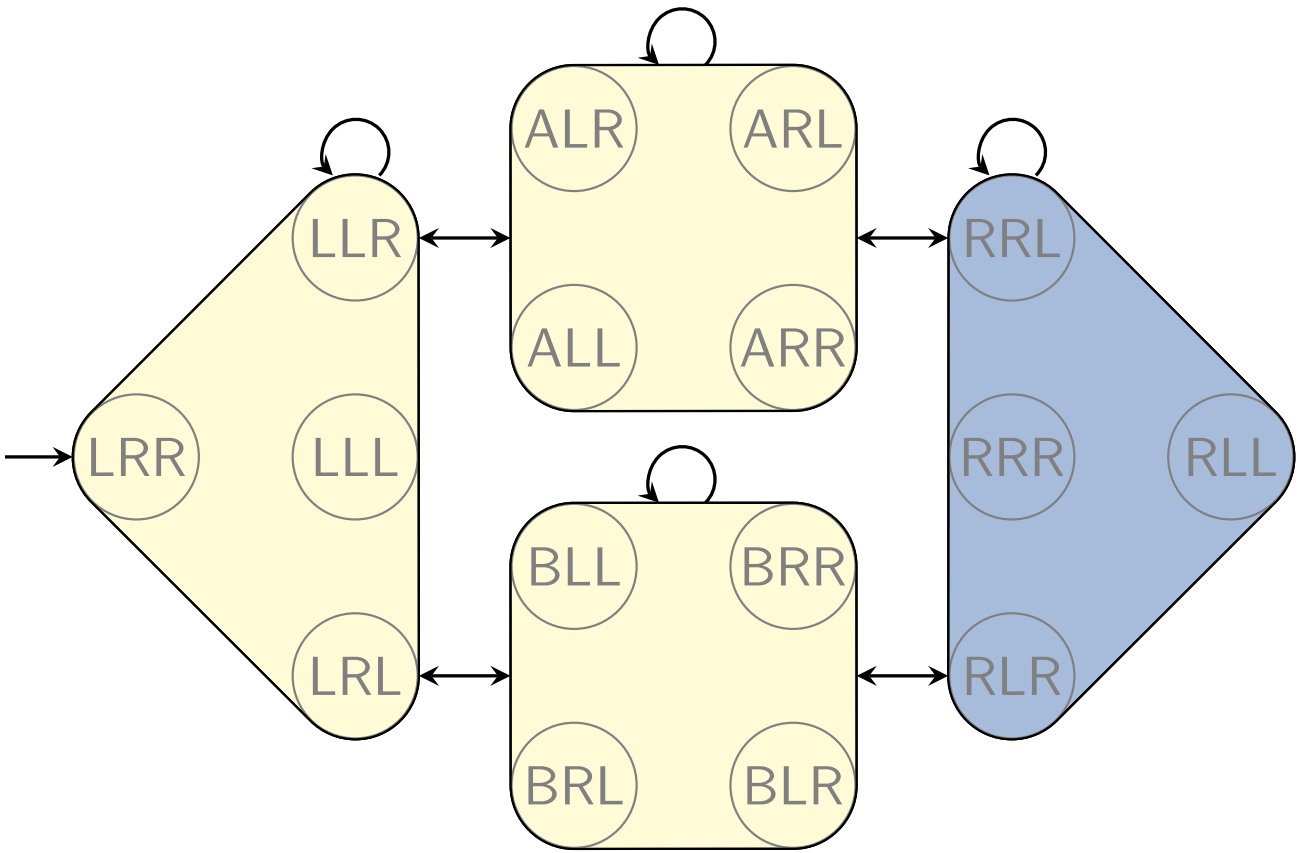


Logistics problem with one package, two trucks, two locations:

- state variable package: $\{L, R, A, B\}$
- state variable truck A: $\{L, R\}$
- state variable truck B: $\{L, R\}$

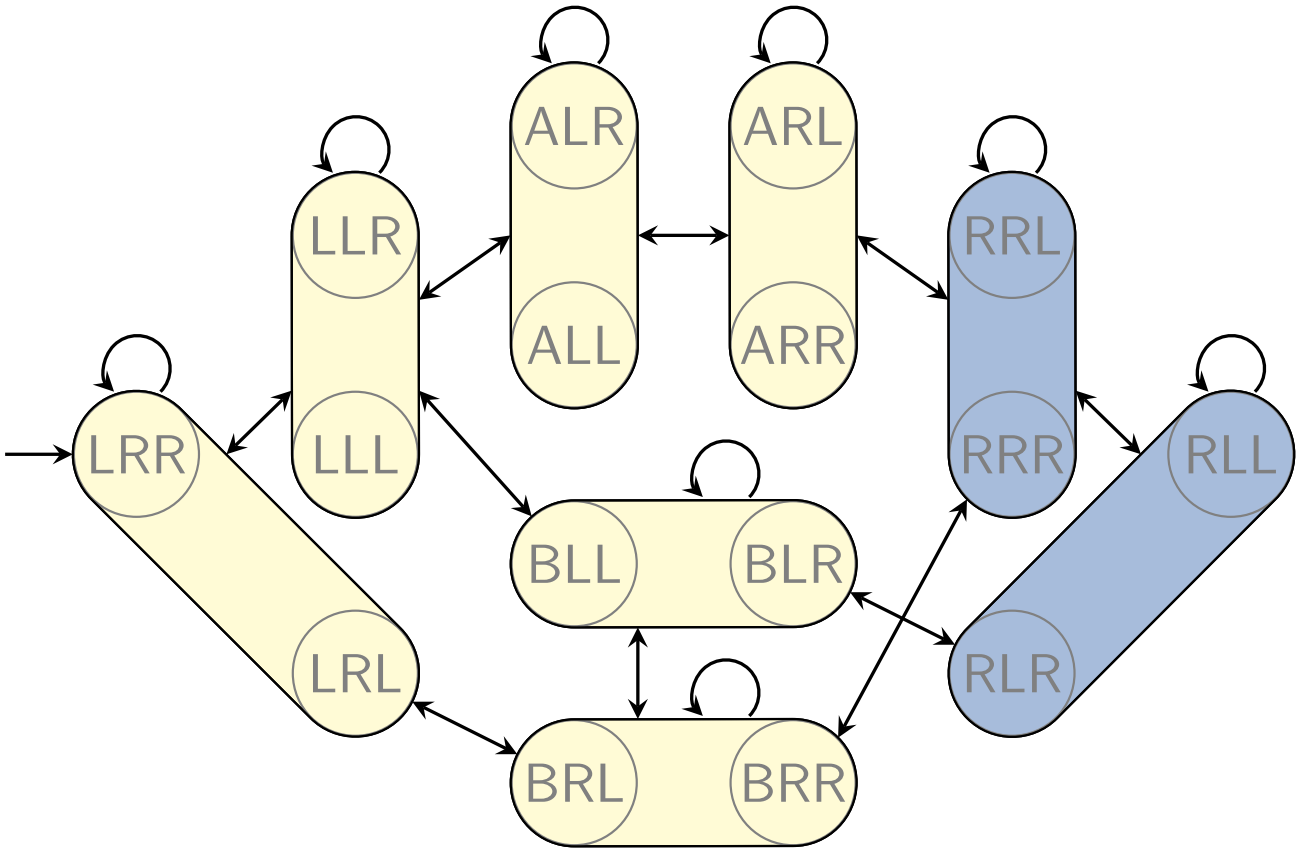
Example: Projection (1)

$\mathcal{T}^{\pi_{\{\text{package}\}}}$:



Example: Projection (2)

$\mathcal{T}^{\pi_{\{\text{package,truck A}\}}}$:



Limitations of Projections

How accurate is the PDB heuristic?

- consider generalization of the example:
  N trucks, 1 package
- consider any pattern that is a proper subset of variable set V
- $h(s_0) \leq 2 \rightsquigarrow$ no better than atomic projection to package

These values cannot be improved by maximizing
over several patterns or using additive patterns.

Merge-and-shrink abstractions can represent heuristics
with $h(s_0) \geq 3$ for tasks of this kind of any size.
Time and space requirements are linear in N.

(In fact, with time/space $O(N^2)$ we can construct a merge-and-shrink abstraction that gives the perfect heuristic h^* for such tasks, but we do not show this here.)

E9.2 Main Idea

Merge-and-Shrink Abstractions: Main Idea

Main Idea of Merge-and-shrink Abstractions

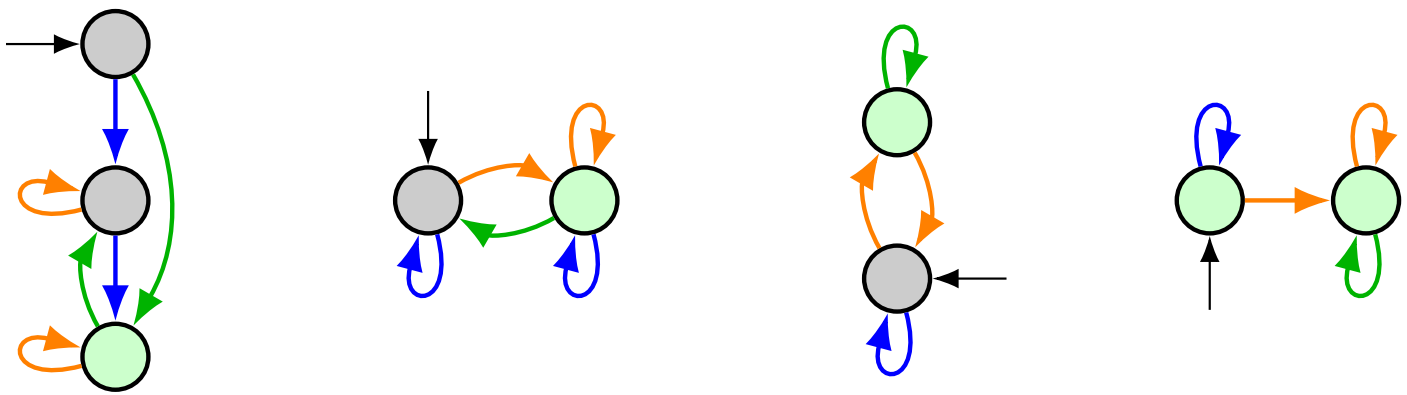
(due to Dräger, Finkbeiner & Podelski, 2006):

Instead of perfectly reflecting a few state variables,
reflect all state variables, but in a potentially lossy way.

- Represent planning task as factored transition system (FTS):
  a set of (small) abstract transition systems (factors)
  that jointly represent the full transition system of the task.
- Iteratively transform FTS by:
  - merging: combining two factors into one
  - shrinking: reducing the size of a single factor by abstraction
- When only a single factor is left, its goal distances
  are the merge-and-shrink heuristic values.

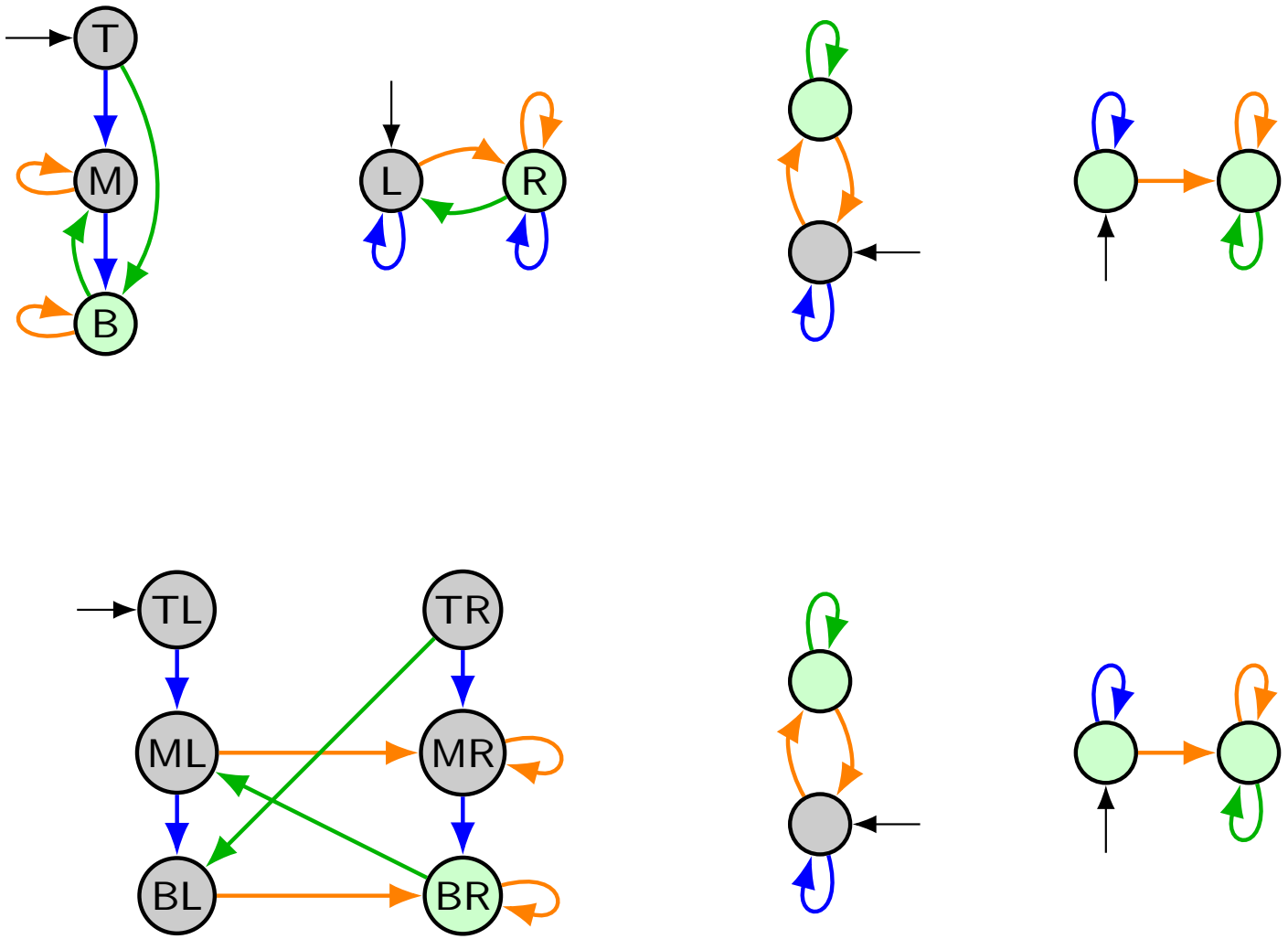
Merge-and-Shrink Abstractions: Idea

Start from atomic factors (projections to single state variables)



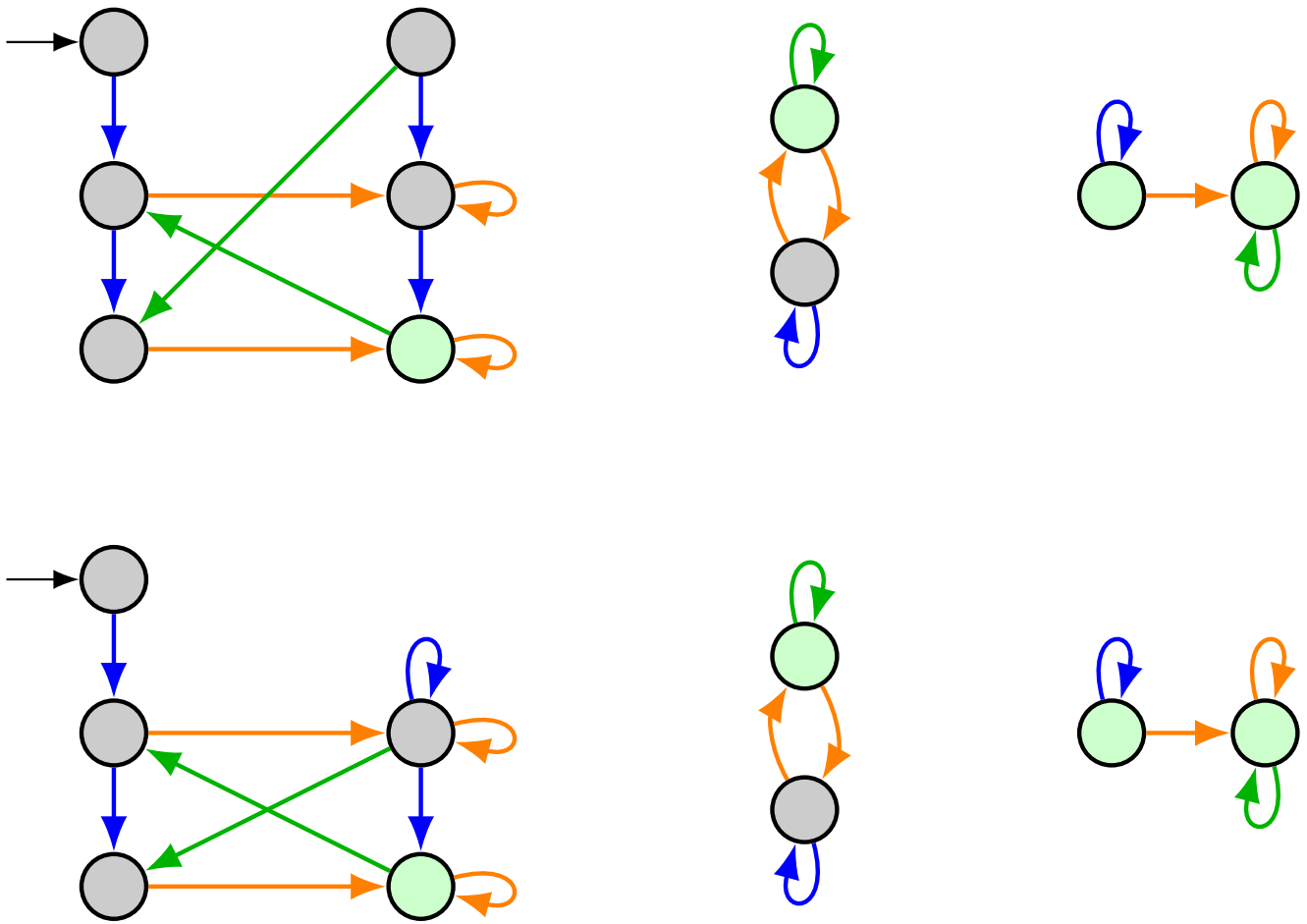
Merge-and-Shrink Abstractions: Idea

Merge: replace two factors with their product



Merge-and-Shrink Abstractions: Idea

Shrink: replace a factor by an abstraction of it



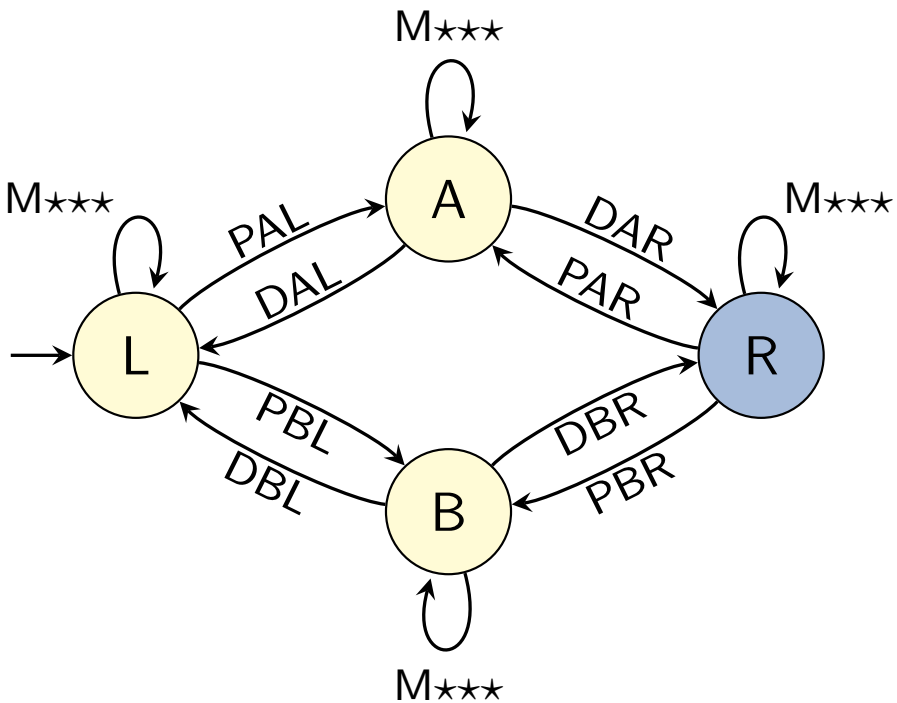
E9.3 Atomic Projections

Running Example: Explanations

- Atomic projections (projections to a single state variable) play an important role for merge-and-shrink abstractions.
- Unlike previous chapters, transition labels are critically important for merge-and-shrink.
- Hence we now look at the transition systems for atomic projections of our example task, including transition labels.
- We abbreviate labels (operator names) as in these examples:
  - MALR: move truck A from left to right
  - DAR: drop package from truck A at right location
  - PBL: pick up package with truck B at left location
- We abbreviate parallel arcs with commas and wildcards ($\star$) as in these examples:
  - PAL, DAL: two parallel arcs labeled PAL and DAL
  - MA$\star\star$: two parallel arcs labeled MALR and MARL

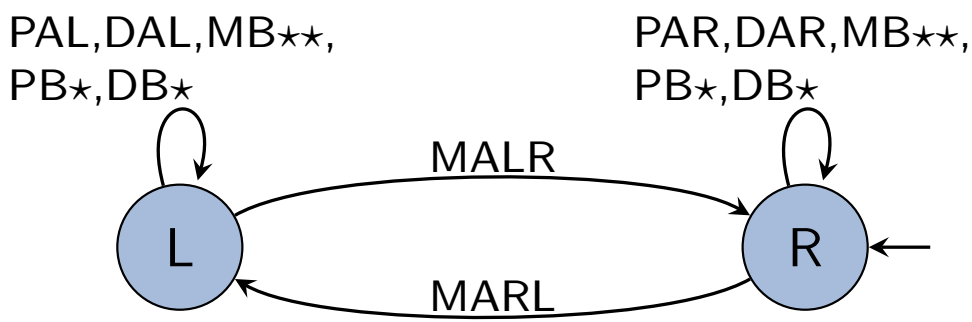
Running Example: Atomic Projection for Package

$\mathcal{T}^{\pi_{\{\text{package}\}}}$:



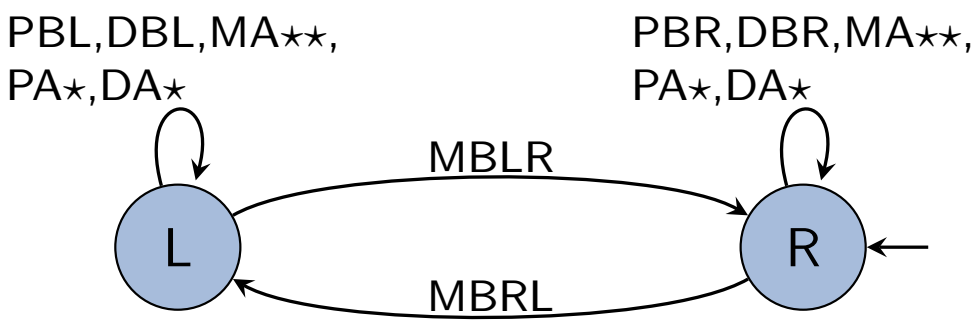
Running Example: Atomic Projection for Truck A

$\mathcal{T}^{\pi_{\{\text{truck A}\}}}$:



Running Example: Atomic Projection for Truck B

$\mathcal{T}^{\pi_{\{\text{truck B}\}}}$:



E9.4 Synchronized Product

Synchronized Product: Idea

- Given two abstract transition systems with the same labels, we can compute a product transition system.
- The product transition system captures all information of both transition systems.
- A sequence of labels is a solution for the product iff it is a solution for both factors.

Synchronized Product of Transition Systems

Definition (Synchronized Product of Transition Systems)

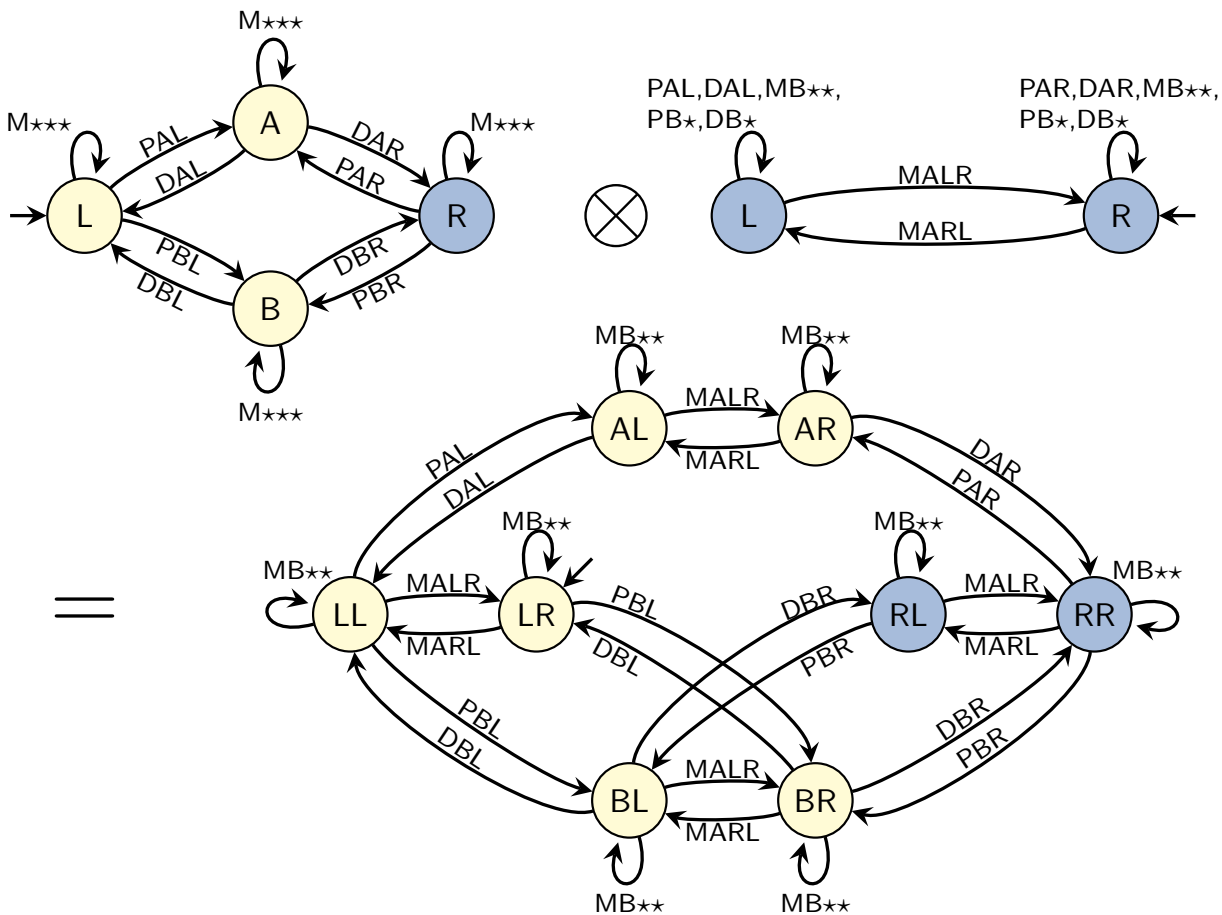
For $i \in \{1, 2\}$, let $\mathcal{T}_i = \langle S_i, L, c, T_i, s_{0i}, S_{\star i} \rangle$ be transition systems with the same labels and cost function.

The synchronized product of $\mathcal{T}_1$ and $\mathcal{T}_2$, in symbols $\mathcal{T}_1 \otimes \mathcal{T}_2$, is the transition system $\mathcal{T}_\otimes = \langle S_\otimes, L, c, T_\otimes, s_{0\otimes}, S_{\star\otimes} \rangle$ with

- $S_\otimes = S_1 \times S_2$
- $T_\otimes = \{\langle s_1, s_2 \rangle \xrightarrow{\ell} \langle t_1, t_2 \rangle \mid s_1 \xrightarrow{\ell} t_1 \in T_1 \text{ and } s_2 \xrightarrow{\ell} t_2 \in T_2\}$
- $s_{0\otimes} = \langle s_{01}, s_{02} \rangle$
- $S_{\star\otimes} = S_{\star 1} \times S_{\star 2}$

Example: Synchronized Product

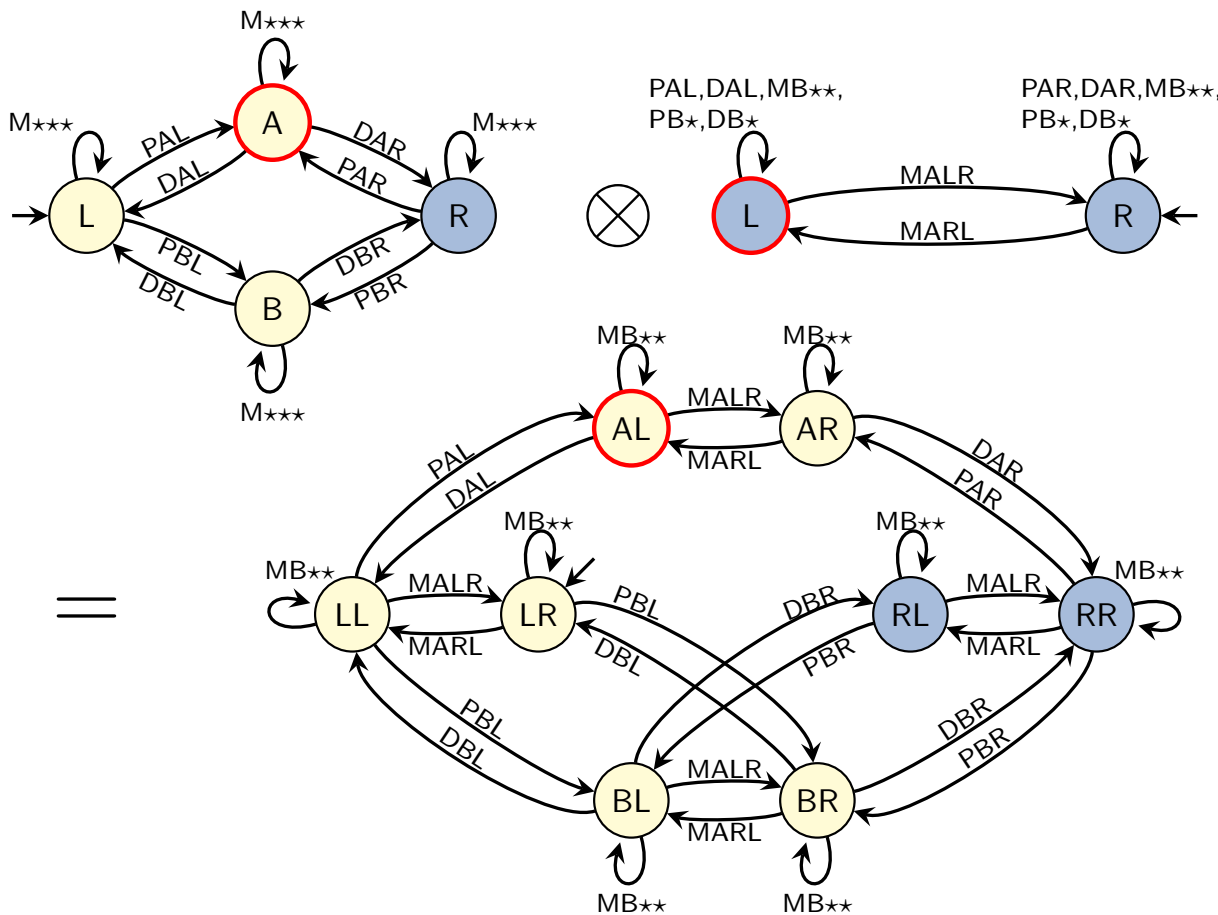
$\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\}$:



Example: Synchronized Product

$\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\}$:

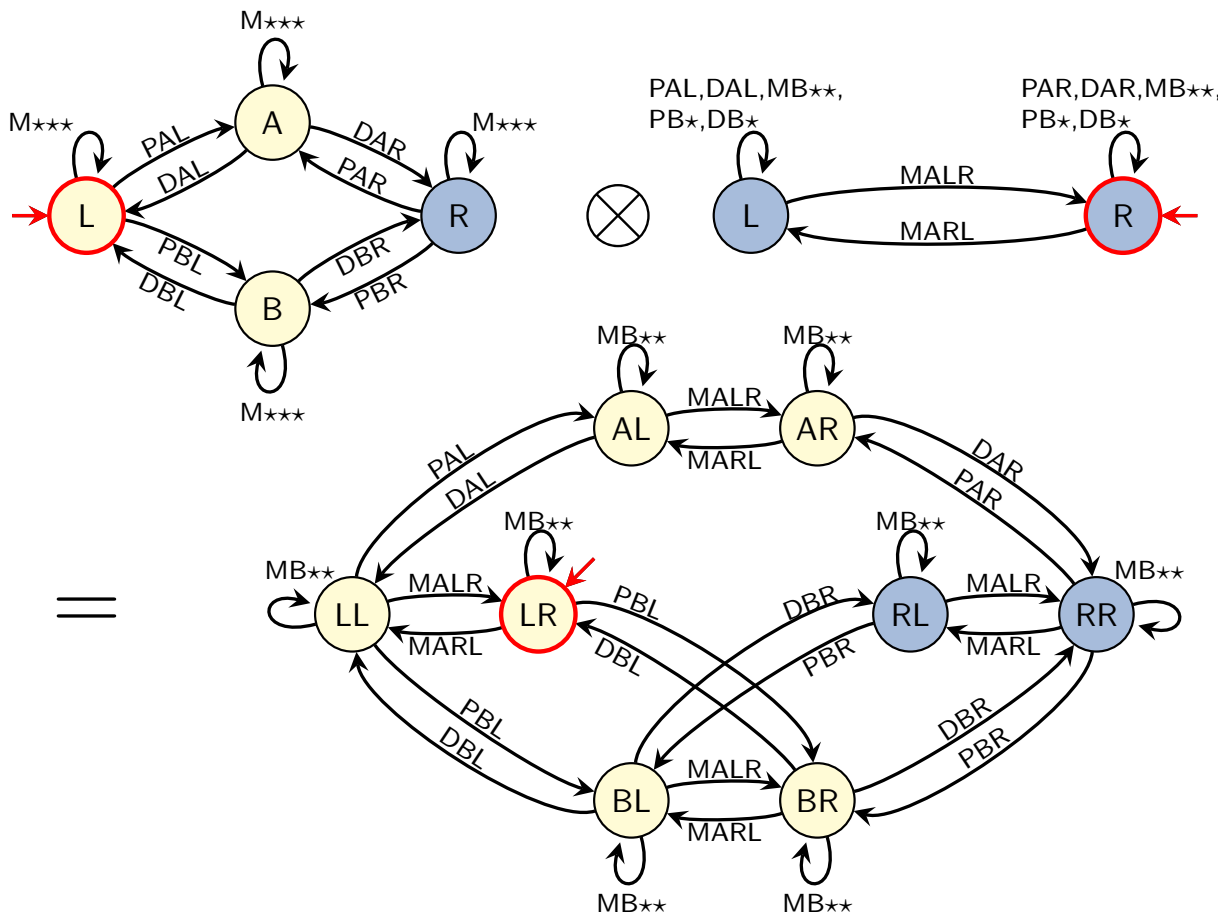
$S_{\otimes} = S_1 \times S_2$



Example: Synchronized Product

$\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\}$:

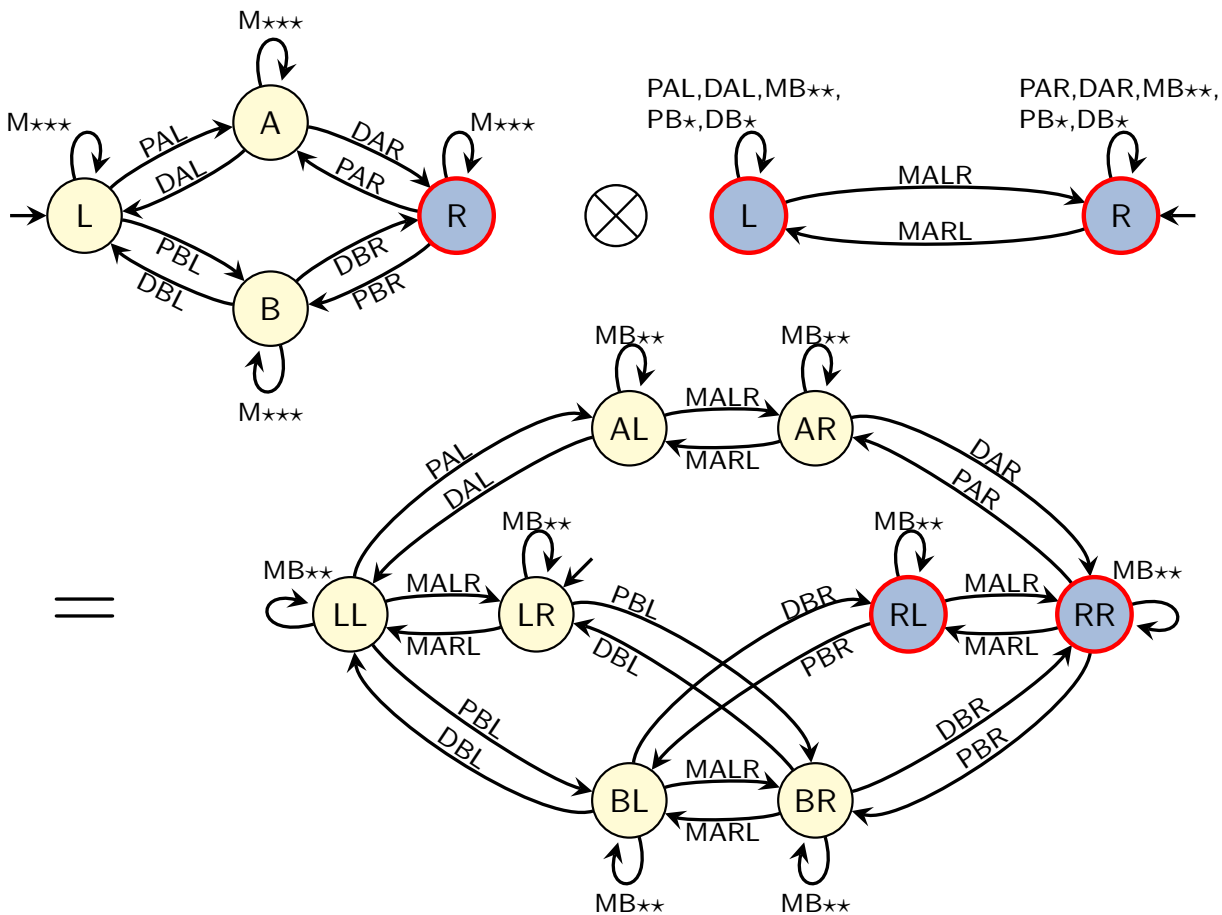
$s_{0\otimes} = \langle s_{01}, s_{02} \rangle$



Example: Synchronized Product

$\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\}$:

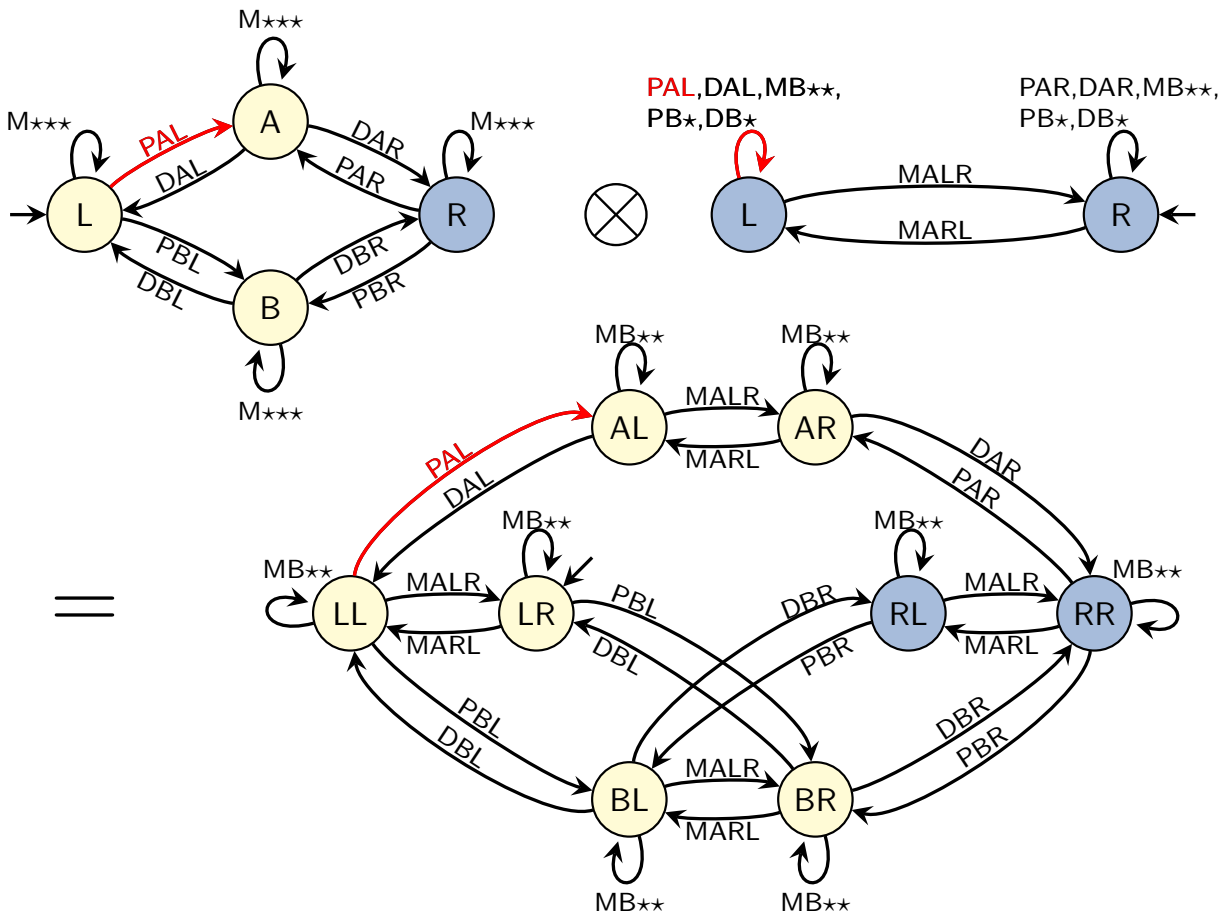
$S_{\star\otimes} = S_{\star 1} \times S_{\star 2}$



Example: Synchronized Product

$\mathcal{T}^{\pi\{\text{package}\}} \otimes \mathcal{T}^{\pi\{\text{truck A}\}}$:

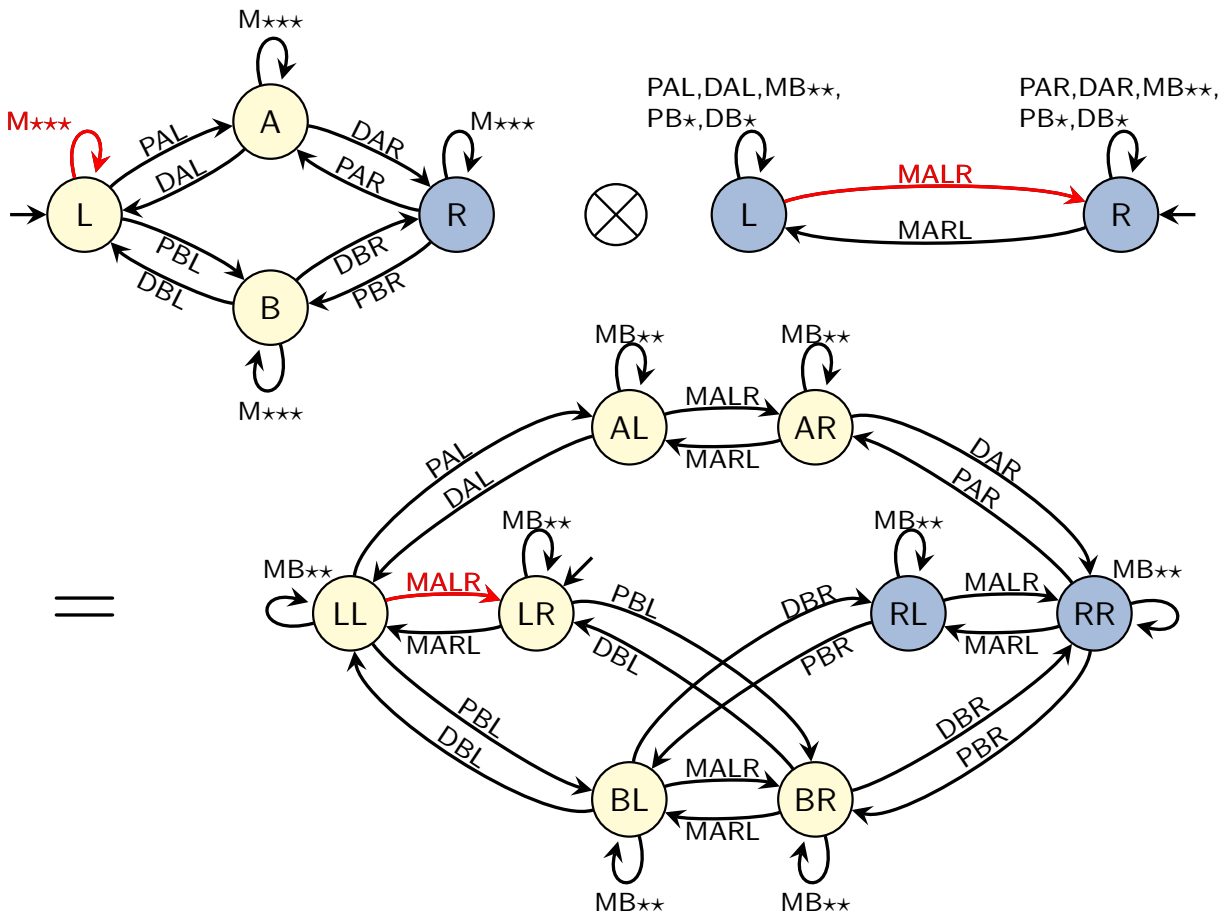
$T_\otimes = \{\langle s_1, s_2\rangle \xrightarrow{\ell} \langle t_1, t_2\rangle \mid s_1 \xrightarrow{\ell} t_1 \in T_1 \text{ and } s_2 \xrightarrow{\ell} t_2 \in T_2\}$



Example: Synchronized Product

$\mathcal{T}^{\pi\{\text{package}\}} \otimes \mathcal{T}^{\pi\{\text{truck A}\}}$:

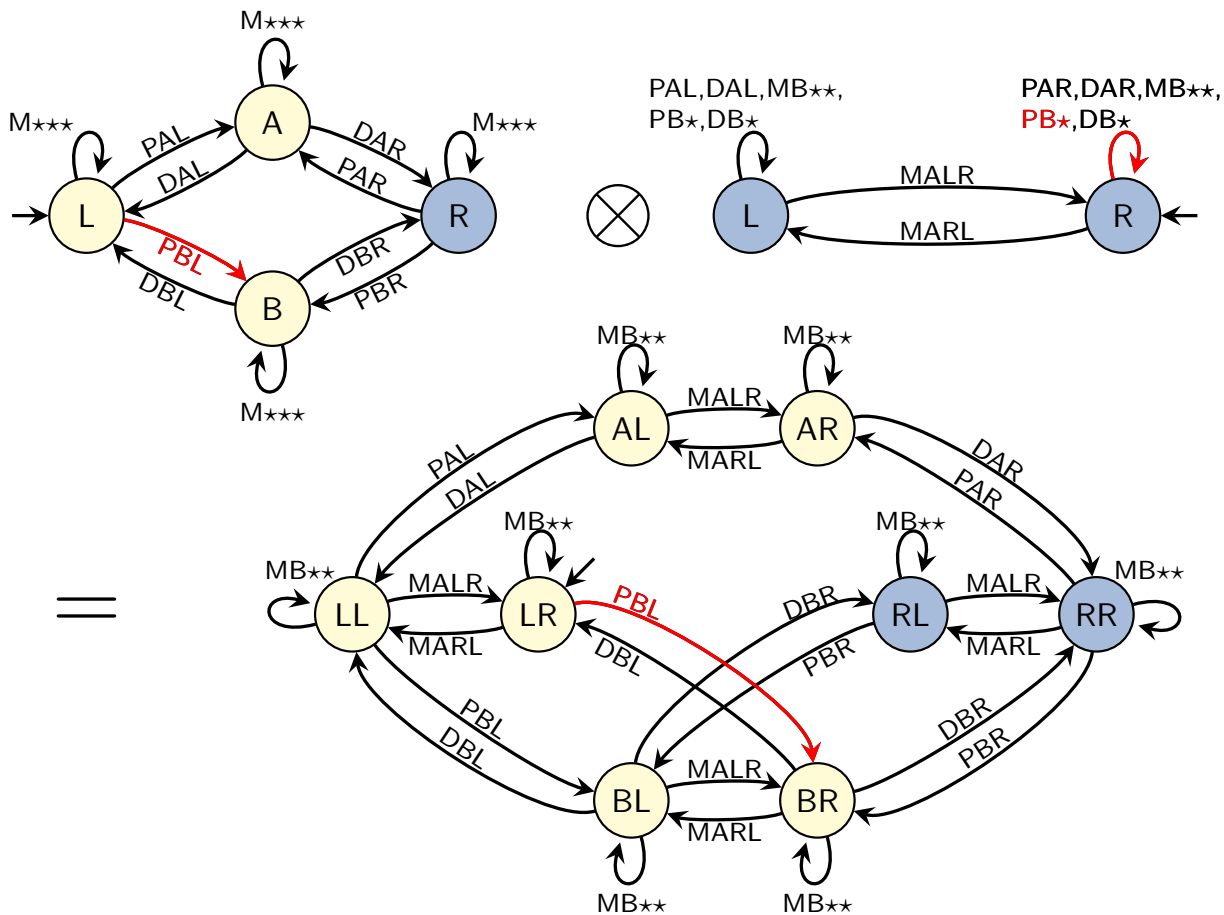
$T_\otimes = \{\langle s_1, s_2\rangle \xrightarrow{\ell} \langle t_1, t_2\rangle \mid s_1 \xrightarrow{\ell} t_1 \in T_1 \text{ and } s_2 \xrightarrow{\ell} t_2 \in T_2\}$



Example: Synchronized Product

$\mathcal{T}^{\pi\{\text{package}\}} \otimes \mathcal{T}^{\pi\{\text{truck A}\}}$:

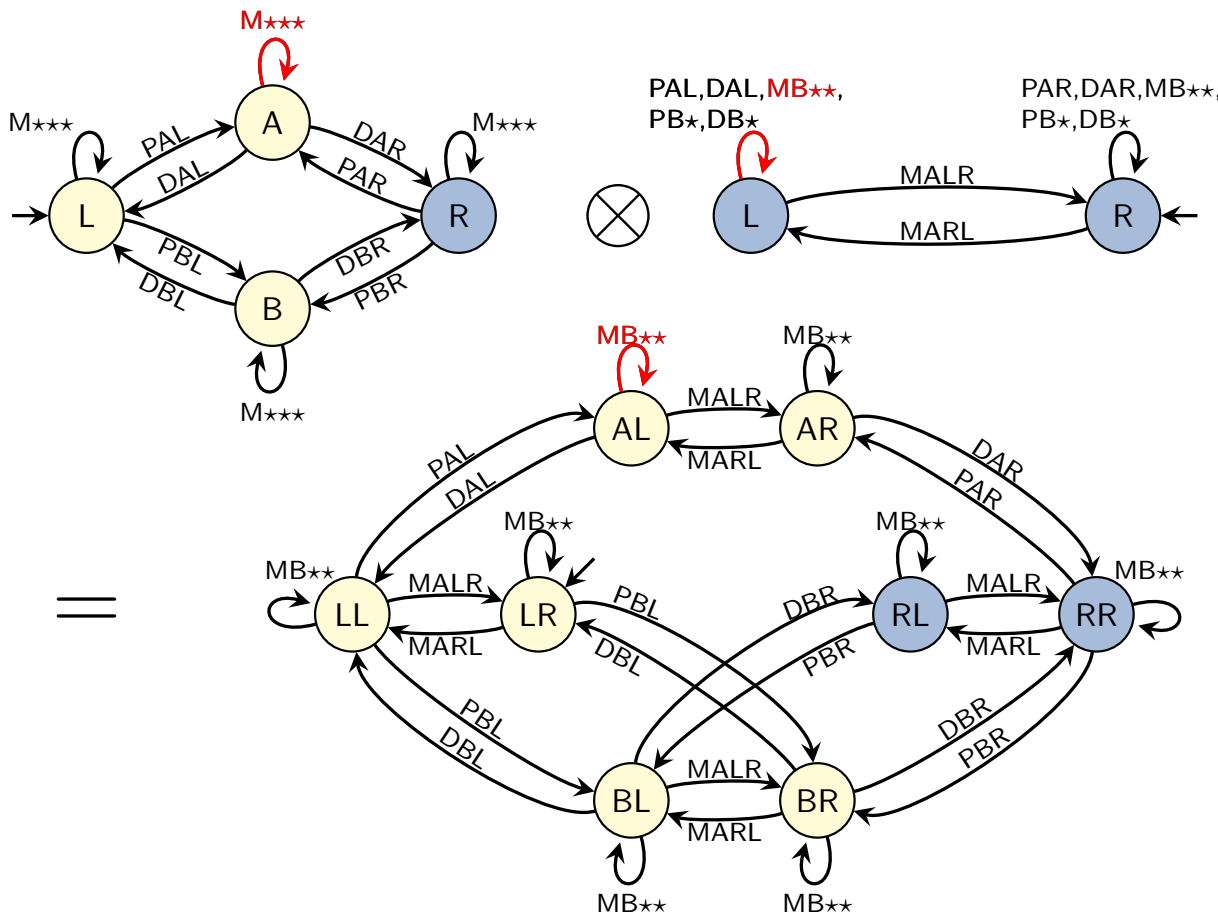
$T_\otimes = \{\langle s_1, s_2\rangle \xrightarrow{\ell} \langle t_1, t_2\rangle \mid s_1 \xrightarrow{\ell} t_1 \in T_1 \text{ and } s_2 \xrightarrow{\ell} t_2 \in T_2\}$



Example: Synchronized Product

$\mathcal{T}^{\pi\{\text{package}\}} \otimes \mathcal{T}^{\pi\{\text{truck A}\}}$:

$T_\otimes = \{\langle s_1, s_2\rangle \xrightarrow{\ell} \langle t_1, t_2\rangle \mid s_1 \xrightarrow{\ell} t_1 \in T_1 \text{ and } s_2 \xrightarrow{\ell} t_2 \in T_2\}$



Associativity and Commutativity

- Up to isomorphism ("names of states"), products are associative and commutative:
  - $(\mathcal{T} \otimes \mathcal{T}') \otimes \mathcal{T}'' \sim \mathcal{T} \otimes (\mathcal{T}' \otimes \mathcal{T}'')$
  - $\mathcal{T} \otimes \mathcal{T}' \sim \mathcal{T}' \otimes \mathcal{T}$
- We do not care about names of states and thus treat products as associative and commutative.
- We can then define the product of a set $F = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ of transition systems: $\bigotimes F := \mathcal{T}_1 \otimes \ldots \otimes \mathcal{T}_n$

E9.5 Factored Transition Systems

Factored Transition System

Definition (Factored Transition System)

A finite set $F = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ of transition systems with the same labels and cost function is called a factored transition system (FTS).

F represents the transition system $\bigotimes F$.

A planning task gives rise to an FTS via its atomic projections:

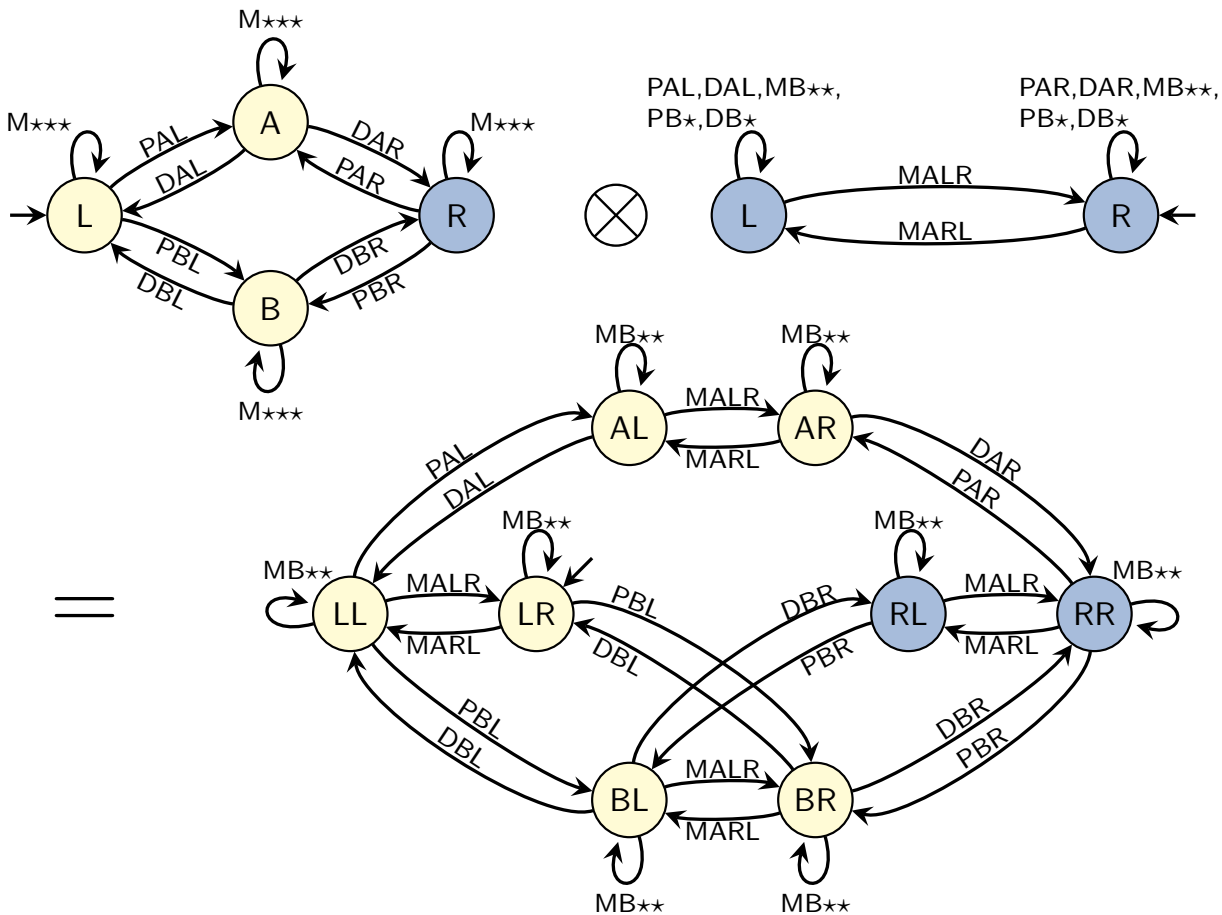
Definition (Factored Transition System Induced by Planning Task)

Let Π be a planning task with state variables V.

The factored transition system induced by Π is the FTS $F(\Pi) = \{\mathcal{T}^{\pi_{\{v\}}} \mid v \in V\}$.

Back to the Example Product

$\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\}$:



We have $\mathcal{T}^{\pi}\{\text{package}\} \otimes \mathcal{T}^{\pi}\{\text{truck A}\} \sim \mathcal{T}^{\pi}\{\text{package,truck A}\}$. Coincidence?

Products of Projections

Theorem (Products of Projections)

Let Π be a SAS$^+$ planning task with variable set V,
and let V_1 and V_2 be disjoint subsets of V.

Then $\mathcal{T}^{\pi^{V_1}} \otimes \mathcal{T}^{\pi^{V_2}} \sim \mathcal{T}^{\pi^{V_1 \cup V_2}}$.

(Proof omitted.)

$\rightsquigarrow$ products allow us to build finer projections from coarser ones

Recovering $\mathcal{T}(\Pi)$ from the Factored Transition System

- By repeated application of the theorem, we can recover all pattern database heuristics of a SAS^+ planning task as products of atomic factors.
- Moreover, by computing the product of all atomic projections, we can recover the identity abstraction $\mathrm{id} = \pi_V$.

This implies:

Corollary (Recovering $\mathcal{T}(\Pi)$ from the Factored Transition System)

Let Π be a SAS^+ planning task. Then $\bigotimes F(\Pi) \sim \mathcal{T}(\Pi)$.

This is an important result because it shows that $F(\Pi)$ represents all important information about Π.

E9.6 Summary

Summary

- A factored transition system is a set of transition systems that represents a larger transition system by focusing on its individual components (factors).
- For planning tasks, these factors are the atomic projections (projections to single state variables).
- The synchronized product $\mathcal{T} \otimes \mathcal{T}'$ of two transition systems with the same labels captures their "joint behaviour".
- For SAS$^+$ tasks, all projections can be obtained as products of atomic projections.
- In particular, the product of all factors of a SAS$^+$ task results in the full transition system of the task.

Planning and Optimization

E10. Merge-and-Shrink: Algorithm

Malte Helmert and Gabriele Röger

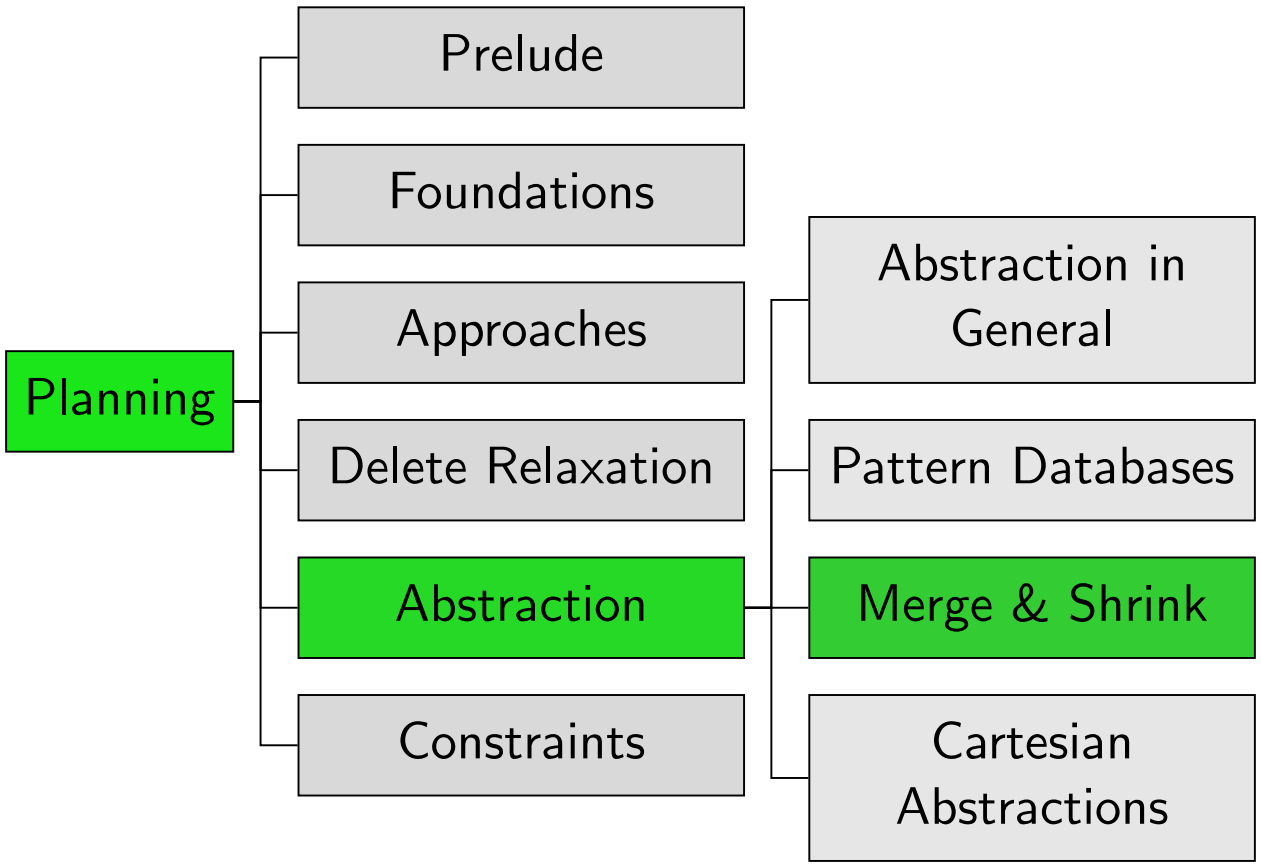
Universität Basel

November 17, 2025

Planning and Optimization

November 17, 2025 — E10. Merge-and-Shrink: Algorithm

Content of the Course



E10.1 Generic Algorithm

Generic Merge-and-shrink Abstractions: Outline

Using the results of the previous chapter, we can develop
a generic abstraction computation procedure
that takes all state variables into account.

- Initialization: Compute the FTS
  consisting of all atomic projections.
- Loop: Repeatedly apply a transformation to the FTS.
  - Merging: Combine two factors by replacing them
    with their synchronized product.
  - Shrinking: If the factors are too large,
    make one of them smaller by abstracting it further
    (applying an arbitrary abstraction to it).
- Termination: Stop when only one factor is left.

The final factor is then used for an abstraction heuristic.

Generic Algorithm Template

Generic Merge & Shrink Algorithm for planning task Π

$F := F(\Pi)$
while $|F| > 1$:
    select $type \in \{\text{merge}, \text{shrink}\}$
    if $type = \text{merge}$:
        select $\mathcal{T}_1, \mathcal{T}_2 \in F$
        $F := (F \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
    if $type = \text{shrink}$:
        select $\mathcal{T} \in F$
        choose an abstraction mapping β on $\mathcal{T}$
        $F := (F \setminus \{\mathcal{T}\}) \cup \{\mathcal{T}^\beta\}$
return the remaining factor $\mathcal{T}^\alpha$ in F

Merge-and-Shrink Strategies

Choices to resolve to instantiate the template:

- When to merge, when to shrink?
  $\rightsquigarrow$ general strategy
- Which abstractions to merge?
  $\rightsquigarrow$ merge strategy
- Which abstraction to shrink, and how to shrink it (which β)?
  $\rightsquigarrow$ shrink strategy

merge and shrink strategies $\rightsquigarrow$ Ch. E11/E12

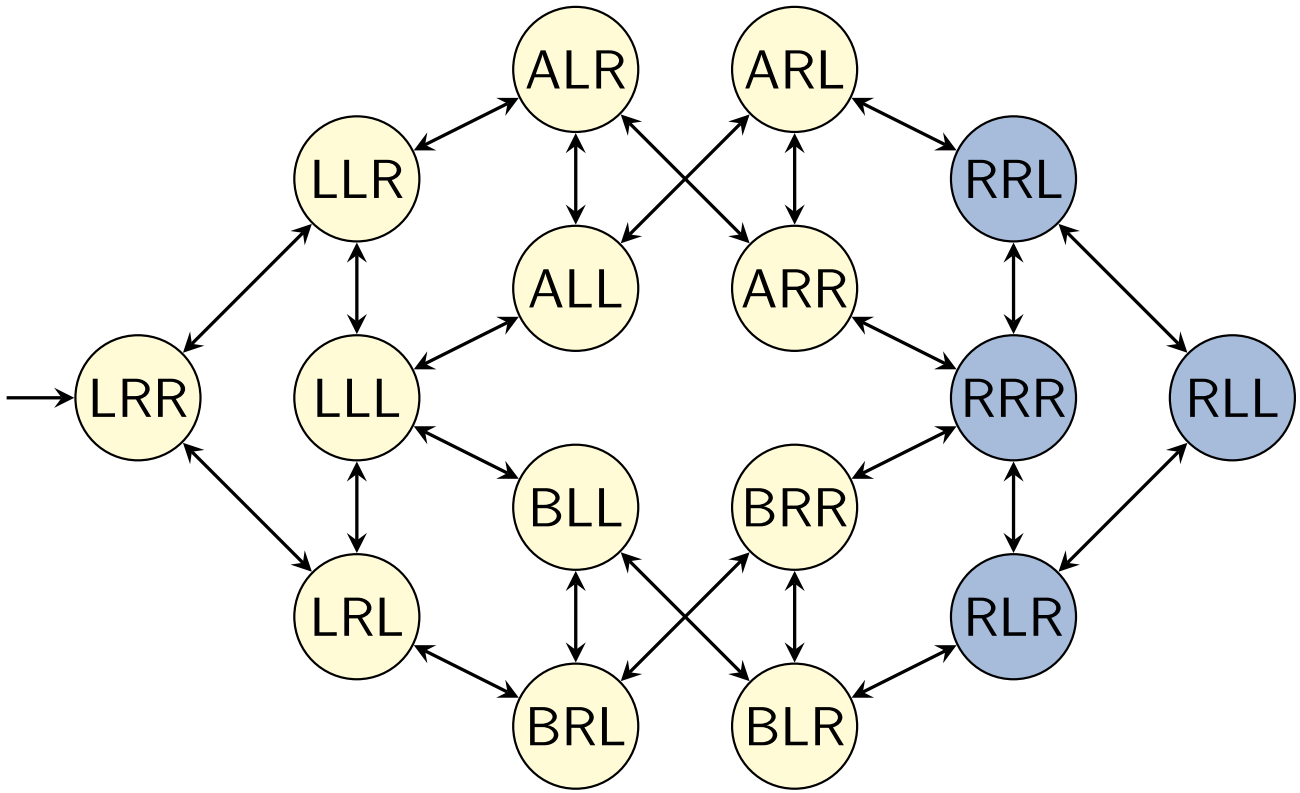
General Strategy

A typical general strategy:

- define a limit N on the number of states allowed in each factor
- in each iteration, select two factors we would like to merge
- merge them if this does not exhaust the state number limit
- otherwise shrink one or both factors just enough to make a subsequent merge possible

E10.2 Example

Back to the Running Example

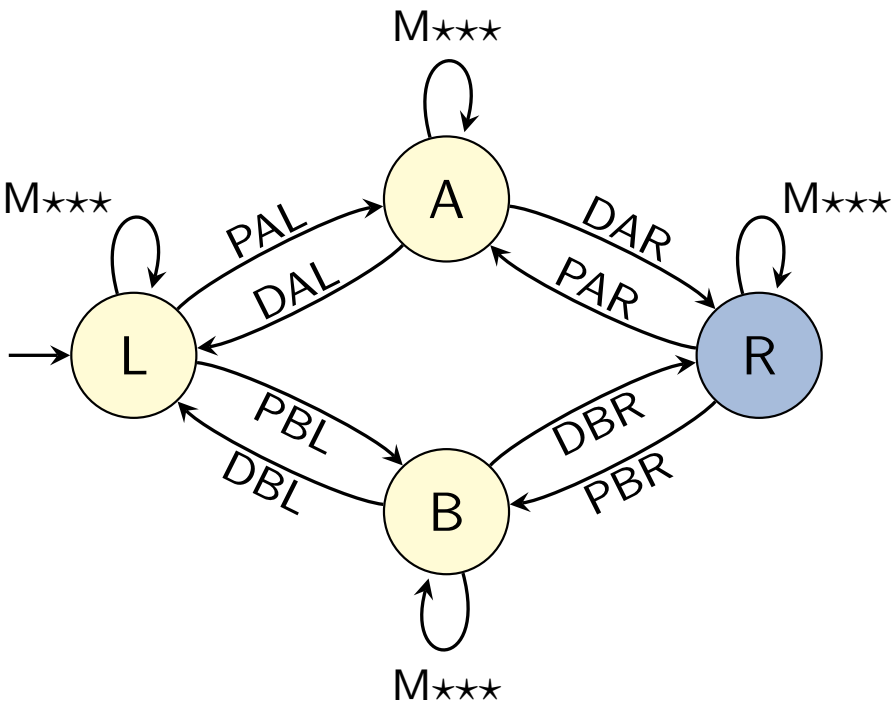


Logistics problem with one package, two trucks, two locations:

- state variable package: $\{L, R, A, B\}$
- state variable truck A: $\{L, R\}$
- state variable truck B: $\{L, R\}$

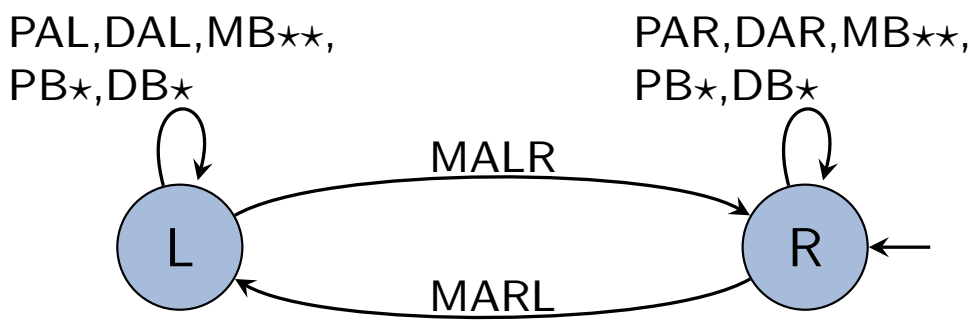
Initialization Step: Atomic Projection for Package

$\mathcal{T}^{\pi\{\text{package}\}}$:



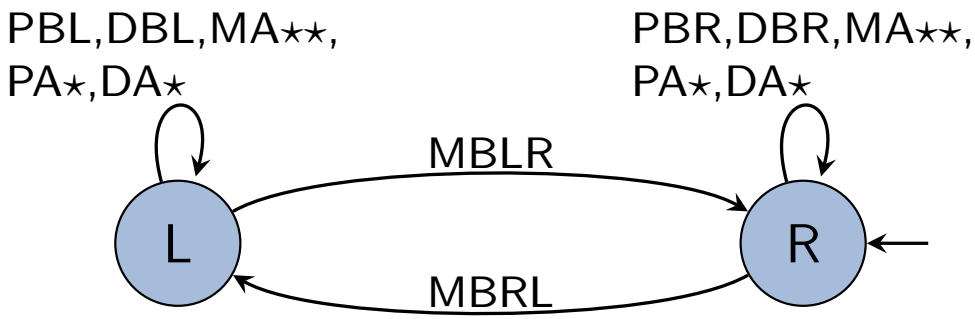
Initialization Step: Atomic Projection for Truck A

$\mathcal{T}^{\pi_{\{\text{truck A}\}}}$:



Initialization Step: Atomic Projection for Truck B

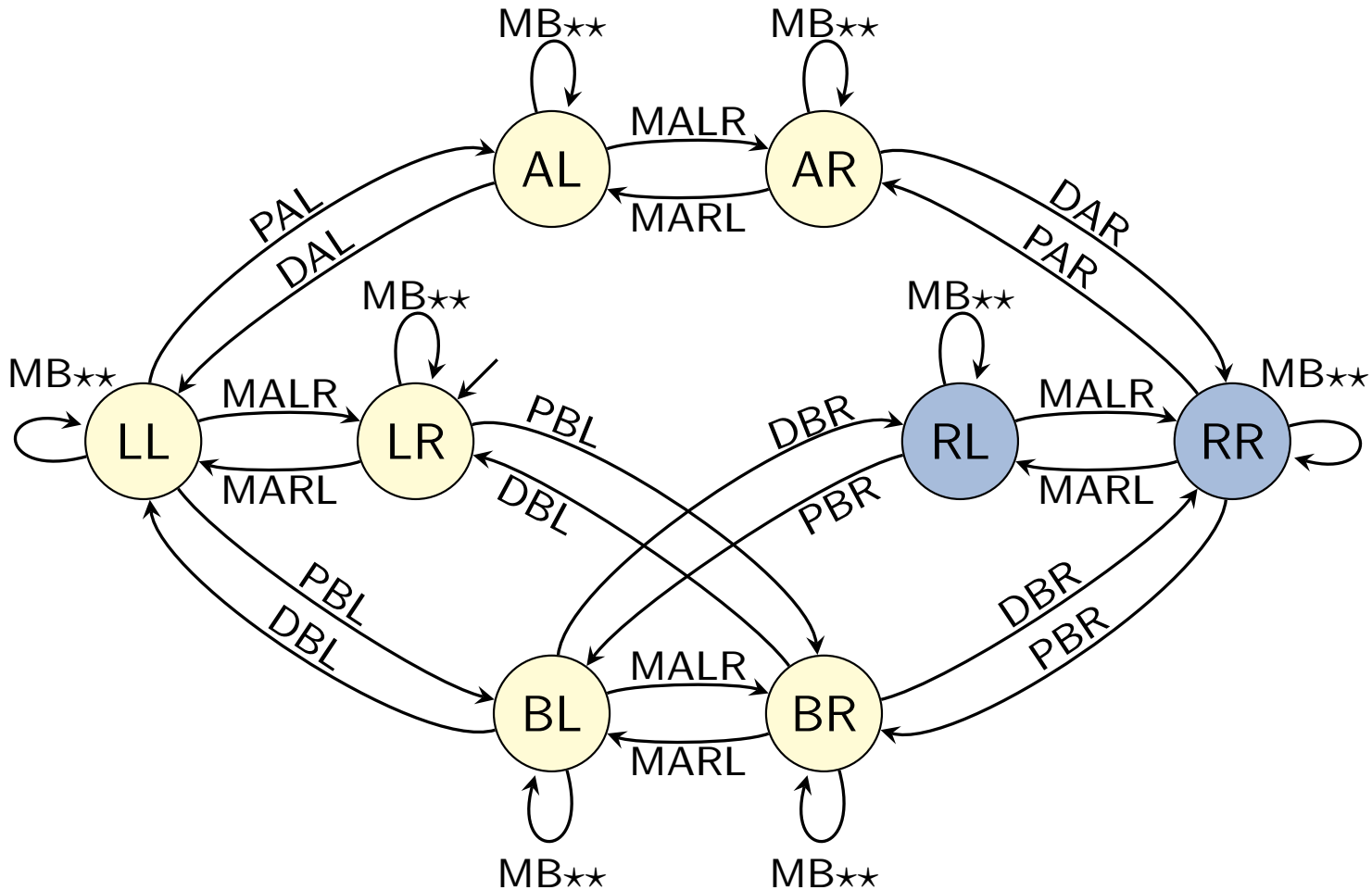
$\mathcal{T}^{\pi_{\{\text{truck B}\}}}$:



current FTS: $\{\mathcal{T}^{\pi_{\{\text{package}\}}}, \mathcal{T}^{\pi_{\{\text{truck A}\}}}, \mathcal{T}^{\pi_{\{\text{truck B}\}}}\}$

First Merge Step

$\mathcal{T}_1 := \mathcal{T}^{\pi\{\text{package}\}} \otimes \mathcal{T}^{\pi\{\text{truck A}\}}$:



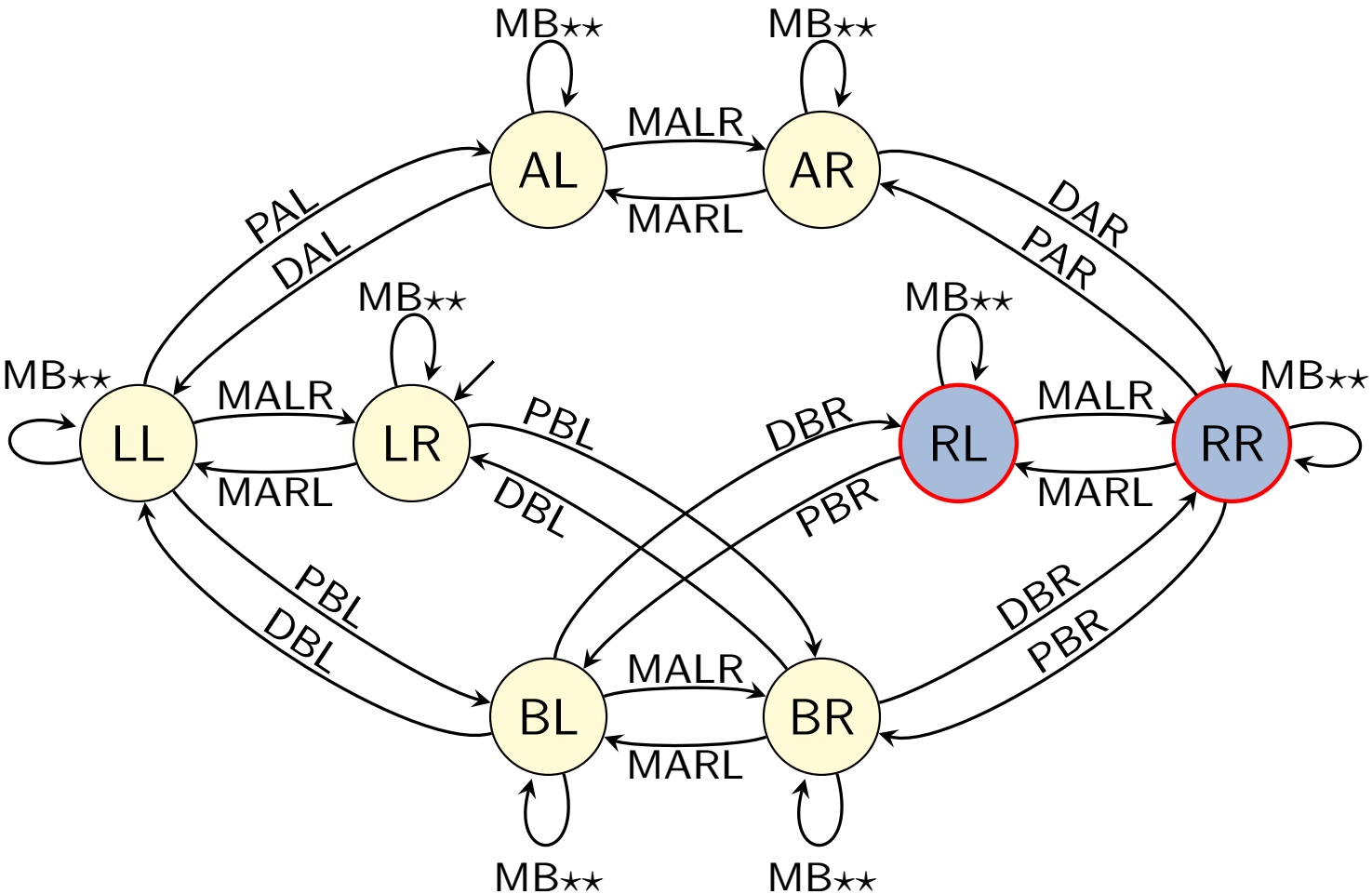
current FTS: $\{\mathcal{T}_1, \mathcal{T}^{\pi\{\text{truck B}\}}\}$

Need to Shrink?

- With sufficient memory, we could now compute $\mathcal{T}_1 \otimes \mathcal{T}^{\pi\{\text{truck B}\}}$ and recover the full transition system of the task.
- However, to illustrate the general idea, we assume that memory is too restricted: we may never create a factor with more than 8 states.
- To make the product fit the bound, we shrink $\mathcal{T}_1$ to 4 states. We can decide freely how exactly to abstract $\mathcal{T}_1$.
- In this example, we manually choose an abstraction that leads to a good result in the end. Making good shrinking decisions algorithmically is the job of the shrink strategy.

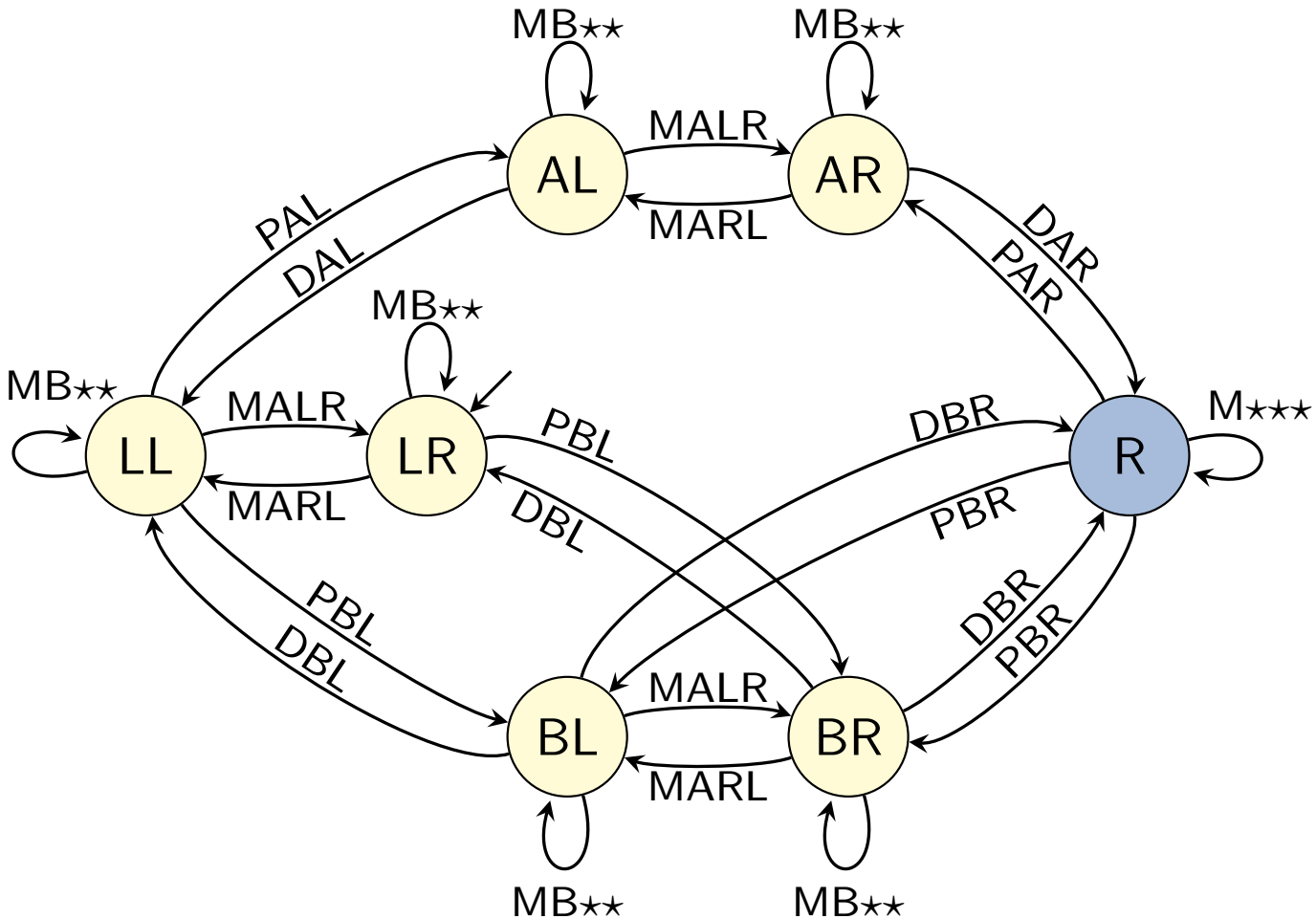
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



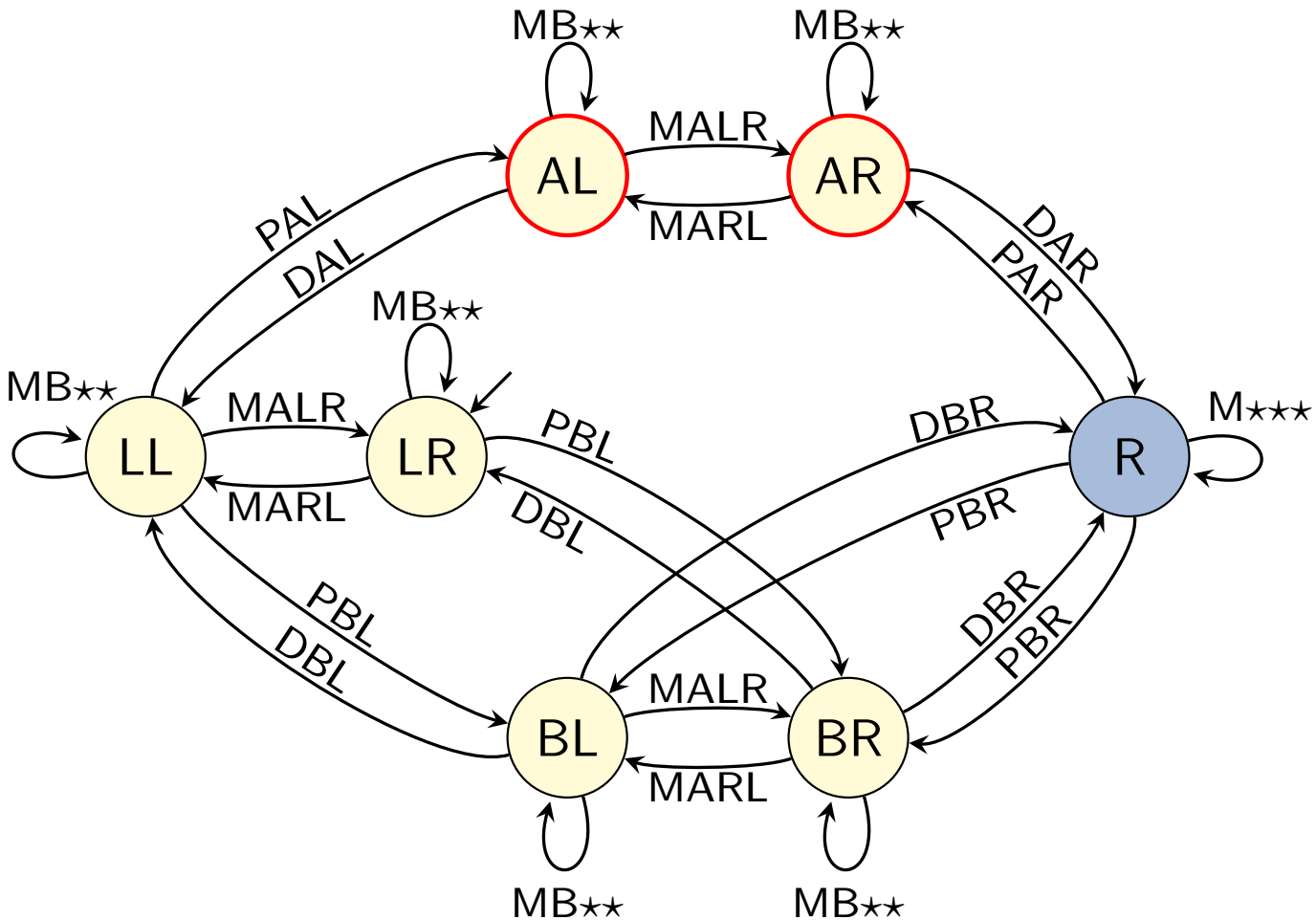
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



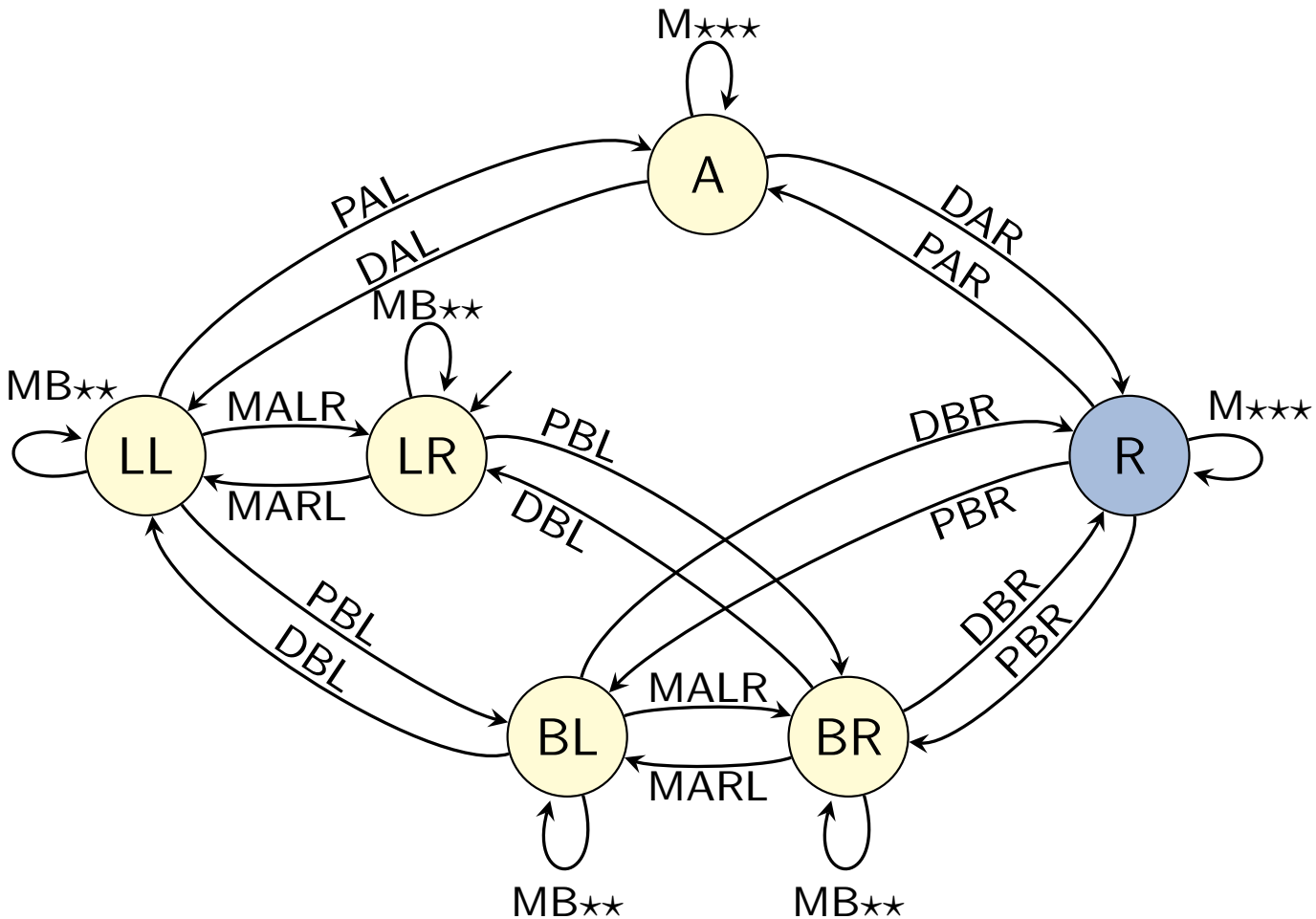
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



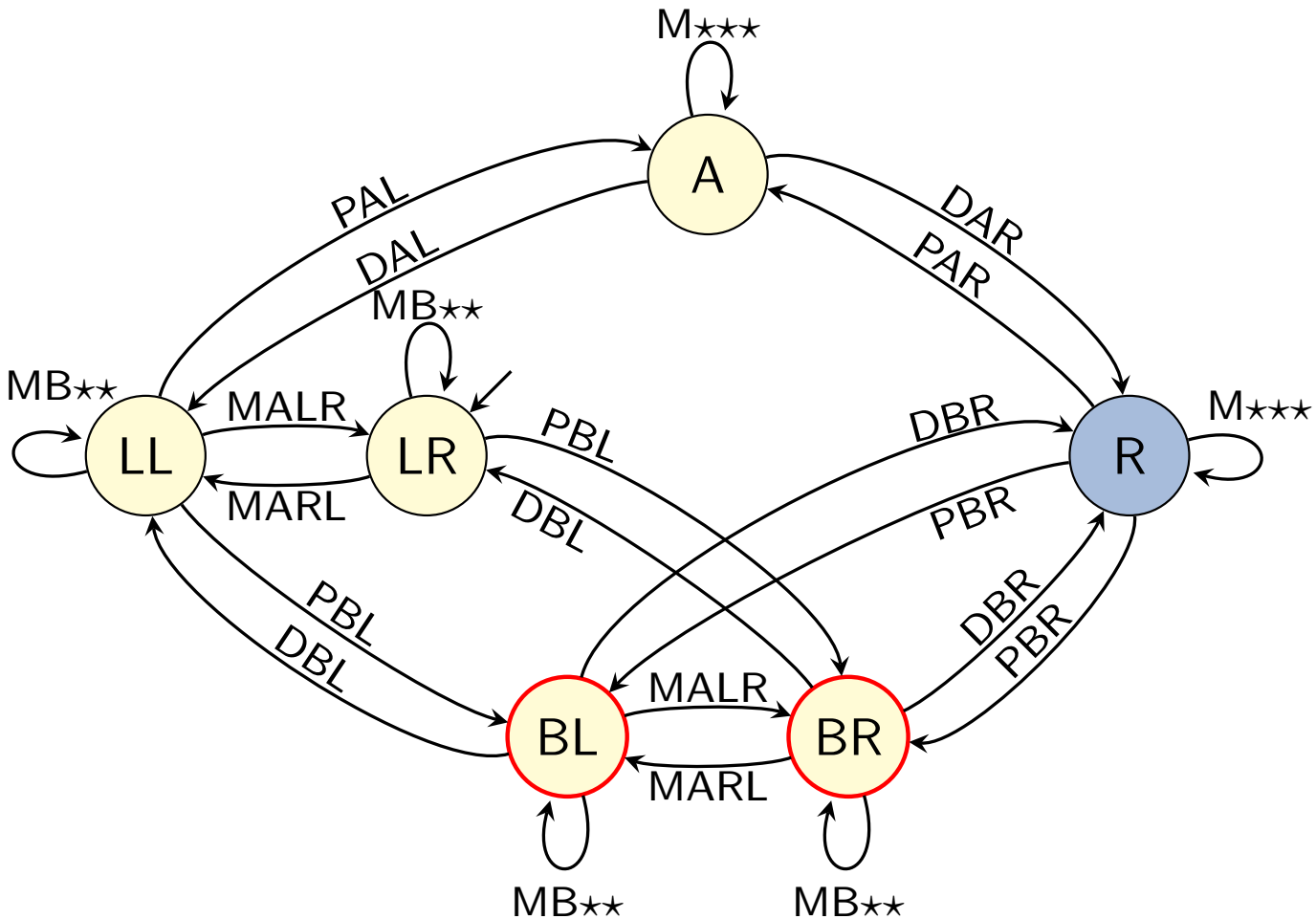
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



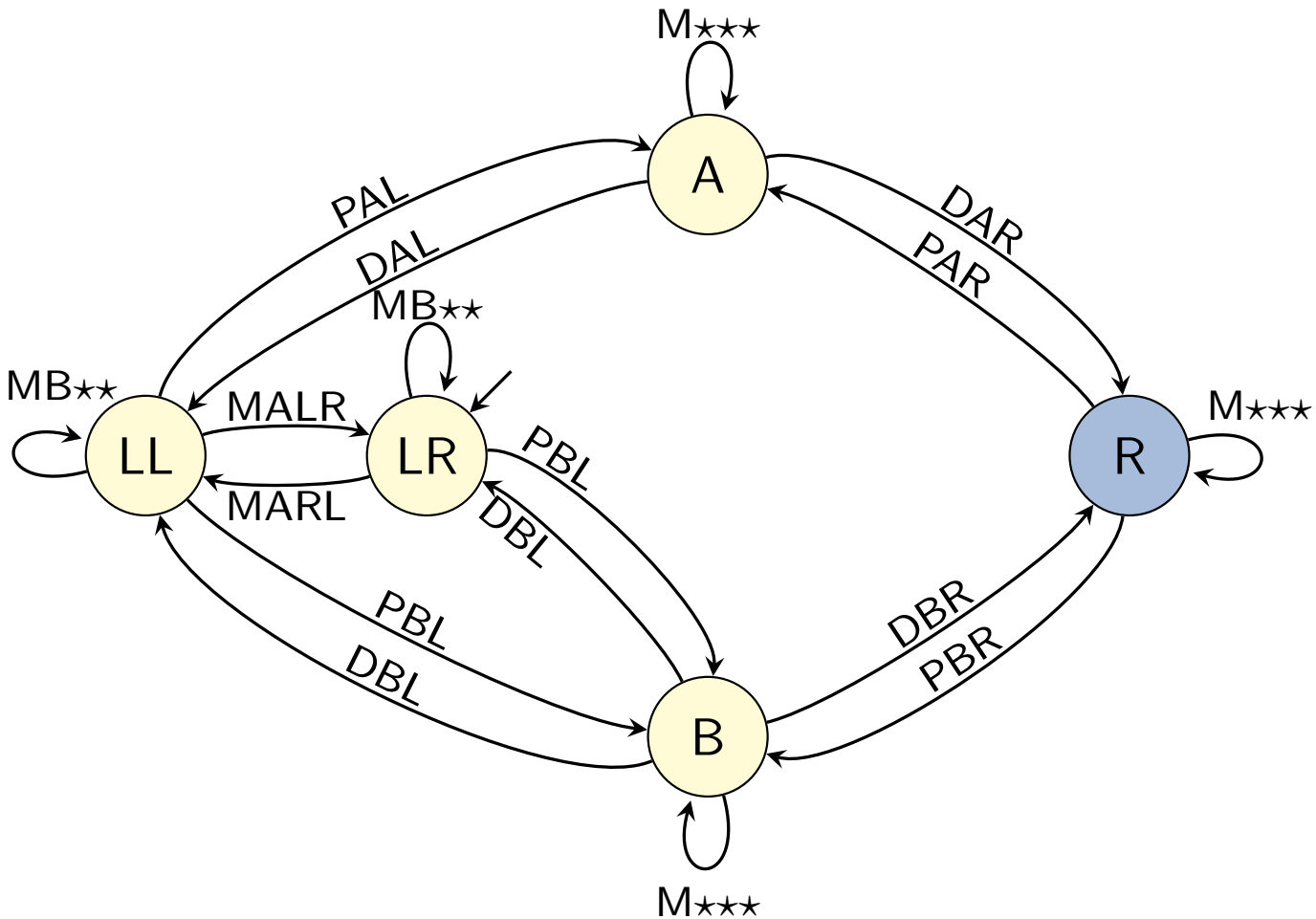
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



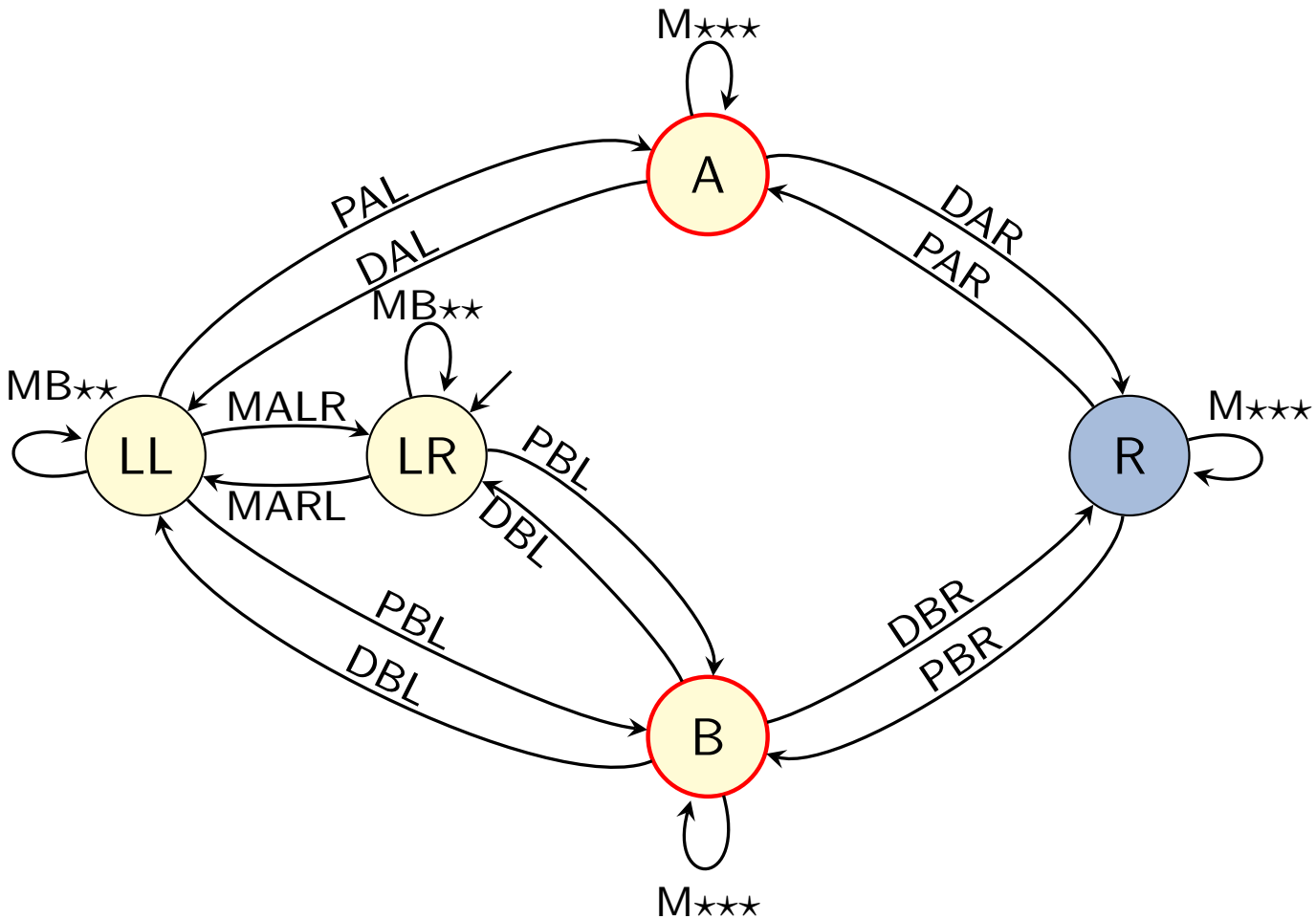
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



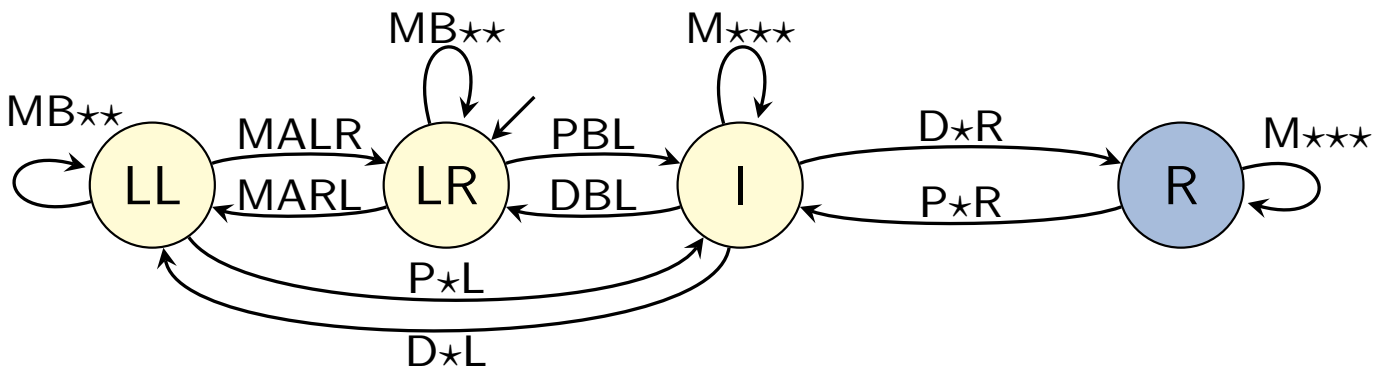
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



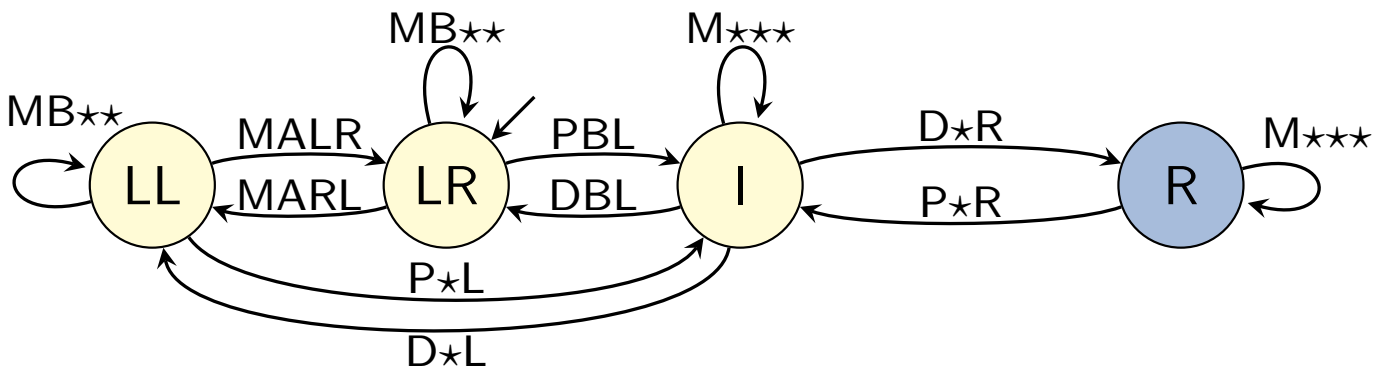
First Shrink Step

$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



First Shrink Step

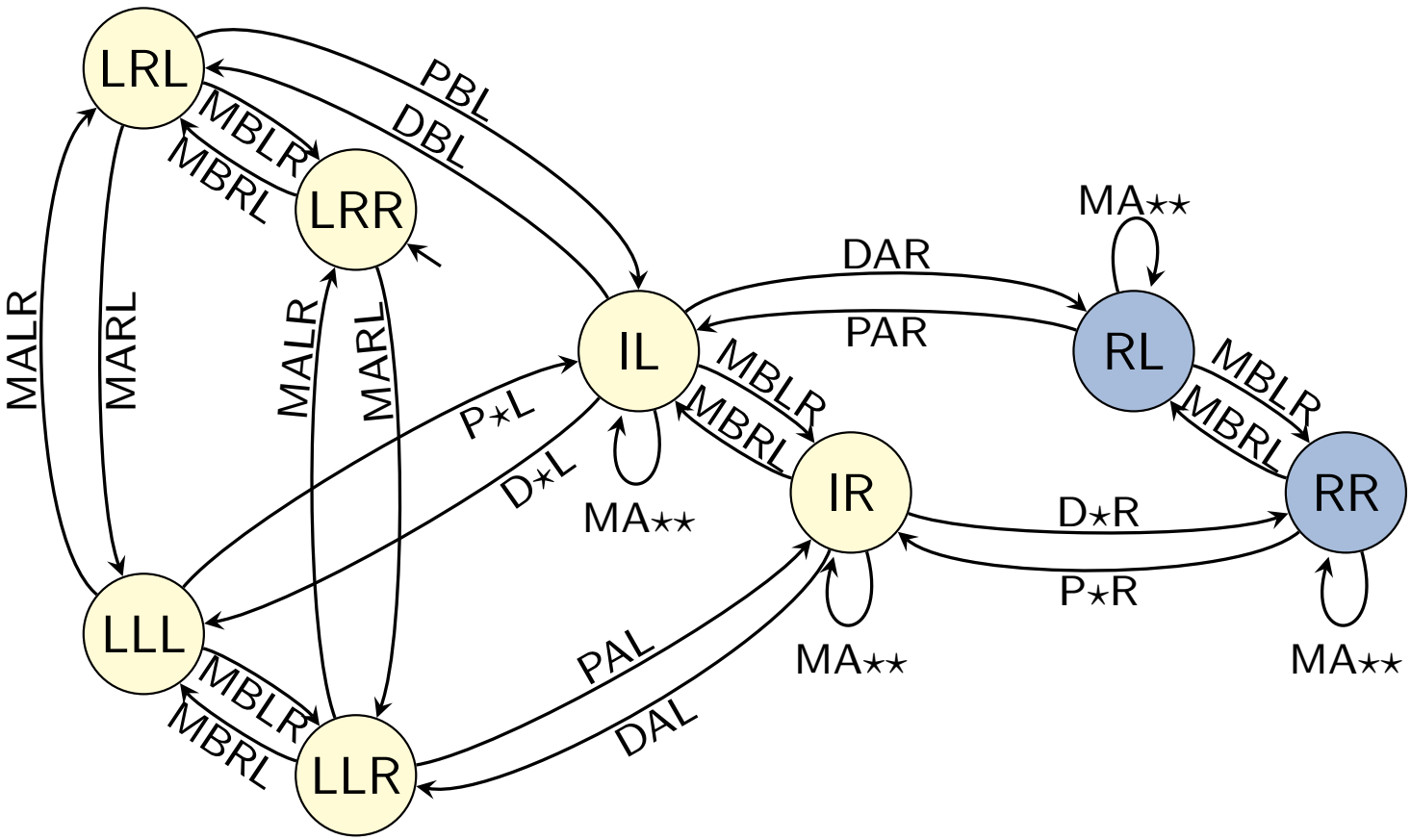
$\mathcal{T}_2 :=$ some abstraction of $\mathcal{T}_1$



current FTS: $\{\mathcal{T}_2, \mathcal{T}^{\pi_{\{\text{truck B}\}}}\}$

Second Merge Step

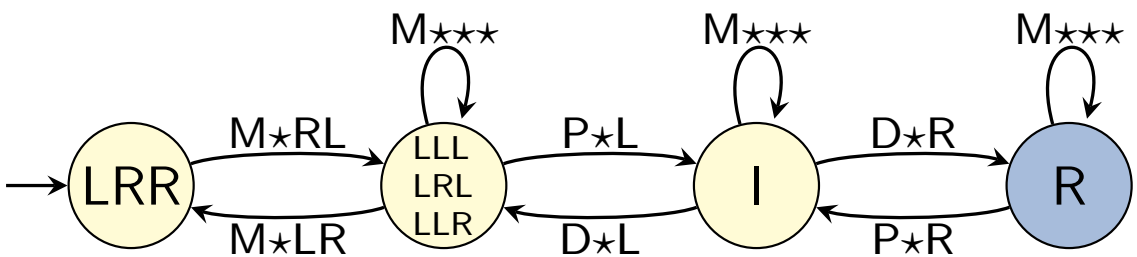
$\mathcal{T}_3 := \mathcal{T}_2 \otimes \mathcal{T}^{\pi\{\text{truck B}\}}$:



current FTS: $\{\mathcal{T}_3\}$

Another Shrink Step?

- At this point, merge-and-shrink construction stops.
  The distances in the final factor define the heuristic function.
- If there were further state variables to integrate,
  we would shrink again, e.g., leading to the following
  abstraction (again with four states):



- We get a heuristic value of 3 for the initial state,
  better than any PDB heuristic that is a proper abstraction.
- The example generalizes to arbitrarily many trucks,
  even if we stick to the fixed size limit of 8.

E10.3 Maintaining the Abstraction

Generic Algorithm Template

Generic Merge & Shrink Algorithm for planning task Π

$F := F(\Pi)$
while $|F| > 1$:
    select $type \in \{\text{merge}, \text{shrink}\}$
    if $type = \text{merge}$:
        select $\mathcal{T}_1, \mathcal{T}_2 \in F$
        $F := (F \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
    if $type = \text{shrink}$:
        select $\mathcal{T} \in F$
        choose an abstraction mapping β on $\mathcal{T}$
        $F := (F \setminus \{\mathcal{T}\}) \cup \{\mathcal{T}^\beta\}$
return the remaining factor $\mathcal{T}^\alpha$ in F

- The algorithm computes an abstract transition system.
- For the heuristic evaluation, we need an abstraction.
- How to maintain and represent the corresponding abstraction?

The Need for Succinct Abstractions

- One major difficulty for non-PDB abstraction heuristics is to succinctly represent the abstraction.
- For pattern databases, this is easy because the abstractions – projections – are very structured.
- For less rigidly structured abstractions, we need another idea.

How to Represent the Abstraction? (1)

Idea: the computation of the abstraction follows the sequence of product computations

- For the atomic abstractions $\pi_{\{v\}}$, we generate a one-dimensional table that denotes which value in $\mathrm{dom}(v)$ corresponds to which abstract state in $\mathcal{T}^{\pi_{\{v\}}}$.
- During the merge (product) step $\mathcal{A} := \mathcal{A}_1 \otimes \mathcal{A}_2$, we generate a two-dimensional table that denotes which pair of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ corresponds to which state of $\mathcal{A}$.
- During the shrink (abstraction) steps, we make sure to keep the table in sync with the abstraction choices.

How to Represent the Abstraction? (2)

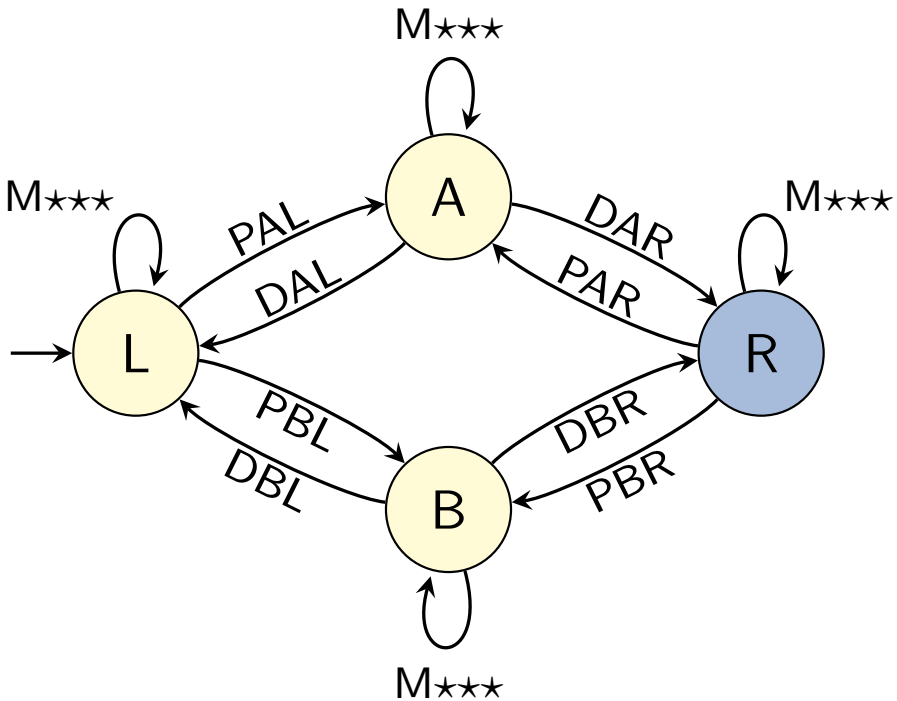
Idea: the computation of the abstraction mapping follows the sequence of product computations

- Once we have computed the final abstract transition system, we compute all abstract goal distances and store them in a one-dimensional table.
- At this point, we can throw away all the abstract transition systems – we just need to keep the tables.
- During search, we do a sequence of table lookups to navigate from the atomic abstraction states to the final abstract state and heuristic value
  $\rightsquigarrow 2|V|$ lookups, $O(|V|)$ time

Again, we illustrate the process with our running example.

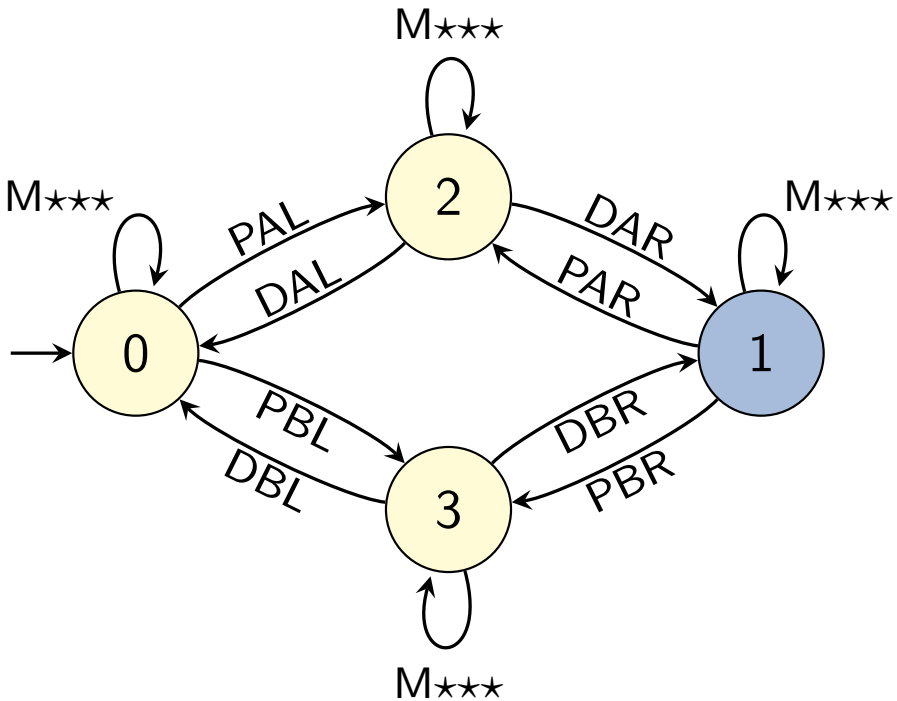
Abstraction Example: Atomic Abstractions

Computing abstractions for the transition systems of atomic abstractions is simple. Just number the states (domain values) consecutively and generate a table of references to the states:



Abstraction Example: Atomic Abstractions

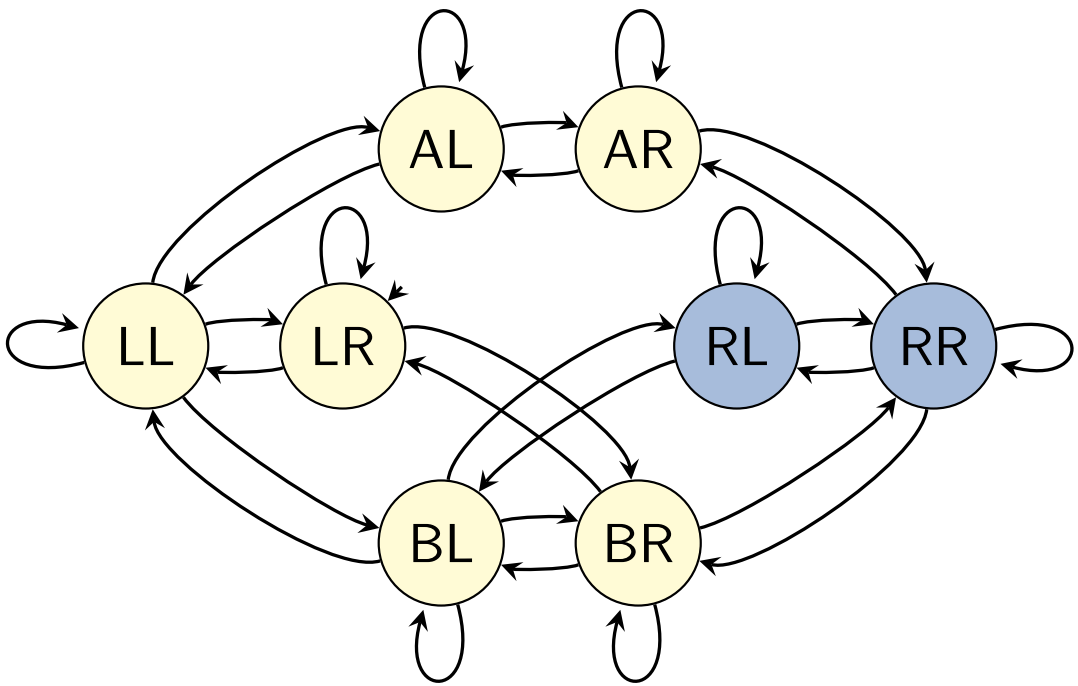
Computing abstractions for the transition systems of atomic abstractions is simple. Just number the states (domain values) consecutively and generate a table of references to the states:



L	R	A	B
0	1	2	3

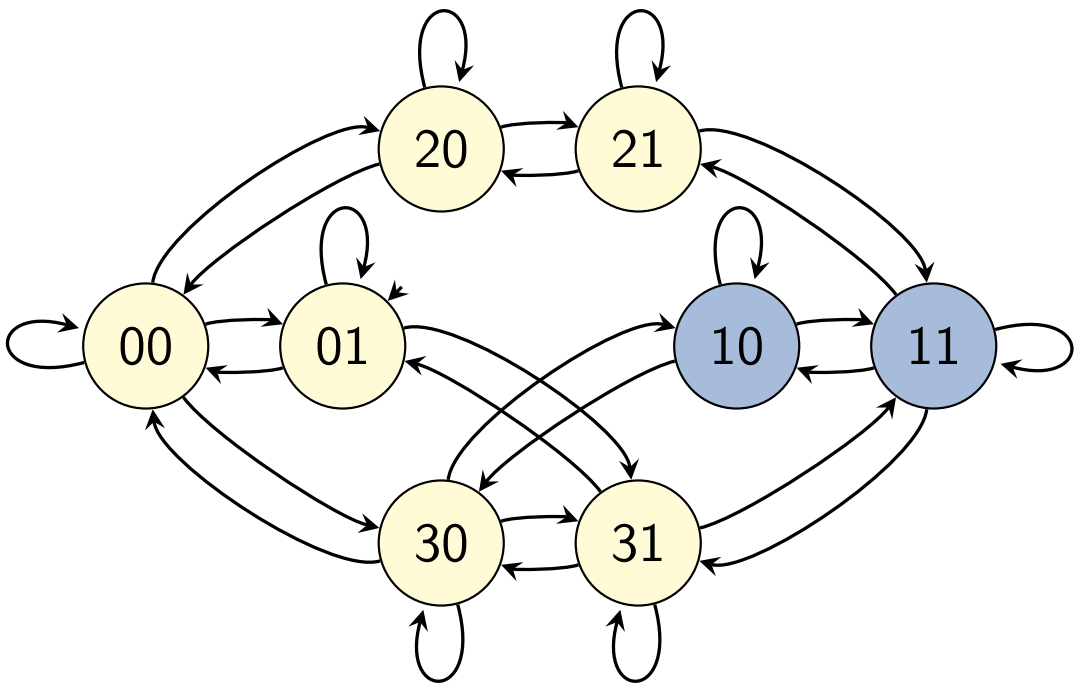
Abstraction Example: Merge Step

For product transition systems $\mathcal{A}_1 \otimes \mathcal{A}_2$, we again number the product states consecutively and generate a table that links state pairs of $\mathcal{A}_1$ and $\mathcal{A}_2$ to states of $\mathcal{A}$:



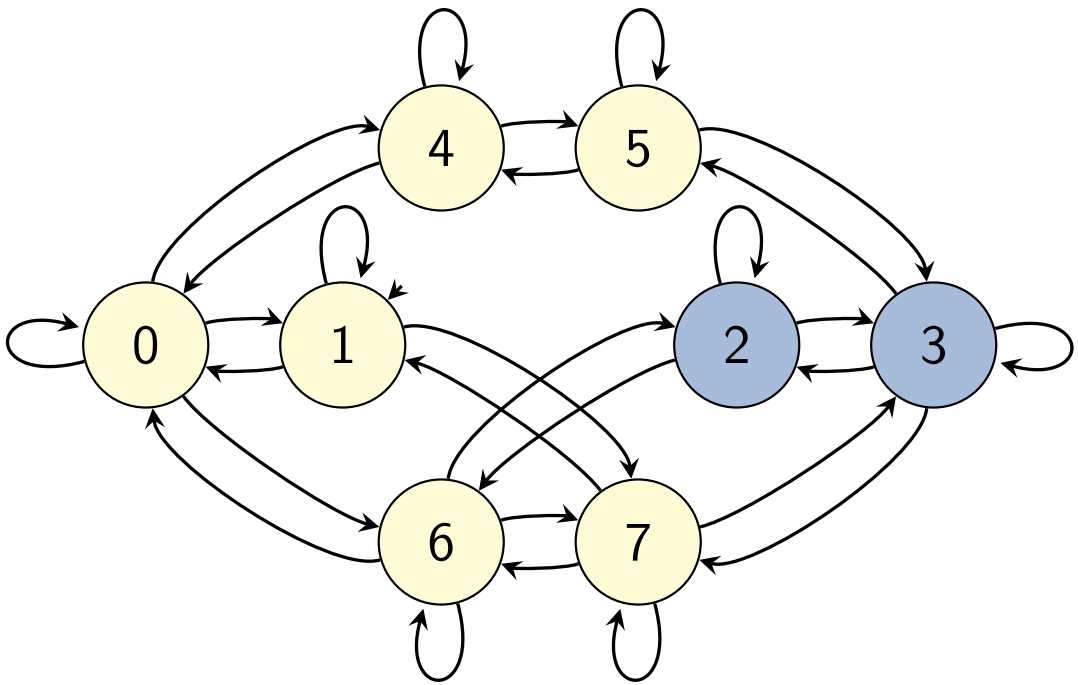
Abstraction Example: Merge Step

For product transition systems $\mathcal{A}_1 \otimes \mathcal{A}_2$, we again number the product states consecutively and generate a table that links state pairs of $\mathcal{A}_1$ and $\mathcal{A}_2$ to states of $\mathcal{A}$:



Abstraction Example: Merge Step

For product transition systems $\mathcal{A}_1 \otimes \mathcal{A}_2$, we again number the product states consecutively and generate a table that links state pairs of $\mathcal{A}_1$ and $\mathcal{A}_2$ to states of $\mathcal{A}$:



	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	0	1
$s_1 = 1$	2	3
$s_1 = 2$	4	5
$s_1 = 3$	6	7

Maintaining the Abstraction when Shrinking

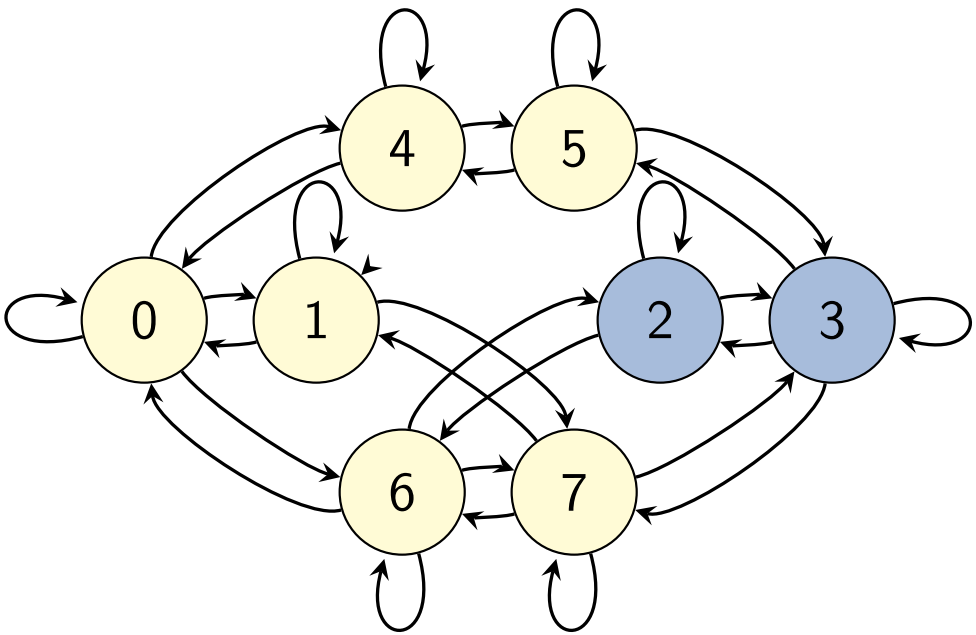
- The hard part in representing the abstraction is to keep it consistent when shrinking.
- In theory, this is easy to do:
  - When combining states i and j, arbitrarily use one of them (say i) as the number of the new state.
  - Find all table entries in the table for this abstraction which map to the other state j and change them to i.
- However, doing a table scan each time two states are combined is very inefficient.
- Fortunately, there also is an efficient implementation which takes constant time per combination.

Maintaining the Abstraction Efficiently

- Associate each abstract state with a linked list, representing all table entries that map to this state.
- Before starting the shrink operation, initialize the lists by scanning through the table, then discard the table.
- While shrinking, when combining i and j, splice the list elements of j into the list elements of i.
  - For linked lists, this is a constant-time operation.
- Once shrinking is completed, renumber all abstract states so that there are no gaps in the numbering.
- Finally, regenerate the mapping table from the linked list information.

Abstraction Example: Shrink Step

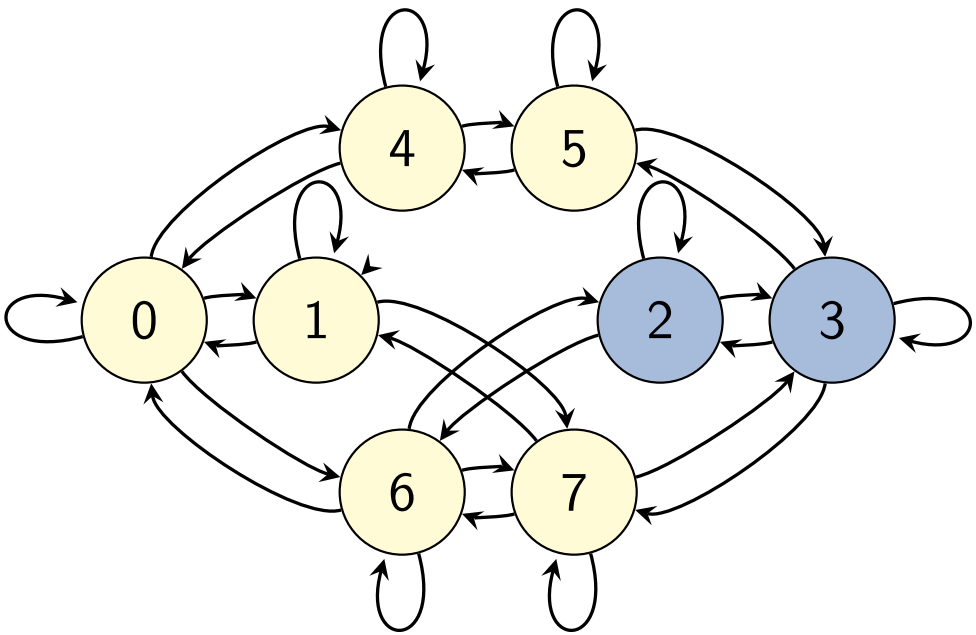
Representation before shrinking:



	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	0	1
$s_1 = 1$	2	3
$s_1 = 2$	4	5
$s_1 = 3$	6	7

Abstraction Example: Shrink Step

1. Convert table to linked lists and discard it.

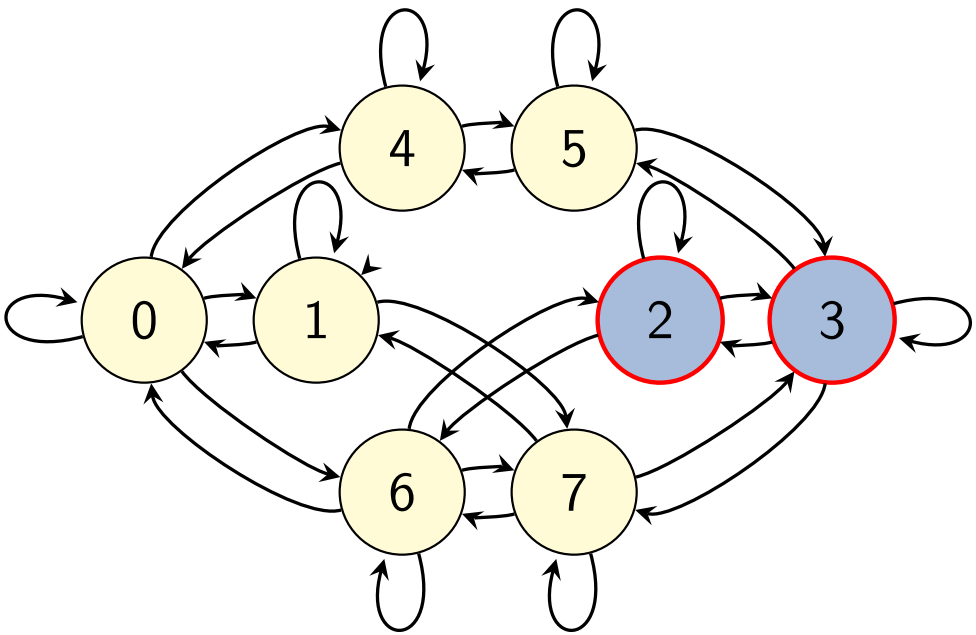


$$list_0 = \{(0,0)\}$$
$$list_1 = \{(0,1)\}$$
$$list_2 = \{(1,0)\}$$
$$list_3 = \{(1,1)\}$$
$$list_4 = \{(2,0)\}$$
$$list_5 = \{(2,1)\}$$
$$list_6 = \{(3,0)\}$$
$$list_7 = \{(3,1)\}$$

	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	0	1
$s_1 = 1$	2	3
$s_1 = 2$	4	5
$s_1 = 3$	6	7

Abstraction Example: Shrink Step

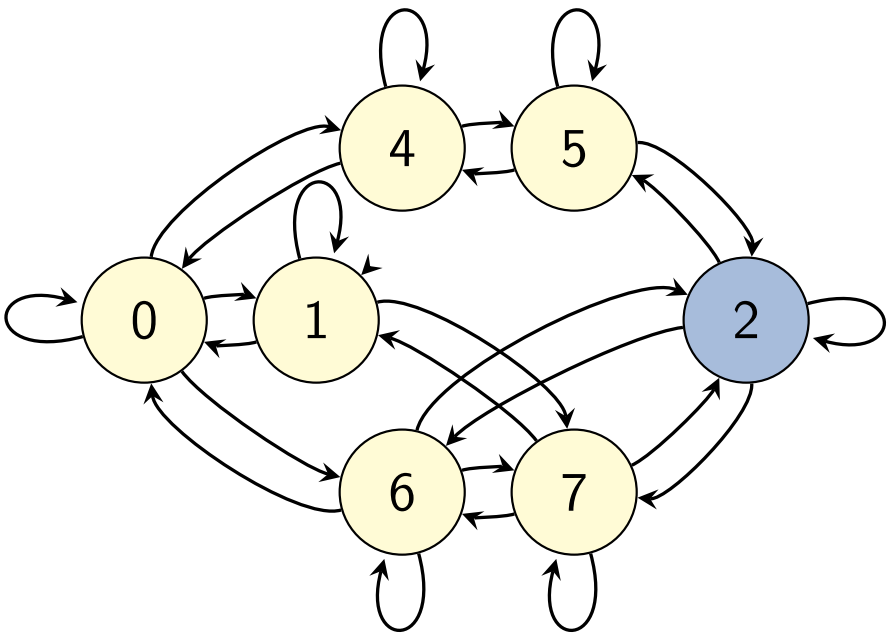
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0)\}$
$list_3 = \{(1,1)\}$
$list_4 = \{(2,0)\}$
$list_5 = \{(2,1)\}$
$list_6 = \{(3,0)\}$
$list_7 = \{(3,1)\}$

Abstraction Example: Shrink Step

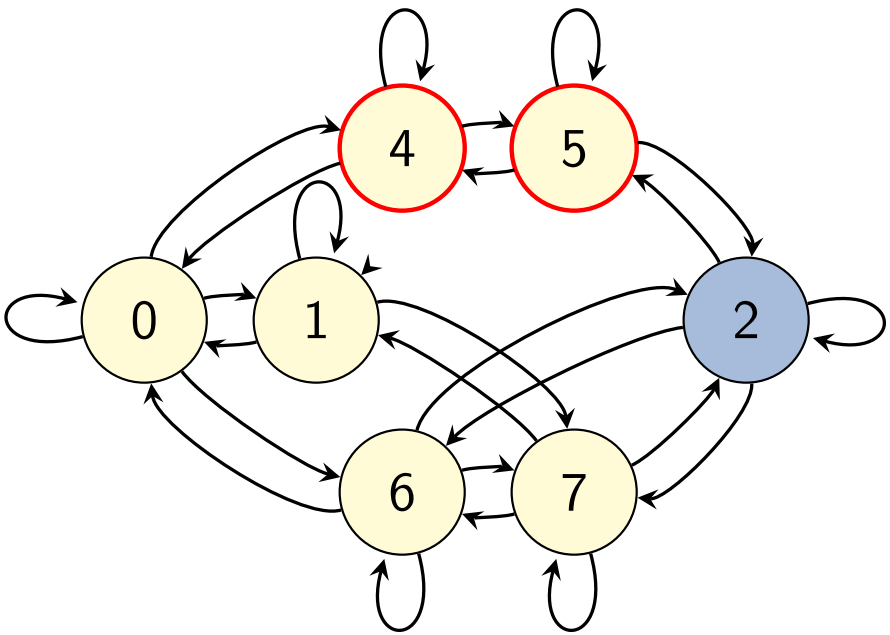
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0)\}$
$list_5 = \{(2,1)\}$
$list_6 = \{(3,0)\}$
$list_7 = \{(3,1)\}$

Abstraction Example: Shrink Step

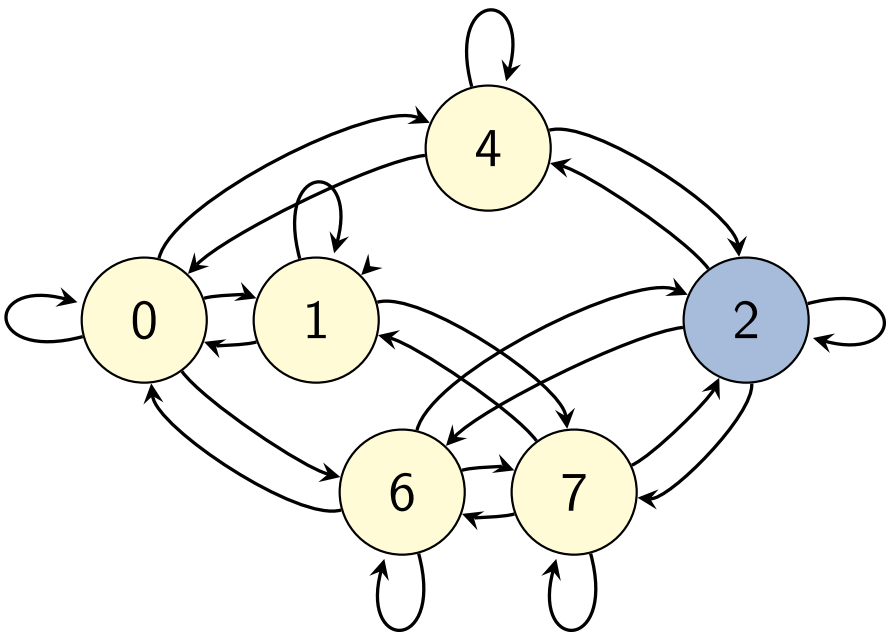
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0)\}$
$list_5 = \{(2,1)\}$
$list_6 = \{(3,0)\}$
$list_7 = \{(3,1)\}$

Abstraction Example: Shrink Step

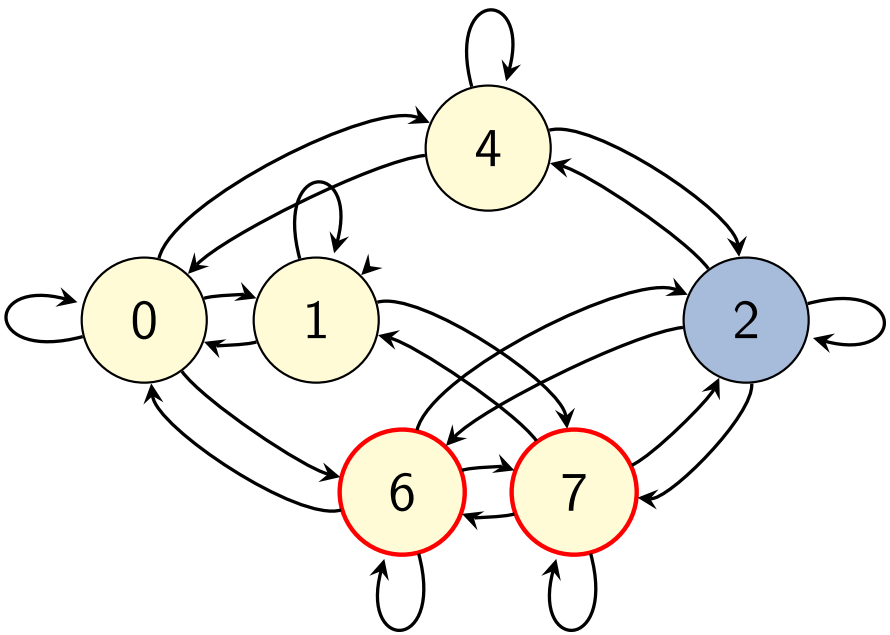
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1)\}$
$list_5 = \emptyset$
$list_6 = \{(3,0)\}$
$list_7 = \{(3,1)\}$

Abstraction Example: Shrink Step

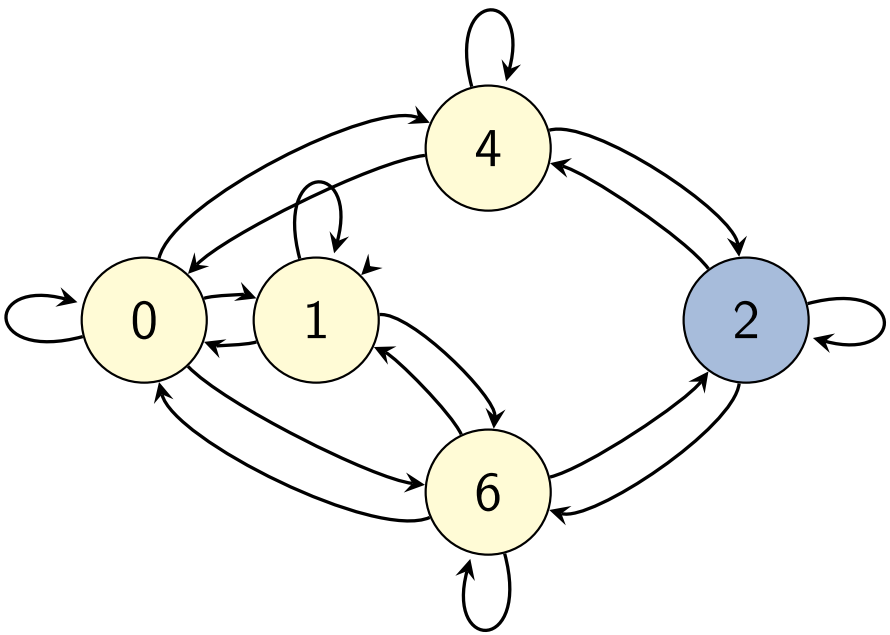
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1)\}$
$list_5 = \emptyset$
$list_6 = \{(3,0)\}$
$list_7 = \{(3,1)\}$

Abstraction Example: Shrink Step

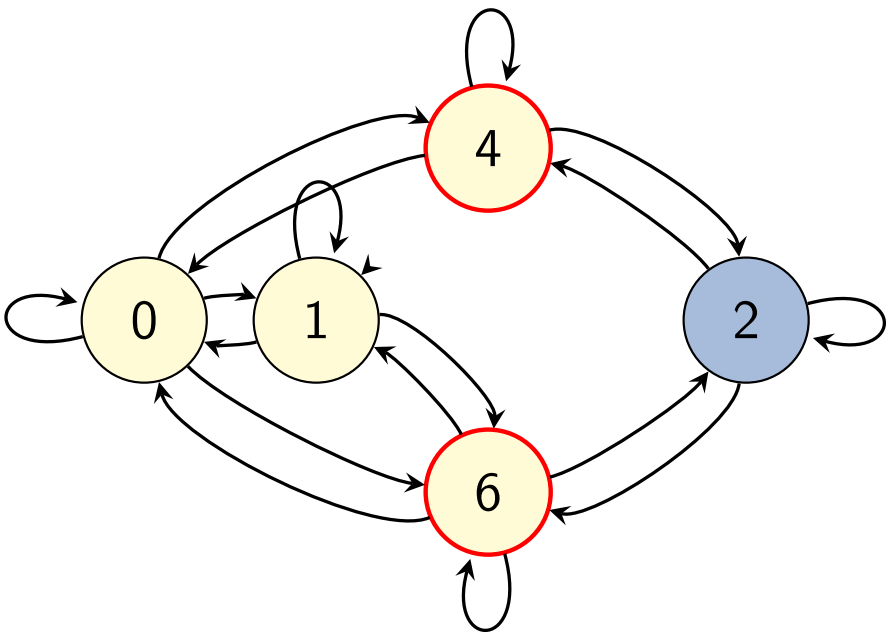
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1)\}$
$list_5 = \emptyset$
$list_6 = \{(3,0),(3,1)\}$
$list_7 = \emptyset$

Abstraction Example: Shrink Step

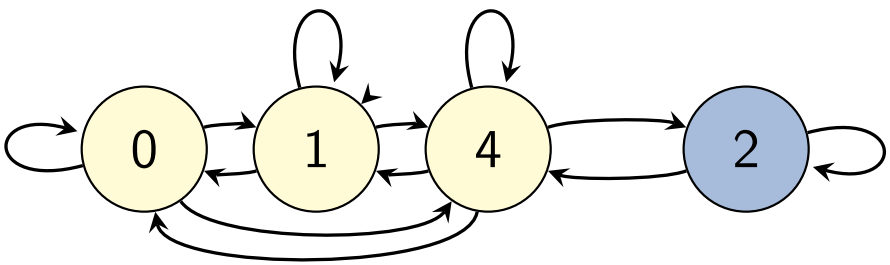
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1)\}$
$list_5 = \emptyset$
$list_6 = \{(3,0),(3,1)\}$
$list_7 = \emptyset$

Abstraction Example: Shrink Step

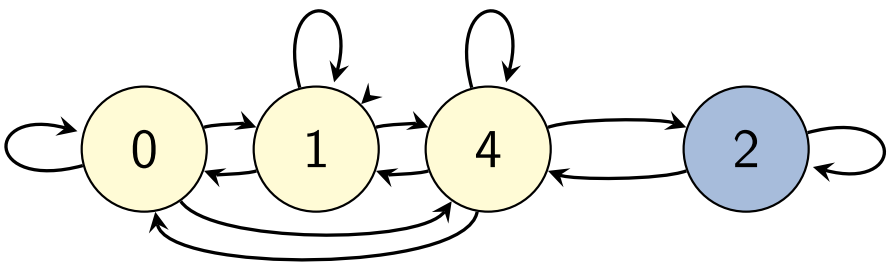
2. When combining i and j, splice $list_j$ into $list_i$.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1),$
$(3,0),(3,1)\}$
$list_5 = \emptyset$
$list_6 = \emptyset$
$list_7 = \emptyset$

Abstraction Example: Shrink Step

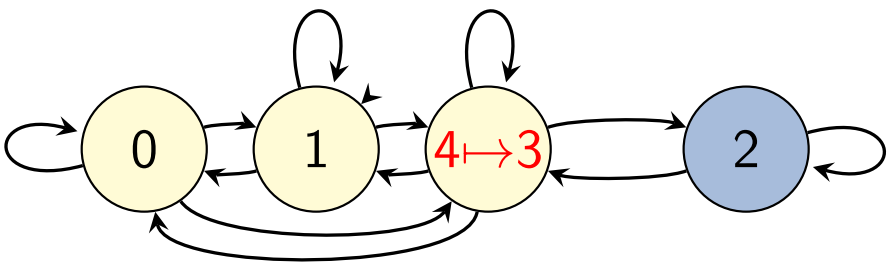
2. When combining i and j, splice $list_j$ into $list_i$.



$$
\begin{aligned}
list_0 &= \{(0,0)\} \\
list_1 &= \{(0,1)\} \\
list_2 &= \{(1,0),(1,1)\} \\
list_3 &= \emptyset \\
list_4 &= \{(2,0),(2,1), \\
&\qquad (3,0),(3,1)\} \\
list_5 &= \emptyset \\
list_6 &= \emptyset \\
list_7 &= \emptyset
\end{aligned}
$$

Abstraction Example: Shrink Step

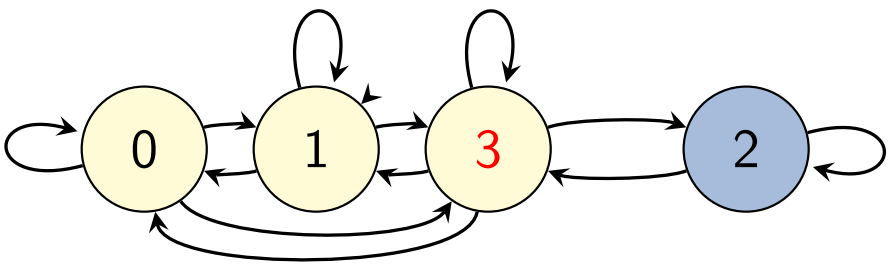
3. Renumber abstract states consecutively.



$list_0 = \{(0,0)\}$
$list_1 = \{(0,1)\}$
$list_2 = \{(1,0),(1,1)\}$
$list_3 = \emptyset$
$list_4 = \{(2,0),(2,1),$
$(3,0),(3,1)\}$
$list_5 = \emptyset$
$list_6 = \emptyset$
$list_7 = \emptyset$

Abstraction Example: Shrink Step

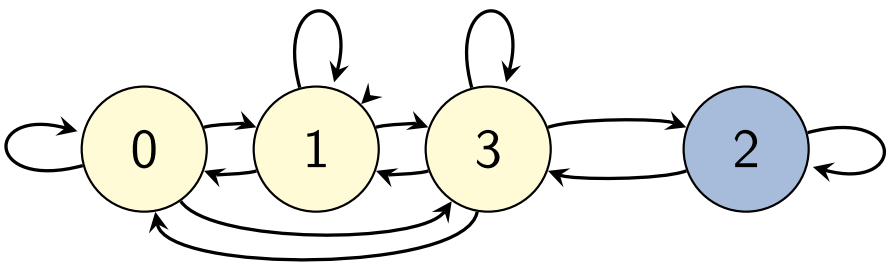
3. Renumber abstract states consecutively.



$$
\begin{aligned}
\mathit{list}_0 &= \{(0,0)\} \\
\mathit{list}_1 &= \{(0,1)\} \\
\mathit{list}_2 &= \{(1,0),(1,1)\} \\
\mathit{list}_3 &= \{(2,0),(2,1), \\
&\qquad (3,0),(3,1)\} \\
\mathit{list}_4 &= \emptyset \\
\mathit{list}_5 &= \emptyset \\
\mathit{list}_6 &= \emptyset \\
\mathit{list}_7 &= \emptyset
\end{aligned}
$$

Abstraction Example: Shrink Step

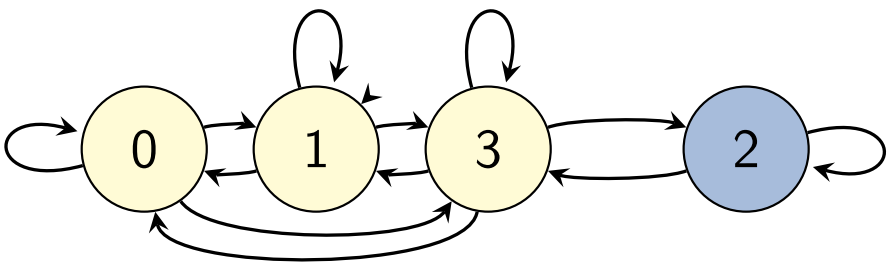
4. Regenerate the mapping table from the linked lists.



$$
\begin{aligned}
\mathit{list}_0 &= \{(0,0)\} \\
\mathit{list}_1 &= \{(0,1)\} \\
\mathit{list}_2 &= \{(1,0),(1,1)\} \\
\mathit{list}_3 &= \{(2,0),(2,1), \\
&\qquad (3,0),(3,1)\} \\
\mathit{list}_4 &= \emptyset \\
\mathit{list}_5 &= \emptyset \\
\mathit{list}_6 &= \emptyset \\
\mathit{list}_7 &= \emptyset
\end{aligned}
$$

Abstraction Example: Shrink Step

4. Regenerate the mapping table from the linked lists.



$$
\begin{aligned}
\mathit{list}_0 &= \{(0,0)\} \\
\mathit{list}_1 &= \{(0,1)\} \\
\mathit{list}_2 &= \{(1,0),(1,1)\} \\
\mathit{list}_3 &= \{(2,0),(2,1), \\
&\qquad (3,0),(3,1)\} \\
\mathit{list}_4 &= \emptyset \\
\mathit{list}_5 &= \emptyset \\
\mathit{list}_6 &= \emptyset \\
\mathit{list}_7 &= \emptyset
\end{aligned}
$$

	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	0	1
$s_1 = 1$	2	2
$s_1 = 2$	3	3
$s_1 = 3$	3	3

The Final Heuristic Representation

At the end, our heuristic is represented by six tables:

- three one-dimensional tables for the atomic abstractions:

T_{package}	L	R	A	B
	0	1	2	3

$T_{\text{truck A}}$	L	R
	0	1

$T_{\text{truck B}}$	L	R
	0	1

- two tables for the two merge and subsequent shrink steps:

$T^1_{\text{m\&s}}$	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	0	1
$s_1 = 1$	2	2
$s_1 = 2$	3	3
$s_1 = 3$	3	3

$T^2_{\text{m\&s}}$	$s_2 = 0$	$s_2 = 1$
$s_1 = 0$	1	1
$s_1 = 1$	1	0
$s_1 = 2$	2	2
$s_1 = 3$	3	3

- one table with goal distances for the final transition system:

T_h	$s = 0$	$s = 1$	$s = 2$	$s = 3$
$h(s)$	3	2	0	1

Given a state $s = \{\text{package} \mapsto L, \text{truck A} \mapsto L, \text{truck B} \mapsto R\}$, its heuristic value is then looked up as:

- $h(s) = T_h[T^2_{\text{m\&s}}[T^1_{\text{m\&s}}[T_{\text{package}}[L], T_{\text{truck A}}[L]], T_{\text{truck B}}[R]]]$

E10.4 Summary

Summary (1)

- Merge-and-shrink abstractions are constructed by iteratively transforming the factored transition system of a planning task.
- Merge transformations combine two factors into their synchronized product.
- Shrink transformations reduce the size of a factor by abstracting it.
- Merge-and-shrink abstractions are represented by a set of reference tables, one for each atomic abstraction and one for each merge-and-shrink step.
- The heuristic representation uses an additional table for the goal distances in the final abstract transition system.

Summary (2)

- Projections of SAS^+ tasks correspond to merges of atomic factors.
- By also including shrinking, merge-and-shrink abstractions generalize projections: they can reflect all state variables, but in a potentially lossy way.

Planning and Optimization

E11. Merge-and-Shrink: Properties and Shrink Strategies

Malte Helmert and Gabriele Röger

Universität Basel

November 19, 2025

Planning and Optimization

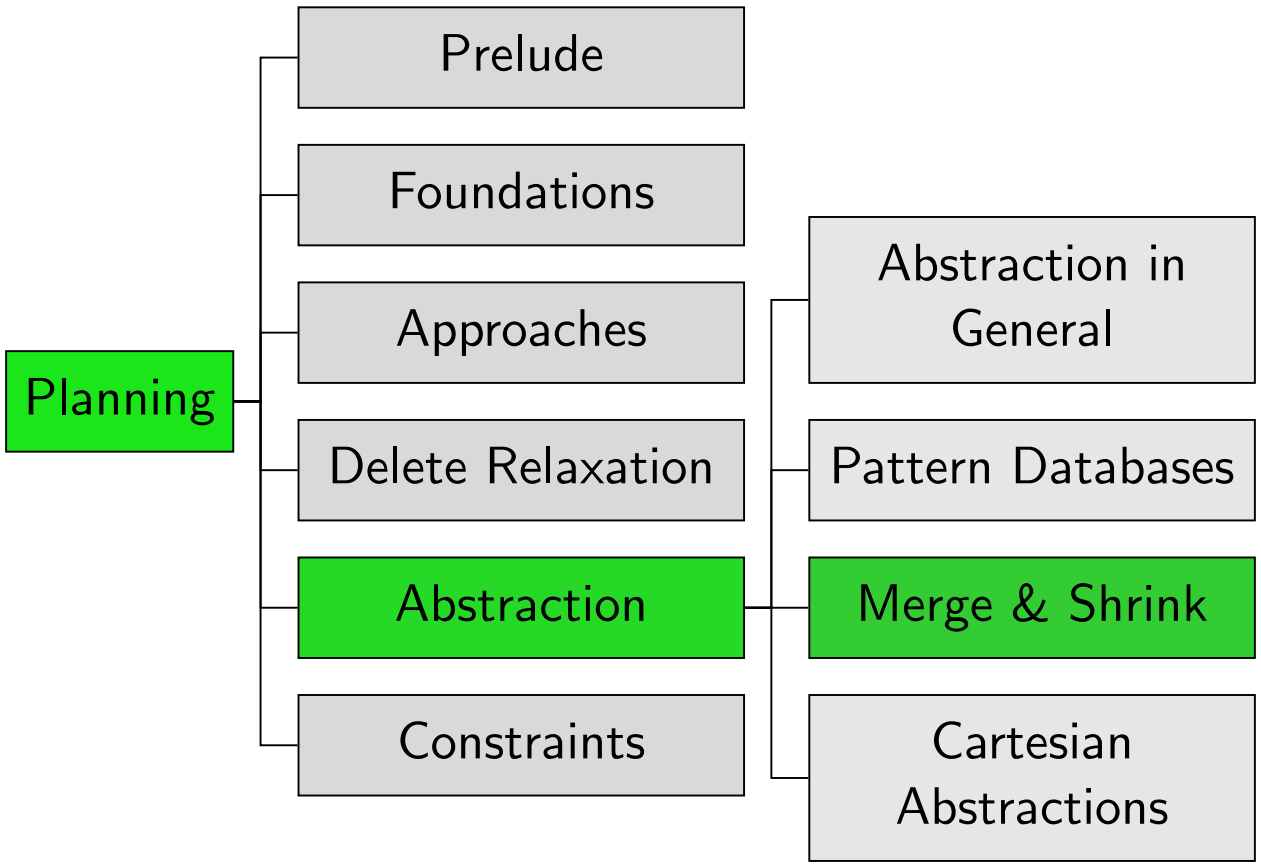
November 19, 2025 — E11. Merge-and-Shrink: Properties and Shrink Strategies

E11.1 Heuristic Properties

E11.2 Shrink Strategies

E11.3 Summary

Content of the Course



Reminder: Generic Algorithm Template

Generic Merge & Shrink Algorithm for planning task Π

$F := F(\Pi)$
while $|F| > 1$:
    select $type \in \{\text{merge}, \text{shrink}\}$
    if $type = \text{merge}$:
        select $\mathcal{T}_1, \mathcal{T}_2 \in F$
        $F := (F \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
    if $type = \text{shrink}$:
        select $\mathcal{T} \in F$
        choose an abstraction mapping β on $\mathcal{T}$
        $F := (F \setminus \{\mathcal{T}\}) \cup \{\mathcal{T}^\beta\}$
return the remaining factor $\mathcal{T}^\alpha$ in F

E11.1 Heuristic Properties

Properties of Merge-and-Shrink Heuristics

To understand merge-and-shrink abstractions better,
we are interested in the properties of the resulting heuristic:

- Is it admissible ($h^\alpha(s) \leq h^*(s)$ for all states s)?
- Is it consistent ($h^\alpha(s) \leq c(o) + h^\alpha(t)$ for all trans. $s \xrightarrow{o} t$)?
- Is it perfect ($h^\alpha(s) = h^*(s)$ for all states s)?

Because merge-and-shrink is a generic procedure,
the answers may depend on how exactly we instantiate it:

- size limits
- merge strategy
- shrink strategy

Merge-and-Shrink as Sequence of Transformations

- Consider a run of the merge-and-shrink construction algorithm with n iterations of the main loop.
- Let F_i $(0 \leq i \leq n)$ be the FTS F after i loop iterations.
- Let $\mathcal{T}_i$ $(0 \leq i \leq n)$ be the transition system represented by F_i, i.e., $\mathcal{T}_i = \bigotimes F_i$.
- In particular, $F_0 = F(\Pi)$ and $F_n = \{\mathcal{T}_n\}$.
- For SAS$^+$ tasks Π, we also know $\mathcal{T}_0 = \mathcal{T}(\Pi)$.

For a formal study, it is useful to view merge-and-shrink construction as a sequence of transformations from $\mathcal{T}_i$ to $\mathcal{T}_{i+1}$.

(We do it in a bit more general fashion than necessary for merge and shrink steps only, to also cover some improvements we will see later.)

Transformations

Definition (Transformation)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ and $\mathcal{T}' = \langle S', L', c', T', s'_0, S'_\star \rangle$ be transition systems.
Let $\sigma : S \to S'$ map the states of $\mathcal{T}$ to the states of $\mathcal{T}'$ and $\lambda : L \to L'$ map the labels of $\mathcal{T}$ to the labels of $\mathcal{T}'$.

The tuple $\tau = \langle \mathcal{T}, \sigma, \lambda, \mathcal{T}' \rangle$ is called a transformation from $\mathcal{T}$ to $\mathcal{T}'$. We also write it as $\mathcal{T} \xrightarrow{\sigma,\lambda} \mathcal{T}'$.

The transformation τ induces the heuristic h^τ for $\mathcal{T}$ defined as $h^\tau(s) = h^*_{\mathcal{T}'}(\sigma(s))$.

Example: If α is an abstraction mapping for transition system $\mathcal{T}$, then $\mathcal{T} \xrightarrow{\alpha,\mathsf{id}} \mathcal{T}^\alpha$ is a transformation.

Conservative Transformations

Definition (Conservative Transformation)

Let $\mathcal{T}$ and $\mathcal{T}'$ be transition systems with label sets L and L' and cost functions c and c', respectively.

A transformation $\langle \mathcal{T}, \sigma, \lambda, \mathcal{T}' \rangle$ is conservative if

- $c'(\lambda(\ell)) \leq c(\ell)$ for all $\ell \in L$,
- for all transitions $\langle s, \ell, t \rangle$ of $\mathcal{T}$ there is a transition $\langle \sigma(s), \lambda(\ell), \sigma(t) \rangle$ of $\mathcal{T}'$, and
- for all goal states s of $\mathcal{T}$, state $\sigma(s)$ is a goal state of $\mathcal{T}'$.

Example: If α is an abstraction mapping for transition system $\mathcal{T}$, then $\mathcal{T} \xrightarrow{\alpha,\mathsf{id}} \mathcal{T}^\alpha$ is a conservative transformation.

Conservative Transformations: Heuristic Properties (1)

Theorem

If τ is a conservative transformation from transition system $\mathcal{T}$ to transition system $\mathcal{T}'$ then h^τ is a safe, consistent, goal-aware and admissible heuristic for $\mathcal{T}$.

Proof.

We prove goal-awareness and consistency, the other properties follow from these two.

Goal-awareness: For all goal states $s_\star$ of $\mathcal{T}$, state $\sigma(s_\star)$ is a goal state of $\mathcal{T}'$ and therefore $h^\tau(s_\star) = h^*_{\mathcal{T}'}(\sigma(s_\star)) = 0$. ...

Conservative Transformations: Heuristic Properties (2)

Proof (continued).

Consistency: Let c and c' be the label cost functions of $\mathcal{T}$ and $\mathcal{T}'$, respectively. Consider state s of $\mathcal{T}$ and transition $\langle s, \ell, t \rangle$.
As $\mathcal{T}'$ has a transition $\langle \sigma(s), \lambda(\ell), \sigma(t) \rangle$, it holds that

$$\begin{aligned} h^\tau(s) &= h^*_{\mathcal{T}'}(\sigma(s)) \\ &\leq c'(\lambda(\ell)) + h^*_{\mathcal{T}'}(\sigma(t)) \\ &= c'(\lambda(\ell)) + h^\tau(t) \\ &\leq c(\ell) + h^\tau(t) \end{aligned}$$

The second inequality holds due to the requirement on the label costs. □

Exact Transformations

Definition (Exact Transformation)

Let $\mathcal{T}$ and $\mathcal{T}'$ be transition systems with label sets L and L' and cost functions c and c', respectively.

A transformation $\langle \mathcal{T}, \sigma, \lambda, \mathcal{T}' \rangle$ is exact if it is conservative and

1. if $\langle s', \ell', t' \rangle$ is a transition of $\mathcal{T}'$ then for all $s \in \sigma^{-1}(s')$ there is a transition $\langle s, \ell, t \rangle$ of $\mathcal{T}$ with $t \in \sigma^{-1}(t')$ and $\ell \in \lambda^{-1}(\ell')$,
2. if s' is a goal state of $\mathcal{T}'$ then all states $s \in \sigma^{-1}(s')$ are goal states of $\mathcal{T}$, and
3. $c(\ell) = c'(\lambda(\ell))$ for all $\ell \in L$.

$\rightsquigarrow$ no "new" transitions and goal states, no cheaper labels

Heuristic Properties with Exact Transformations (1)

Theorem

If τ is an exact transformation from transition system $\mathcal{T}$ to transition system $\mathcal{T}'$ then h^τ is the perfect heuristic h^* for $\mathcal{T}$.

Proof.

As the transformation is conservative, h^τ is admissible for $\mathcal{T}$ and therefore $h^*_{\mathcal{T}}(s) \geq h^\tau(s)$.

For the other direction, we show that for every state s' of $\mathcal{T}'$ and goal path π' for s', there is for each $s \in \sigma^{-1}(s')$ a goal path in $\mathcal{T}$ that has the same cost. ...

Heuristic Properties with Exact Transformations (2)

Proof (continued).

Proof via induction over the length of π'.

$|\pi'| = 0$: If s' is a goal state of $\mathcal{T}'$ then each $s \in \sigma^{-1}(s')$ is a goal state of $\mathcal{T}$ and the empty path is a goal path for s in $\mathcal{T}$.

$|\pi'| = i + 1$: Let $\pi' = \langle s', \ell', t' \rangle \pi'_{t'}$, where $\pi'_{t'}$ is a goal path of length i from t'. Then there is for each $t \in \sigma^{-1}(t')$ a goal path π_t of the same cost in $\mathcal{T}$ (by ind. hypothesis). Furthermore, for all $s \in \sigma^{-1}(s')$ there is a state $t \in \sigma^{-1}(t')$ and a label $\ell \in \lambda^{-1}(\ell')$ such that $\mathcal{T}$ has a transition $\langle s, \ell, t \rangle$. The path $\pi = \langle s, \ell, t \rangle \pi_t$ is a solution for s in $\mathcal{T}$. As ℓ and ℓ' must have the same cost and π_t and $\pi'_{t'}$ have the same cost, π has the same cost as π'. □

Composing Transformations

Merge-and-shrink performs many transformations in sequence.
We can formalize this with a notion of composition:

- Given $\tau = \mathcal{T} \xrightarrow{\sigma, \lambda} \mathcal{T}'$ and $\tau' = \mathcal{T}' \xrightarrow{\sigma', \lambda'} \mathcal{T}''$, their composition $\tau'' = \tau' \circ \tau$ is defined as $\tau'' = \mathcal{T} \xrightarrow{\sigma' \circ \sigma, \lambda' \circ \lambda} \mathcal{T}''$.
- If τ and τ' are conservative, then $\tau' \circ \tau$ is conservative.
- If τ and τ' are exact, then $\tau' \circ \tau$ is exact.

Merge-and-Shrink Transformations

F: factored transition system

Replacement with Synchronized Product is Conservative and Exact

Let $\mathcal{T}_1, \mathcal{T}_2 \in F$ with $\mathcal{T}_1 \neq \mathcal{T}_2$.
Let $F' := (X \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$.
Then there is an exact transformation $\langle \otimes F, \sigma, \mathrm{id}, \otimes F' \rangle$.

Up to the isomorphism we know from the synchronized product, we can use $\sigma = \mathrm{id}$.

Abstraction is Conservative

Let α be an abstraction of $\mathcal{T}_i \in F$ and let $F' := (F \setminus \{\mathcal{T}_i\}) \cup \{\mathcal{T}_i^\alpha\}$.
The transformation $\langle \otimes F, \sigma, \mathrm{id}, \otimes F' \rangle$ with
$\sigma(\langle s_1, \ldots, s_n \rangle) = \langle s_1, \ldots, s_{i-1}, \alpha(s_i), s_{i+1}, \ldots, s_n \rangle$ is conservative.

(Proofs omitted.)

Properties of Merge-and-Shrink Heuristics

We can conclude the following properties of merge-and-shrink heuristics for SAS^+ tasks:

- The heuristic is always admissible and consistent (because it is induced by a a composition of conservative transformations).
- If all shrink transformation used are exact, the heuristic is perfect (because it is induced by a composition of exact transformations).

E11.2 Shrink Strategies

Reminder: Generic Algorithm Template

$F := F(\Pi)$
while $|F| > 1$:
    select $type \in \{\text{merge}, \text{shrink}\}$
    if $type = \text{merge}$:
        select $\mathcal{T}_1, \mathcal{T}_2 \in F$
        $F := (F \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
    if $type = \text{shrink}$:
        select $\mathcal{T} \in F$
        choose an abstraction mapping β on $\mathcal{T}$
        $F := (F \setminus \{\mathcal{T}\}) \cup \{\mathcal{T}^\beta\}$
return the remaining factor $\mathcal{T}^\alpha$ in F

Remaining Questions:

- Which abstractions to select for merging? $\rightsquigarrow$ merge strategy
- How to shrink an abstraction? $\rightsquigarrow$ shrink strategy

Shrink Strategies

How to shrink an abstraction?

We cover two common approaches:

- f-preserving shrinking
- bisimulation-based shrinking

f-preserving Shrink Strategy

f-preserving Shrink Strategy

Repeatedly combine abstract states with
identical abstract goal distances (h values) and
identical abstract initial state distances (g values).

Rationale: preserves heuristic value and overall graph shape

Tie-breaking Criterion

Prefer combining states where $g + h$ is high.
In case of ties, combine states where h is high.

Rationale: states with high $g + h$ values are less likely to be explored by A*, so inaccuracies there matter less

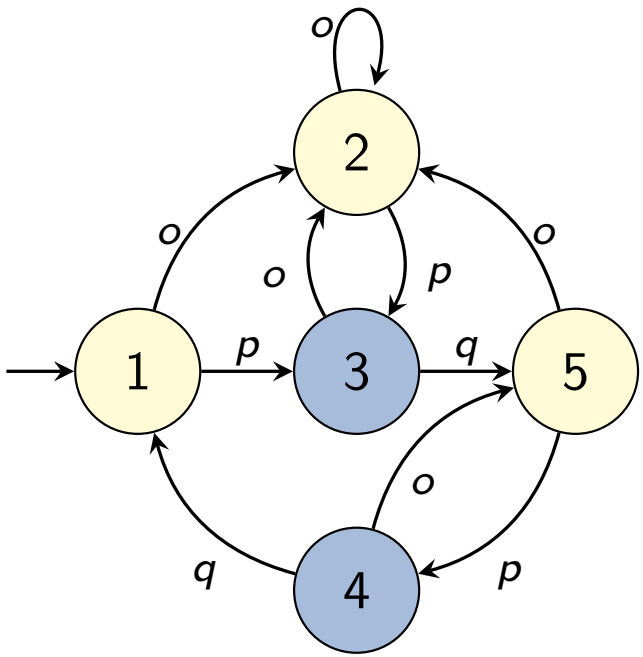
Bisimulation

Definition (Bisimulation)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ be a transition system. An equivalence relation $\sim$ on S is a bisimulation for $\mathcal{T}$ if for every $\langle s, \ell, s' \rangle \in T$ and every $t \sim s$ there is a transition $\langle t, \ell, t' \rangle \in T$ with $t' \sim s'$.

A bisimulation $\sim$ is goal-respecting if $s \sim t$ implies that either $s, t \in S_\star$ or $s, t \notin S_\star$.

Bisimulation: Example



$\sim$ with equivalence classes $\{\{1, 2, 5\}, \{3, 4\}\}$ is a goal-respecting bisimulation.

Bisimulation Abstractions

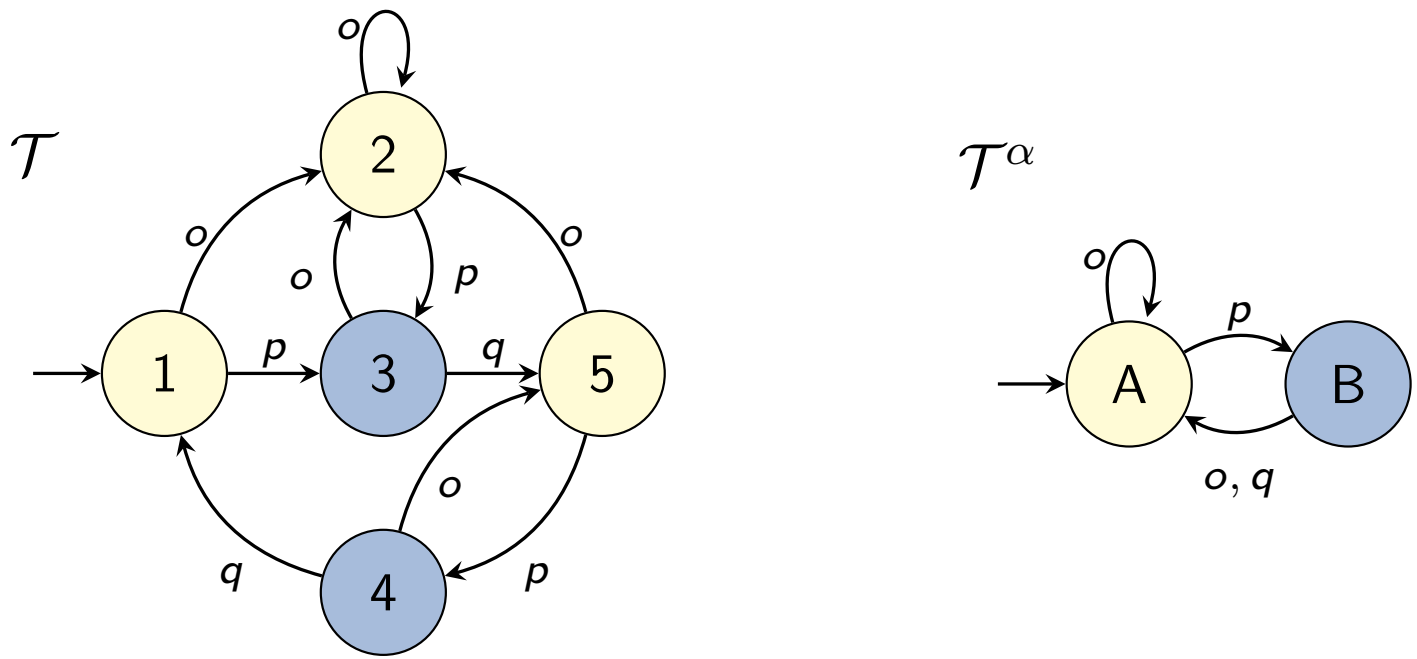
Definition (Abstractions as Bisimulation)

Let $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ be a transition system and $\alpha : S \to S'$ be an abstraction of $\mathcal{T}$. The abstraction induces the equivalence relation $\sim_\alpha$ as $s \sim_\alpha t$ iff $\alpha(s) = \alpha(t)$.

We say that α is a (goal-respecting) bisimulation for $\mathcal{T}$ if $\sim_\alpha$ is a (goal-respecting) bisimulation for $\mathcal{T}$.

Abstraction as Bisimulations: Example

Abstraction α with
$\alpha(1) = \alpha(2) = \alpha(5) = A$ and $\alpha(3) = \alpha(4) = B$
is a goal-respecting bisimulation for $\mathcal{T}$.



Goal-respecting Bisimulations are Exact

Theorem

Let F be a factored transition system and α be an abstraction of $\mathcal{T}_i \in F$.
If α is a goal-respecting bisimulation then the transformation $\langle \otimes F, \sigma, id, \otimes F' \rangle$ with

- $\sigma(\langle s_1, \ldots, s_n \rangle) = \langle s_1, \ldots, s_{i-1}, \alpha(s_i), s_{i+1}, \ldots, s_n \rangle$ *and*
- $F' := (F \setminus \{\mathcal{T}_i\}) \cup \{\mathcal{T}_i^\alpha\}$

is exact.

(Proofs omitted.)

Shrinking with bisimulation preserves the heuristic estimates.

Bisimulations: Discussion

- As all bisimulations preserve all relevant information, we are interested in the coarsest such abstraction (to shrink as much as possible).
- There is always a unique coarsest bisimulation for $\mathcal{T}$ and it can be computed efficiently (from the explicit representation).
- In some cases, computing the bisimulation is still too expensive or it cannot sufficiently shrink a transition system.

E11.3 Summary

Summary

- Merge-and-shrink abstractions can be analyzed by viewing them as a sequence of transformations.
- We only use conservative transformations, and hence merge-and-shrink heuristics for SAS^+ tasks are admissible and consistent.
- Merge-and-shrink heuristics for SAS^+ tasks that only use exact transformations are perfect.
- Bisimulation is an exact shrinking method.

Planning and Optimization

E12. Merge-and-Shrink: Merge Strategies & Outlook

Malte Helmert and Gabriele Röger

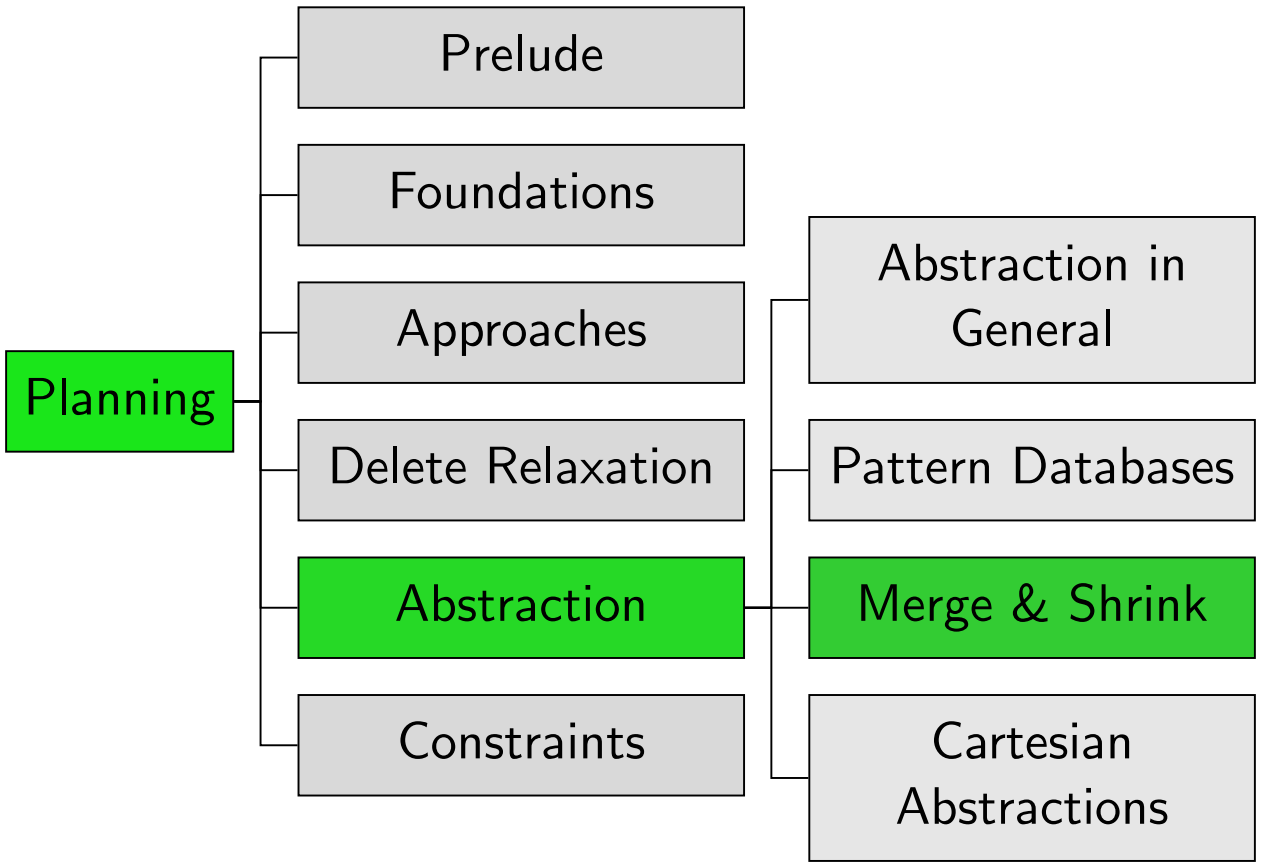
Universität Basel

November 19, 2025

Planning and Optimization

November 19, 2025 — E12. Merge-and-Shrink: Merge Strategies & Outlook

Content of the Course



E12.1 Merge Strategies

Reminder: Generic Algorithm Template

Generic Merge & Shrink Algorithm for planning task Π

$F := F(\Pi)$
while $|F| > 1$:
    select $type \in \{\text{merge}, \text{shrink}\}$
    if $type = \text{merge}$:
        select $\mathcal{T}_1, \mathcal{T}_2 \in F$
        $F := (F \setminus \{\mathcal{T}_1, \mathcal{T}_2\}) \cup \{\mathcal{T}_1 \otimes \mathcal{T}_2\}$
    if $type = \text{shrink}$:
        select $\mathcal{T} \in F$
        choose an abstraction mapping β on $\mathcal{T}$
        $F := (F \setminus \{\mathcal{T}\}) \cup \{\mathcal{T}^\beta\}$
return the remaining factor $\mathcal{T}^\alpha$ in F

Remaining Question:

- Which abstractions to select for merging? $\rightsquigarrow$ merge strategy

Linear vs. Non-linear Merge Strategies

Linear Merge Strategy

In each iteration after the first, choose the abstraction computed in the previous iteration as $\mathcal{T}_1$.

Rationale: only maintains one "complex" abstraction at a time

- Fully defined by an ordering of atomic projections/variables.
- Each merge-and-shrink heuristic computed with a non-linear merge strategy can also be computed with a linear merge strategy.
- However, linear merging can require a super-polynomial blow-up of the final representation size.
- Recent research turned from linear to non-linear strategies, also because better label reduction techniques (later in this chapter) enabled a more efficient computation.

Classes of Merge Strategies

We can distinguish two major types of merge strategies:

- precomputed merge strategies fix a unique merge order up-front.
  One-time effort but cannot react to other transformations applied to the factors.
- stateless merge strategies only consider the current FTS and decide what factors to merge.
  Typically computing a score for each pair of factors and naturally non-linear; easy to implement but cannot capture dependencies between more than two factors.

Hybrid strategies combine ideas from precomputed and stateless strategies.

Example Linear Precomputed Merge Strategy

Idea: Use similar causal graph criteria as for growing patterns.

Example: Strategy of h_{HHH}

h_{HHH}: Ordering of atomic projections

- Start with a goal variable.
- Add variables that appear in preconditions of operators affecting previous variables.
- If that is not possible, add a goal variable.

Rationale: increases h quickly

Example Non-linear Precomputed Merge Strategy

Idea: Build clusters of variables with strong interactions and first merge variables within each cluster.

Example: MIASM ("maximum intermediate abstraction size minimizing merging strategy")

MIASM strategy

- Measure interaction by ratio of unnecessary states in the merged system (= states not traversed by any abstract plan).
- Best-first search to identify interesting variable sets.
- Disjoint variable sets chosen by a greedy algorithm for maximum weighted set packing.

Rationale: increase power of pruning (later in this chapter)

Example Non-linear Stateless Merge Strategy

Idea: Preferrably merge transition systems that must synchronize on labels that occur close to a goal state.

Example: DFP (named after Dräger, Finkbeiner and Podelski)

DFP strategy

- $\mathit{labelrank}(\ell, \mathcal{T}) = \min\{h^*(t) \mid \langle s, \ell, t\rangle \text{ transition in } \mathcal{T}\}$
- $\mathit{score}(\mathcal{T}, \mathcal{T}') = \min\{\max\{\mathit{labelrank}(\ell, \mathcal{T}), \mathit{labelrank}(\ell, \mathcal{T}')\} \mid \ell \text{ label in } \mathcal{T} \text{ and } \mathcal{T}'\}$
- Select two transition systems with minimum score.

Rationale: abstraction fine-grained in the goal region, which is likely to be searched by A^*.

Example Hybrid Merge Strategy

Idea: first combine the variables within each strongly connected component of the causal graph.

Example: SCC framework

SCC strategy

- Compute strongly connected components of causal graph
- Secondary strategies for order in which
  - the SCCs are considered (e.g. topologic order),
  - the factors within an SCC are merged, and
  - the resulting product systems are merged.

Rationale: reflect strong interactions of variables well

State of the art: SCC+DFP or a stateless MIASM variant

E12.2 Outlook: Label Reduction and Pruning

Further Transformations

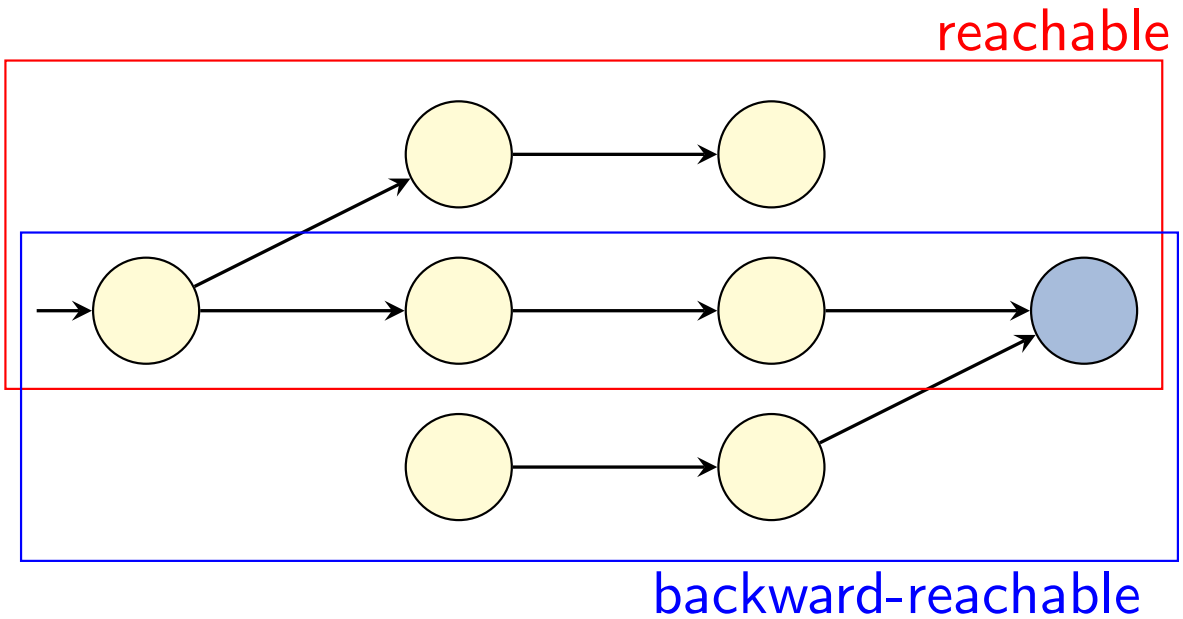
State-of-the-art Merge & Shrink uses two further transformations:

- Label reduction
- Pruning

Label Reduction

- Do no longer distinguish certain labels, similar to abstraction that does not distinguish certain states.
- A label reduction $\langle \lambda, c' \rangle$ for a FTS F with label set L is given by a function $\lambda : L \to L'$, where L' is an arbitrary set of labels, and a label cost function c' on L' such that for all $\ell \in L$, $c'(\lambda(\ell)) \leq c(\ell)$.
  The label-reduced TSs have L' and c' for the labels and cost, and in each transition the original label ℓ is replaced with $\lambda(\ell)$.
- Label reduction is a conservative transformation.
- There are also clear criteria when label reduction is exact.
- Reduces the time and memory requirement for merge and shrink steps and enables coarser bisimulation abstractions.

Alive States



- state s is reachable if we can reach it from the initial state
- state s is backward-reachable if we can reach the goal from s
- state s is alive if it is reachable and backward-reachable
  $\rightarrow$ only alive states can be traversed by a solution
- a state s is dead if it is not alive.

Pruning States (1)

- If in a factor, state s is dead/not backward-reachable then all states that "cover" s in a synchronized product are dead/not backward-reachable in the synchronized product.
- Removing such states and all adjacent transitions in a factor does not remove any solutions from the synchronized product.
- This pruning leads to states in the original state space for which the merge-and-shrink abstraction does not define an abstract state.
  $\rightarrow$ use heuristic estimate ∞

Pruning States (2)

- Keeping exactly all backward-reachable states we still obtain safe, consistent, goal-aware and admissible (with conservative transformations) or perfect heuristics (with exact transformations).
- Pruning unreachable, backward-reachable states can render the heuristic unsafe because pruned states lead to infinite estimates.
- However, all reachable states in the original state space will have admissible estimates, so we can use the heuristic like an admissible one in a forward state-space search such as A*(but not in other contexts like such as orbit search).
  We usually prune all dead states to keep the factors small.

E12.3 Summary

Summary

- There is a wide range of merge strategies. We only covered some important ones.
- Label reduction is crucial for the performance of the merge-and-shrink algorithm, especially when using bisimilarity for shrinking.
- Pruning is used to keep the size of the factors small. It depends on the intended application how aggressive the pruning can be.

E12.4 Literature

Literature (1)

References on merge-and-shrink abstractions:


Klaus Dräger, Bernd Finkbeiner and Andreas Podelski.
Directed Model Checking with Distance-Preserving Abstractions.
Proc. SPIN 2006, pp. 19–34, 2006.
Introduces merge-and-shrink abstractions (for model checking) and DFP merging strategy.


Malte Helmert, Patrik Haslum and Jörg Hoffmann.
Flexible Abstraction Heuristics for Optimal Sequential Planning.
Proc. ICAPS 2007, pp. 176–183, 2007.
Introduces merge-and-shrink abstractions for planning.

Literature (2)


Raz Nissim, Jörg Hoffmann and Malte Helmert.
Computing Perfect Heuristics in Polynomial Time:
On Bisimulation and Merge-and-Shrink Abstractions
in Optimal Planning.
Proc. IJCAI 2011, pp. 1983–1990, 2011.
Introduces bisimulation-based shrinking.


Malte Helmert, Patrik Haslum, Jörg Hoffmann
and Raz Nissim.
Merge-and-Shrink Abstraction: A Method
for Generating Lower Bounds in Factored State Spaces.
Journal of the ACM 61 (3), pp. 16:1–63, 2014.
Detailed journal version of the previous two publications.

Literature (3)


Silvan Sievers, Martin Wehrle and Malte Helmert.
Generalized Label Reduction for Merge-and-Shrink Heuristics.
Proc. AAAI 2014, pp. 2358–2366, 2014.
Introduces modern version of label reduction.
(There was a more complicated version before.)


Gaojian Fan, Martin Müller and Robert Holte.
Non-linear merging strategies for merge-and-shrink
based on variable interactions.
Proc. SoCS 2014, pp. 53–61, 2014.
Introduces UMC and MIASM merging strategies

Literature (4)

Malte Helmert, Gabriele Röger and Silvan Sievers.
On the Expressive Power of Non-Linear Merge-and-Shrink Representations.
Proc. ICAPS 2015, pp. 106–114, 2015.
Shows that linear merging can require a super-polynomial blow-up in representation size.

Silvan Sievers and Malte Helmert.
Merge-and-Shrink: A Compositional Theory of Transformations of Factored Transition Systems.
JAIR 71, pp. 781–883, 2021.
Detailed theoretical analysis of task transformations as sequence of transformations.

Literature (5)

Silvan Sievers, Florian Pommerening , Thomas Keller and Malte Helmert.
Cost-Partitioned Merge-and-Shrink Heuristics for Optimal Classical Planning.
Proc. IJCAI 2020, pp. 4152–4160, 2020.
Extends saturated cost partitioning to merge-and-shrink.

Planning and Optimization

E13. Cartesian Abstractions

Malte Helmert and Gabriele Röger

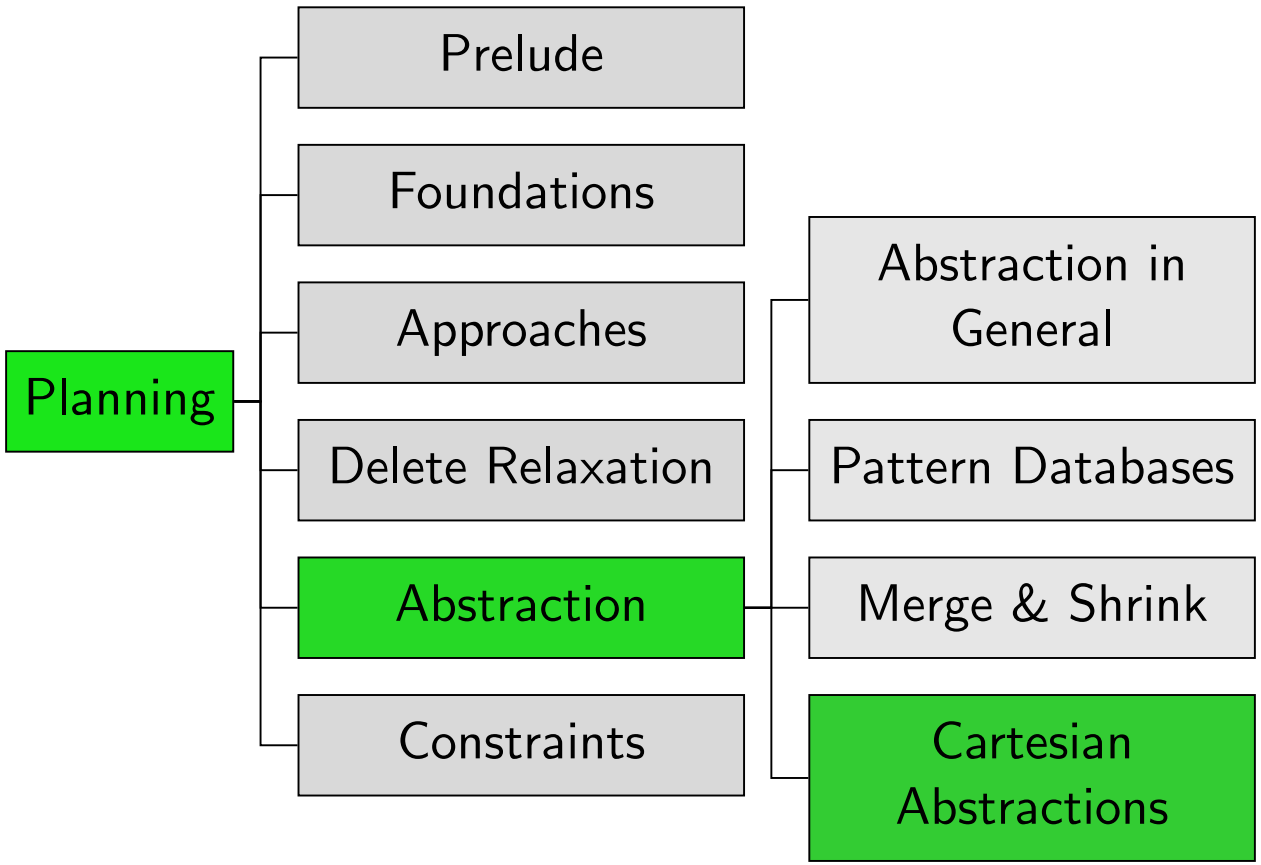
Universität Basel

November 24, 2025

Planning and Optimization

November 24, 2025 — E13. Cartesian Abstractions

Content of the Course



E13.1 Introduction

Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR) is an approach to compute a tailored abstraction for a task (or to solve it).

- Start with a very coarse abstraction.
- Iteratively compute an (optimal) abstract solution and check whether it works for the concrete tasks.
  - If yes, the task is solved.
  - If not, refine the abstraction so that the same flaw will not be encountered in future iterations.

CEGAR is another technique originally introduced for model checking.

Our Plan for Today

- For a certain class of abstractions (the Cartesian abstractions), CEGAR can be efficiently implemented.
- In this chapter, we get to know this class of abstractions and the necessary foundations.
- In the next chapter, we see how they can be used within CEGAR.

Remarks

- In Ch. E13 and E14 we continue to only consider SAS^+ tasks.
- To facilitate notation, we will use an arbitrary (but fixed) order on the variables.
  $\rightarrow$ Tuple of variables instead of set of variables.
- These chapters are based on:
  Jendrik Seipp and Malte Helmert.
  Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. Journal of Artificial Intelligence Research 62, pp. 535-577. 2018.

Example Task: Two Packages, One Truck

In E13 and E14 we use the following running example.

Example (Two Packages, One Truck)

Consider the following FDR planning task $\langle V, I, O, \gamma \rangle$:

- $V = \{p_A, p_B, t\}$ with
  - $\text{dom}(p)_A = \text{dom}(p_B) = \{L, I, R\}$
  - $\text{dom}(t) = \{L, R\}$
- $I = \{p_A \mapsto L, p_B \mapsto L, t \mapsto L\}$
- $O = \{\text{pickup}_{i,j} \mid i \in \{A, B\}, j \in \{L, R\}\}$
  $\cup \{\text{drop}_{i,j} \mid i \in \{A, B\}, j \in \{L, R\}\}$
  $\cup \{\text{move}_{i,j} \mid i, j \in \{L, R\}, i \neq j\}$, where
  - $\text{pickup}_{i,j} = \langle p_i = j \wedge t = j, p_i := I, 1 \rangle$
  - $\text{drop}_{i,j} = \langle p_i = I \wedge t = j, p_i := j, 1 \rangle$
  - $\text{move}_{i,j} = \langle t = i, t := j, 1 \rangle$
- $\gamma = (p_A = R \wedge p_B = R)$

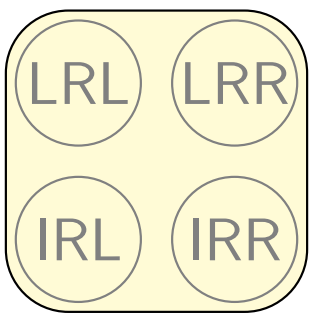
E13.2 Cartesian Sets

Cartesian Sets

Definition

A set of states for a planning task with variables $\langle v_1, \ldots, v_n \rangle$ is called Cartesian if it is of the form $A_1 \times \cdots \times A_n$, where $A_i \subseteq \mathrm{dom}(v_i)$ for all $1 \leq i \leq n$.

$\{L, I\} \times \{R\} \times \{L, R\} = \{(L, R, L), (L, R, R), (I, R, L), (I, R, R)\}$
for variables $\langle p_A, p_B, t \rangle$



Conjunctions of Atoms as Cartesian Sets

For a conjunction φ of atoms, the set of all states s with $s \models \varphi$ is Cartesian and can be defined as follows:

Definition

Let φ be a conjunction of atoms over finite domain variables $V = \langle v_1, \ldots, v_n \rangle$. The Cartesian set induced by φ is $\mathit{Cartesian}(\varphi) = A_1 \times \cdots \times A_n$, where

$$A_i = \begin{cases} \mathrm{dom}(v_i) & \text{if } \varphi \text{ contains no atom } v_i = d, \\ \{d\} & \text{if } \varphi \text{ contains an atom } v_i = d \text{ and} \\ & \quad \text{no atom } v_i = d' \text{ with } d \neq d' \\ \emptyset & \text{otherwise (conflicting atoms for } v_i). \end{cases}$$

Conjunctions of Atoms as Cartesian Sets: Examples

In the running example with variables $\langle p_A, p_B, t \rangle$

- $\mathit{Cartesian}(p_A = R \wedge t = L) = \{R\} \times \{L, I, R\} \times \{L\}$
- $\mathit{Cartesian}(p_A = R \wedge t = L \wedge t = R) = \{R\} \times \{L, I, R\} \times \emptyset$

Properties of Cartesian Sets

Theorem

Let $\Pi = \langle V, O, I, \gamma \rangle$ be a SAS$^+$ planning task.

1. *The set of goal states of Π is Cartesian.*
2. *For all $o \in O$, the set of states in which o is applicable is Cartesian.*
3. *The intersection of Cartesian sets over the same variables is Cartesian.*
4. *For all operators o, the regression of a Cartesian set wrt. o is Cartesian.*

From the proofs we will see that the corresponding Cartesian sets are easy to determine.

Properties of Cartesian Sets

Proof Sketch.

1. The set of goal states is $\mathit{Cartesian}(\gamma)$.
2. For $o \in O$, the set of states in which o is applicable is $\mathit{Cartesian}(\mathit{pre}(o))$.
3. The intersection of Cartesian sets $A_1 \times \cdots \times A_n$ and $B_1 \times \cdots \times B_n$ is $(A_1 \cap B_1) \times \cdots \times (A_n \cap B_n)$.

...

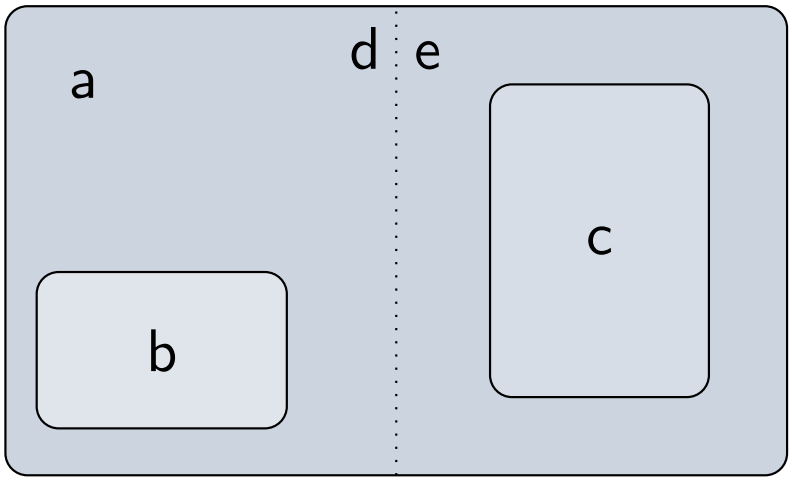
Properties of Cartesian Sets

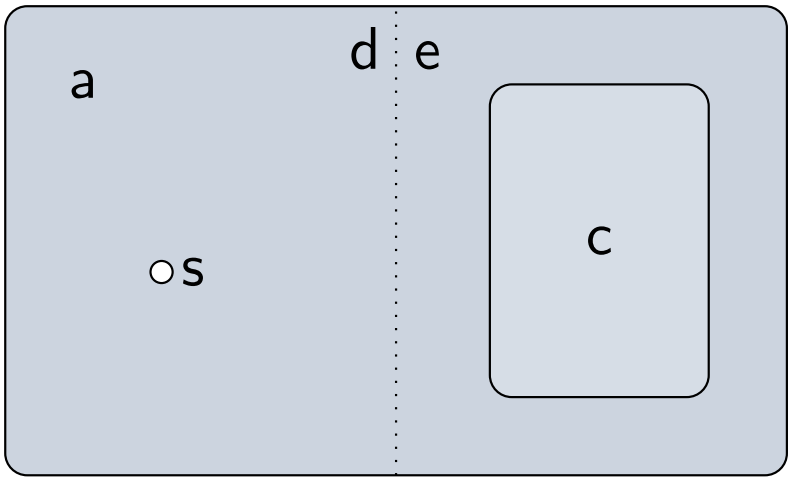
Proof Sketch (continued).

4. With variables $\langle v_1, \ldots, v_n \rangle$, the regression of Cartesian set $b = B_1 \times \cdots \times B_n$ wrt. o is $regr(b, o) = A_1 \times \cdots \times A_n$, where

$$A_i = \begin{cases} B_i & \text{if } v_i \text{ does not occur in } pre(o) \text{ and } eff(o) \\ \emptyset & \text{if } o \text{ has an effect setting } v_i \text{ to } d' \notin B_i \\ & \text{or if } o \text{ has no effect on } v_i \\ & \quad \text{but a precondition } v_i = d \text{ with } d \notin B_i. \\ \mathrm{dom}(v_i) & \text{if } o \text{ has no precondition on } v_i \text{ and} \\ & \quad \text{an effect setting } v_i \text{ to } d' \in B_i \\ \{d\} & \text{if } o \text{ has a precondition } v_i = d \text{ and} \\ & \quad \text{an effect setting } v_i \text{ to } d' \in B_i \\ & \text{or if } o \text{ has precondition } v_i = d \text{ with } d \in B_i \\ & \quad \text{and no effect on } v_i \end{cases}$$

Splitting Cartesian Sets

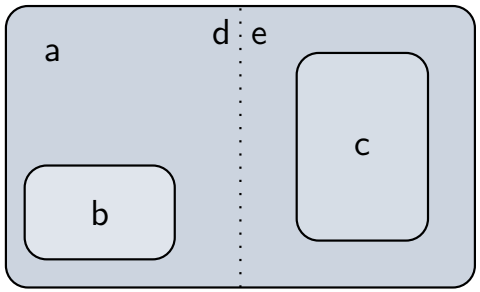




Theorem (Splits)

1. *If $b \subseteq a$ and $c \subseteq a$ are disjoint Cartesian subsets of the Cartesian set a, then a can be partitioned into Cartesian sets d and e with $b \subseteq d$ and $c \subseteq e$.*
2. *If $c \subseteq a$ is a Cartesian subset of the Cartesian set a and $s \in a \setminus c$, then a can be partitioned into Cartesian sets d and e with $s \in d$ and $c \subseteq e$.*

Splitting Cartesian Sets



Proof.

For 1), let $a = A_1 \times \cdots \times A_n$, $b = B_1 \times \cdots \times B_n$ and $c = C_1 \times \cdots \times C_n$.

Let j be such that B_j and C_j are disjoint. It must exist because otherwise b and c are not disjoint (we could select for each variable v_i a value in $B_i \cap C_i$).

Partition A_j into D_j and E_j with $B_j \subseteq D_j$ and $C_j \subseteq E_j$, e.g. $E_j = C_j$ and $D_j = A_j \setminus C_j$.

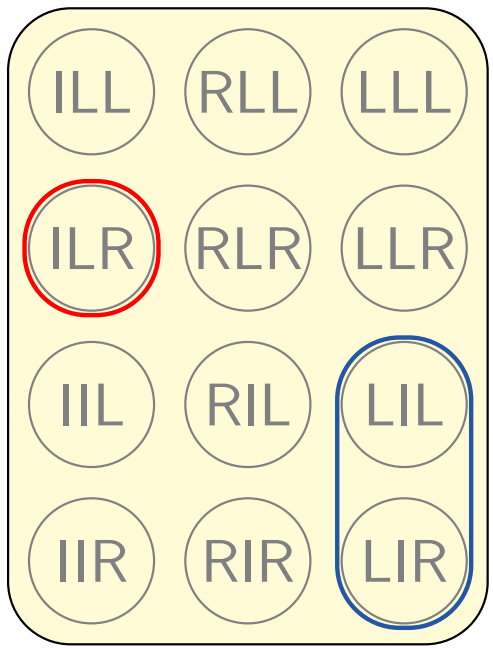
Then $d = A_1 \times \cdots \times A_{j-1} \times D_j \times A_{j+1} \times \cdots \times A_n$ and $e = A_1 \times \cdots \times A_{j-1} \times E_j \times A_{j+1} \times \cdots \times A_n$

2) follows from 1) by setting $b = \{s\}$ (a Cartesian set). □

Splitting Cartesian Sets: Example

$$a : \{I, R, L\} \times \{L, I\} \times \{L, R\}$$

$b : \{I\} \times \{L\} \times \{R\}$



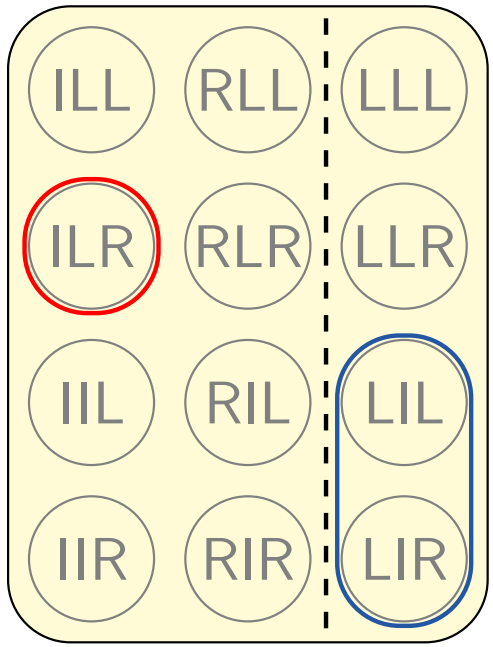
$c : \{L\} \times \{I\} \times \{L, R\}$

On which variable(s) can we split? $\rightsquigarrow$ first or second.
What are the two Cartesian sets d and e in each case?

Splitting Cartesian Sets: Example

$a : \{I, R, L\} \times \{L, I\} \times \{L, R\}$

$b : \{I\} \times \{L\} \times \{R\}$



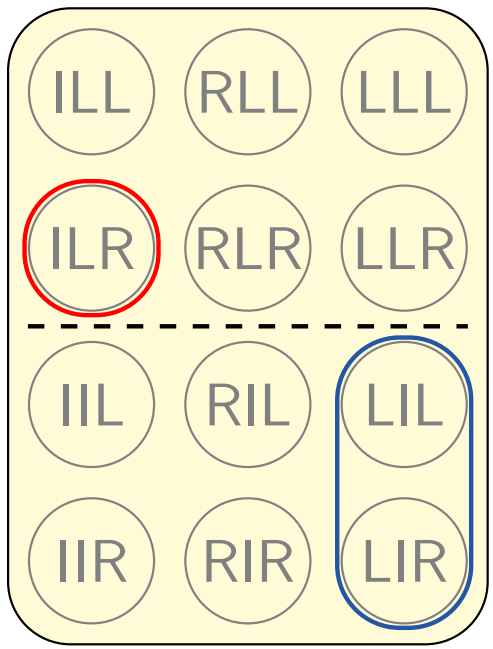
$c : \{L\} \times \{I\} \times \{L, R\}$

Split on first variable:
$d = \{I, R\} \times \{L, I\} \times \{L, R\}$ and $e = \{L\} \times \{L, I\} \times \{L, R\}$

Splitting Cartesian Sets: Example

$a : \{I, R, L\} \times \{L, I\} \times \{L, R\}$

$b : \{I\} \times \{L\} \times \{R\}$



$c : \{L\} \times \{I\} \times \{L, R\}$

Split on second variable:
$d = \{I, R, L\} \times \{L\} \times \{L, R\}$ and $e = \{I, R, L\} \times \{I\} \times \{L, R\}$

E13.3 Cartesian Abstractions

Reminder: Abstractions as Equivalence Relations

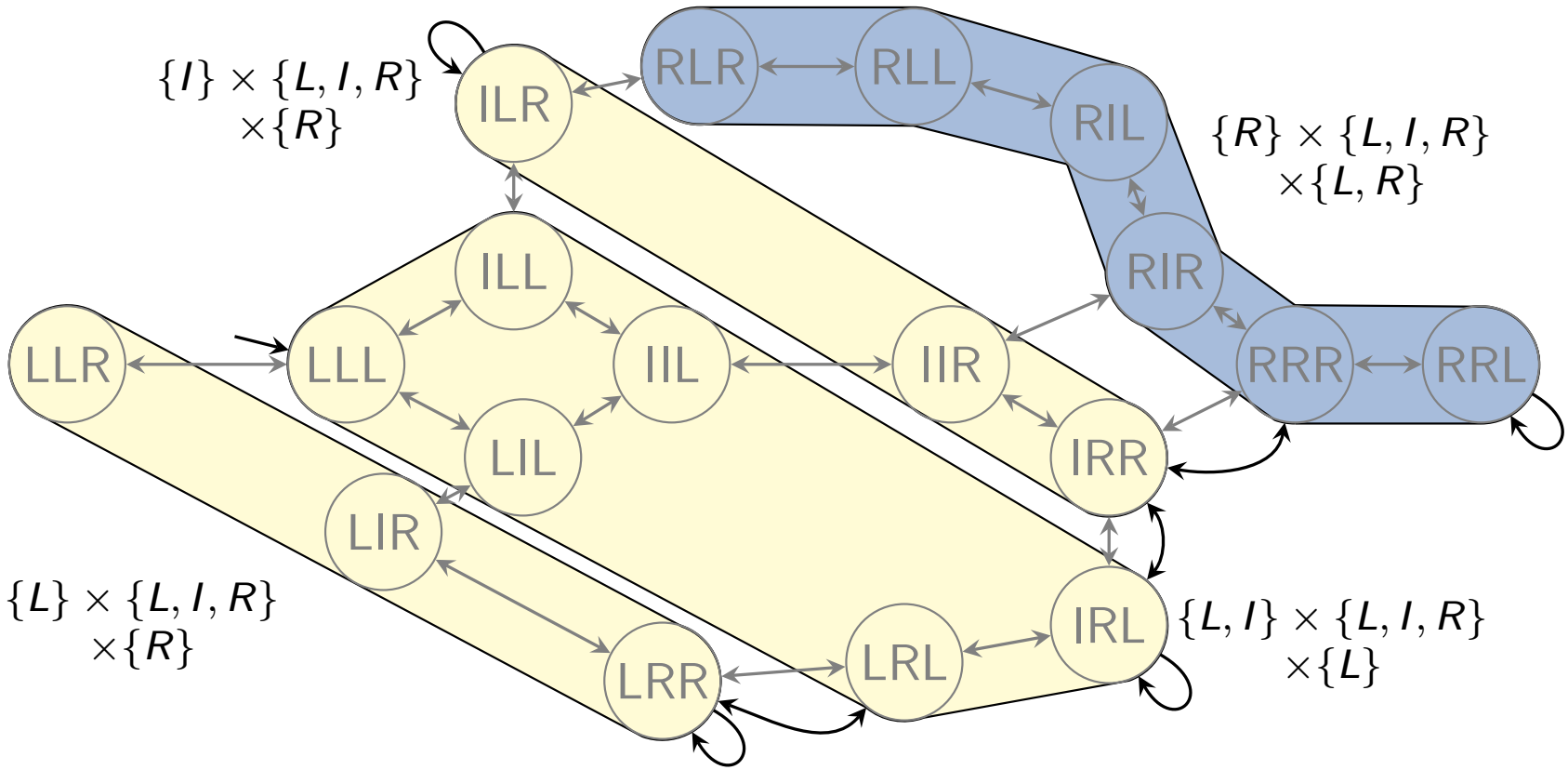
- An abstraction α induces the equivalence relation $\sim_\alpha$ over the set of (concrete) states as $s \sim_\alpha t$ iff $\alpha(s) = \alpha(t)$.
- The equivalence class $[s]_\alpha$ of state s is the set of all concrete states that are mapped to the same abstract state as s.
- We write $\sim$ and $[s]$, if α is clear from context.

Cartesian Abstraction

Definition

An abstraction α is called Cartesian if all equivalence classes of $\sim_\alpha$ are Cartesian sets.

Example



Labels omitted for clarity.

Relationship to other Classes of Abstractions

- Cartesian abstractions generalize projections (PDBs): the equivalence classes of projections are Cartesian.
- Merge & Shrink abstractions are more general than Cartesian abstractions (every abstraction can be represented as Merge & Shrink abstraction).
- Merge & Shrink and Cartesian abstractions are incomparable in representation size: there are compact Cartesian abstractions that do not have a compact Merge & Shrink representation and vice versa.

E13.4 Summary

Summary

- Cartesian sets are sets of states that can be represented as a Cartesian product of possible values for each variable.
- In Cartesian abstractions the sets of states that do not get distinguished must be Cartesian.

Planning and Optimization

E14. Cartesian Abstractions: CEGAR

Malte Helmert and Gabriele Röger

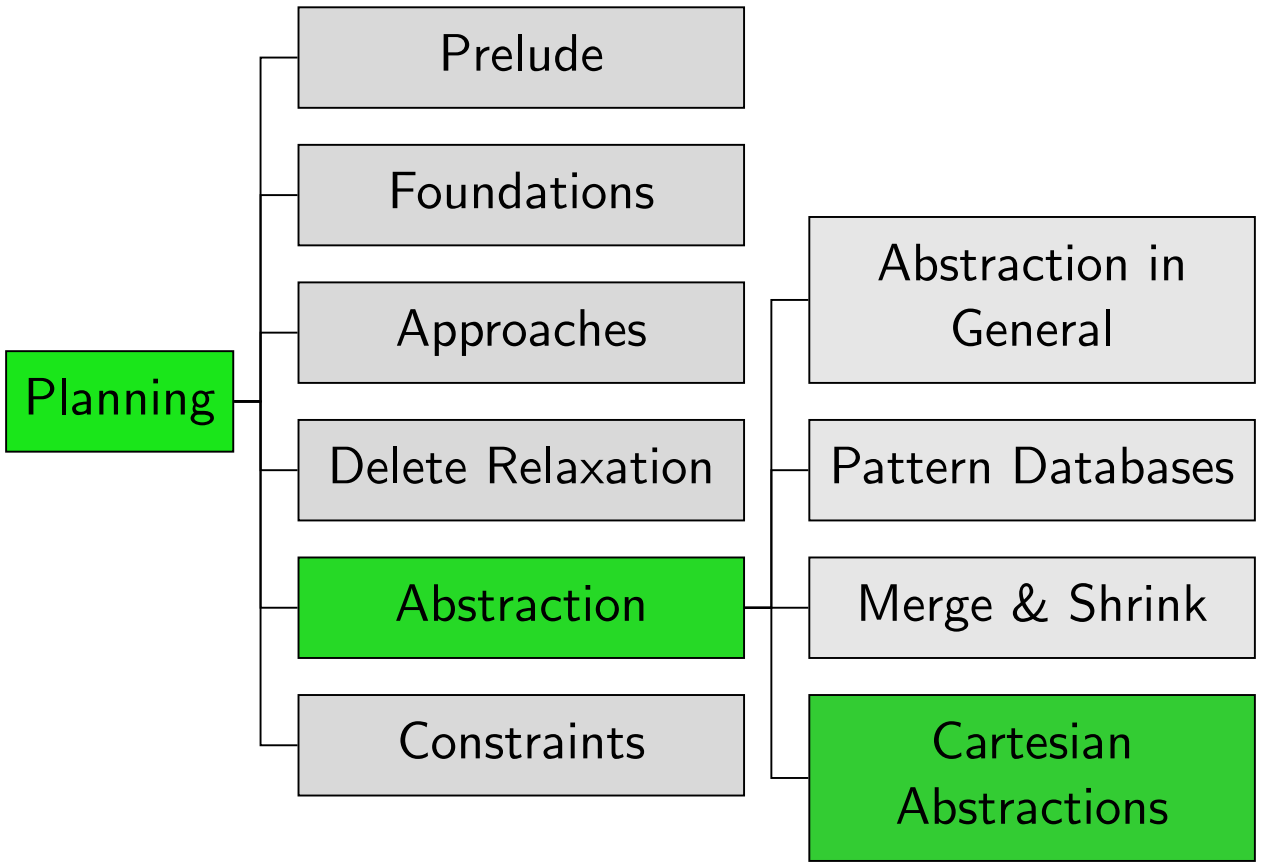
Universität Basel

November 24, 2025

Planning and Optimization

November 24, 2025 — E14. Cartesian Abstractions: CEGAR

Content of the Course



E14.1 CEGAR

Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR) is an approach to compute a tailored abstraction for a task (or to solve it).

- Start with a very coarse abstraction.
- Iteratively compute an (optimal) abstract solution and check whether it works for the concrete tasks.
  - If yes, the task is solved.
  - If not, refine the abstraction so that the same flaw will not be encountered in future iterations.

CEGAR Algorithm

Generic CEGAR algorithm for planning task Π

$\mathcal{T} :=$ TrivialAbstractTransitionSystem(Π) $\leftarrow$ one abstract state
while not TerminationCondition(): $\leftarrow$ e.g. time/memory limit
    $\tau :=$ FindOptimalTrace($\mathcal{T}$) $\leftarrow$ abstract solution (path in $\mathcal{T}$)
    if τ is "no trace" then **return** Π unsolvable
    $F :=$ FindFlaw($\tau, \Pi, \mathcal{T}$)
    if F is "no flaw" then
        return label sequence of τ as plan for Π
    $\mathcal{T} :=$ Refine($\mathcal{T}, F$)
return $\mathcal{T}$

Open questions:

- What are flaws (and how to find them)? $\rightsquigarrow$ next
- How do we refine the system?

E14.2 Flaws

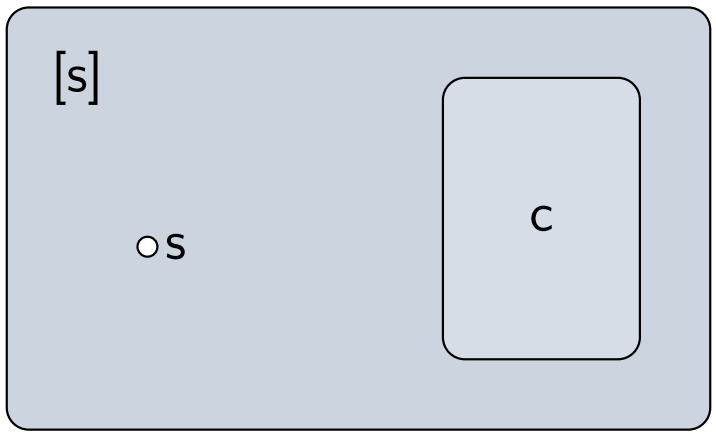
Flaws

A flaw is a reason why (the label sequence of) τ does not solve Π the way it solves the abstract system $\mathcal{T}$ (with abstraction α).

Start from the initial state of Π and iteratively apply the next operator (label) o from τ.

- Precondition flaw: o is not applicable in the current state s.
- Goal flaw: the final state is not a goal state.
- Deviation flaw: the next abstract transition is $a \xrightarrow{o} a'$, the current concrete state is s with $\alpha(s) = a$ but for successor state $s' = s[\![o]\!]$ we have $\alpha(s') \neq a'$ (deviating from the abstract path).

Extracting Flaws



For the refinement, we represent flaws in the form $\langle s, c \rangle$, where

- s is a concrete state,
- $c \subseteq [s]$ is a non-empty Cartesian set,
- the abstract plan relied on "being in c" but $s \notin c$.

$\langle s, c \rangle$ will define the split for the refinement step.

Extracting Different Kinds of Flaws

- Precondition flaw: if o is not applicabe in state s, use $\langle s, c \rangle$, where c is the set of concrete states in $[s]$ in which o is applicable.
- Goal flaw: if the final state s is not a goal state, use $\langle s, c \rangle$, where c is the set of concrete goal states in $[s]$.
- Deviation flaw: the next abstract transition is $a \xrightarrow{o} a'$, the current concrete state is s with $\alpha(s) = a$ but for successor state $s' = s[\![o]\!]$ we have $\alpha(s') \neq a'$ (deviating from the abstract path). Use (s, c), where c is the intersection of $[s]$ and $\mathit{regr}(a', o)$.

Easy for Cartesian abstractions, using the results from Ch. E13.

E14.3 Refinement

CEGAR Algorithm

Generic CEGAR algorithm for planning task Π

$\mathcal{T} :=$ TrivialAbstractTransitionSystem(Π)
while not TerminationCondition():
    $\tau :=$ FindOptimalTrace($\mathcal{T}$)
    if τ is "no trace" then **return** Π unsolvable
    $F :=$ FindFlaw($\tau, \Pi, \mathcal{T}$)
    if F is "no flaw" then
        return label sequence of τ as plan for Π
    $\mathcal{T} :=$ Refine($\mathcal{T}, F$)
return $\mathcal{T}$

Open questions:

- How do we refine the system?

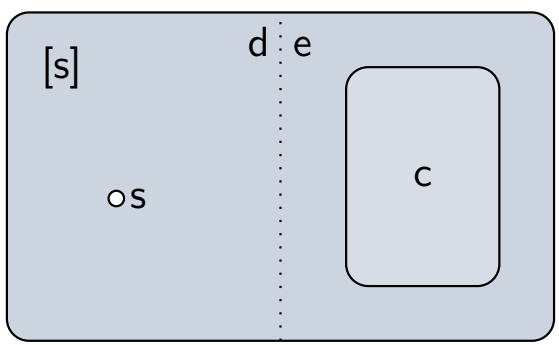
Refinement

Refinement splits abstract state $[s]$ and maintains the transition system induced by the underlying abstraction.

Refine($\langle S', L', c', T', s'_0, S'_\star \rangle$, $\langle s, c \rangle$)

$\langle d, e \rangle := \text{Split}([s], s, c)$
$S'' := S' \setminus \{[s]\} \cup \{d, e\}$
$T'' := \text{RewireTransitions}(T', [s], d, e)$
if $[s] = s'_0$ then $s''_0 := d$ else $s''_0 := s'_0$
if $[s] \in S'_\star$ then $S''_\star := (S''_\star \setminus \{[s]\}) \cup \{e\}$ else $S''_\star := S'_\star$
return $\langle S'', L', c', T'', s''_0, S''_\star \rangle$

Split $[s]$ into d and e.



Refinement

Refinement splits abstract state $[s]$ and maintains the transition system induced by the underlying abstraction.

Refine($\langle S', L', c', T', s'_0, S'_\star \rangle$, $\langle s, c \rangle$)

$\langle d, e \rangle := \text{Split}([s], s, c)$
$S'' := S' \setminus \{[s]\} \cup \{d, e\}$
$T'' := \text{RewireTransitions}(T', [s], d, e)$
if $[s] = s'_0$ then $s''_0 := d$ else $s''_0 := s'_0$
if $[s] \in S'_\star$ then $S''_\star := (S''_\star \setminus \{[s]\}) \cup \{e\}$ else $S''_\star := S'_\star$
return $\langle S'', L', c', T'', s''_0, S''_\star \rangle$

Update incident transitions of $[s]$.

- Check for each incoming and outgoing transition of $[s]$ (including self-loops) whether it needs to be rewired from/to d, from/to e, or both.
- Easy for SAS$^+$ operators and Cartesian abstract states.

Refinement

Refinement splits abstract state $[s]$ and maintains the transition system induced by the underlying abstraction.

Refine($\langle S', L', c', T', s'_0, S'_\star \rangle$, $\langle s, c \rangle$)

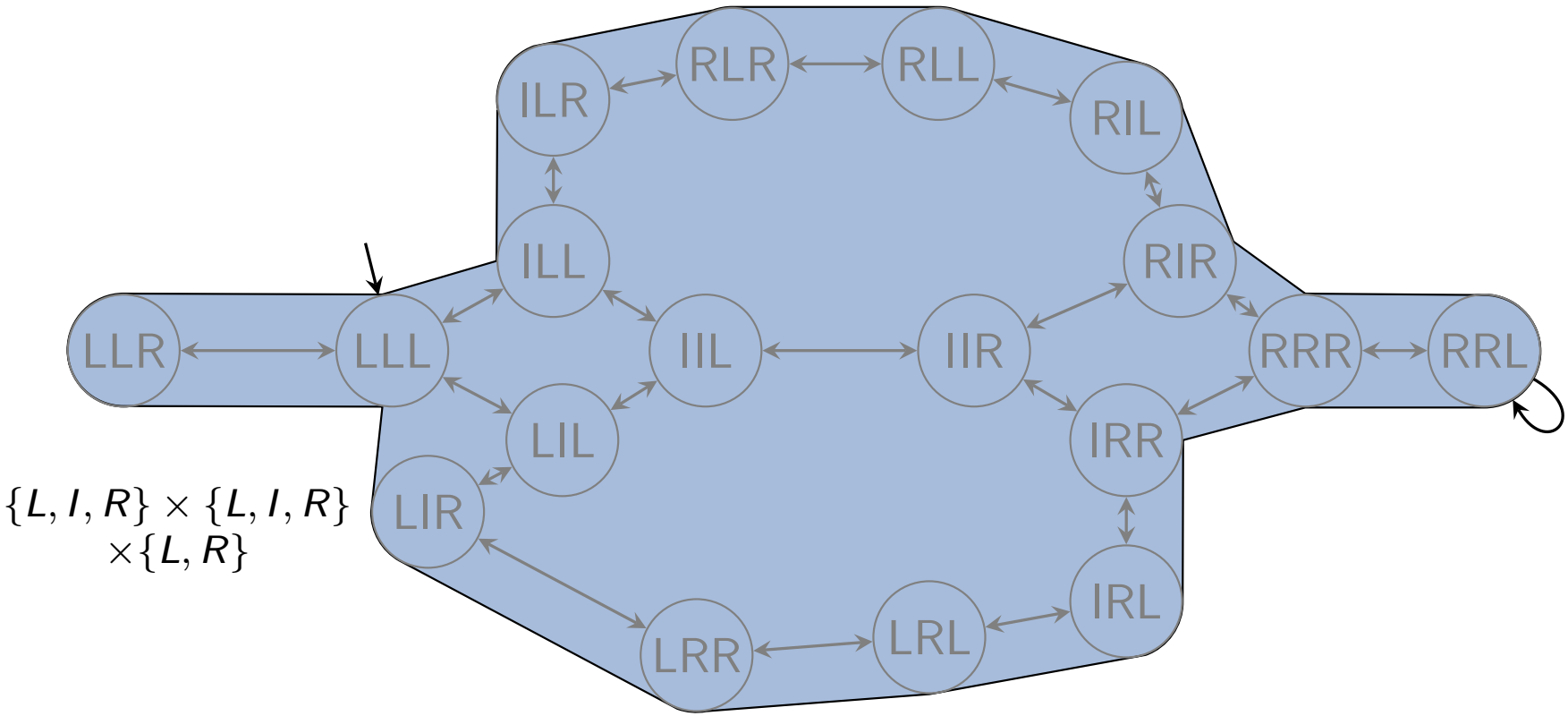
$\langle d, e \rangle := \text{Split}([s], s, c)$
$S'' := S' \setminus \{[s]\} \cup \{d, e\}$
$T'' := \text{RewireTransitions}(T', [s], d, e)$
if $[s] = s'_0$ then $s''_0 := d$ else $s''_0 := s'_0$
if $[s] \in S'_\star$ then $S''_\star := (S''_\star \setminus \{[s]\}) \cup \{e\}$ else $S''_\star := S'_\star$
return $\langle S'', L', c', T'', s''_0, S''_\star \rangle$

Update abstract initial state and goal states.

The way we defined the flaws, e can never be the abstract initial state and d never be an abstract goal state.

E14.4 Example

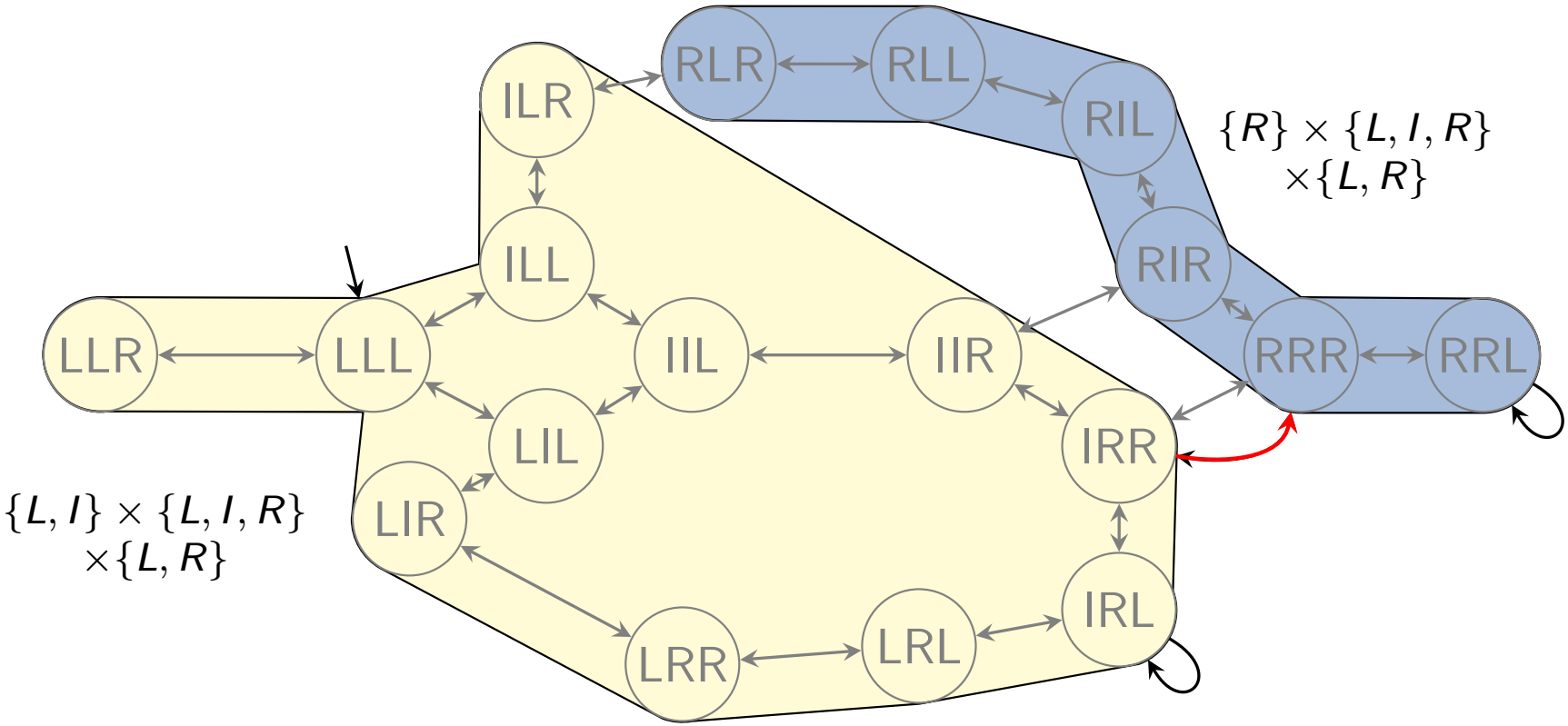
Example: Two Packages, One Truck



Abstract plan $\langle\rangle$ ends in state LLL, which is not a goal.
Refine $\{L, I, R\} \times \{L, I, R\} \times \{L, R\}$ with split $(LLL, \{R\} \times \{R\} \times \{L, R\})$.

$\rightsquigarrow$ split on first or second variable;
$\rightsquigarrow \{L, I\} \times \{L, I, R\} \times \{L, R\}$ and $\{R\} \times \{L, I, R\} \times \{L, R\}$

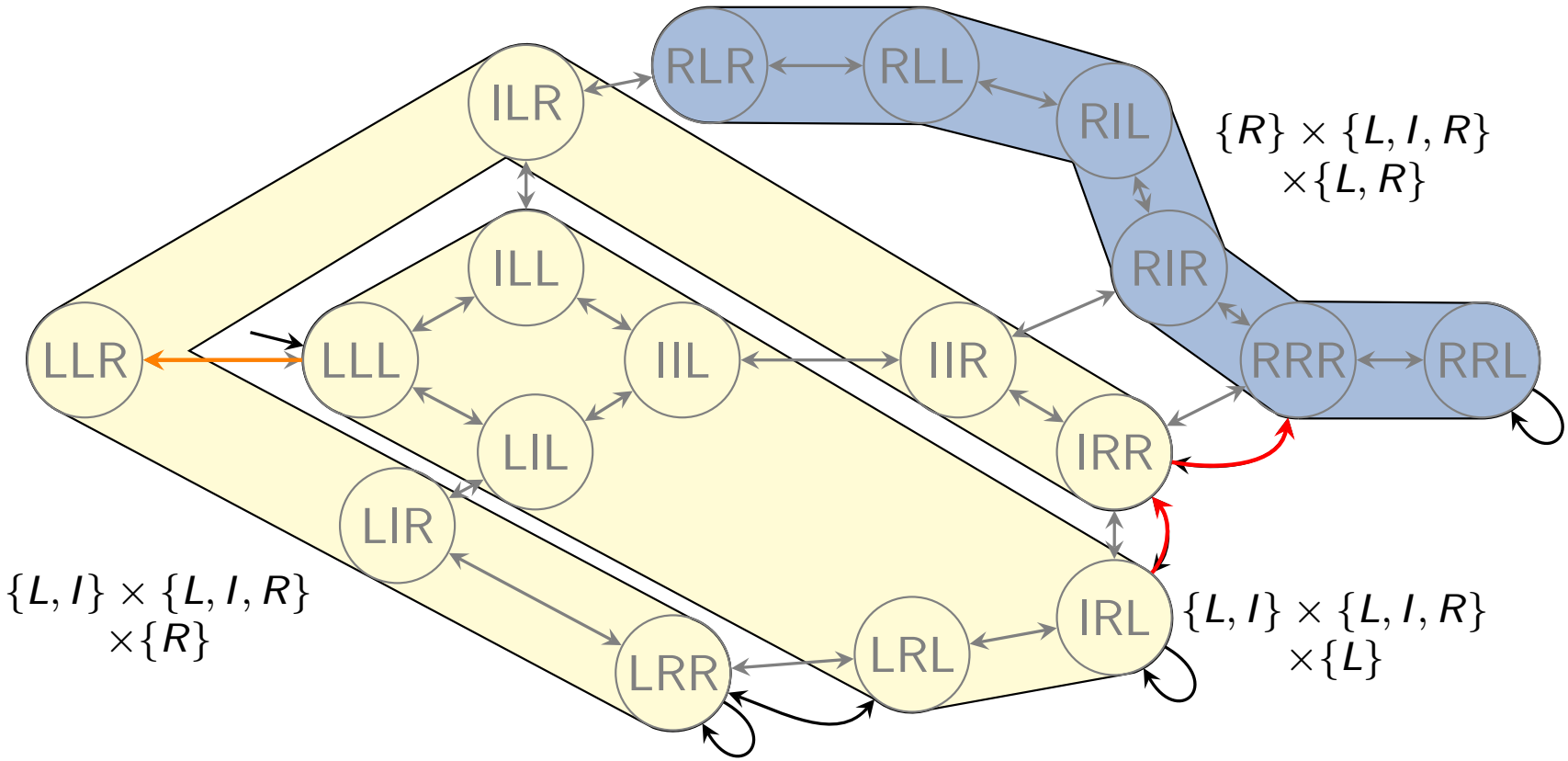
Example: Two Packages, One Truck



Abstract plan $\langle \text{drop}_{A,R} \rangle$; first action inapplicable in LLL.
Refine $\{L, I\} \times \{L, I, R\} \times \{L, R\}$ with split $(LLL, \{I\} \times \{L, I, R\} \times \{R\})$.

$\rightsquigarrow$ split on first or third variable;
$\rightsquigarrow \{L, I\} \times \{L, I, R\} \times \{L\}$ and $\{L, I\} \times \{L, I, R\} \times \{R\}$

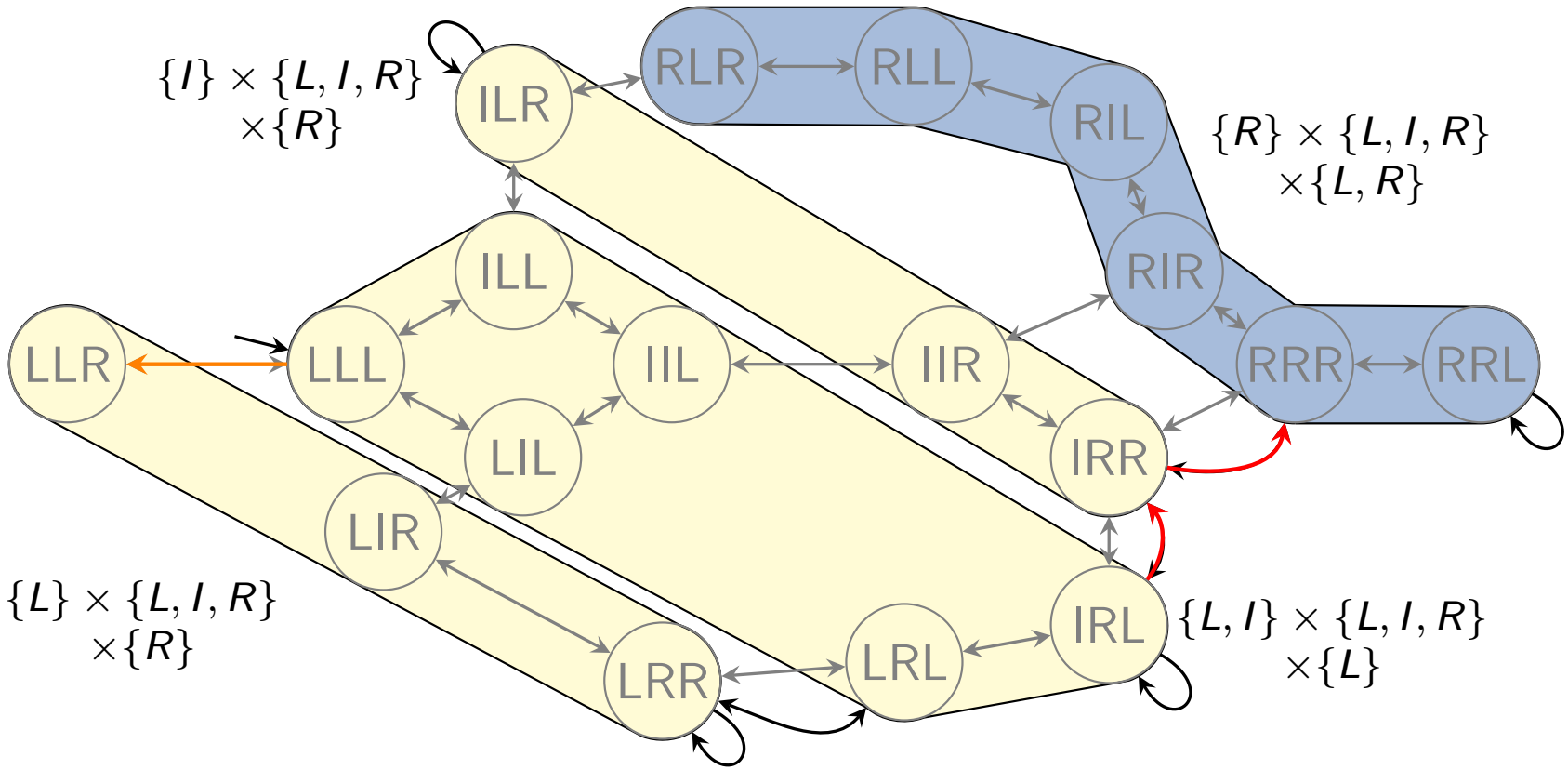
Example: Two Packages, One Truck



Abstract plan $\langle \text{move}_{L,R}, \text{drop}_{A,R} \rangle$; second action inapplicable in LLR.
Refine $\{L, I\} \times \{L, I, R\} \times \{R\}$ with split $(LLR, \{I\} \times \{L, I, R\} \times \{R\})$.

$\rightsquigarrow$ split on first variable;
$\rightsquigarrow \{L\} \times \{L, I, R\} \times \{R\}$ and $\{I\} \times \{L, I, R\} \times \{R\}$

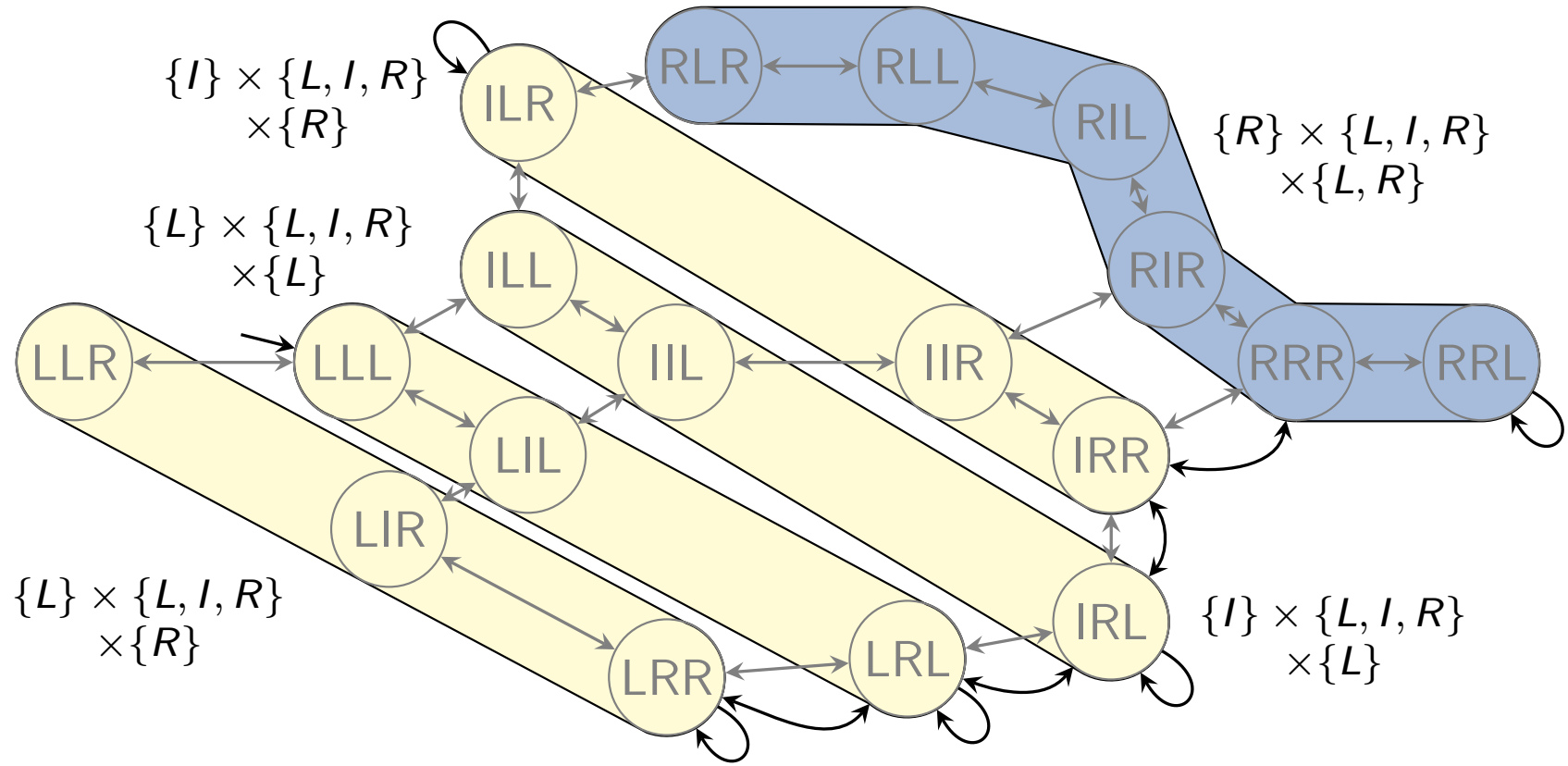
Example: Two Packages, One Truck



Abstract plan $\langle \text{move}_{L,R}, \text{drop}_{A,R} \rangle$; deviation flaw at first transition.
Refine $\{L, I\} \times \{L, I, R\} \times \{L\}$ with split $(LLL, \{I\} \times \{L, I, R\} \times \{L\})$.

$\rightsquigarrow$ split on first variable;
$\rightsquigarrow \{L\} \times \{L, I, R\} \times \{L\}$ and $\{I\} \times \{L, I, R\} \times \{L\}$

Example: Two Packages, One Truck



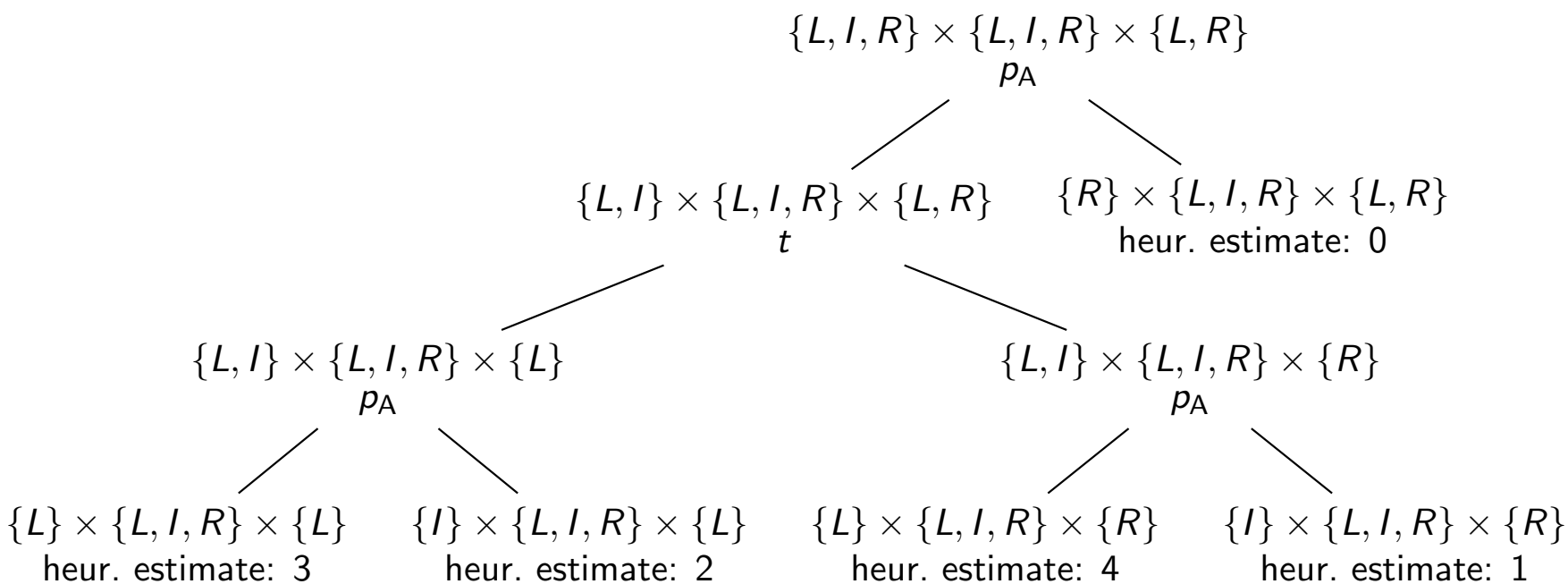
...

E14.5 Heuristic Representation

Representation

- In every iteration, we split one abstract state based on one variable.
- Represent abstraction as binary tree of abstract states.
  - Root: Single state of trivial abstraction
  - Leaves: Abstract states of final abstraction
- With each inner node, we store the variable on which the state was split.

Representation: Running Example



E14.6 Summary

Summary

Counterexample-guided abstraction refinement (CEGAR):

- Iteratively improve a coarse abstraction:
  - Find an optimal abstract solution.
  - Try it in the concrete transition system.
  - If it fails, extract a flaw and refine the abstraction.
- Flaws: unsatisfied precondition, unsatisfied goal, deviation.
- Refinement: split abstract state based on flaw to avoid repeating it.
- Can be efficiently implemented for Cartesian abstractions.
- Can stop at any time. The resulting heuristic is safe, goal-aware, admissible and consistent.

Planning and Optimization

F1. Constraints: Introduction

Malte Helmert and Gabriele Röger

Universität Basel

November 26, 2025

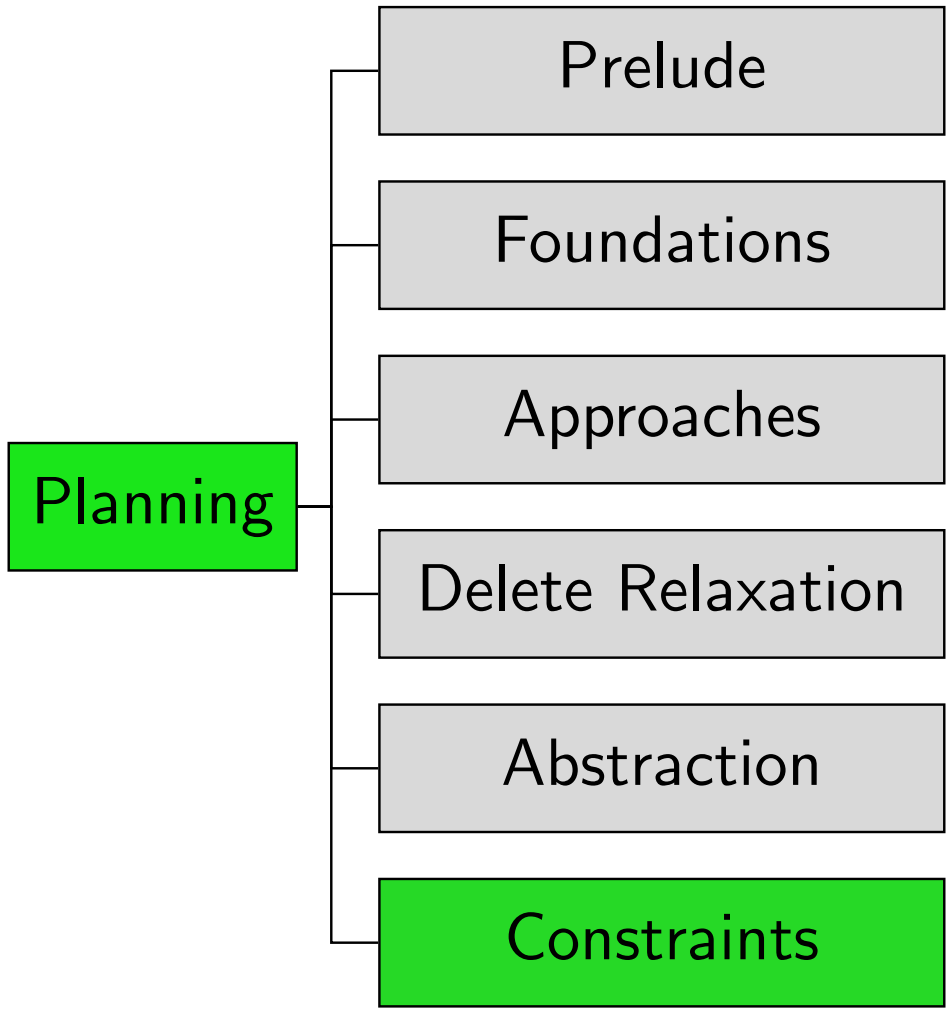
Planning and Optimization
November 26, 2025 — F1. Constraints: Introduction

F1.1 Constraint-based Heuristics

F1.2 Multiple Heuristics

F1.3 Summary

Content of the Course



F1.1 Constraint-based Heuristics

Coming Up with Heuristics in a Principled Way

General Procedure for Obtaining a Heuristic

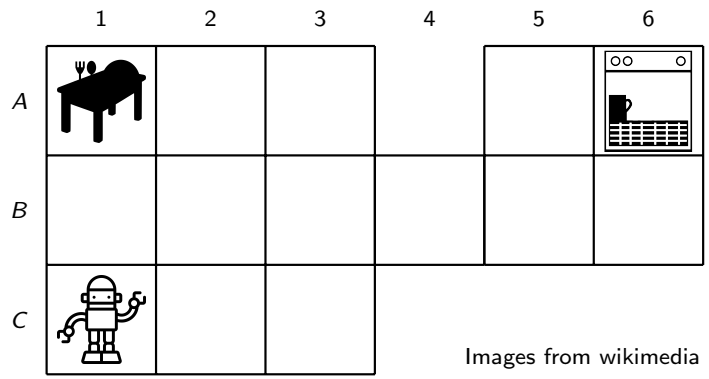
Solve a simplified version of the problem.

Major ideas for heuristics in the planning literature:

- delete relaxation
- abstraction
- critical paths
- landmarks
- network flows
- potential heuristic

Landmarks, network flows and potential heuristics are based on constraints that can be specified for a planning task.

Constraints: Example



Images from wikimedia

FDR planning task $\langle V, I, O, \gamma \rangle$ with

- $V = \{\mathit{robot\text{-}at}, \mathit{dishes\text{-}at}\}$ with
  - $\mathrm{dom}(\mathit{robot\text{-}at}) = \{\mathrm{A1}, \ldots, \mathrm{C3}, \mathrm{B4}, \mathrm{A5}, \ldots, \mathrm{B6}\}$
  - $\mathrm{dom}(\mathit{dishes\text{-}at}) = \{\mathrm{Table}, \mathrm{Robot}, \mathrm{Dishwasher}\}$
- $I = \{\mathit{robot\text{-}at} \mapsto \mathrm{C1}, \mathit{dishes\text{-}at} \mapsto \mathrm{Table}\}$
- operators
  - move-x-y to move from cell x to adjacent cell y
  - pickup dishes, and
  - load dishes into the dishwasher.
- $\gamma = (\mathit{robot\text{-}at} = B6) \wedge (\mathit{dishes\text{-}at} = \mathrm{Dishwasher})$

Constraints

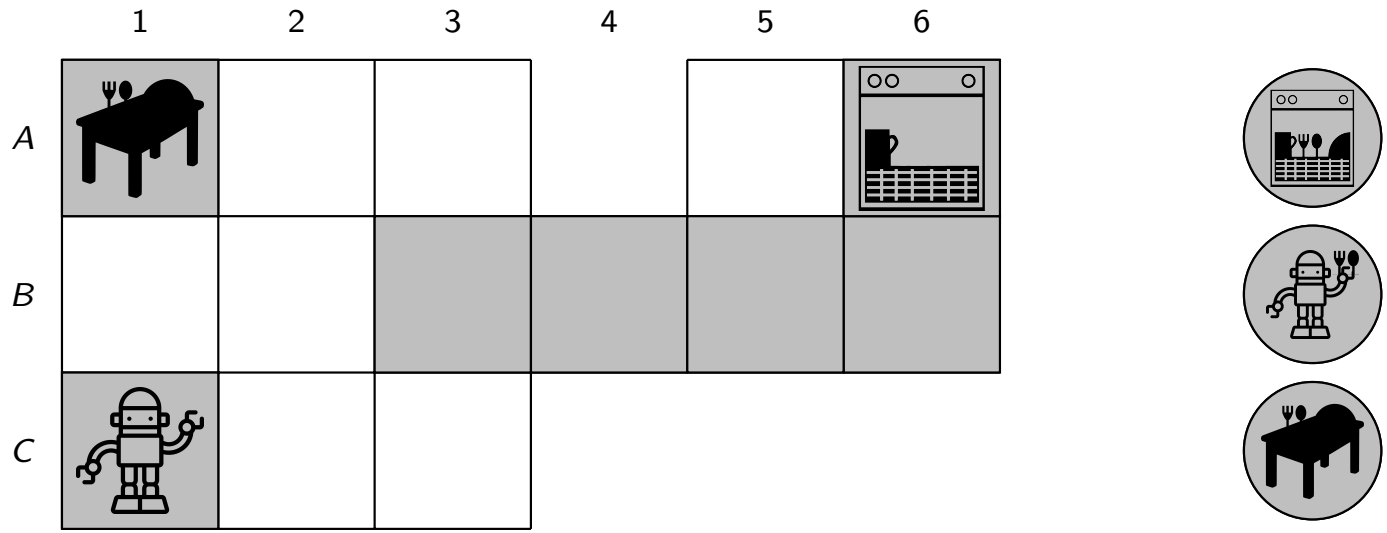
Some heuristics exploit constraints that describe something that holds in every solution of the task.

For instance, every solution is such that

- a variable takes a certain value in at least one visited state. (a fact landmark constraint)

Fact Landmarks: Example

Which values do *robot-at* and *dishes-at* take in every solution?



- `robot-at` = C1, `dishes-at` = Table (initial state)
- `robot-at` = B6, `dishes-at` = Dishwasher (goal state)
- `robot-at` = A1, `robot-at` = B3, `robot-at` = B4, `robot-at` = B5, `robot-at` = A6, `dishes-at` = Robot

Constraints

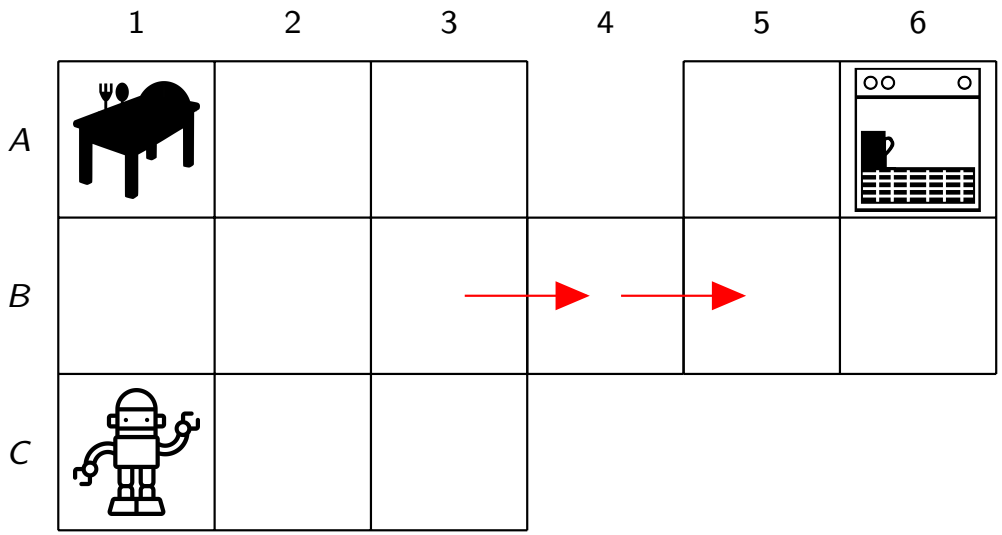
Some heuristics exploit constraints that describe something that holds in every solution of the task.

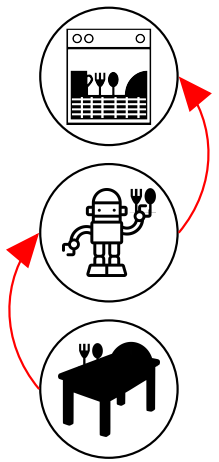
For instance, every solution is such that

- a variable takes some value in at least one visited state.
  (a fact landmark constraint)
- an action must be applied.
  (an action landmark constraint)

Action Landmarks: Example

Which actions must be applied in every solution?





- pickup
- load
- move-B3-B4
- move-B4-B5

Constraints

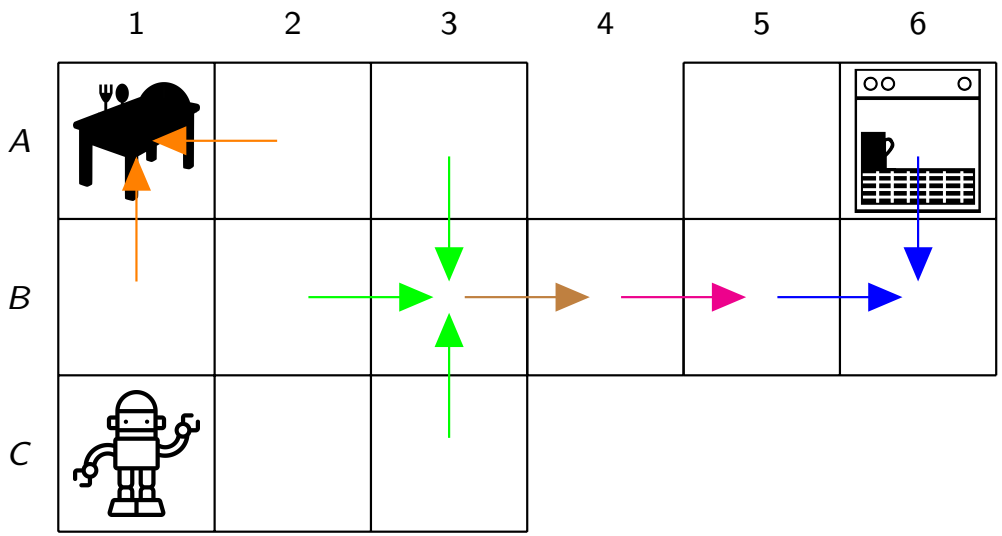
Some heuristics exploit constraints that describe something that holds in every solution of the task.

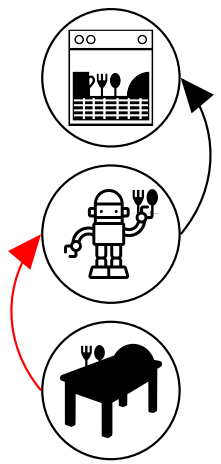
For instance, every solution is such that

- a variable takes some value in at least one visited state.
  (a fact landmark constraint)
- an action must be applied.
  (an action landmark constraint)
- at least one action from a set of actions must be applied.
  (a disjunctive action landmark constraint)

Disjunctive Action Landmarks: Example

Which set of actions is such that at least one must be applied?





- {pickup}
- {load}
- {move-B3-B4}
- {move-B4-B5}
- {move-A6-B6, move-B5-B6}
- {move-A3-B3, move-B2-B3, move-C3-B3}
- {move-B1-A1, move-A2-A1}
- ...

Constraints

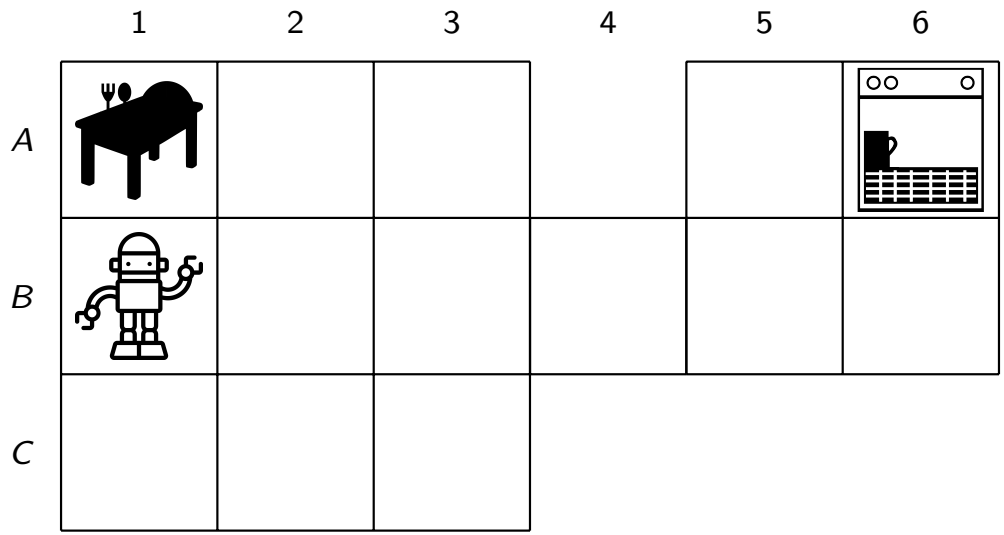
Some heuristics exploit constraints that describe something that holds in every solution of the task.

For instance, every solution is such that

- a variable takes some value in at least one visited state.
  (a fact landmark constraint)
- at least one action from a set of actions must be applied.
  (a disjunctive action landmark constraint)
- fact consumption and production is "balanced".
  (a network flow constraint)

Network Flow: Example

Consider the fact robot-at $= B1$.
How often are actions used that enter this cell?



Answer: as often as actions that leave this cell

If Count_o denotes how often operator o is applied, we have:

$$\text{Count}_{\text{move-A1-B1}} + \text{Count}_{\text{move-B2-B1}} + \text{Count}_{\text{move-C1-B1}} =$$
$$\text{Count}_{\text{move-B1-A1}} + \text{Count}_{\text{move-B1-B2}} + \text{Count}_{\text{move-B1-C1}}$$

F1.2 Multiple Heuristics

Combining Admissible Heuristics Admissibly

Major ideas to combine heuristics admissibly:

- maximize
- canoncial heuristic (for abstractions)
- minimum hitting set (for landmarks)
- cost partitioning
- operator counting

Often computed as solution to a (integer) linear program.

Combining Heuristics Admissibly: Example

Example

Consider an FDR planning task $\langle V, I, \{o_1, o_2, o_3, o_4\}, \gamma \rangle$ with $V = \{v_1, v_2, v_3\}$ with $dom(v_1) = \{A, B\}$ and $\mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \{A, B, C\}$, $I = \{v_1 \mapsto A, v_2 \mapsto A, v_3 \mapsto A\}$,

$$o_1 = \langle v_1 = \mathrm{A}, v_1 := \mathrm{B}, 1 \rangle$$
$$o_2 = \langle v_2 = \mathrm{A} \wedge v_3 = \mathrm{A}, v_2 := \mathrm{B} \wedge v_3 := \mathrm{B}, 1 \rangle$$
$$o_3 = \langle v_2 = \mathrm{B}, v_2 := \mathrm{C}, 1 \rangle$$
$$o_4 = \langle v_3 = \mathrm{B}, v_3 := \mathrm{C}, 1 \rangle$$

and $\gamma = (v_1 = \mathrm{B}) \wedge (v_2 = \mathrm{C}) \wedge (v_3 = \mathrm{C})$.

Let $\mathcal{C}$ be the pattern collection that contains all atomic projections. What is the canonical heuristic function $h^{\mathcal{C}}$?

Answer: Let $h_i := h^{v_i}$. Then $h^{\mathcal{C}} = \max\{h_1 + h_2, h_1 + h_3\}$.

Reminder: Orthogonality and Additivity

Why can we add h_1 and h_2 (h_1 and h_3) admissibly?

Theorem (Additivity for Orthogonal Abstractions)

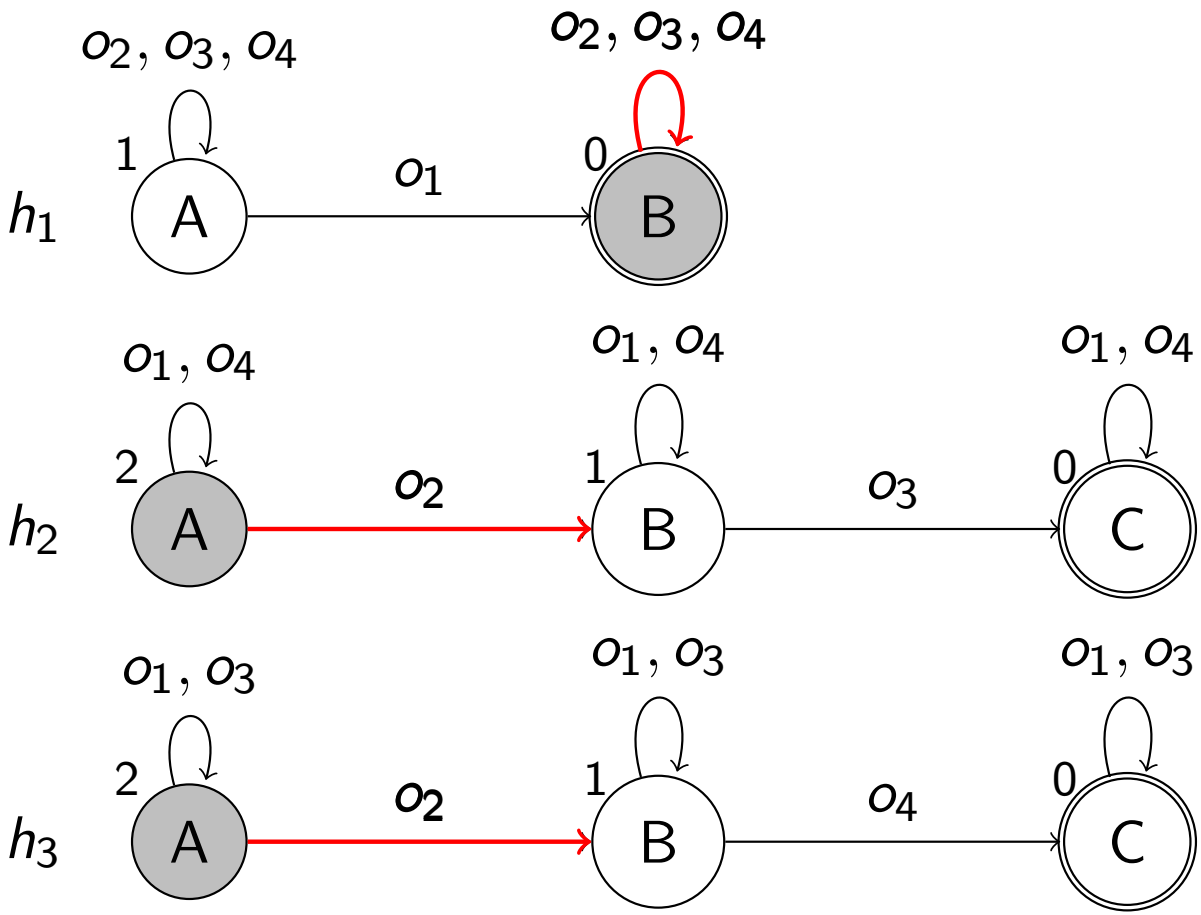
Let $h^{\alpha_1}, \ldots, h^{\alpha_n}$ be abstraction heuristics of the same transition system such that α_i and α_j are orthogonal for all $i \neq j$.

Then $\sum_{i=1}^{n} h^{\alpha_i}$ is a safe, goal-aware, admissible and consistent heuristic for Π.

The proof exploits that every concrete transition induces state-changing transition in at most one abstraction.

Combining Heuristics (In)admissibly: Example

Let $h = h_1 + h_2 + h_3$.



$\langle o_2, o_3, o_4 \rangle$ is a plan for $s = \langle B, A, A \rangle$ but $h(s) = 4$.
Heuristics h_2 and h_3 both account for the single application of o_2.

Prevent Inadmissibility

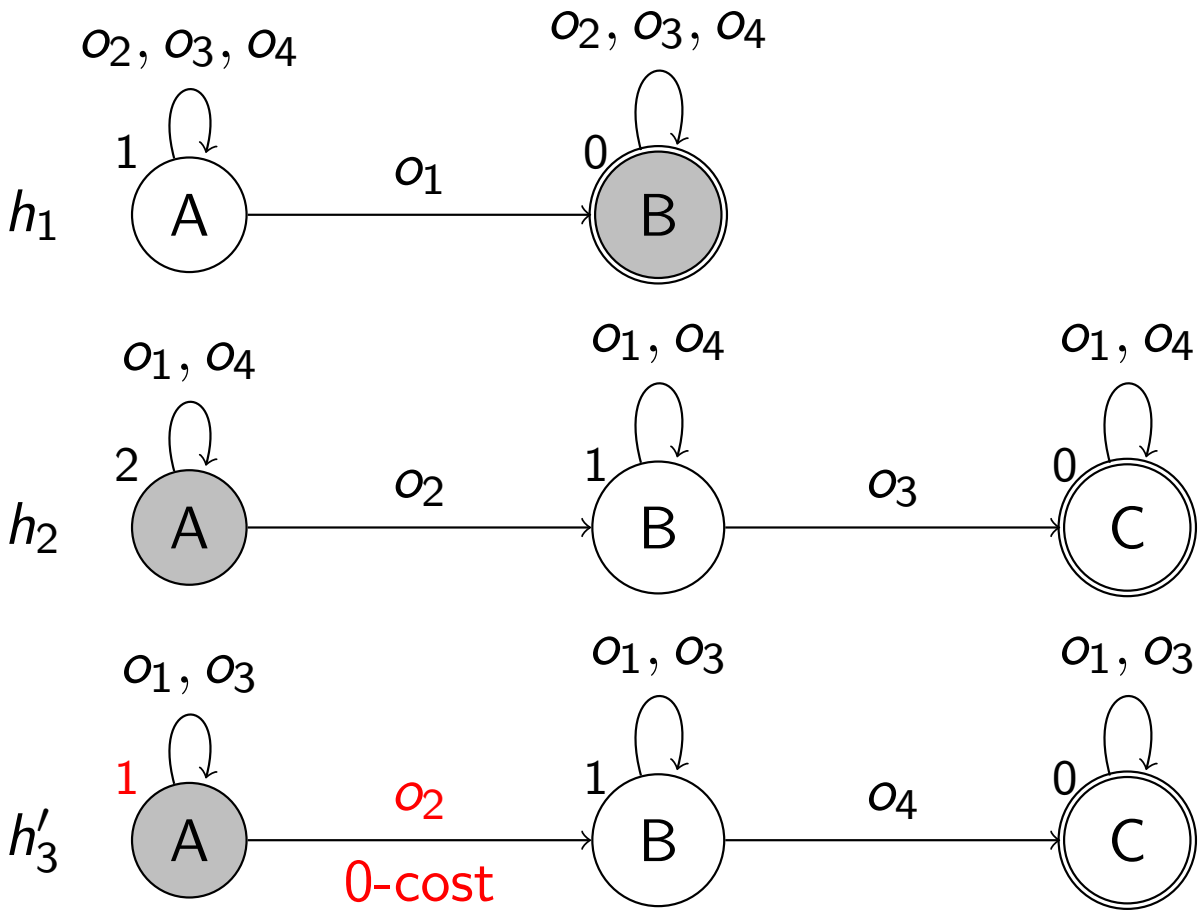
The reason that h_2 and h_3 are not additive is because the cost of o_2 is considered in both.

Is there anything we can do about this?

Solution: We can ignore the cost of o_2 in one heuristic by setting its cost to 0 (e.g., $cost_3(o_2) = 0$).

Combining Heuristics Admissibly: Example

Let $h' = h_1 + h_2 + h'_3$, where $h'_3 = h^{v_3}$ assuming $cost_3(o_2) = 0$.



$\langle o_2, o_3, o_4 \rangle$ is an optimal plan for $s = \langle B, A, A \rangle$ and $h'(s) = 3$ an admissible estimate.

Cost partitioning

Using the cost of every operator only in one heuristic is called a zero-one cost partitioning.

More generally, heuristics are additive if all operator costs are distributed in a way that the sum of the individual costs is no larger than the cost of the operator.

This can also be expressed as a constraint,
the cost partitioning constraint:

$$\sum_{i=1}^{n} cost_i(o) \leq cost(o) \text{ for all } o \in O$$

(more details later)

F1.3 Summary

Summary

- Landmarks and network flows are constraints that describe something that holds in every solution of the task.
- Heuristics can be combined admissibly if the cost partitioning constraint is satisfied.

Planning and Optimization

F2. Landmarks: RTG Landmarks

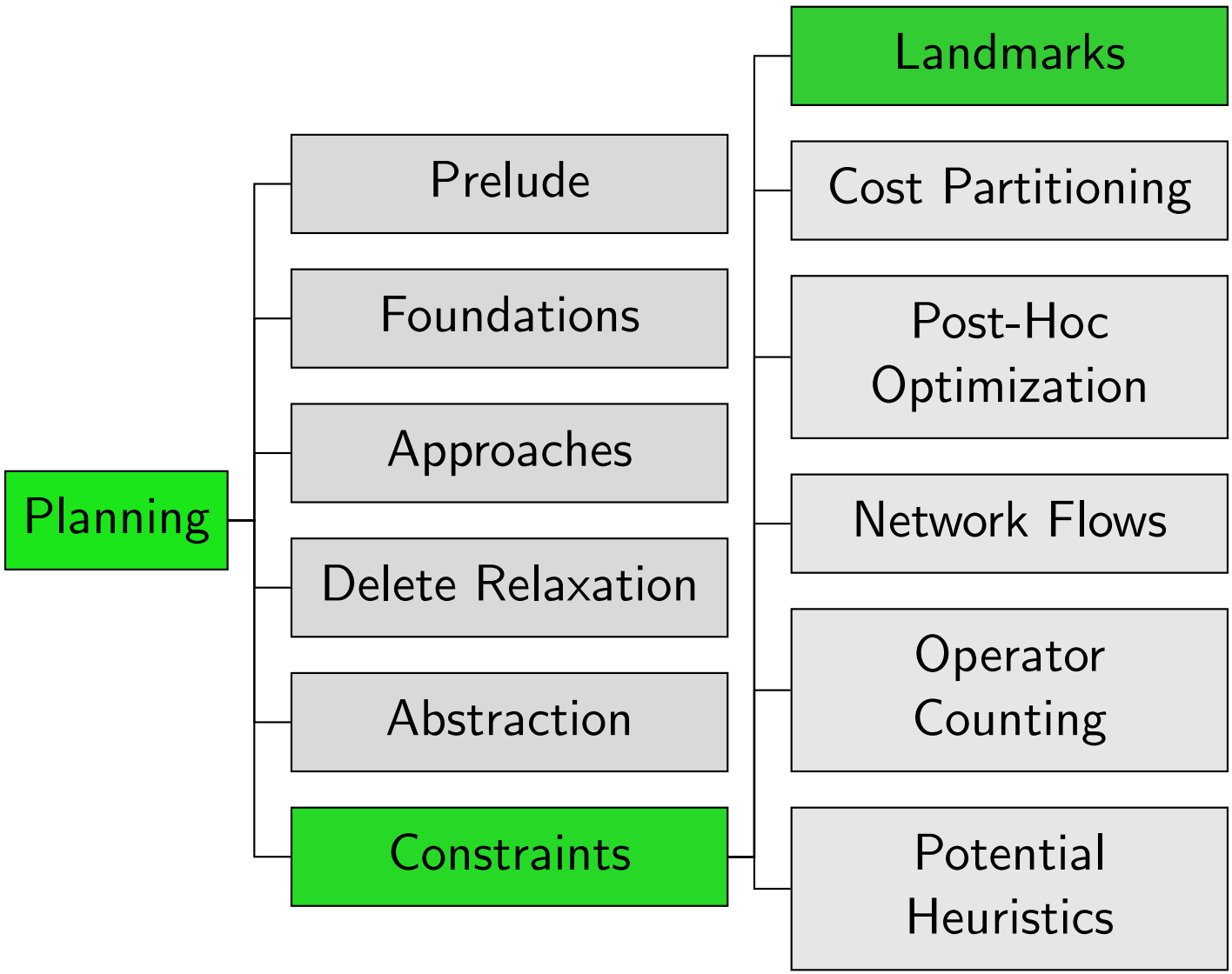
Malte Helmert and Gabriele Röger

Universität Basel

November 26, 2025

Planning and Optimization
November 26, 2025 — F2. Landmarks: RTG Landmarks

Content of the Course



F2.1 Landmarks

Landmarks

Basic Idea: Something that must happen in every solution

For example

- some operator must be applied (action landmark)
- some atomic proposition must hold (fact landmark)
- some formula must be true (formula landmark)

→ Derive heuristic estimate from this kind of information.

We mostly consider fact and disjunctive action landmarks.

Reminder: Terminology

Consider sequence of transitions $s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n$ such that $s^0 = s$ and $s^n = s'$.

- $s^0, \ldots, s^n$ is called (state) path from s to s'
- $\ell_1, \ldots, \ell_n$ is called (label) path from s to s'

Disjunctive Action Landmarks

Definition (Disjunctive Action Landmark)

Let s be a state of a propositional or FDR planning task $\Pi = \langle V, I, O, \gamma \rangle$.

A disjunctive action landmark for s is a set of operators $L \subseteq O$ such that every label path from s to a goal state contains an operator from L.
The cost of landmark L is $cost(L) = \min_{o \in L} cost(o)$.

If we talk about landmarks for the initial state, we omit "for I".

Fact and Formula Landmarks

Definition (Formula and Fact Landmark)

Let s be a state of a propositional or FDR planning task $\Pi = \langle V, I, O, \gamma \rangle$.

A formula landmark for s is a formula λ over V such that every state path from s to a goal state contains a state s' with $s' \models \lambda$.

If λ is an atomic proposition then λ is a fact landmark.

If we talk about landmarks for the initial state, we omit "for I".

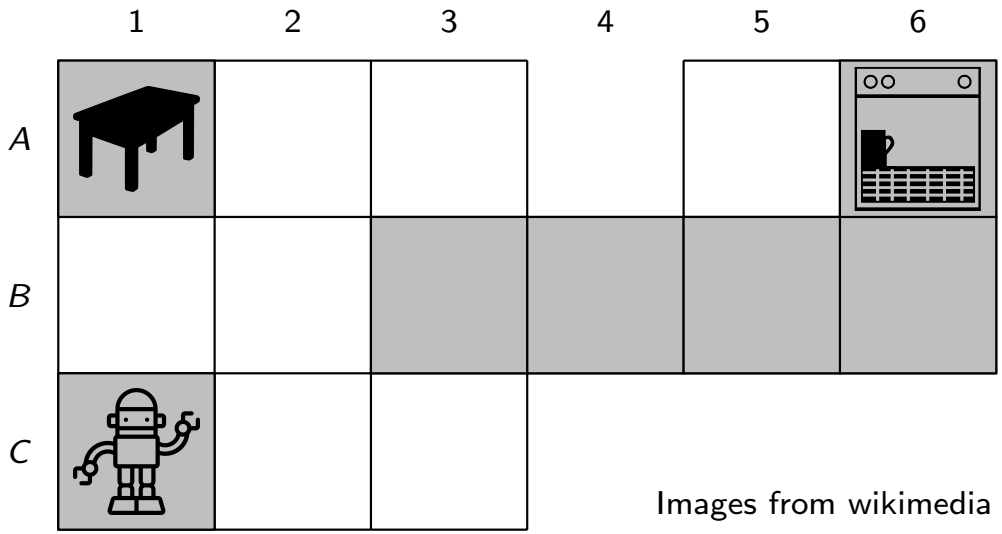
Landmarks: Example

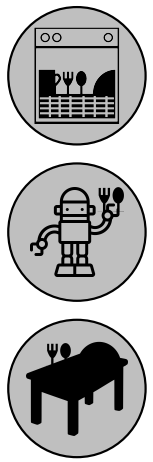
Example

Consider a FDR planning task $\langle V, I, O, \gamma \rangle$ with

- $V = \{\mathit{robot\text{-}at}, \mathit{dishes\text{-}at}\}$ with
  - $\mathrm{dom}(\mathit{robot\text{-}at}) = \{\mathrm{A1}, \ldots, \mathrm{C3}, \mathrm{B4}, \mathrm{A5}, \ldots, \mathrm{B6}\}$
  - $\mathrm{dom}(\mathit{dishes\text{-}at}) = \{\mathrm{Table}, \mathrm{Robot}, \mathrm{Dishwasher}\}$
- $I = \{\mathit{robot\text{-}at} \mapsto \mathrm{C1}, \mathit{dishes\text{-}at} \mapsto \mathrm{Table}\}$
- operators
  - move-x-y to move from cell x to adjacent cell y
  - pickup dishes, and
  - load dishes into the dishwasher.
- $\gamma = (\mathit{robot\text{-}at} = B6) \wedge (\mathit{dishes\text{-}at} = \mathrm{Dishwasher})$

Fact and Formula Landmarks: Example





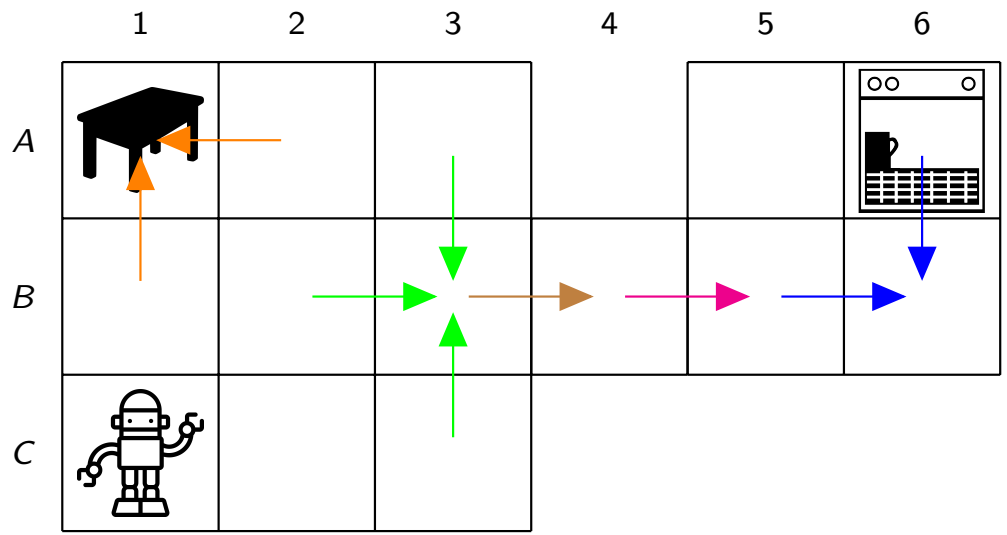
Each fact in gray is a fact landmark:

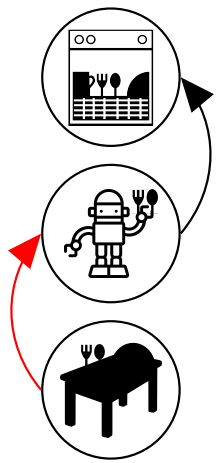
- $\texttt{robot-at} = x$ for $x \in \{\text{A1}, \text{A6}, \text{B3}, \text{B4}, \text{B5}, \text{B6}, \text{C1}\}$
- $\texttt{dishes-at} = x$ for $x \in \{\text{Dishwasher}, \text{Robot}, \text{Table}\}$

Formula landmarks:

- $\texttt{dishes-at} = \text{Robot} \wedge \texttt{robot-at} = \text{B4}$
- $\texttt{robot-at} = \text{B1} \vee \texttt{robot-at} = \text{A2}$

Disjunctive Action Landmarks: Example





Actions of same color form disjunctive action landmark:

- {pickup}
- {load}
- {move-B3-B4}
- {move-B4-B5}
- {move-A6-B6, move-B5-B6}
- {move-A3-B3, move-B2-B3, move-C3-B3}
- {move-B1-A1, move-A2-A1}
- ...

Remarks

- Not every landmark is informative. Some examples:
  - The set of all operators is a disjunctive action landmark unless the initial state is already a goal state.
  - Every variable that is initially true is a fact landmark.
  - The goal formula is a formula landmark.
- Every fact landmark v that is initially false induces a disjunctive action landmark consisting of all operators that possibly make v true.

Complexity: Disjunctive Action Landmarks

Theorem

Deciding whether a given operator set is a disjunctive action landmark is as hard as the plan existence problem.

Proof.

Given a propositional planning task $\Pi = \langle V, I, O, \gamma \rangle$, create a new planning task Π' with goal $g \notin V$ as $\Pi' = \langle V \cup \{g\}, I \cup \{g \mapsto \mathbf{F}\}, O \cup \{o_\gamma, o_\top\}, g \rangle$, where

$$o_\gamma = \langle \gamma, g, 0 \rangle, \text{ and}$$
$$o_\top = \langle \top, g, 0 \rangle.$$

If $\gamma = \top$ then Π is trivially solvable. Otherwise Π is solvable iff $\{o_\top\}$ is not a disjunctive action landmark of Π'. □

Complexity: Fact Landmarks

Theorem

Deciding whether a given atomic proposition is a fact landmark is as hard as the plan existence problem.

Proof.

Given a propositional planning task $\Pi = \langle V, I, O, \gamma \rangle$,
let $p, g \notin V$ be new atomic propositions and create a new planning task $\Pi' = \langle V \cup \{p, g\}, I \cup \{p \mapsto \mathbf{F}, g \mapsto \mathbf{F}\}, O \cup \{o, o'\}, g \rangle$, where

$$o = \langle \gamma, g, 0 \rangle, \text{ and}$$
$$o' = \langle \top, g \wedge p, 0 \rangle.$$

Then p is a fact landmark of Π' iff Π is not solvable. □

Complexity: Discussion

- Does this mean that the idea of exploiting landmarks is fruitless?– No!
- We do not need to know all landmarks, so we can use incomplete methods to identify landmarks.
  - The way we generate the landmarks guarantees that they are indeed landmarks.
  - Efficient landmark generation methods do not guarantee to generate all possible landmarks.

Computing Landmarks

How can we come up with landmarks?

Most landmarks are derived from the relaxed task graph:

- RHW landmarks: Richter, Helmert & Westphal. Landmarks Revisited. (AAAI 2008)
- LM-Cut: Helmert & Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? (ICAPS 2009)
- h^m landmarks: Keyder, Richter & Helmert: Sound and Complete Landmarks for And/Or Graphs (ECAI 2010)

Today we will discuss the special case of h^m landmarks for $m = 1$, restricted to STRIPS planning tasks.

F2.2 Set Representation

Set Representation of STRIPS Planning Tasks

In this (and the following) sections, we only consider STRIPS. For a more convenient notation, we will use a set representation of STRIPS planning task...

Three differences:

- Represent conjunctions of variables as sets of variables.
- Use two sets to represent add and delete effects of operators separately.
- Represent states as sets of the true variables.

STRIPS Operators in Set Representation

- Every STRIPS operator is of the form

$$\langle v_1 \wedge \cdots \wedge v_p, \quad a_1 \wedge \cdots \wedge a_q \wedge \neg d_1 \wedge \cdots \wedge \neg d_r, c \rangle$$

  where v_i, a_j, d_k are state variables and c is the cost.
- The same operator o in set representation is $\langle pre(o), add(o), del(o), cost(o) \rangle$, where
  - $pre(o) = \{v_1, \ldots, v_p\}$ are the preconditions,
  - $add(o) = \{a_1, \ldots, a_q\}$ are the add effects,
  - $del(o) = \{d_1, \ldots, d_r\}$ are the delete effects, and
  - $cost(o) = c$ is the operator cost.
- Since STRIPS operators must be conflict-free, $add(o) \cap del(o) = \emptyset$

STRIPS Planning Tasks in Set Representation

A STRIPS planning task in set representation is given as a tuple $\langle V, I, O, G \rangle$, where

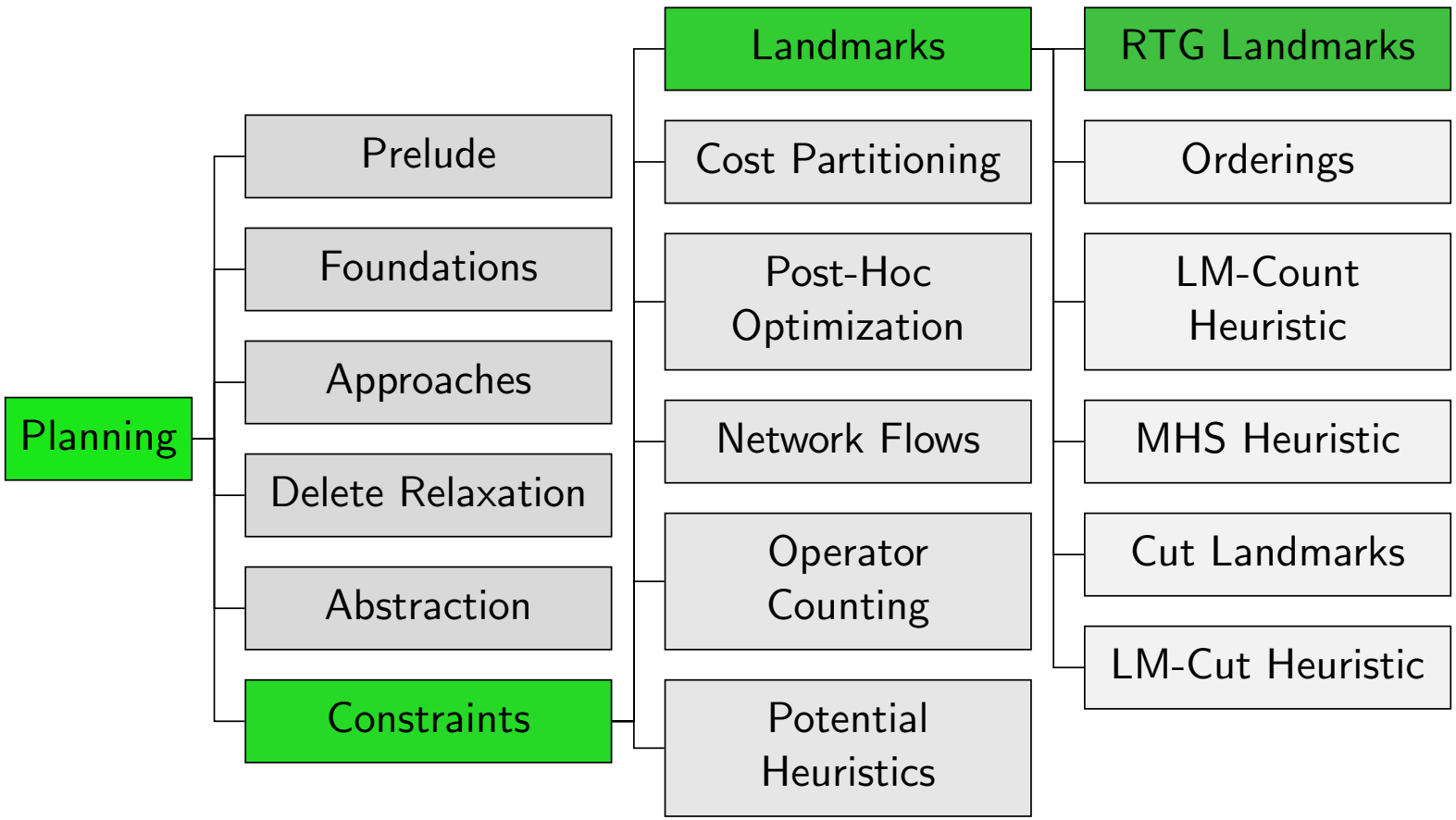
- V is a finite set of state variables,
- $I \subseteq V$ is the initial state,
- O is a finite set of STRIPS operators in set representation,
- $G \subseteq V$ is the goal.

The corresponding planning task in the previous notation is $\langle V, I', O', \gamma \rangle$, where

- $I'(v) = \mathbf{T}$ iff $v \in I$,
- $O' = \{\langle \bigwedge_{v \in pre(o)} v, \bigwedge_{v \in add(o)} v \wedge \bigwedge_{v \in del(o)} \neg v, cost(o) \rangle \mid o \in O\}$,
- $\gamma = \bigwedge_{v \in G} v$.

F2.3 Landmarks from RTGs

Content of the Course



Incidental Landmarks: Example

Example (Incidental Landmarks)

Consider a STRIPS planning task $\langle V, I, \{o_1, o_2\}, G \rangle$ with

$$
\begin{aligned}
V &= \{a, b, c, d, e, f\}, \\
I &= \{a, b, e\}, \\
o_1 &= \langle \{a\}, \{c, d, e\}, \{b\} \rangle, \\
o_2 &= \langle \{d, e\}, \{f\}, \{a\} \rangle, \text{ and} \\
G &= \{e, f\}.
\end{aligned}
$$

Single solution: $\langle o_1, o_2 \rangle$

- All variables are fact landmarks.
- Variable b is initially true but irrelevant for the plan.
- Variable c gets true as "side effect" of o_1 but it is not necessary for the goal or to make an operator applicable.

Causal Landmarks (1)

Definition (Causal Formula Landmark)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a propositional or FDR planning task.

A formula λ over V is a causal formula landmark for I if $\gamma \models \lambda$ or if for all plans $\pi = \langle o_1, \ldots, o_n \rangle$ there is an o_i with $pre(o_i) \models \lambda$.

Causal Landmarks (2)

Special case: Fact Landmark for STRIPS task

Definition (Causal Fact Landmark)

Let $\Pi = \langle V, I, O, G \rangle$ be a STRIPS planning task (in set representation).

A variable $v \in V$ is a causal fact landmark for I

- if $v \in G$ or
- if for all plans $\pi = \langle o_1, \ldots, o_n \rangle$ there is an o_i with $v \in pre(o_i)$.

Causal Landmarks: Example

Example (Causal Landmarks)

Consider a STRIPS planning task $\langle V, I, \{o_1, o_2\}, G \rangle$ with

$$
\begin{aligned}
V &= \{a, b, c, d, e, f\}, \\
I &= \{a, b, e\}, \\
o_1 &= \langle \{a\}, \{c, d, e\}, \{b\} \rangle, \\
o_2 &= \langle \{d, e\}, \{f\}, \{a\} \rangle, \text{ and} \\
G &= \{e, f\}.
\end{aligned}
$$

Single solution: $\langle o_1, o_2 \rangle$

- All variables are fact landmarks for the initial state.
- Only a, d, e and f are causal landmarks.

What We Are Doing Next

- Causal landmarks are the desirable landmarks.
- We can use a simplified version of RTGs for STRIPS to compute causal landmarks for STRIPS planning tasks.
- We will define landmarks of AND/OR graphs, ...
- and show how they can be computed.
- Afterwards we establish that these are landmarks of the planning task.

Simplified Relaxed Task Graph

Definition

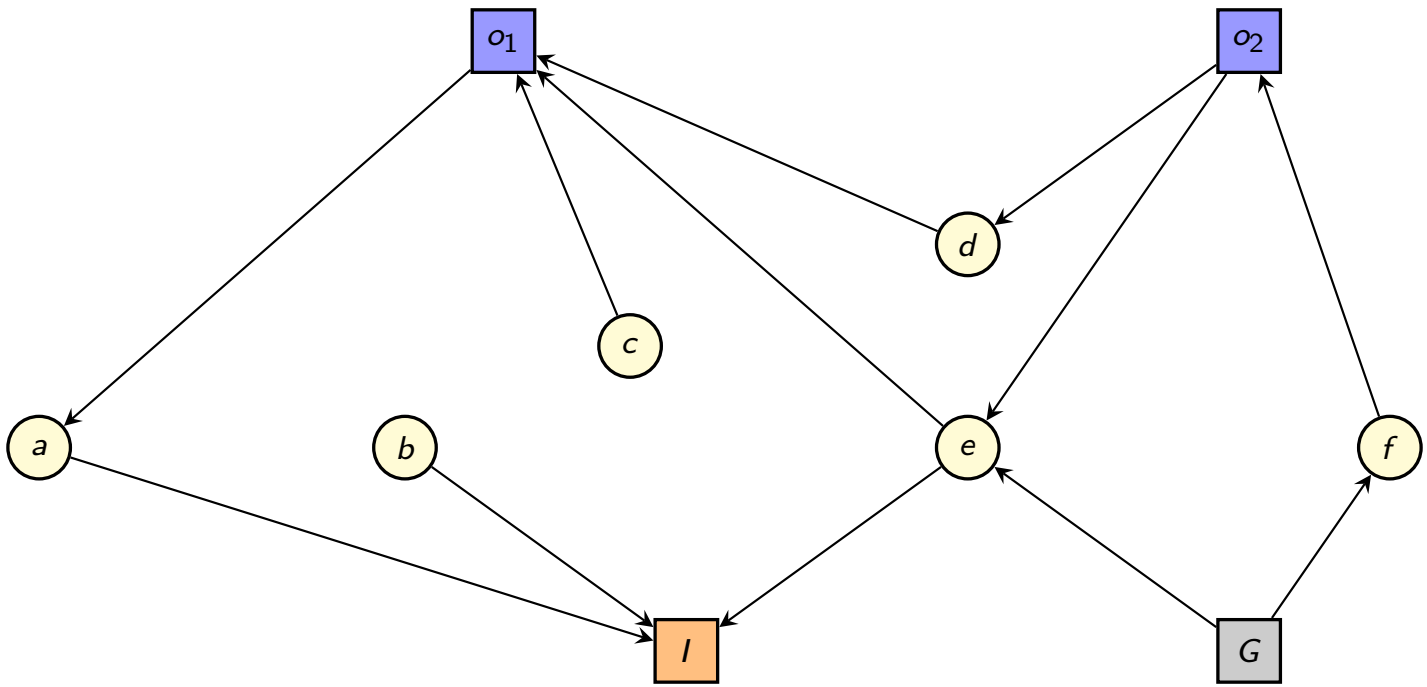
For a STRIPS planning task $\Pi = \langle V, I, O, G \rangle$ (in set representation), the simplified relaxed task graph $sRTG(\Pi^+)$ is the AND/OR graph $\langle N_{\text{and}} \cup N_{\text{or}}, A, type \rangle$ with

- $N_{\text{and}} = \{n_o \mid o \in O\} \cup \{v_I, v_G\}$ with $type(n) = \wedge$ for all $n \in N_{\text{and}}$,
- $N_{\text{or}} = \{n_v \mid v \in V\}$ with $type(n) = \vee$ for all $n \in N_{\text{or}}$, and
- $A = \{\langle n_a, n_o \rangle \mid o \in O, a \in add(o)\} \cup$
  $\{\langle n_o, n_p \rangle \mid o \in O, p \in pre(o)\} \cup$
  $\{\langle n_v, n_I \rangle \mid v \in I\} \cup$
  $\{\langle n_G, n_v \rangle \mid v \in G\}$

Like RTG but without extra nodes to support arbitrary conditions.

Simplified RTG: Example

The simplified RTG for our example task is:



Justification

Definition (Justification)

Let $G = \langle N, A, type \rangle$ be an AND/OR graph.

A subgraph $J = \langle N^J, A^J, type^J \rangle$ with $N^J \subseteq N$ and $A^J \subseteq A$ and $type^J = type|_{N^J}$ justifies $n_\star \in N$ iff

- $n_\star \in N^J$,
- $\forall n \in N^J$ with $type(n) = \wedge$:
  $\forall \langle n, n' \rangle \in A : n' \in N^J$ and $\langle n, n' \rangle \in A^J$
- $\forall n \in N^J$ with $type(n) = \vee$:
  $\exists \langle n, n' \rangle \in A : n' \in N^J$ and $\langle n, n' \rangle \in A^J$, and
- J is acyclic.

"Proves" that $n_\star$ is forced true.

Landmarks in AND/OR Graphs

Definition (Landmarks in AND/OR Graphs)

Let $G = \langle N, A, \mathit{type} \rangle$ be an AND/OR graph.

A node $n \in N$ is a landmark for reaching $n_\star \in N$
if $n \in V^J$ for all justifications J for $n_\star$.

But: exponential number of possible justifications

Characterizing Equation System

Theorem

Let $G = \langle N, A, type \rangle$ be an AND/OR graph. Consider the following system of equations:

$$LM(n) = \{n\} \cup \bigcap_{\langle n,n' \rangle \in A} LM(n') \quad type(n) = \vee$$

$$LM(n) = \{n\} \cup \bigcup_{\langle n,n' \rangle \in A} LM(n') \quad type(n) = \wedge$$

The equation system has a unique maximal solution (maximal with regard to set inclusion), and for this solution it holds that

$n' \in LM(n)$ *iff* n' *is a landmark for reaching* n *in* G.

Computation of Maximal Solution

Theorem

Let $G = \langle N, A, type \rangle$ be an AND/OR graph. Consider the following system of equations:

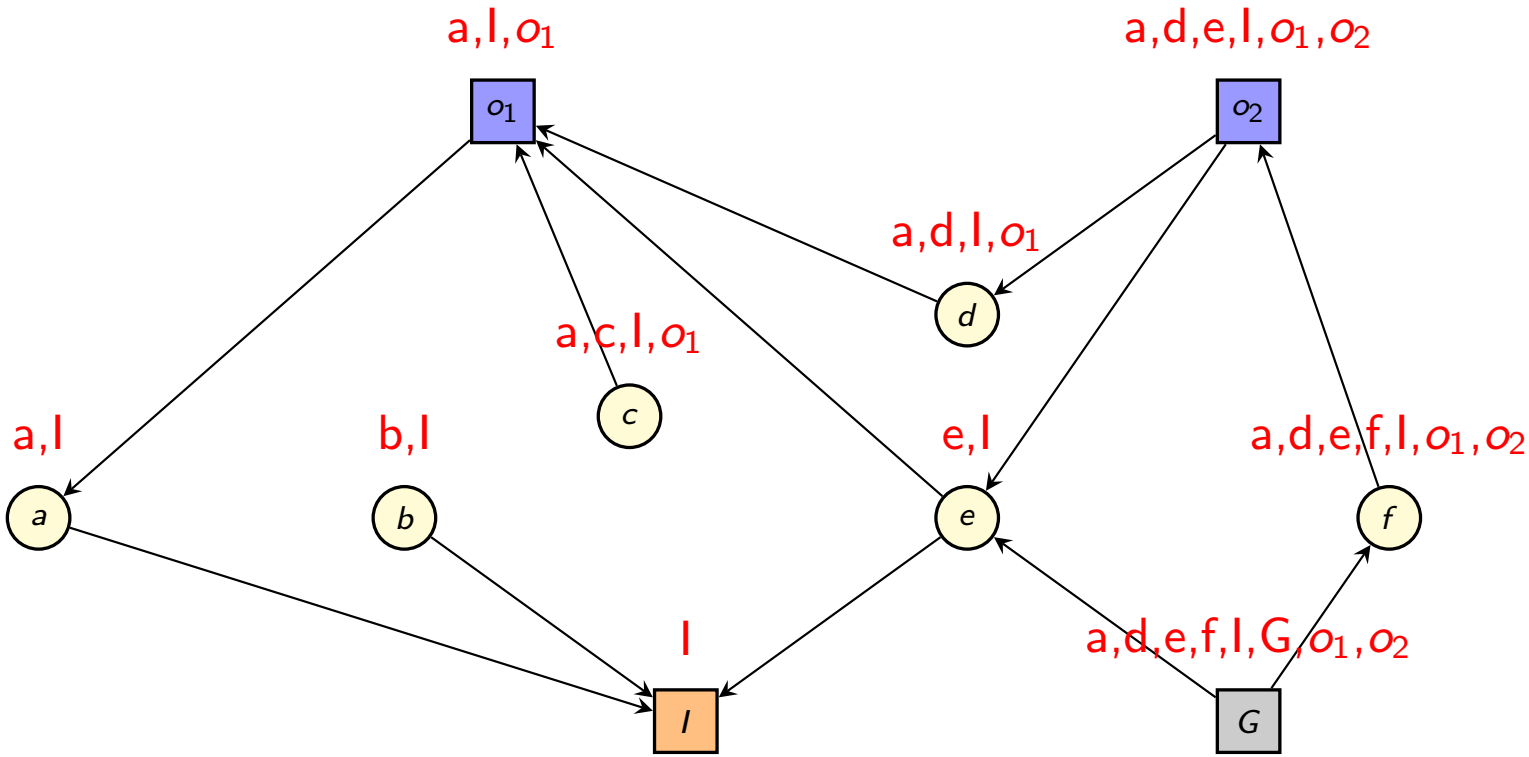
$$LM(n) = \{n\} \cup \bigcap_{\langle n,n' \rangle \in A} LM(n') \quad type(n) = \vee$$

$$LM(n) = \{n\} \cup \bigcup_{\langle n,n' \rangle \in A} LM(n') \quad type(n) = \wedge$$

The equation system has a unique maximal solution (maximal with regard to set inclusion).

Computation: Initialize landmark sets as $LM(n) = N$ and apply equations as update rules until fixpoint.

Computation: Example



(cf. screen version of slides for step-wise computation)

Relation to Planning Task Landmarks

Theorem

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a STRIPS planning task and let $\mathcal{L}$ be the set of landmarks for reaching n_G in sRTG(Π^+).

The set $\{v \in V \mid n_v \in \mathcal{L}\}$ is exactly the set of causal fact landmarks in Π^+.

For operators $o \in O$, if $n_o \in \mathcal{L}$ then $\{o\}$ is a disjunctive action landmark in Π^+.
There are no other disjunctive action landmarks of size 1.

(Proofs omitted.)

Computed RTG Landmarks: Example

Example (Computed RTG Landmarks)

Consider a STRIPS planning task $\langle V, I, \{o_1, o_2\}, G \rangle$ with

$$
\begin{aligned}
V &= \{a, b, c, d, e, f\}, \\
I &= \{a, b, e\}, \\
o_1 &= \langle \{a\}, \{c, d, e\}, \{b\} \rangle, \\
o_2 &= \langle \{d, e\}, \{f\}, \{a\} \rangle, \text{ and} \\
G &= \{e, f\}.
\end{aligned}
$$

- $LM(n_G) = \{a, d, e, f, I, G, o_1, o_2\}$
- a, d, e, and f are causal fact landmarks of Π^+.
- $\{o_1\}$ and $\{o_2\}$ are disjunctive action landmarks of Π^+.

(Some) Landmarks of Π^+ Are Landmarks of Π

Theorem

Let Π be a STRIPS planning task.

All fact landmarks of Π^+ are fact landmarks of Π and all disjunctive action landmarks of Π^+ are disjunctive action landmarks of Π.

Proof.

Let L be a disjunctive action landmark of Π^+ and π be a plan for Π. Then π is also a plan for Π^+ and, thus, π contains an operator from L.

Let f be a fact landmark of Π^+. If f is already true in the initial state, then it is also a landmark of Π. Otherwise, every plan for Π^+ contains an operator that adds f and the set of all these operators is a disjunctive action landmark of Π^+. Therefore, also each plan of Π contains such an operator, making f a fact landmark of Π. □

Not All Landmarks of Π^+ are Landmarks of Π

Example

Consider STRIPS task $\langle\{a, b, c\}, \emptyset, \{o_1, o_2\}, \{c\}\rangle$ with $o_1 = \langle\{\}, \{a\}, \{\}, 1\rangle$ and $o_2 = \langle\{a\}, \{c\}, \{a\}, 1\rangle$.

$a \wedge c$ is a formula landmark of Π^+ but not of Π.

F2.4 Summary

Summary

- Fact landmark: atomic proposition that is true in each state path to a goal
- Disjunctive action landmark: set L of operators such that every plan uses some operator from L
- We can efficiently compute all causal fact landmarks of a delete-free STRIPS task from the (simplified) RTG.
- Fact landmarks of the delete relaxed task are also landmarks of the original task.

Planning and Optimization

F3. Landmarks: Orderings & LM-Count Heuristic

Malte Helmert and Gabriele Röger

Universität Basel

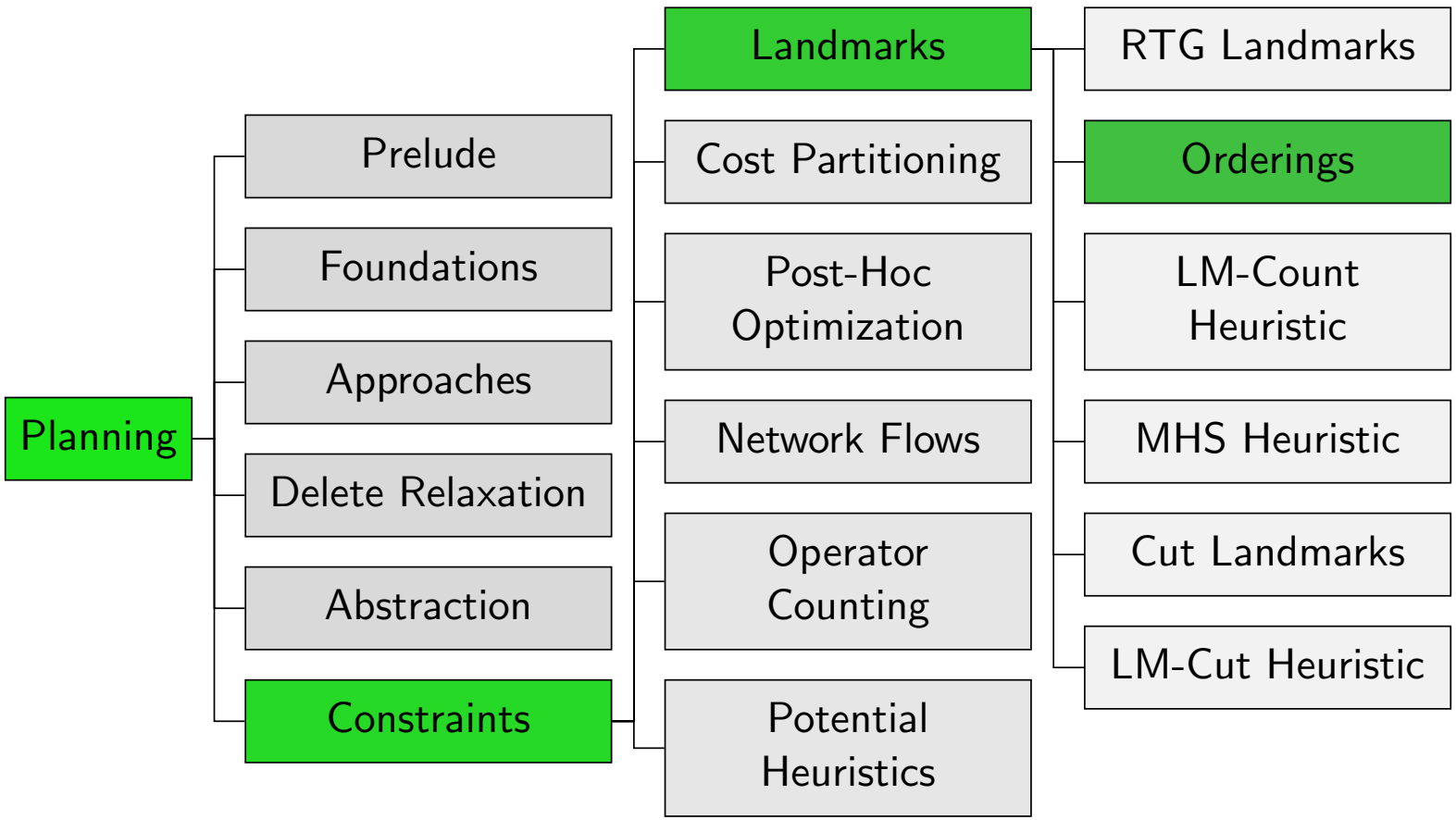
December 1, 2025

Planning and Optimization

December 1, 2025 — F3. Landmarks: Orderings & LM-Count Heuristic

F3.1 Landmark Orderings

Content of the Course



Why Landmark Orderings?

- To compute a landmark heuristic estimate for state s we need landmarks for s.
- We could invest the time to compute them for every state from scratch.
- Alternatively, we can compute landmarks once and propagate them over operator applications.
- Landmark orderings are used to detect landmarks that should be further considered because they (again) need to be satisfied later.

- (We will later see yet another approach, where heuristic computation and landmark computation are integrated $\rightsquigarrow$ LM-Cut.)

Example

Consider task $\langle \{a, b, c, d\}, I, \{o_1, o_2, \ldots, o_n\}, d \rangle$ with

- $I(v) = \bot$ for $v \in \{a, b, c, d\}$,
- $o_1 = \langle \top, a \wedge b \rangle$, and
- $o_2 = \langle a, c \wedge \neg a \wedge \neg b \rangle$ (plus some more operators).

You know that a, b, c and d are all fact landmarks for I.

- What landmarks are still required to be made true in state $I[\![\langle o_1, o_2 \rangle]\!]$?
- You get the additional information that variable a must be true immediately before d is first made true. Any changes?

Terminology

Let $\pi = \langle o_1, \ldots, o_n \rangle$ be a sequence of operators applicable in state I and let φ be a formula over the state variables.

- φ is true at time i if $I[\![\langle o_1, \ldots, o_i \rangle]\!] \models \varphi$.
- Also special case $i = 0$: φ is true at time 0 if $I \models \varphi$.
- No formula is true at time $i < 0$.
- φ is added at time i if it is true at time i but not at time $i - 1$.
- φ is first added at time i if it is true at time i but not at any time $j < i$.
  We denote this i by $\mathit{first}(\varphi, \pi)$.
- $\mathit{last}(\varphi, \pi)$ denotes the last time in which φ is added in π.

Landmark Orderings

Definition (Landmark Orderings)

Let φ and ψ be formula landmarks. There is

- a natural ordering between φ and ψ (written $\varphi \rightarrow \psi$) if in each plan π it holds that $\mathit{first}(\varphi, \pi) < \mathit{first}(\psi, \pi)$.
  "φ must be true some time strictly before ψ is first added."
- a greedy-necessary ordering between φ and ψ (written $\varphi \rightarrow_{\mathsf{gn}} \psi$) if for every plan $\pi = \langle o_1, \ldots, o_n \rangle$ it holds that $s[\![\langle o_1, \ldots, o_{\mathit{first}(\psi,\pi)-1} \rangle]\!] \models \varphi$.
  "φ must be true immediately before ψ is first added."
- a weak ordering between φ and ψ (written $\varphi \rightarrow_{\mathsf{w}} \psi$) if in each plan π it holds that $\mathit{first}(\varphi, \pi) < \mathit{last}(\psi, \pi)$.
  "φ must be true some time before ψ is last added."

Not covered: reasonable orderings, which generalize weak orderings

Natural Orderings

Definition

There is a natural ordering between φ and ψ (written $\varphi \rightarrow \psi$) if in each plan π it holds that $\mathit{first}(\varphi, \pi) < \mathit{first}(\psi, \pi)$.

- We can directly determine natural orderings from the LM sets computed from the simplified relaxed task graph.
- For fact landmarks v, v' with $v \neq v'$, if $n_{v'} \in LM(n_v)$ then $v' \rightarrow v$.

Greedy-necessary Orderings

Definition

There is a greedy-necessary ordering between φ and ψ (written $\varphi \rightarrow_{\text{gn}} \psi$) if in each plan where ψ is first added at time i, φ is true at time $i - 1$.

- We can again determine such orderings from the sRTG.
- For an OR node n_v, we define the set of first achievers as $FA(n_v) = \{n_o \mid n_o \in succ(n_v) \text{ and } n_v \notin LM(n_o)\}$.
- Then $v' \rightarrow_{\text{gn}} v$ if $n_{v'} \in succ(n_o)$ for all $n_o \in FA(n_v)$.

F3.2 Landmark Propagation

Example Revisited

Consider task $\langle \{a, b, c, d\}, I, \{o_1, o_2, \ldots, o_n\}, d \rangle$ with

- $I(v) = \bot$ for $v \in \{a, b, c, d\}$,
- $o_1 = \langle \top, a \wedge b \rangle$ and $o_2 = \langle a, c \wedge \neg a \wedge \neg b \rangle$ (plus some more).

You know that a, b, c and d are all fact landmarks for I.

- What landmarks are still required to be made true in state $I[\![\langle o_1, o_2 \rangle]\!]$? All not achieved yet on the state path
- You get the additional information that variable a must be true immediately before d is first made true. Any changes? Exploit orderings to determine landmarks that are still required.
- There is another path to the same state where b was never true. What now?
  Exploit information from multiple paths.

Past and Future Landmarks

- In the following, $\mathcal{L}_I$ is always a set of formula landmarks for the initial state with set of orderings $\mathcal{O}_I$.
- The set $\mathcal{L}^*_{\text{past}}(s)$ of past landmarks of a state s contains all landmarks from $\mathcal{L}_I$ that are at some point true in every path from the initial state to s.
- The set $\mathcal{L}^*_{\text{fut}}(s)$ of future landmarks of a state s contains all landmarks from $\mathcal{L}_I$ that are also landmarks of s but not true in s.
- Past landmarks are important for inferring which orderings are still relevant, future landmarks are relevant for the heuristic estimates.
- Since the exact sets are defined over all paths between certain states, we use approximations.

Landmark State

Definition

Let $\mathcal{L}_I$ be a set of formula landmarks for the initial state.

A landmark state $\mathbb{L}$ is $\bot$ or a pair $\langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ such that $\mathcal{L}_{\text{fut}} \cup \mathcal{L}_{\text{past}} = \mathcal{L}_I$.

$\mathbb{L}$ is valid in state s if

- $\mathbb{L} = \bot$ and Π has no s-plan, or
- $\mathbb{L} = \langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ with $\mathcal{L}_{\text{past}} \supseteq \mathcal{L}^*_{\text{past}}$ and $\mathcal{L}_{\text{fut}} \subseteq \mathcal{L}^*_{\text{fut}}$.

Context in Search: LM-BFS Algorithm

$\mathbb{L}(\text{init}), \mathcal{L}_I, \mathcal{O}_I := \text{compute\_landmark\_info}(\text{init}())$
if $h(\text{init}(), \mathbb{L}(\text{init})) < \infty$ **then**
    $\mathit{open}.\text{insert}(\langle \text{init}(), 0, h(\text{init}(), \mathbb{L}(\text{init})) \rangle)$
while $\mathit{open} \neq \emptyset$ **do**
    $\langle s, g, v \rangle = \mathit{open}.\text{pop}()$
    if $v < h(s, \mathbb{L}(s))$ **then**
        $\mathit{open}.\text{insert}(\langle s, g, h(s, \mathbb{L}(s)) \rangle)$
    else if $g < \mathit{distances}(s)$ **then**
        $\mathit{distances}(s) := g$
        if $\text{is\_goal}(s)$ **then return** $\text{extract\_plan}(s)$;
        foreach $\langle a, s' \rangle \in \mathit{succ}(s)$ **do**
            $\mathbb{L}' := \text{progress\_landmark\_state}(\mathbb{L}(s), \langle s, a, s' \rangle)$
            $\mathbb{L}(s') := \text{merge\_landmark\_states}(\mathbb{L}(s'), \mathbb{L}')$
            if $\mathbb{L}(s') \neq \bot$ *and* $h(s', \mathbb{L}(s')) < \infty$ **then**
                $\mathit{open}.\text{insert}(\langle s', g + \mathit{cost}(a), h(s', \mathbb{L}(s')) \rangle)$

$\mathbb{L}(s) := \langle \mathcal{L}_I, \emptyset \rangle$ and $\mathit{distances}(s) := \infty$ if read before set.

Context: Exploit Information from Multiple Paths

$\mathbb{L}(\text{init}), \mathcal{L}_I, \mathcal{O}_I := \text{compute\_landmark\_info}(\text{init}())$
if $h(\text{init}(), \mathbb{L}(\text{init})) < \infty$ **then**
    $\mathit{open}.\text{insert}(\langle \text{init}(), 0, h(\text{init}(), \mathbb{L}(\text{init})) \rangle)$
while $\mathit{open} \neq \emptyset$ **do**
    $\langle s, g, v \rangle = \mathit{open}.\text{pop}()$
    if $v < h(s, \mathbb{L}(s))$ **then**
        $\mathit{open}.\text{insert}(\langle s, g, h(s, \mathbb{L}(s)) \rangle)$
    else if $g < \mathit{distances}(s)$ **then**
        $\mathit{distances}(s) := g$
        if $\text{is\_goal}(s)$ **then return** $\text{extract\_plan}(s)$;
        foreach $\langle a, s' \rangle \in \mathit{succ}(s)$ **do**
            $\mathbb{L}' := \text{progress\_landmark\_state}(\mathbb{L}(s), \langle s, a, s' \rangle)$
            $\mathbb{L}(s') := \text{merge\_landmark\_states}(\mathbb{L}(s'), \mathbb{L}')$
            if $\mathbb{L}(s') \neq \bot$ *and* $h(s', \mathbb{L}(s')) < \infty$ **then**
                $\mathit{open}.\text{insert}(\langle s', g + \mathit{cost}(a), h(s', \mathbb{L}(s')) \rangle)$

$\mathbb{L}(s) := \langle \mathcal{L}_I, \emptyset \rangle$ and $\mathit{distances}(s) := \infty$ if read before set.

Merging Landmark States

Merging combines the information from two landmark states.

merge_landmark_states($\mathbb{L}, \mathbb{L}'$)

if $\mathbb{L} = \bot$ *or* $\mathbb{L}' = \bot$ **then** return $\bot$;
$\langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle := \mathbb{L}$
$\langle \mathcal{L}'_{\text{past}}, \mathcal{L}'_{\text{fut}} \rangle := \mathbb{L}'$
return $\langle \mathcal{L}_{\text{past}} \cap \mathcal{L}'_{\text{past}}, \mathcal{L}_{\text{fut}} \cup \mathcal{L}'_{\text{fut}} \rangle$

Theorem

If $\mathbb{L}$ and $\mathbb{L}'$ are valid in a state s then also merge_landmark_states($\mathbb{L}, \mathbb{L}'$) is valid in s.

Context: Progression for a Transition

$\mathbb{L}(\text{init}), \mathcal{L}_I, \mathcal{O}_I := \text{compute\_landmark\_info}(\text{init}())$
if $h(\text{init}(), \mathbb{L}(\text{init})) < \infty$ **then**
    $\mathit{open}.\text{insert}(\langle \text{init}(), 0, h(\text{init}(), \mathbb{L}(\text{init}))\rangle)$
while $\mathit{open} \neq \emptyset$ **do**
    $\langle s, g, v\rangle = \mathit{open}.\text{pop}()$
    if $v < h(s, \mathbb{L}(s))$ **then**
        $\mathit{open}.\text{insert}(\langle s, g, h(s, \mathbb{L}(s))\rangle)$
    else if $g < \mathit{distances}(s)$ **then**
        $\mathit{distances}(s) := g$
        if is_goal(s) **then return** extract_plan(s);
        foreach $\langle a, s'\rangle \in \mathit{succ}(s)$ **do**
            $\mathbb{L}' := \text{progress\_landmark\_state}(\mathbb{L}(s), \langle s, a, s'\rangle)$
            $\mathbb{L}(s') := \text{merge\_landmark\_states}(\mathbb{L}(s'), \mathbb{L}')$
            if $\mathbb{L}(s') \neq \bot$ *and* $h(s', \mathbb{L}(s')) < \infty$ **then**
                $\mathit{open}.\text{insert}(\langle s', g + \mathit{cost}(a), h(s', \mathbb{L}(s'))\rangle)$

$\mathbb{L}(s) := \langle \mathcal{L}_I, \emptyset\rangle$ and $\mathit{distances}(s) := \infty$ if read before set.

Progressing Landmark States

- If we expand a state s with transition $\langle s, o, s' \rangle$, we use progression to determine a landmark state for s' from the one we know for s.
- We will only introduce progression methods that preserve the validity of landmark states.
- Since every progression method gives a valid landmark state, we can merge results from different methods into a valid landmark state.

Basic Progression

Definition (Basic Progression)

Basic progression maps landmark state $\langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ and transition $\langle s, o, s' \rangle$ to landmark state $\langle \mathcal{L}_{\text{past}} \cup \mathcal{L}_{\text{add}}, \mathcal{L}_{\text{fut}} \setminus \mathcal{L}_{\text{add}} \rangle$, where $\mathcal{L}_{\text{add}} = \{\varphi \in \mathcal{L}_I \mid s \not\models \varphi \text{ and } s' \models \varphi\}$.

"Extend the past with all landmarks added in s' and remove them from the future."

Goal Progression

Definition (Goal Progression)

Let γ be the goal of the task.
Goal progression maps landmark state $\langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ and transition $\langle s, o, s' \rangle$ to landmark state $\langle \mathcal{L}_I, \mathcal{L}_{\text{goal}} \rangle$, where
$\mathcal{L}_{\text{goal}} = \{\varphi \in \mathcal{L}_I \mid \gamma \models \varphi \text{ and } s' \not\models \varphi\}$.

"All landmarks that must be true in the goal but are false in s' must be achieved in the future."

Weak Ordering Progression

$\varphi \rightarrow_{\mathsf{w}} \psi$: "$\varphi$ must be true some time before ψ is last added."

Definition (Weak Ordering Progression)

The weak ordering progression maps landmark state $\langle \mathcal{L}_{\mathsf{past}}, \mathcal{L}_{\mathsf{fut}} \rangle$ and transition $\langle s, o, s' \rangle$ to landmark state $\langle \mathcal{L}_I, \{\psi \mid \exists \varphi \rightarrow_{\mathsf{w}} \psi : \varphi \notin \mathcal{L}_{\mathsf{past}}\} \rangle$.

"Landmark ψ must be added in the future because we haven't done something that must be done before ψ is last added."

Greedy-necessary Ordering Progression

$\varphi \rightarrow_{\mathsf{gn}} \psi$: "$\varphi$ must be true immediately before ψ is first added."

Definition (Greedy-necessary Ordering Progression)

The greedy necessary ordering progression maps landmark state $\langle \mathcal{L}_{\mathsf{past}}, \mathcal{L}_{\mathsf{fut}} \rangle$ and transition $\langle s, o, s' \rangle$ to landmark state

- $\bot$ if there is a $\varphi \rightarrow_{\mathsf{gn}} \psi \in \mathcal{O}_I$ with $\psi \notin \mathcal{L}_{\mathsf{past}}, s \not\models \varphi$ and $s' \models \psi$, and
- $\langle \mathcal{L}_I, \{\varphi \mid s' \not\models \varphi \text{ and } \exists \varphi \rightarrow_{\mathsf{gn}} \psi \in \mathcal{O}_I : \psi \notin \mathcal{L}_{\mathsf{past}}, s' \not\models \psi\} \rangle$ otherwise.

"Landmark ψ has not been true, yet, and φ must be true immediately before it becomes true. Since φ is currently false, we must make it true in the future (before making ψ true)."

Natural Ordering Progression

$\varphi \rightarrow \psi$: φ must be true some time strictly before ψ is first added.

Definition (Natural Ordering Progression)

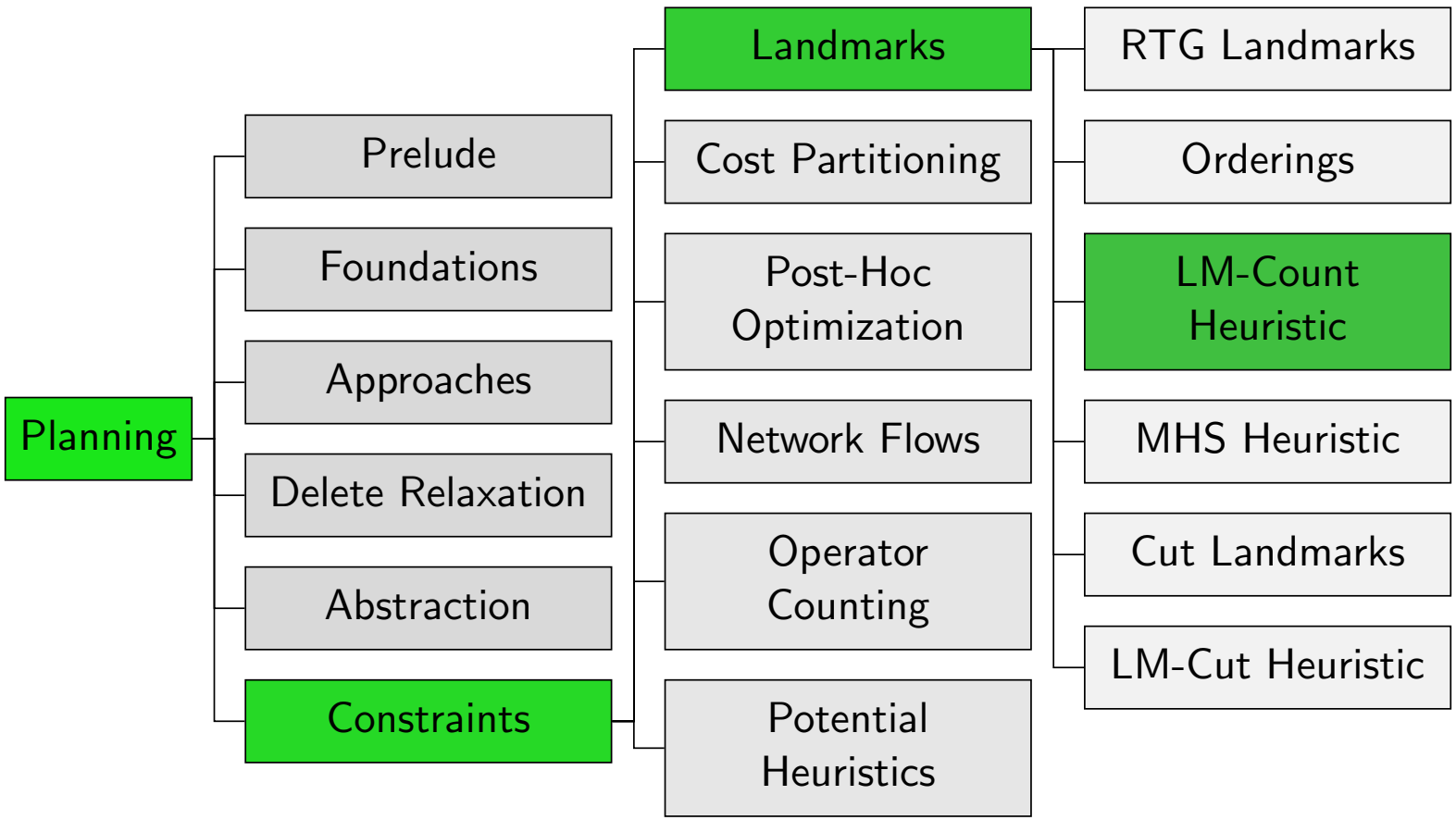
The natural ordering progression maps landmark state $\langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ and transition $\langle s, o, s' \rangle$ to landmark state

- $\bot$ if there is a $\varphi \rightarrow \psi \in \mathcal{O}_I$ with $\varphi \notin \mathcal{L}_{\text{past}}$ and $s' \models \psi$, and
- $\langle \mathcal{L}_I, \emptyset \rangle$ otherwise.

Not (yet) useful: All known methods only find natural orderings that are true for every applicable operator sequence, so the interesting first case never happens in LM-BFS.

F3.3 Landmark-count Heuristic

Content of the Course



Landmark-count Heuristic

The landmark-count heuristic counts the landmarks that still have to be achieved.

Definition (LM-count Heuristic)

Let Π be a planning task, s be a state and $\mathbb{L} = \langle \mathcal{L}_{\text{past}}, \mathcal{L}_{\text{fut}} \rangle$ be a valid landmark state for s.

The LM-count heuristic for s and $\mathbb{L}$ is

$$h^{\text{LM-count}}(s, \mathbb{L}) = \begin{cases} \infty & \text{if } \mathbb{L} = \bot, \\ |\mathcal{L}_{\text{fut}}| & \text{otherwise} \end{cases}$$

In the original work, $\mathcal{L}_{\text{fut}}$ was determined without considering information from multiple paths and could not detect dead-ends.

LM-count Heuristic is Path-dependent

- LM-count heuristic gives estimates for landmark states, which depend on the considered paths.
- Search algorithms need estimates for states.
- $\rightsquigarrow$ we use estimate from the current landmark state.
- $\rightsquigarrow$ heuristic estimate for a state is not well-defined.

LM-count Heuristic is Inadmissible

Example

Consider STRIPS planning task $\Pi = \langle \{a, b\}, I, \{o\}, \{a, b\} \rangle$ with $I = \emptyset$, $o = \langle \emptyset, \{a, b\}, \emptyset, 1 \rangle$. Let $\mathcal{L} = \{a, b\}$ and $\mathcal{O} = \emptyset$.

Landmark state $\langle \emptyset, \mathcal{L} \rangle$ for the initial state is valid and the estimate is $h^{\text{LM-count}}(I, \langle \emptyset, \{a, b\} \rangle) = 2$
while $h^*(I) = 1$.

$\rightsquigarrow$ $h^{\text{LM-count}}$ is inadmissible.

LM-count Heuristic: Comments

- LM-Count alone is not a particularily informative heuristic.
- On the positive side, it complements h^{FF} very well.
- For example, the LAMA planning system alternates between expanding a state with minimal h^{FF} and minimal $h^{\text{LM-count}}$ estimate.
- The LM-sum heuristic is a cost-aware variant of the heuristic that sums up the costs of the cheapest achiever ($=$ operator that adds the fact landmark) of each landmark.
- There is an admissible variant of the heuristic based on operator cost partitioning.

F3.4 Summary

Summary

- We can propagate landmark sets over action applications.
- Landmark orderings can be useful for detecting when a landmark that has already been achieved should be further considered.
- We can combine the landmark information from several paths to the same state.
- The LM-count heuristic counts how many landmarks still need to be satisfied.
- The LM-count heuristic is inadmissible (but there is an admissible variant).

Planning and Optimization

F4. Landmarks: Minimum Hitting Set Heuristic

Malte Helmert and Gabriele Röger

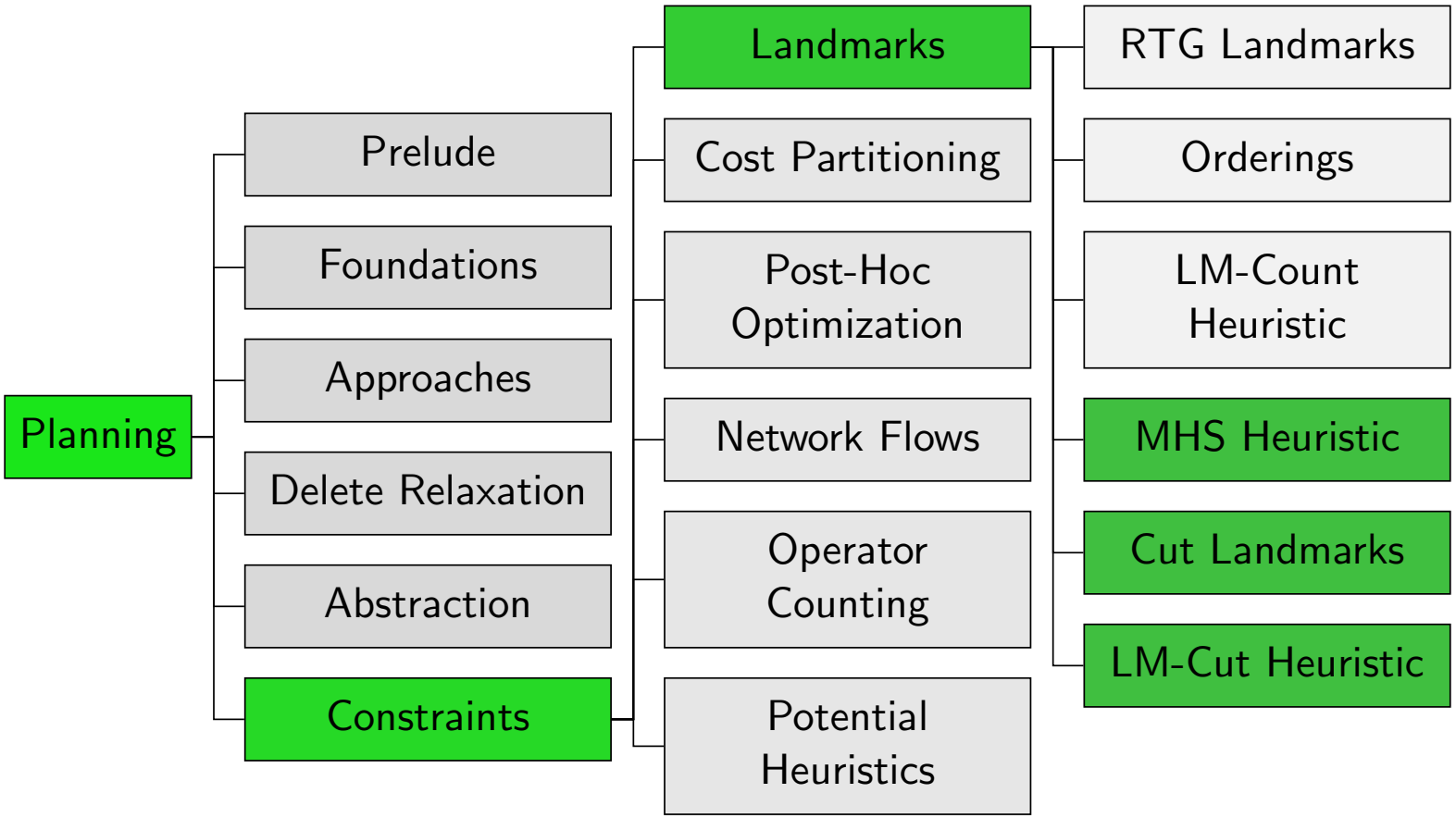
Universität Basel

December 1, 2025

F4.1 Minimum Hitting Set Heuristic

F4.2 Summary

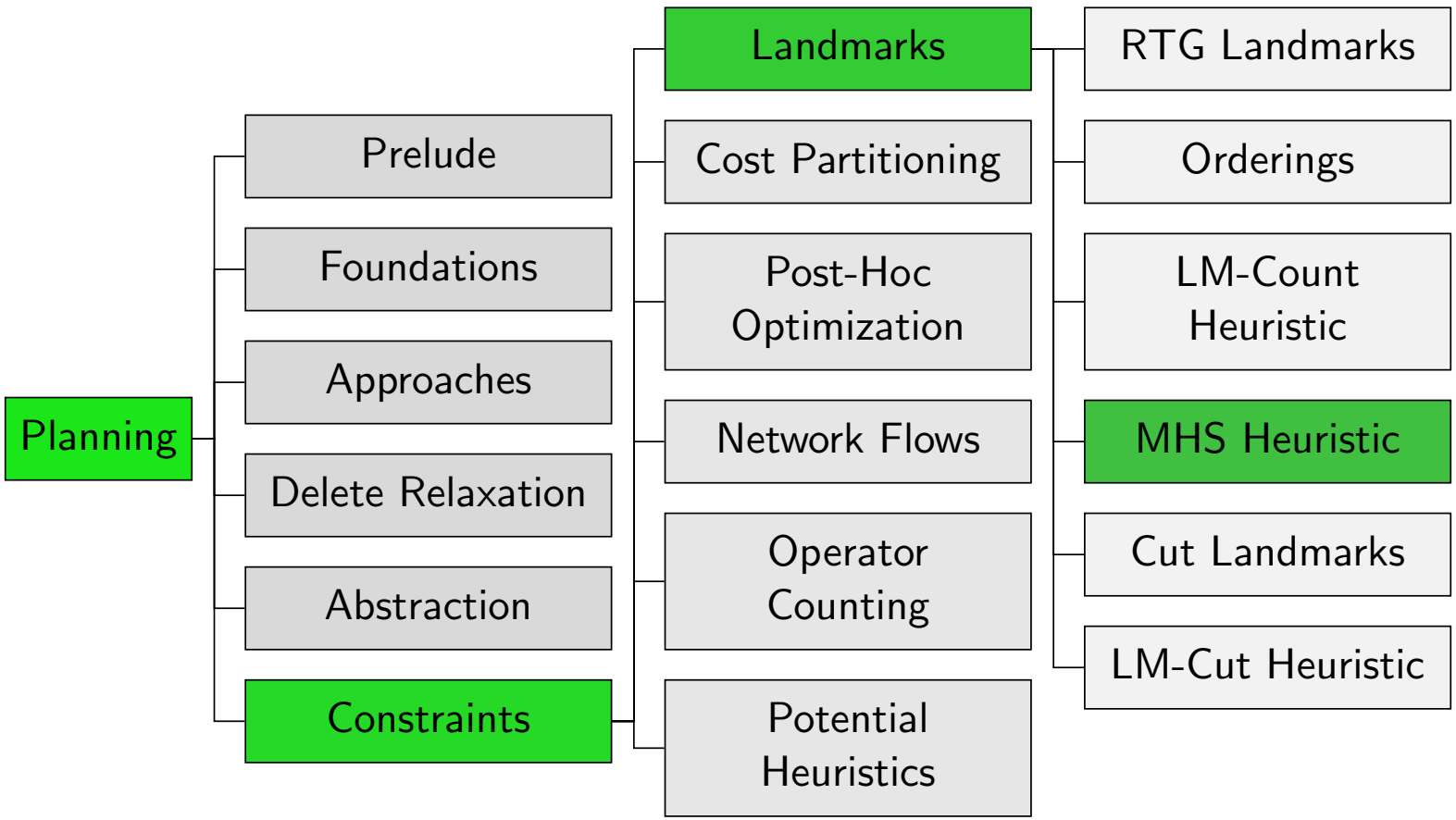
Content of the Course



The remaining landmark topics focus on disjunctive action landmarks.

F4.1 Minimum Hitting Set Heuristic

Content of the Course



Exploiting Disjunctive Action Landmarks

- The cost $cost(L)$ of a disjunctive action landmark L is an admissible heuristic, but it is usually not very informative.
- Landmark heuristics typically aim to combine multiple disjunctive action landmarks.

How can we exploit a given set $\mathcal{L}$ of disjunctive action landmarks?

- Sum of costs $\sum_{L \in \mathcal{L}} cost(L)$?
  $\rightsquigarrow$ not admissible!
- Maximize costs $\max_{L \in \mathcal{L}} cost(L)$?
  $\rightsquigarrow$ usually very weak heuristic
- better: Hitting sets

Hitting Sets

Definition (Hitting Set)

Let X be a set, $\mathcal{F} = \{F_1, \ldots, F_n\} \subseteq 2^X$ be a family of subsets of X and $c : X \to \mathbb{R}_0^+$ be a cost function for X.

A hitting set is a subset $H \subseteq X$ that "hits" all subsets in $\mathcal{F}$, i.e., $H \cap F \neq \emptyset$ for all $F \in \mathcal{F}$. The cost of H is $\sum_{x \in H} c(x)$.

A minimum hitting set (MHS) is a hitting set with minimal cost.

MHS is a "classical" NP-complete problem (Karp, 1972)

Example: Hitting Sets

Example

$X = \{o_1, o_2, o_3, o_4\}$

$\mathcal{F} = \{\{o_4\}, \{o_1, o_2\}, \{o_1, o_3\}, \{o_2, o_3\}\}$

$c(o_1) = 3, \quad c(o_2) = 4, \quad c(o_3) = 5, \quad c(o_4) = 0$

Specify a minimum hitting set.

Solution: $\{o_1, o_2, o_4\}$ with cost $3 + 4 + 0 = 7$

Hitting Sets for Disjunctive Action Landmarks

Idea: disjunctive action landmarks are interpreted as instance of minimum hitting set

Definition (Hitting Set Heuristic)

Let $\mathcal{L}$ be a set of disjunctive action landmarks. The hitting set heuristic $h^{\mathrm{MHS}}(\mathcal{L})$ is defined as the cost of a minimum hitting set for $\mathcal{L}$ with $c(o) = cost(o)$.

Proposition (Hitting Set Heuristic is Admissible)

Let $\mathcal{L}$ be a set of disjunctive action landmarks for state s. Then $h^{MHS}(\mathcal{L})$ is an admissible estimate for s.

Hitting Set Heuristic: Discussion

- The hitting set heuristic is the best possible heuristic that only uses the given information...
- ...but is NP-hard to compute.
- $\rightsquigarrow$ Use approximations that can be efficiently computed.
  $\Rightarrow$ LP-relaxation, cost partitioning (both discussed later)

F4.2 Summary

Summary

- Hitting sets yield the most accurate heuristic for a given set of disjunctive action landmarks.
- The computation of a minimal hitting set is NP-hard.

Planning and Optimization

F5. Landmarks: Cut Landmarks & LM-Cut Heuristic

Malte Helmert and Gabriele Röger

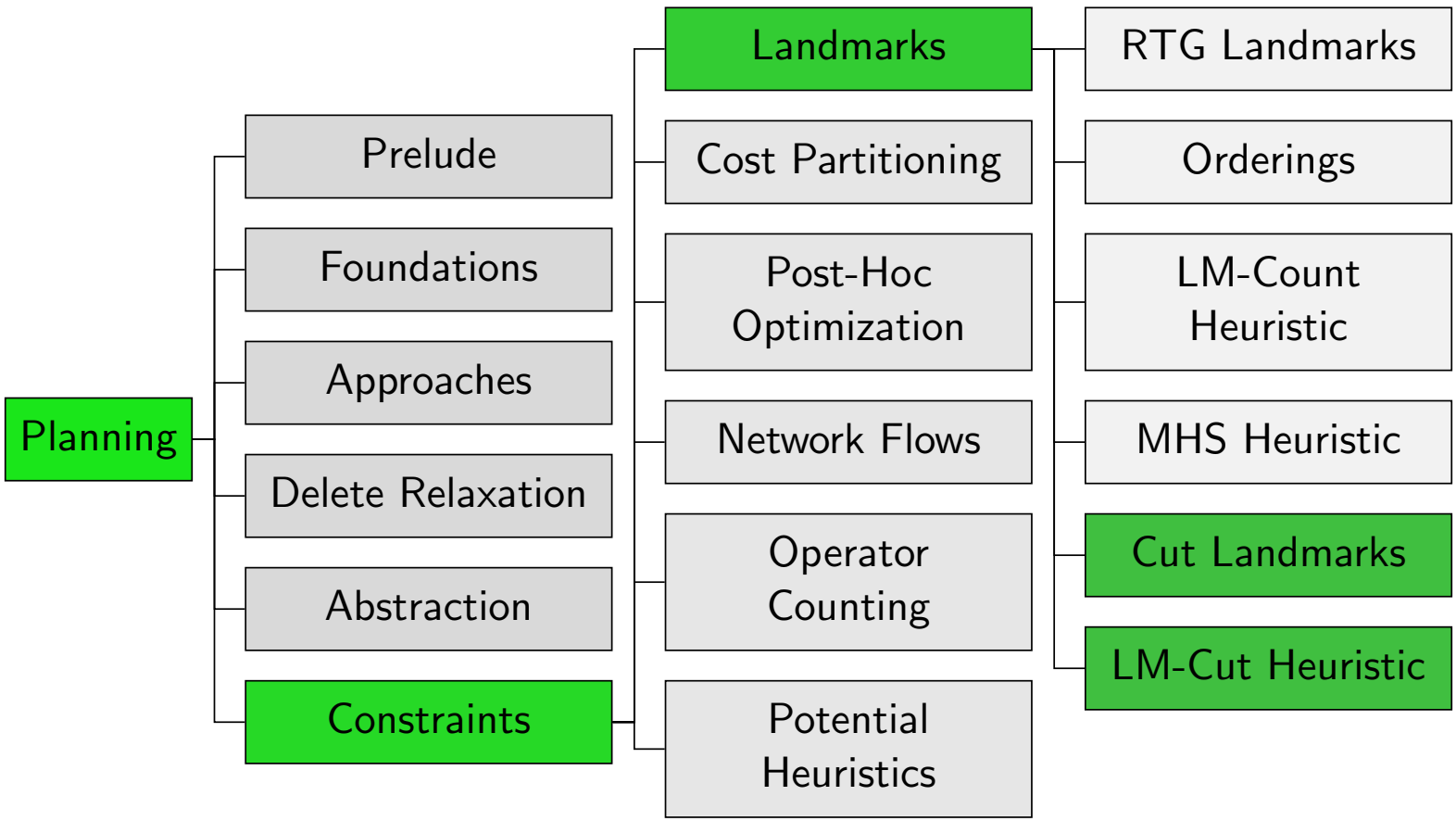
Universität Basel

December 3, 2025

Planning and Optimization

December 3, 2025 — F5. Landmarks: Cut Landmarks & LM-Cut Heuristic

Content of the Course



Roadmap for this Chapter

- We first introduce a new normal form for delete-free STRIPS tasks that simplifies later definitions.
- We then present a method that computes disjunctive action landmarks for such tasks.
- We conclude with the LM-cut heuristic that builds on this method.

F5.1 i-g Form

Delete-Free STRIPS Planning Task in i-g Form (1)

In this chapter, we only consider delete-free STRIPS tasks in a special form:

Definition (i-g Form for Delete-free STRIPS)

A delete-free STRIPS planning task $\langle V, I, O, \gamma \rangle$ is in i-g form if

- V contains atoms i and g
- Initially exactly i is true: $I(v) = \mathbf{T}$ iff $v = i$
- g is the only goal atom: $\gamma = \{g\}$
- Every action has at least one precondition.

Transformation to i-g Form

Every delete-free STRIPS task $\Pi = \langle V, I, O, \gamma \rangle$ can easily be transformed into an analogous task in i-g form.

- If i or g are in V already, rename them everywhere.
- Add i and g to V.
- Add an operator $\langle \{i\}, \{v \in V \mid I(v) = \mathbf{T}\}, \{\}, 0 \rangle$.
- Add an operator $\langle \gamma, \{g\}, \{\}, 0 \rangle$.
- Replace all operator preconditions $\top$ with i.
- Replace initial state and goal.

For the remainder of this chapter, we assume tasks in i-g form.

Example: Delete-Free Planning Task in i-g Form

Example

Consider a delete-relaxed STRIPS planning $\langle V, I, O, \gamma \rangle$ with $V = \{i, a, b, c, d, g\}$, $I = \{i \mapsto \mathbf{T}\} \cup \{v \mapsto \mathbf{F} \mid v \in V \setminus \{i\}\}$, $\gamma = g$ and operators

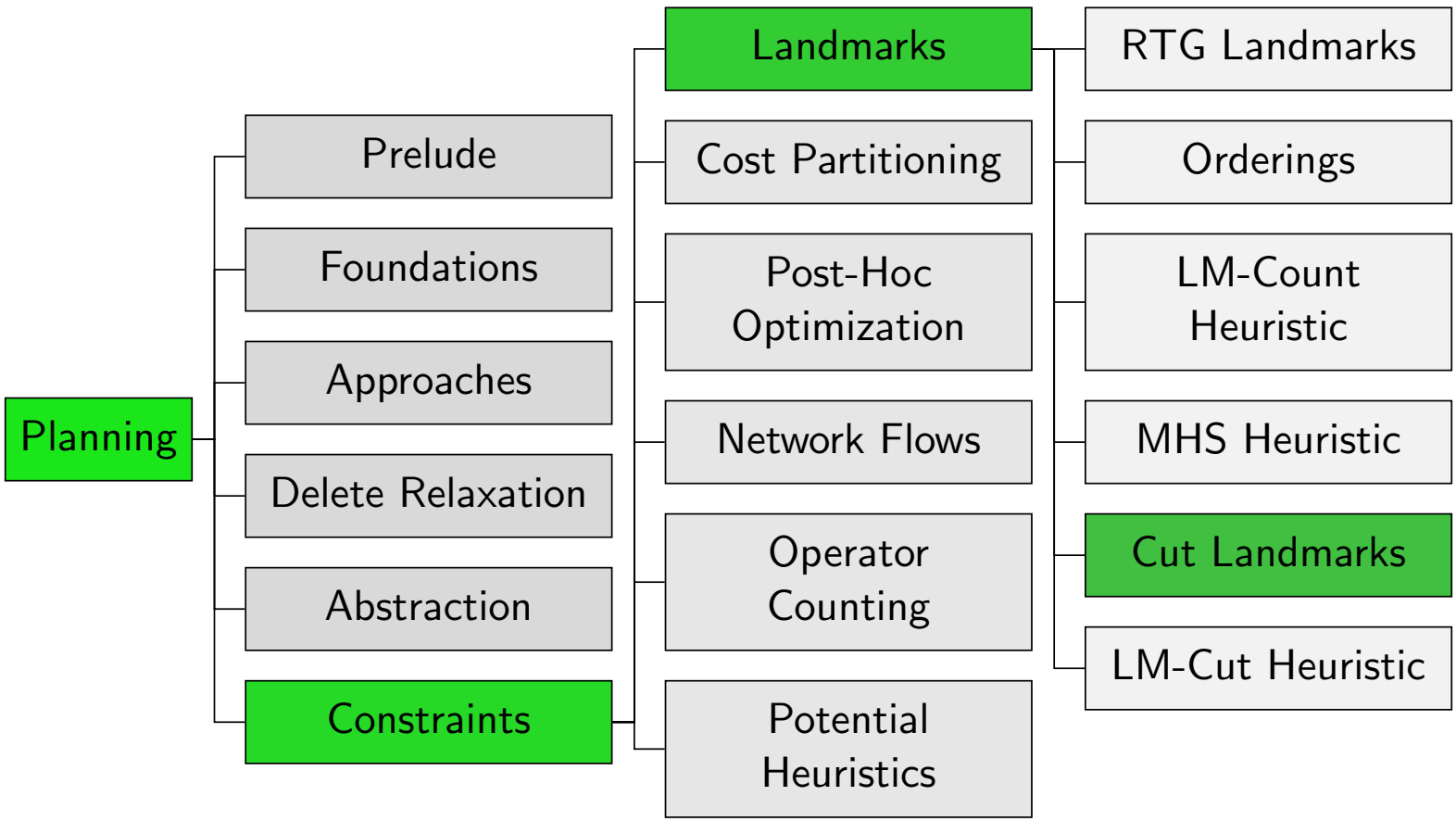
- $o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$,
- $o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$,
- $o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$,
- $o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 2 \rangle$, and
- $o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$.

optimal solution to reach g from i:

- plan: $\langle o_{\text{blue}}, o_{\text{black}}, o_{\text{red}}, o_{\text{orange}} \rangle$
- cost: $4 + 3 + 2 + 0 = 9$ $\quad (= h^+(I)$ because plan is optimal$)$

F5.2 Cut Landmarks

Content of the Course



Justification Graphs

Definition (Precondition Choice Function)

A precondition choice function (pcf) $P : O \to V$ for a delete-free STRIPS task $\Pi = \langle V, I, O, \gamma \rangle$ in i-g form maps each operator to one of its preconditions (i.e. $P(o) \in pre(o)$ for all $o \in O$).

Definition (Justification Graphs)

Let P be a pcf for $\langle V, I, O, \gamma \rangle$ in i-g form. The justification graph for P is the directed, edge-labeled graph $J = \langle V, E \rangle$, where

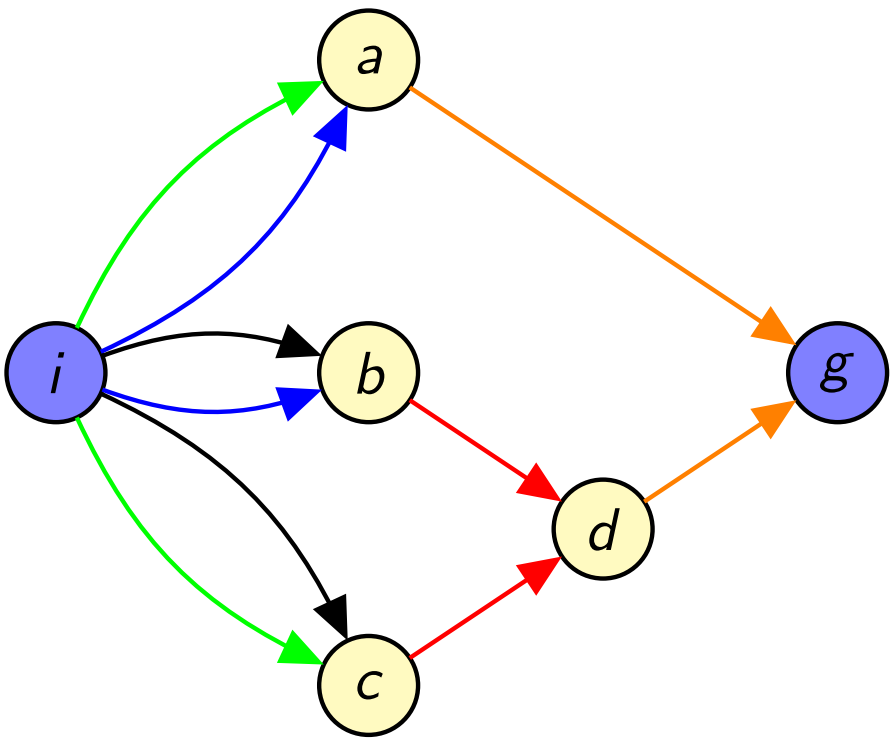
- the vertices are the variables from V, and
- E contains an edge $P(o) \xrightarrow{o} a$ for each $o \in O$, $a \in add(o)$.

Example: Justification Graph

Example (Precondition Choice Function)

$P(o_{\text{blue}}) = P(o_{\text{green}}) = P(o_{\text{black}}) = i$, $P(o_{\text{red}}) = b$, $P(o_{\text{orange}}) = a$

$P'(o_{\text{blue}}) = P'(o_{\text{green}}) = P'(o_{\text{black}}) = i$, $P'(o_{\text{red}}) = c$, $P'(o_{\text{orange}}) = d$

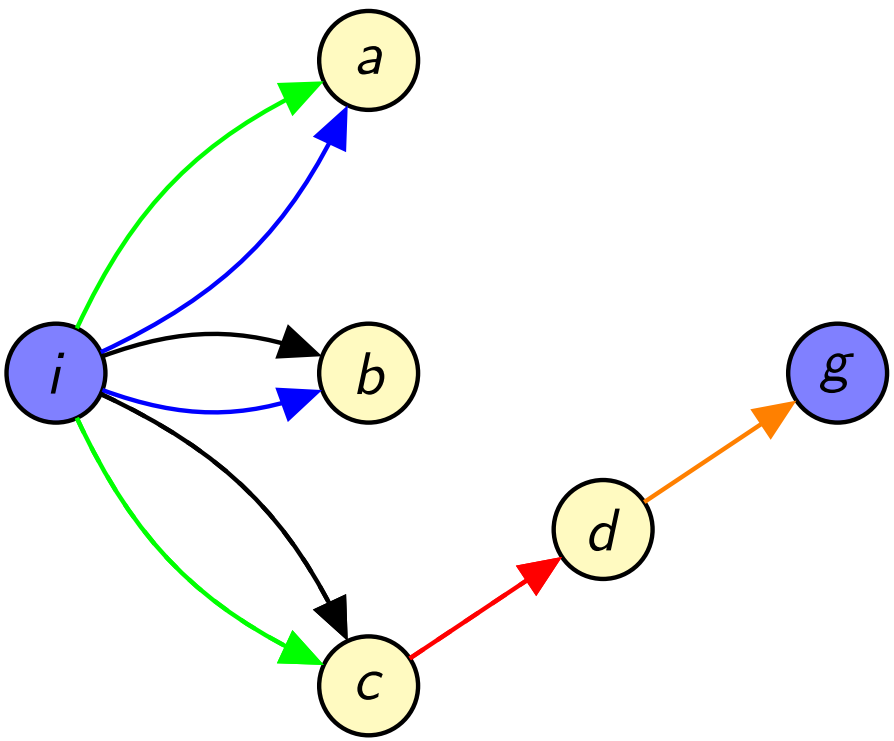


$$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$$
$$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$$
$$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$$
$$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 2 \rangle$$
$$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$$

Cuts

Definition (Cut)

A cut in a justification graph is a subset C of its edges such that all paths from i to g contain an edge from C.



$$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$$
$$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$$
$$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$$
$$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 2 \rangle$$
$$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$$

Cuts are Disjunctive Action Landmarks

Theorem (Cuts are Disjunctive Action Landmarks)

Let P be a pcf for $\langle V, I, O, \gamma \rangle$ (in i-g form) and C be a cut in the justification graph for P.

The set of edge labels from C (formally $\{o \mid \langle v, o, v' \rangle \in C\}$) is a disjunctive action landmark for I.

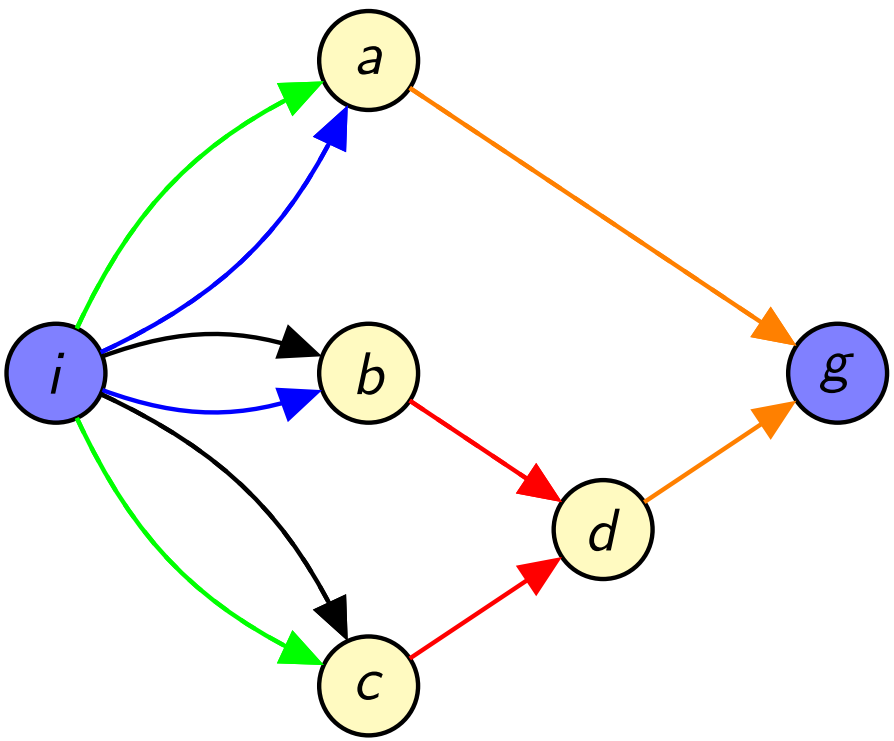
Proof idea:

- The justification graph corresponds to a simpler problem where some preconditions (those not picked by the pcf) are ignored.
- Cuts are landmarks for this simplified problem.
- Hence they are also landmarks for the original problem.

Example: Cuts in Justification Graphs

Example (Landmarks)

- $L_1 = \{o_{\text{orange}}\}$ (cost $= 0$)
- $L_2 = \{o_{\text{green}}, o_{\text{black}}\}$ (cost $= 3$)
- $L_3 = \{o_{\text{red}}\}$ (cost $= 2$)
- $L_4 = \{o_{\text{green}}, o_{\text{blue}}\}$ (cost $= 4$)



$$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$$
$$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$$
$$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$$
$$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 2 \rangle$$
$$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$$

Power of Cuts in Justification Graphs

- Which landmarks can be computed with the cut method?
- all interesting ones!

Proposition (perfect hitting set heuristics)

Let $\mathcal{L}$ be the set of all "cut landmarks" of a given planning task with initial state I. Then $h^{MHS}(\mathcal{L}) = h^+(I)$.

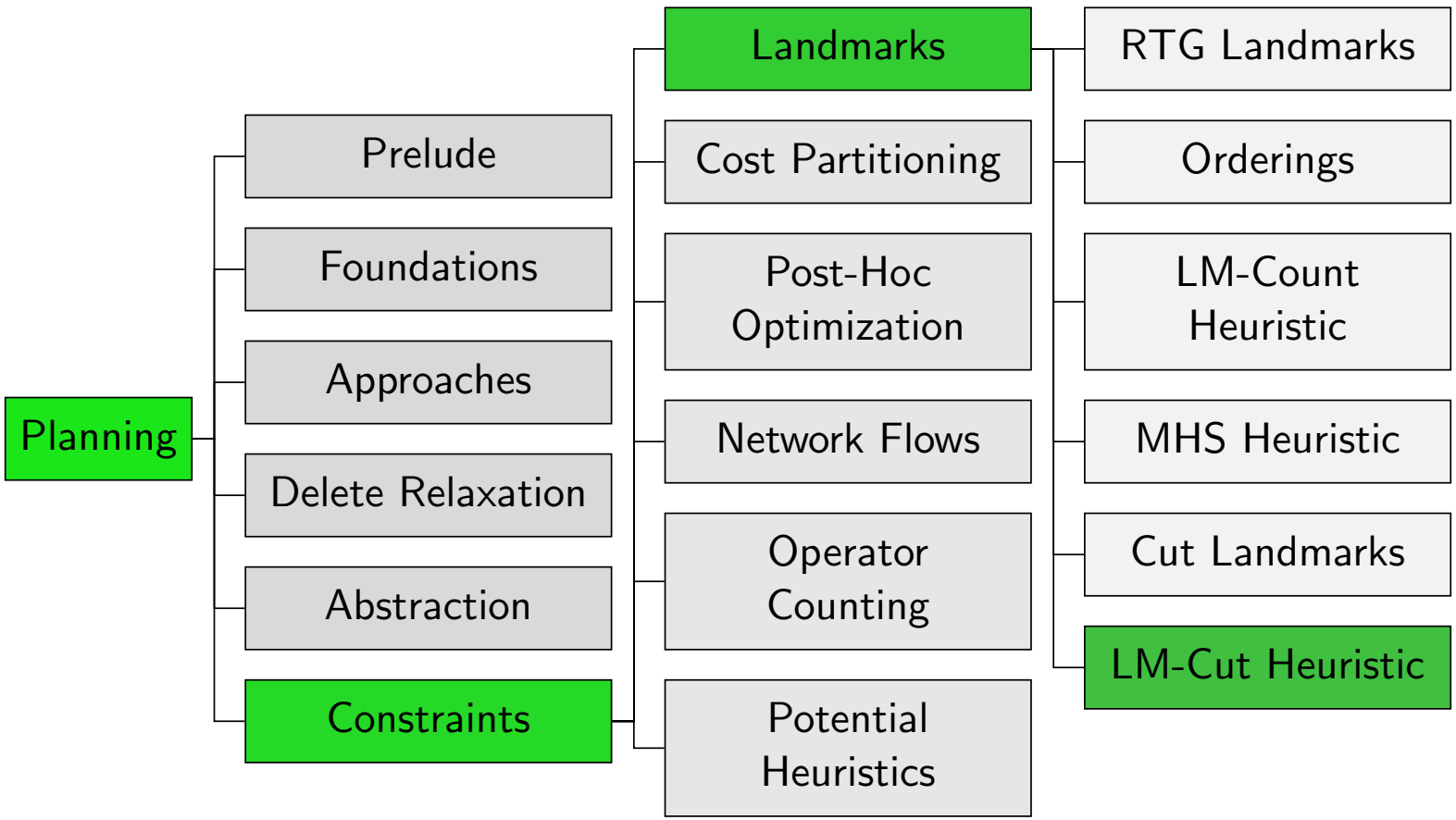
$\rightsquigarrow$ Hitting set heuristic for $\mathcal{L}$ is perfect.

Proof idea:

- Show 1:1 correspondence of hitting sets H for $\mathcal{L}$ and plans, i.e., each hitting set for $\mathcal{L}$ corresponds to a plan, and vice versa.

F5.3 The LM-Cut Heuristic

Content of the Course



LM-Cut Heuristic: Motivation

- In general, there are exponentially many pcfs, hence computing all relevant landmarks is not tractable.
- The LM-cut heuristic is a method that chooses pcfs and computes cuts in a goal-oriented way.
- As a side effect, it computes a
  - a cost partitioning over multiple instances of $h^{\max}$ that is also
  - a saturated cost partitioning over disjunctive action landmarks.
    $\rightsquigarrow$ next week

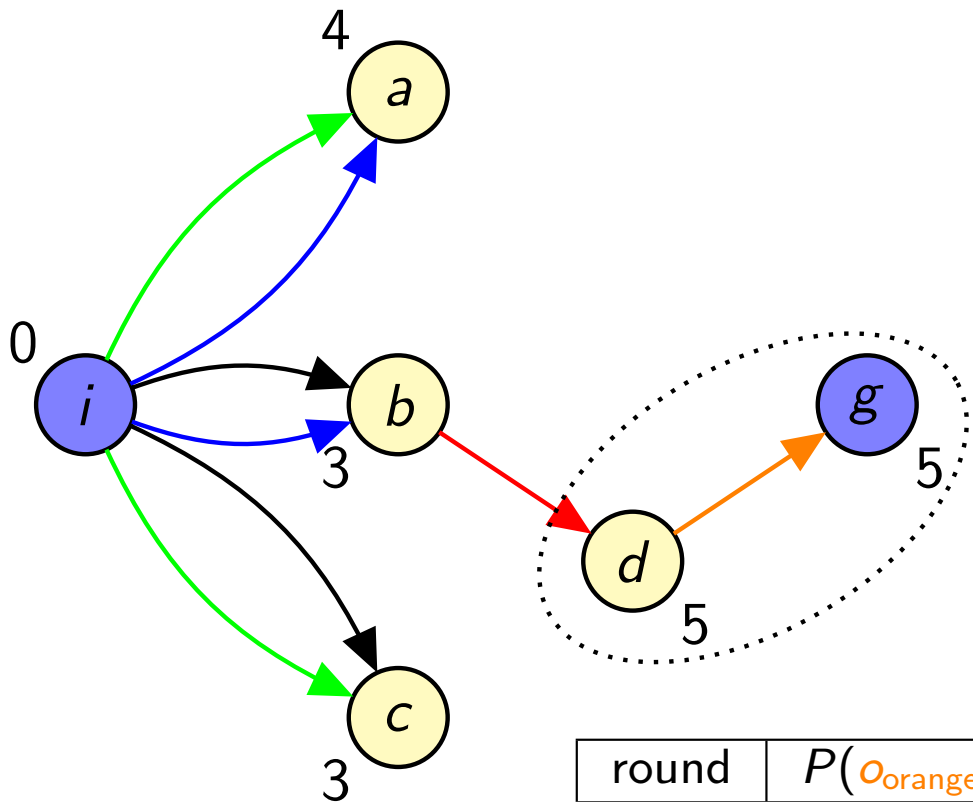
LM-Cut Heuristic

$h^{\text{LM-cut}}$: Helmert & Domshlak (2009)

Initialize $h^{\text{LM-cut}}(I) := 0$. Then iterate:

1. Compute $h^{\max}$ values of the variables. Stop if $h^{\max}(g) = 0$.
2. Compute justification graph G for the P that chooses preconditions with maximal $h^{\max}$ value
3. Determine the goal zone V_g of G that consists of all nodes that have a zero-cost path to g.
4. Compute the cut L that contains the labels of all edges $\langle v, o, v' \rangle$ such that $v \notin V_g$, $v' \in V_g$ and v can be reached from i without traversing a node in V_g.
   It is guaranteed that $\mathit{cost}(L) > 0$.
5. Increase $h^{\text{LM-cut}}(I)$ by $\mathit{cost}(L)$.
6. Decrease $\mathit{cost}(o)$ by $\mathit{cost}(L)$ for all $o \in L$.

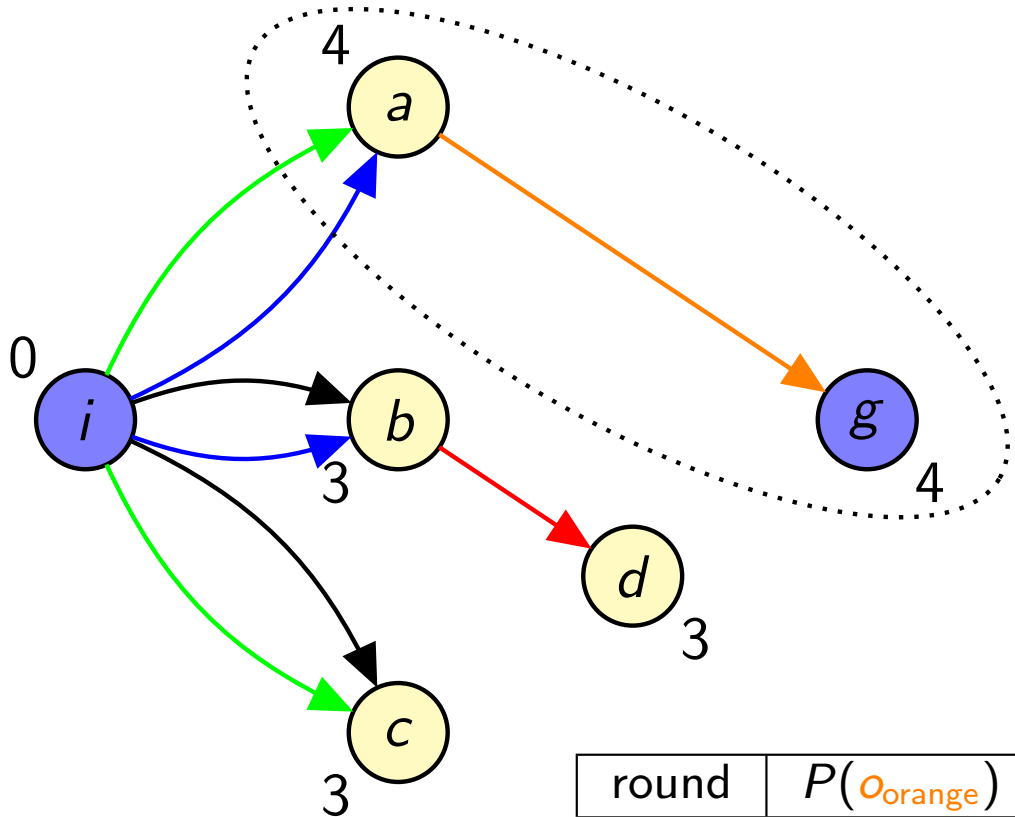
Example: Computation of LM-Cut



$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$
$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$
$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$
$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 0 \rangle$
$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$

round	$P(o_{\text{orange}})$	$P(o_{\text{red}})$	landmark	cost
1	d	b	$\{o_{\text{red}}\}$	2
			$h^{\text{LM-cut}}(I)$	2

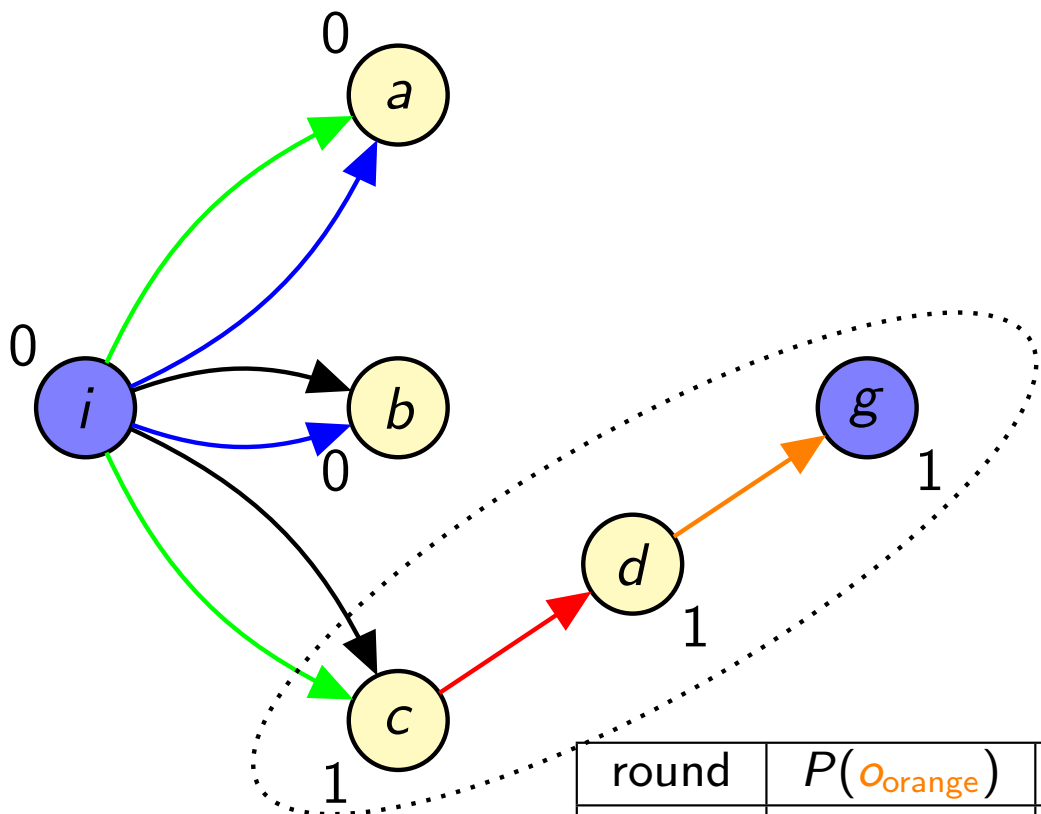
Example: Computation of LM-Cut



$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 0 \rangle$
$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 1 \rangle$
$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$
$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 0 \rangle$
$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$

round	$P(o_{\text{orange}})$	$P(o_{\text{red}})$	landmark	cost
1	d	b	$\{o_{\text{red}}\}$	2
2	a	b	$\{o_{\text{green}}, o_{\text{blue}}\}$	4
			$h^{\text{LM-cut}}(I)$	6

Example: Computation of LM-Cut



$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 0 \rangle$
$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 0 \rangle$
$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 2 \rangle$
$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 0 \rangle$
$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$

round	$P(o_{\text{orange}})$	$P(o_{\text{red}})$	landmark	cost
1	d	b	$\{o_{\text{red}}\}$	2
2	a	b	$\{o_{\text{green}}, o_{\text{blue}}\}$	4
3	d	c	$\{o_{\text{green}}, o_{\text{black}}\}$	1
			$h^{\text{LM-cut}}(I)$	7

Properties of LM-Cut Heuristic

Theorem

Let $\langle V, I, O, \gamma \rangle$ be a delete-free STRIPS task in i-g normal form. The LM-cut heuristic is admissible: $h^{LM\text{-}cut}(I) \leq h^*(I)$.

Proof omitted.

If Π is not delete-free, we can compute $h^{\text{LM-cut}}$ on Π^+.
Then $h^{\text{LM-cut}}$ is bounded by h^+.

F5.4 Summary

Summary

- Cuts in justification graphs are a general method to find disjunctive action landmarks.
- The minimum hitting set over all cut landmarks is a perfect heuristic for delete-free planning tasks.
- The LM-cut heuristic is an admissible heuristic based on these ideas.

Planning and Optimization

F6. Linear & Integer Programming

Malte Helmert and Gabriele Röger

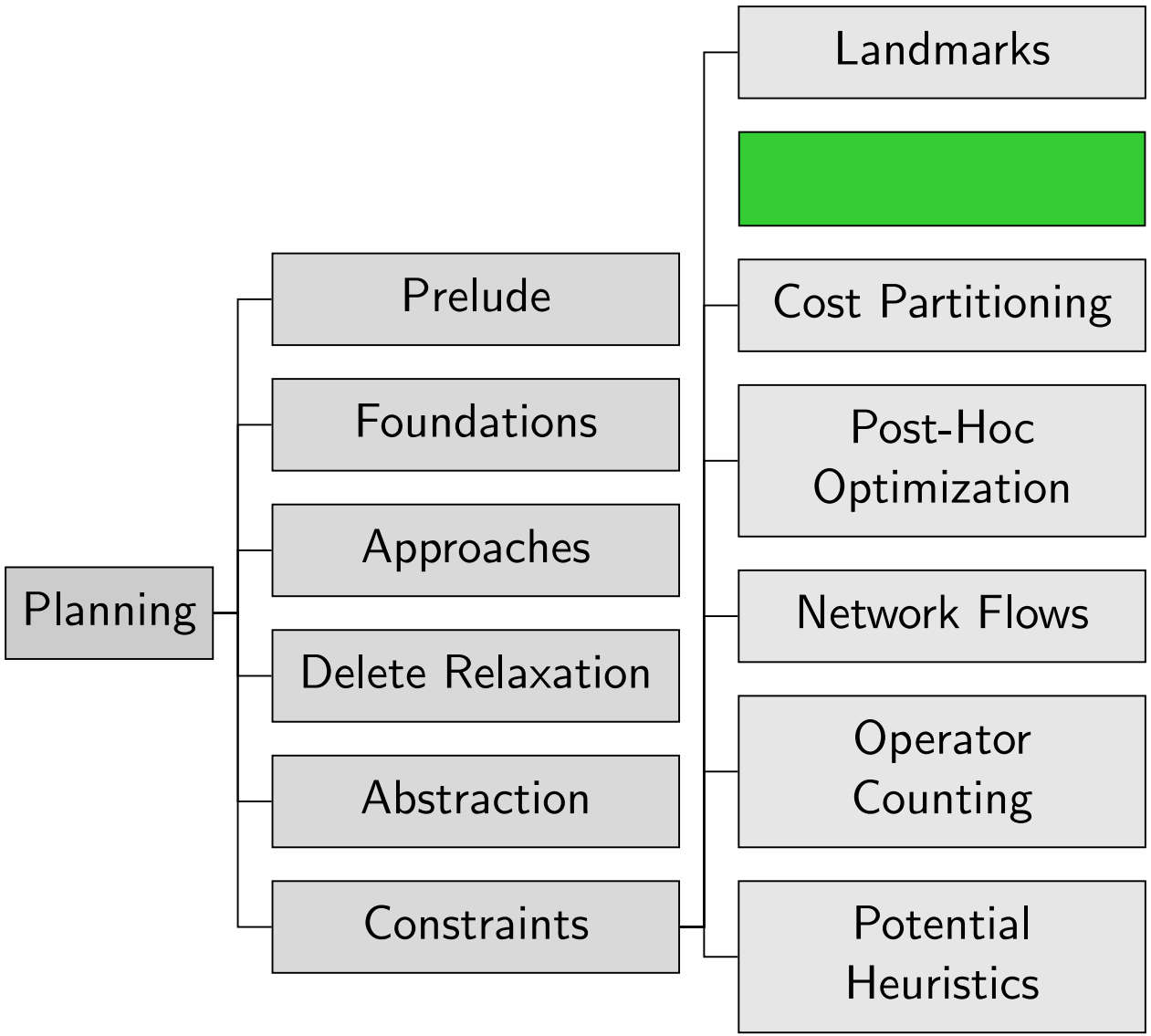
Universität Basel

December 3, 2025

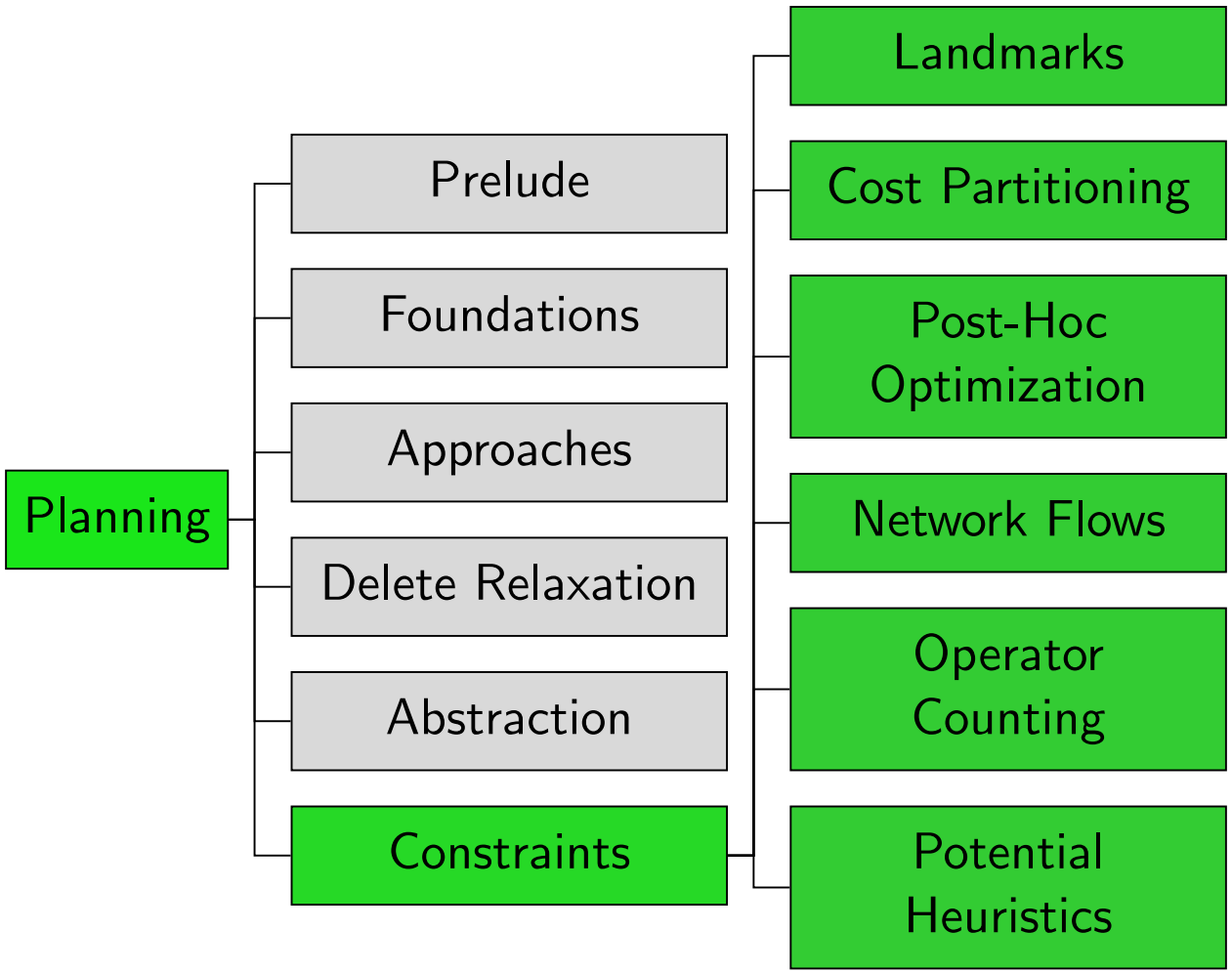
Planning and Optimization

December 3, 2025 — F6. Linear & Integer Programming

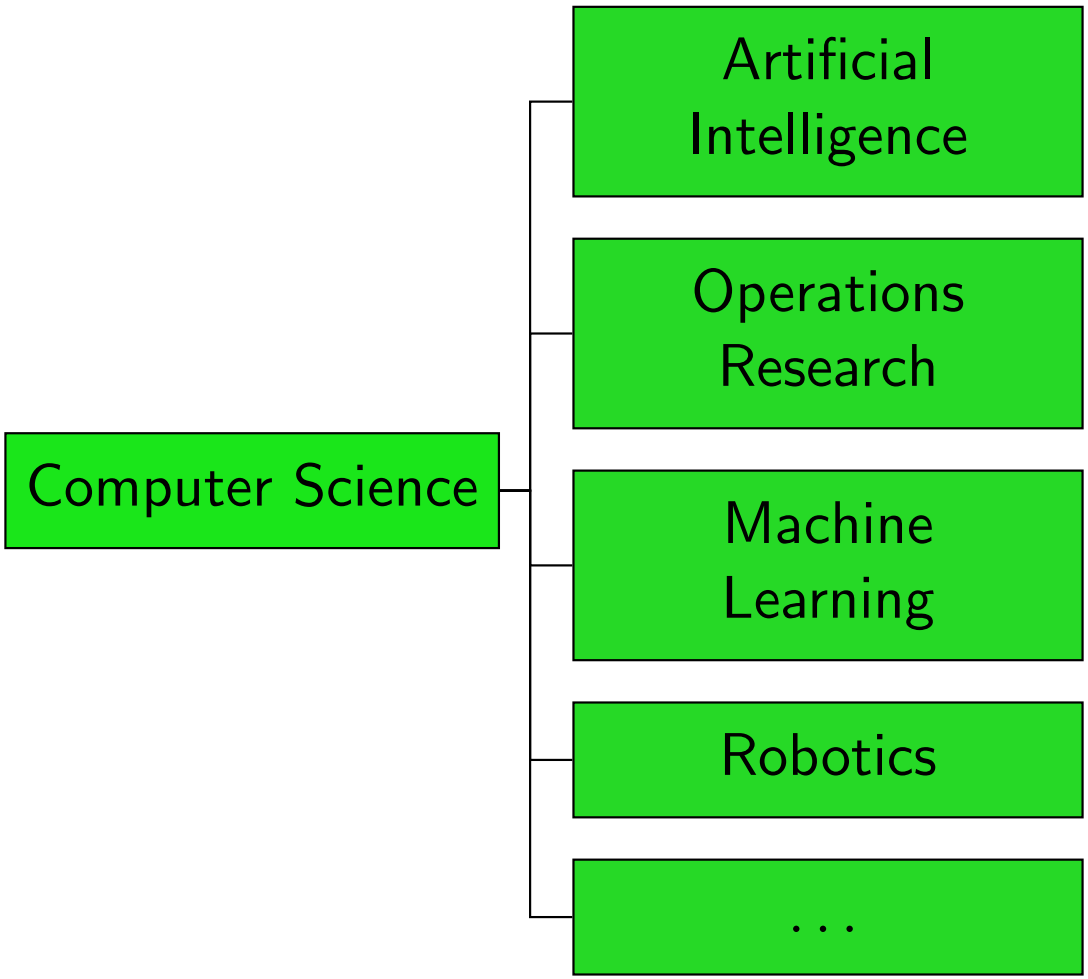
Content of the Course (Timeline)



Content of the Course (Relevance)



Not Content of this Course (Relevance)



F6.1 Integer Programs

Motivation

- This goes on beyond Computer Science
- Active research on IPs and LPs in
  - Operation Research
  - Mathematics
- Many application areas, for instance:
  - Manufacturing
  - Agriculture
  - Mining
  - Logistics
  - Planning
- As an application, we treat LPs / IPs as a blackbox
- We just look at the fundamentals
- However, even on the application side there is much more (e.g., modelling tricks or solver parameters to speed up computation)

Motivation

Example (Optimization Problem)

Consider the following scenario:

- A factory produces two products A and B
- Selling one (unit of) B yields 5 times the profit of selling one A
- A client places the unusual order to "buy anything that can be produced on that day as long as two plus twice the units of A is not smaller than the number of B"
- More than 12 products in total cannot be produced per day
- There is only material for 6 units of A
  (there is enough material to produce any amount of B)

How many units of A and B does the client receive
if the factory owner aims to maximize her profit?

Integer Program: Example

Let X_A and X_B be the (integer) number of produced A and B

Example (Optimization Problem as Integer Program)

$$\text{maximize} \quad X_A + 5X_B \quad \text{subject to}$$

$$2 + 2X_A \geq X_B$$
$$X_A + X_B \leq 12$$
$$X_A \leq 6$$

$$X_A \geq 0, \quad X_B \geq 0$$

$\rightsquigarrow$ unique optimal solution:
produce 4 A ($X_A = 4$) and 8 B ($X_B = 8$) for a profit of 44

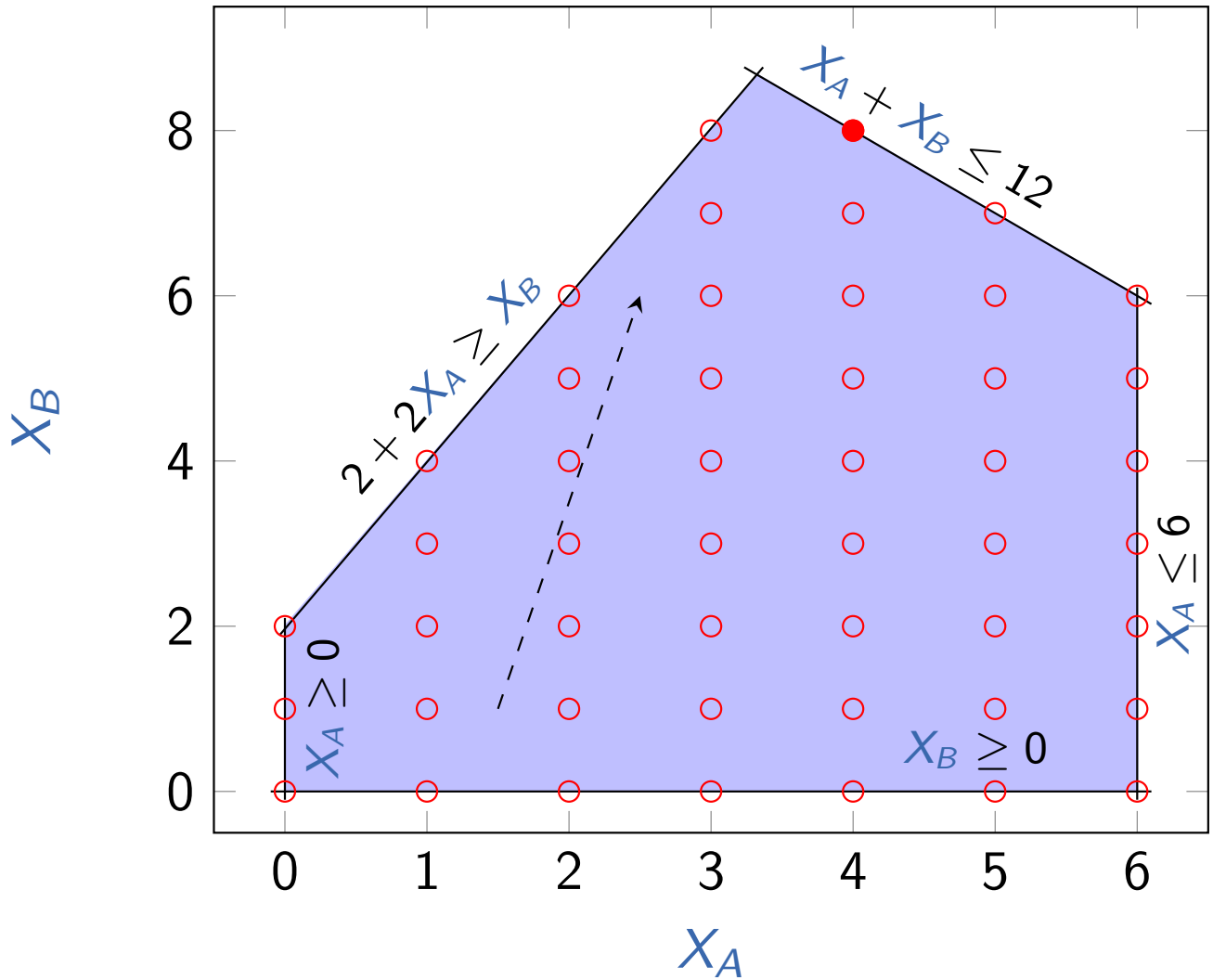
Same Program as Input for the Solver

File `ip.lp`

```
Maximize
 obj: X_A + 5 X_B
Subject To
 c1: -2 X_A + X_B <= 2
 c2: X_A + X_B <= 12
Bounds
 0 <= X_A <= 6
 0 <= X_B
General
 X_A X_B
End
```

→ Demo (Gurobi; same format also works with CPLEX and others)

Integer Program Example: Visualization



Integer Programs

Integer Program

An integer program (IP) consists of:

- a finite set of integer-valued variables V
- a finite set of linear inequalities (constraints) over V
- an objective function, which is a linear combination of V
- which should be minimized or maximized.

Terminology

- An integer assignment to all variables in V is feasible if it satisfies the constraints.
- An integer program is feasible if there is such a feasible assignment. Otherwise it is infeasible.
- A feasible maximum (resp. minimum) problem is unbounded if the objective function can assume arbitrarily large positive (resp. negative) values at feasible assignments. Otherwise it is bounded.
- The objective value of a bounded feasible maximum (resp. minimum) problem is the maximum (resp. minimum) value of the objective function with a feasible assignment.

Another Example

Example

$$\text{minimize} \quad 3X_{o_1} + 4X_{o_2} + 5X_{o_3} \quad \text{subject to}$$

$$\begin{aligned} X_{o_4} &\geq 1 \\ X_{o_1} + X_{o_2} &\geq 1 \\ X_{o_1} + X_{o_3} &\geq 1 \\ X_{o_2} + X_{o_3} &\geq 1 \end{aligned}$$

$$X_{o_1} \geq 0, \quad X_{o_2} \geq 0, \quad X_{o_3} \geq 0, \quad X_{o_4} \geq 0$$

What example from a recent chapter does this IP encode?

$\rightsquigarrow$ the minimum hitting set from Chapter F4

Complexity of Solving Integer Programs

- As an IP can compute an MHS, solving an IP must be at least as complex as computing an MHS
- Reminder: MHS is a "classical" NP-complete problem
- Good news: Solving an IP is not harder

$\rightsquigarrow$ Finding solutions for IPs is NP-complete.

Removing the requirement that solutions must be integer-valued leads to a simpler problem

F6.2 Linear Programs

Linear Programs

Linear Program

A linear program (LP) consists of:

- a finite set of real-valued variables V
- a finite set of linear inequalities (constraints) over V
- an objective function, which is a linear combination of V
- which should be minimized or maximized.

We use the introduced IP terminology also for LPs.

Mixed IPs (MIPs) are something between IPs and LPs:
some variables are integer-valued, some are real-valued.

Linear Program: Example

Let X_A and X_B be the (real-valued) number of produced A and B

Example (Optimization Problem as Linear Program)

$$\text{maximize} \quad X_A + 5X_B \quad \text{subject to}$$

$$2 + 2X_A \geq X_B$$
$$X_A + X_B \leq 12$$
$$X_A \leq 6$$

$$X_A \geq 0, \quad X_B \geq 0$$

$\rightsquigarrow$ unique optimal solution:
$X_A = 3\frac{1}{3}$ and $X_B = 8\frac{2}{3}$ with objective value $46\frac{2}{3}$

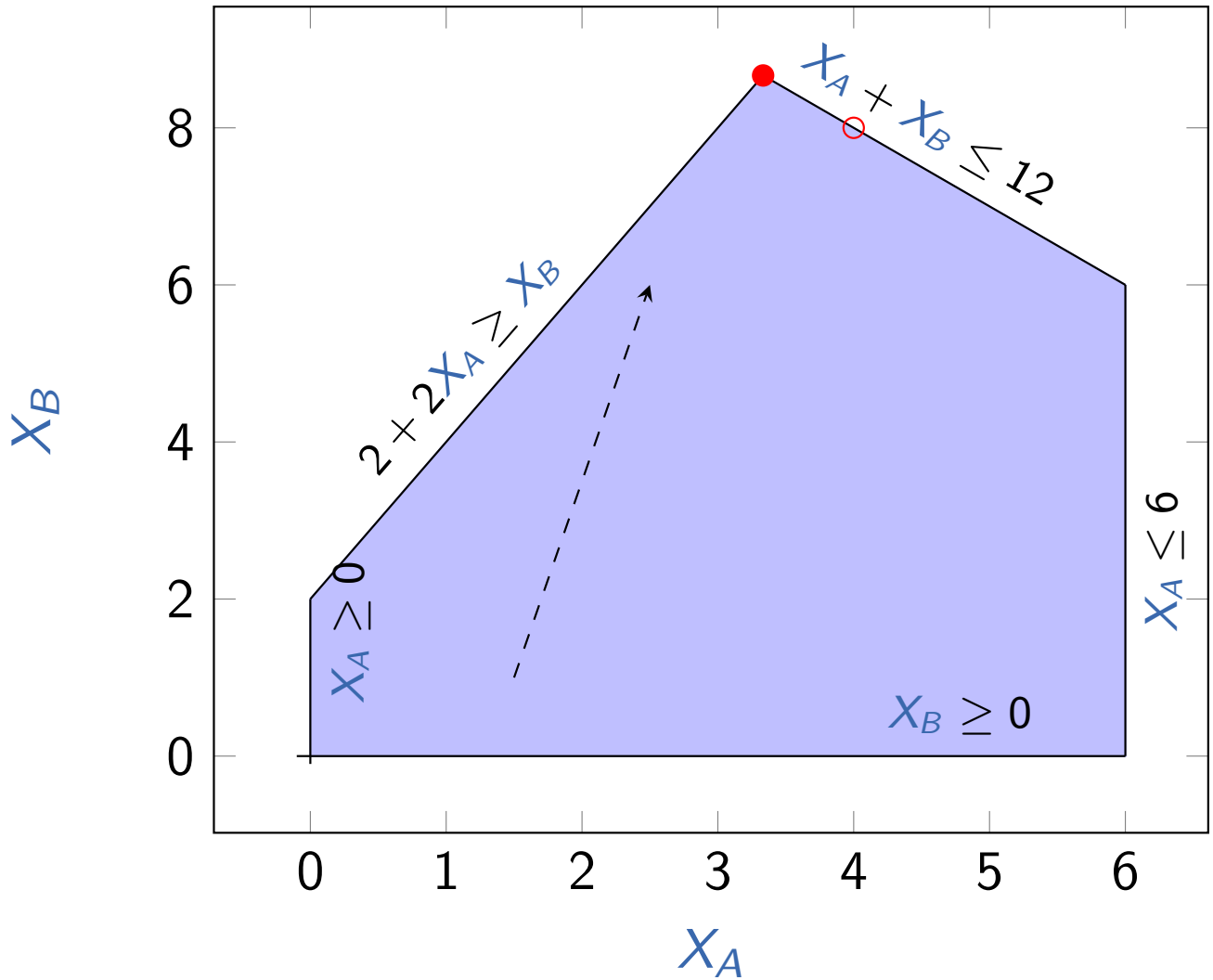
Same Program as Input for the Solver

File `lp.lp`

```
Maximize
 obj: X_A + 5 X_B
Subject To
 c1: -2 X_A + X_B <= 2
 c2: X_A + X_B <= 12
Bounds
 0 <= X_A <= 6
 0 <= X_B
End
```

→ Demo (Gurobi; same format also works with CPLEX and others)

Linear Program Example: Visualization



Solving Linear Programs

- Observation:
  Here, LP solution is an upper bound for the corresponding IP.
- Complexity:
  LP solving is a polynomial-time problem.
- Common idea:
  Approximate IP solution with corresponding LP
  (LP relaxation).

LP Relaxation

Theorem (LP Relaxation)

The LP relaxation of an integer program is the problem that arises by removing the requirement that variables are integer-valued.

For a maximization (resp. minimization) problem, the objective value of the LP relaxation is an upper (resp. lower) bound on the value of the IP.

Proof idea.

Every feasible assignment for the IP is also feasible for the LP. □

LP Relaxation of MHS heuristic

Example (Minimum Hitting Set)

$$\text{minimize} \quad 3X_{o_1} + 4X_{o_2} + 5X_{o_3} \quad \text{subject to}$$

$$\begin{aligned} X_{o_4} &\geq 1 \\ X_{o_1} + X_{o_2} &\geq 1 \\ X_{o_1} + X_{o_3} &\geq 1 \\ X_{o_2} + X_{o_3} &\geq 1 \end{aligned}$$

$$X_{o_1} \geq 0, \quad X_{o_2} \geq 0, \quad X_{o_3} \geq 0, \quad X_{o_4} \geq 0$$

- $\rightsquigarrow$ optimal solution of LP relaxation:
  $X_{o_4} = 1$ and $X_{o_1} = X_{o_2} = X_{o_3} = 0.5$ with objective value 6
- $\rightsquigarrow$ LP relaxation of MHS heuristic is admissible
  and can be computed in polynomial time

F6.3 Normal Forms and Duality

Standard Maximum Problem

Normal form for maximization problems:

Definition (Standard Maximum Problem)

Find values for $x_1, \ldots, x_n$, to maximize

$$c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

subject to the constraints

$$\begin{aligned} a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n &\leq b_1 \\ a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n &\leq b_2 \\ &\vdots \\ a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n &\leq b_m \end{aligned}$$

and $x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0$.

Standard Maximum Problem: Matrix and Vectors

A standard maximum problem is often given by

- an m-vector $\mathbf{b} = \langle b_1, \ldots, b_m \rangle^T$ (bounds),
- an n-vector $\mathbf{c} = \langle c_1, \ldots, c_n \rangle^T$ (objective coefficients),
- and an $m \times n$ matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \text{ (coefficients)}$$

- Then the problem is to find a vector $\mathbf{x} = \langle x_1, \ldots, x_n \rangle^T$ to maximize $\mathbf{c}^T\mathbf{x}$ subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$.

Standard Minimum Problem

- there is also a standard minimum problem
- it's form is identical to the standard maximum problem, except that
  - the aim is to minimize the objective function
  - subject to $\mathbf{Ax} \geq \mathbf{b}$
- All linear programs can efficiently be converted into a standard maximum/minimum problem.

Some LP Theory: Duality

Every LP has an alternative view (its dual LP).

Primal	Dual
maximization (or minimization)	minimization (or maximization)
objective coefficients	bounds
bounds	objective coefficients
bounded variable	$\geq$-constraint
$\leq$-constraint	bounded variable
free variable	$=$-constraint
$=$-constraint	free variable

dual of dual: original LP

Dual Problem

Definition (Dual Problem)

The dual of the standard maximum problem

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } \mathbf{A}\mathbf{x} \leq \mathbf{b} \text{ and } \mathbf{x} \geq \mathbf{0}$$

is the standard minimum problem

$$\text{minimize } \mathbf{b}^T\mathbf{y} \text{ subject to } \mathbf{A}^T\mathbf{y} \geq \mathbf{c} \text{ and } \mathbf{y} \geq \mathbf{0}$$

Dual Problem: Example

Example (Dual of the Optimization Problem)

$$\text{maximize} \quad X_A + 5X_B \quad \text{subject to}$$

$$\begin{aligned} [Y_1] \quad & -2X_A + X_B \leq 2 \\ [Y_2] \quad & X_A + X_B \leq 12 \\ [Y_3] \quad & X_A \leq 6 \end{aligned}$$

$$X_A \geq 0, \quad X_B \geq 0$$

Dual Problem: Example

Example (Dual of the Optimization Problem)

$$\text{minimize} \quad 2Y_1 + 12Y_2 + 6Y_3 \quad \text{subject to}$$

$$\begin{aligned} [X_A] \qquad -2Y_1 + Y_2 + Y_3 &\geq 1 \\ [X_B] \qquad Y_1 + Y_2 &\geq 5 \end{aligned}$$

$$Y_1 \geq 0, \quad Y_2 \geq 0, \quad Y_3 \geq 0$$

Duality Theorem

Theorem (Duality Theorem)

If a standard linear program is bounded feasible, then so is its dual, and their objective values are equal.

(Proof omitted.)

The dual provides a different perspective on a problem.

F6.4 Summary

Summary

- Linear (and integer) programs consist of an objective function that should be maximized or minimized subject to a set of given linear constraints.
- Finding solutions for integer programs is NP-complete.
- LP solving is a polynomial time problem.
- The dual of a maximization LP is a minimization LP and vice versa.
- The dual of a bounded feasible LP has the same objective value.

Further Reading

The slides in this chapter are based on the following excellent tutorial on LP solving:

Thomas S. Ferguson.
Linear Programming – A Concise Introduction.
UCLA, unpublished document available online.

Planning and Optimization

F7. Cost Partitioning

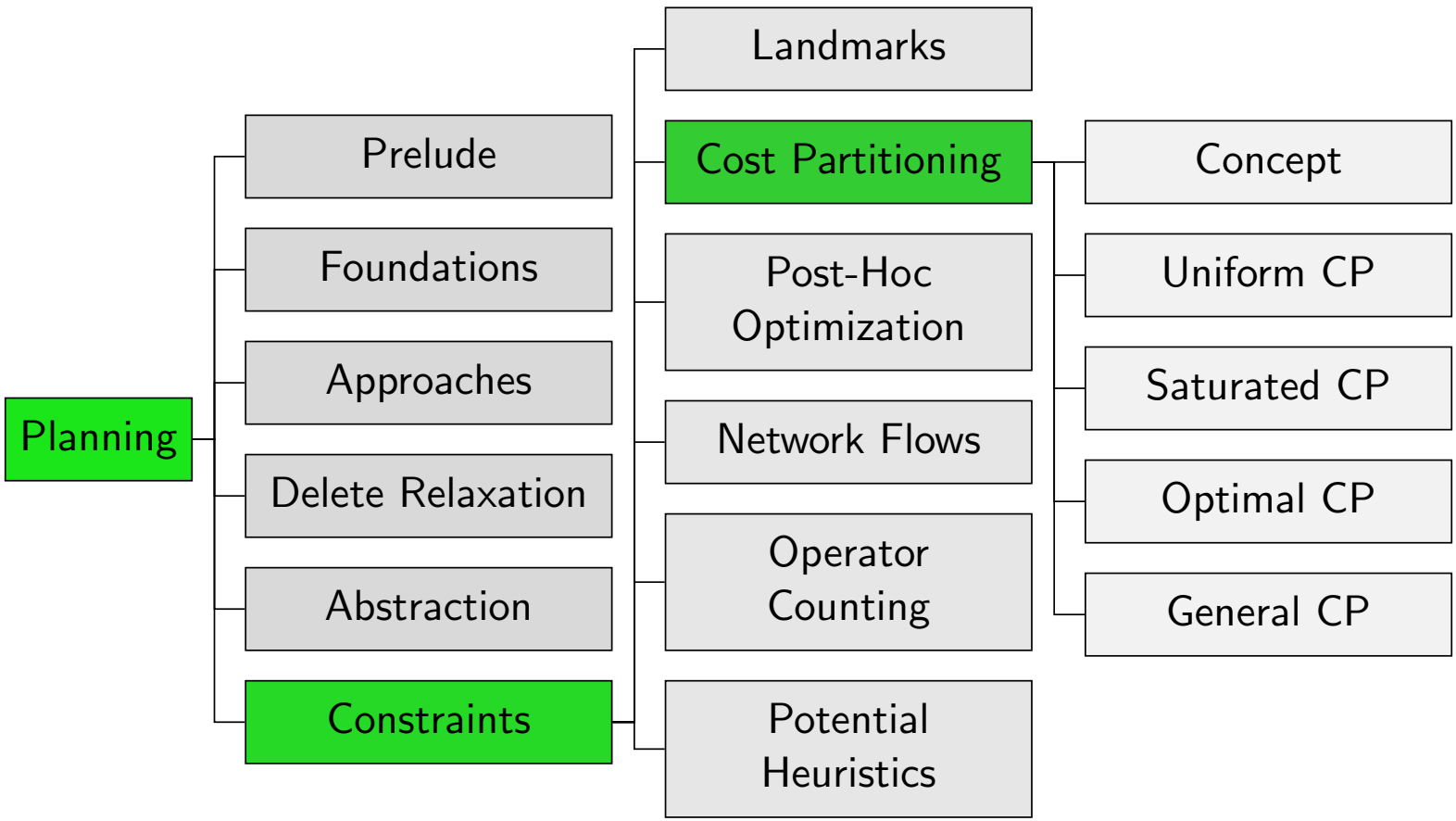
Malte Helmert and Gabriele Röger

Universität Basel

December 8, 2025

Planning and Optimization
December 8, 2025 — F7. Cost Partitioning

Content of the Course



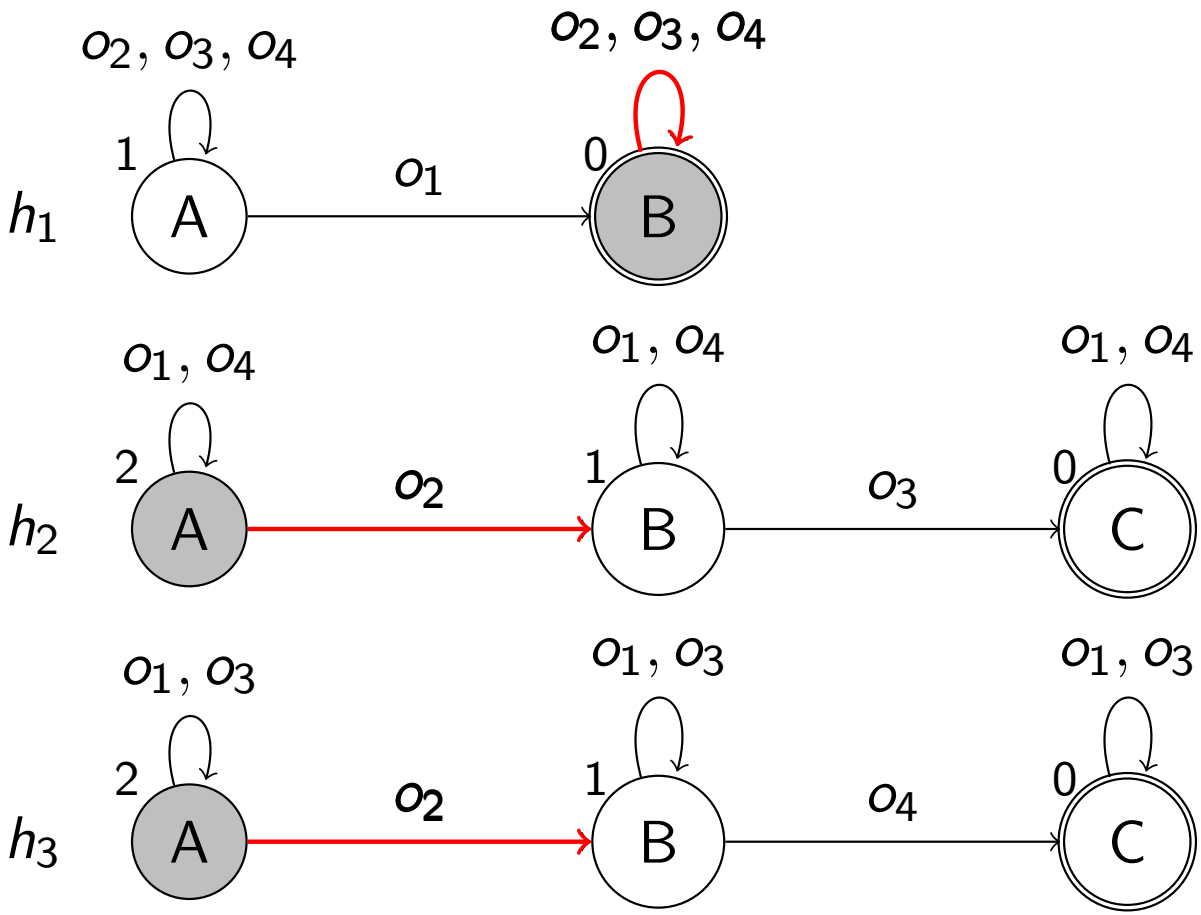
F7.1 Introduction

Exploiting Additivity

- Additivity allows to add up heuristic estimates admissibly. This gives better heuristic estimates than the maximum.
- For example, the canonical heuristic for PDBs sums up where addition is admissible (by an additivity criterion) and takes the maximum otherwise.
- Cost partitioning provides a more general additivity criterion, based on an adaption of the operator costs.

Combining Heuristics (In)admissibly: Example

Let $h = h_1 + h_2 + h_3$.



$\langle o_2, o_3, o_4 \rangle$ is a plan for $s = \langle B, A, A \rangle$ but $h(s) = 4$.
Heuristics h_2 and h_3 both account for the single application of o_2.

Solution: Cost Partitioning

The reason that h_2 and h_3 are not additive is because the cost of o_2 is considered in both.

Solution 1: We can ignore the cost of o_2 in all but one heuristic by setting its cost to 0 (e.g., $cost_3(o_2) = 0$).
This is a Zero-One cost partitioning.

Solution 2: We can equally distribute the cost of o_2 between the abstractions that use it (i.e. $cost_1(o_2) = 0$, $cost_2(o_2) = cost_3(o_2) = 0.5$).
This is a uniform cost partitioning.

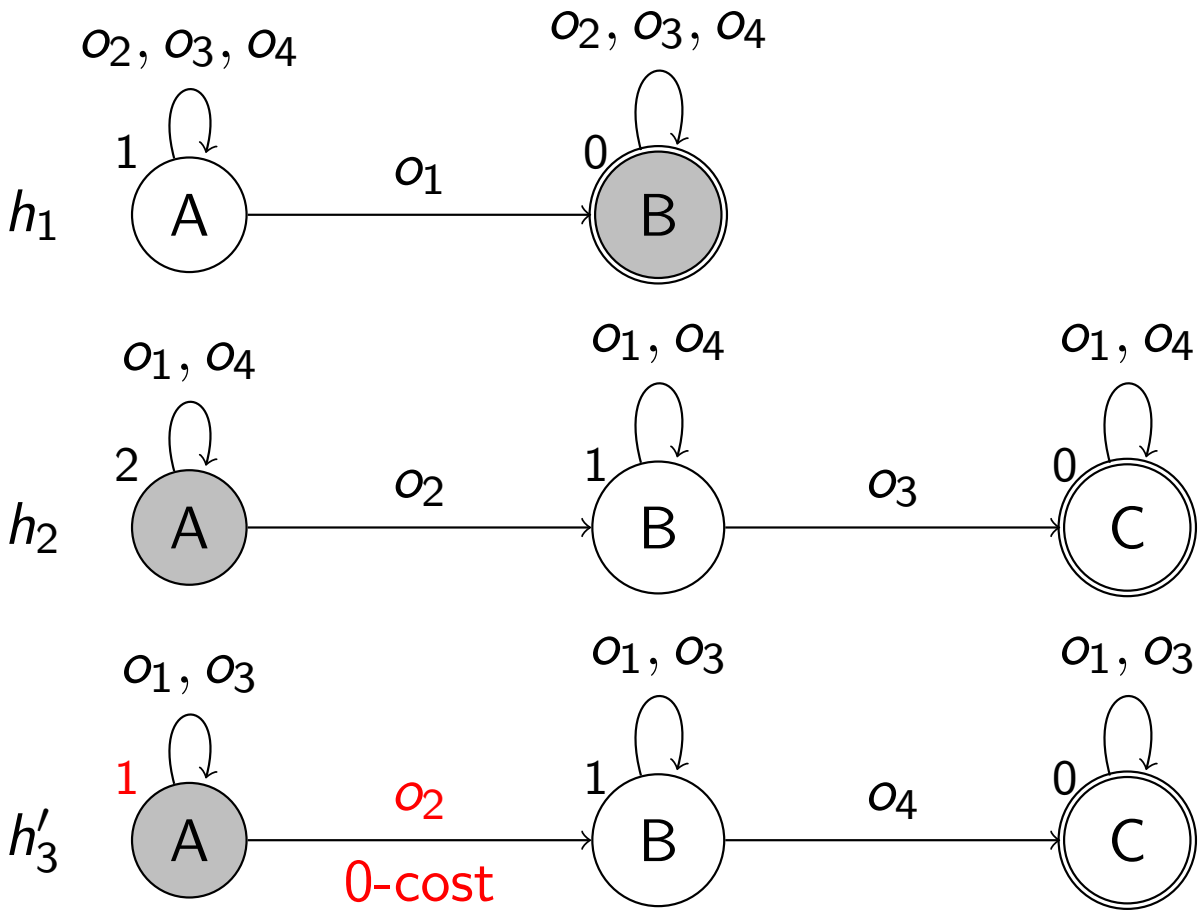
General solution: satisfy cost partitioning constraint

$$\sum_{i=1}^{n} cost_i(o) \leq cost(o) \text{ for all } o \in O$$

What about o_1, o_3 and o_4?

Combining Heuristics Admissibly: Example

Let $h' = h_1 + h_2 + h'_3$, where $h'_3 = h^{v_3}$ assuming $cost_3(o_2) = 0$.



$\langle o_2, o_3, o_4 \rangle$ is an optimal plan for $s = \langle B, A, A \rangle$ and $h'(s) = 3$ an admissible estimate.

Solution: Cost Partitioning

The reason that h_2 and h_3 are not additive is because the cost of o_2 is considered in both.

Solution 1: We can ignore the cost of o_2 in all but one heuristic by setting its cost to 0 (e.g., $cost_3(o_2) = 0$).
This is a Zero-One cost partitioning.

Solution 2: We can equally distribute the cost of o_2 between the abstractions that use it (i.e. $cost_1(o_2) = 0$, $cost_2(o_2) = cost_3(o_2) = 0.5$).
This is a uniform cost partitioning.

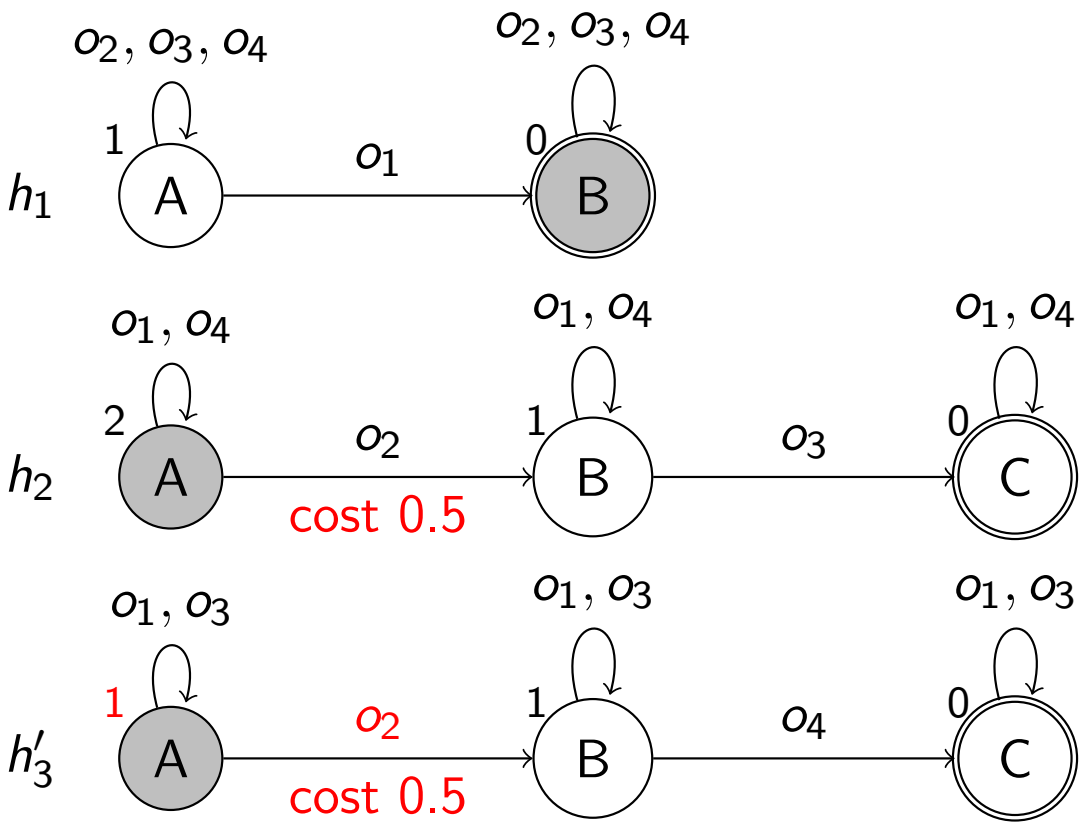
General solution: satisfy cost partitioning constraint

$$\sum_{i=1}^{n} cost_i(o) \leq cost(o) \text{ for all } o \in O$$

What about o_1, o_3 and o_4?

Combining Heuristics Admissibly: Example

Let $h' = h'_1 + h'_2 + h'_3$, where $h'_i = h^{v_i}$ assuming $cost_1(o_2) = 0$, $cost_2(o_2) = cost_3(o_2) = 0.5$.



$\langle o_2, o_3, o_4 \rangle$ is an optimal plan for $s = \langle B, A, A \rangle$ and $h'(s) = 0 + 1.5 + 1.5 = 3$ an admissible estimate.

Solution: Cost Partitioning

The reason that h_2 and h_3 are not additive is because the cost of o_2 is considered in both.

Solution 1: We can ignore the cost of o_2 in all but one heuristic by setting its cost to 0 (e.g., $cost_3(o_2) = 0$).
This is a Zero-One cost partitioning.

Solution 2: We can equally distribute the cost of o_2 between the abstractions that use it (i.e. $cost_1(o_2) = 0$, $cost_2(o_2) = cost_3(o_2) = 0.5$).
This is a uniform cost partitioning.

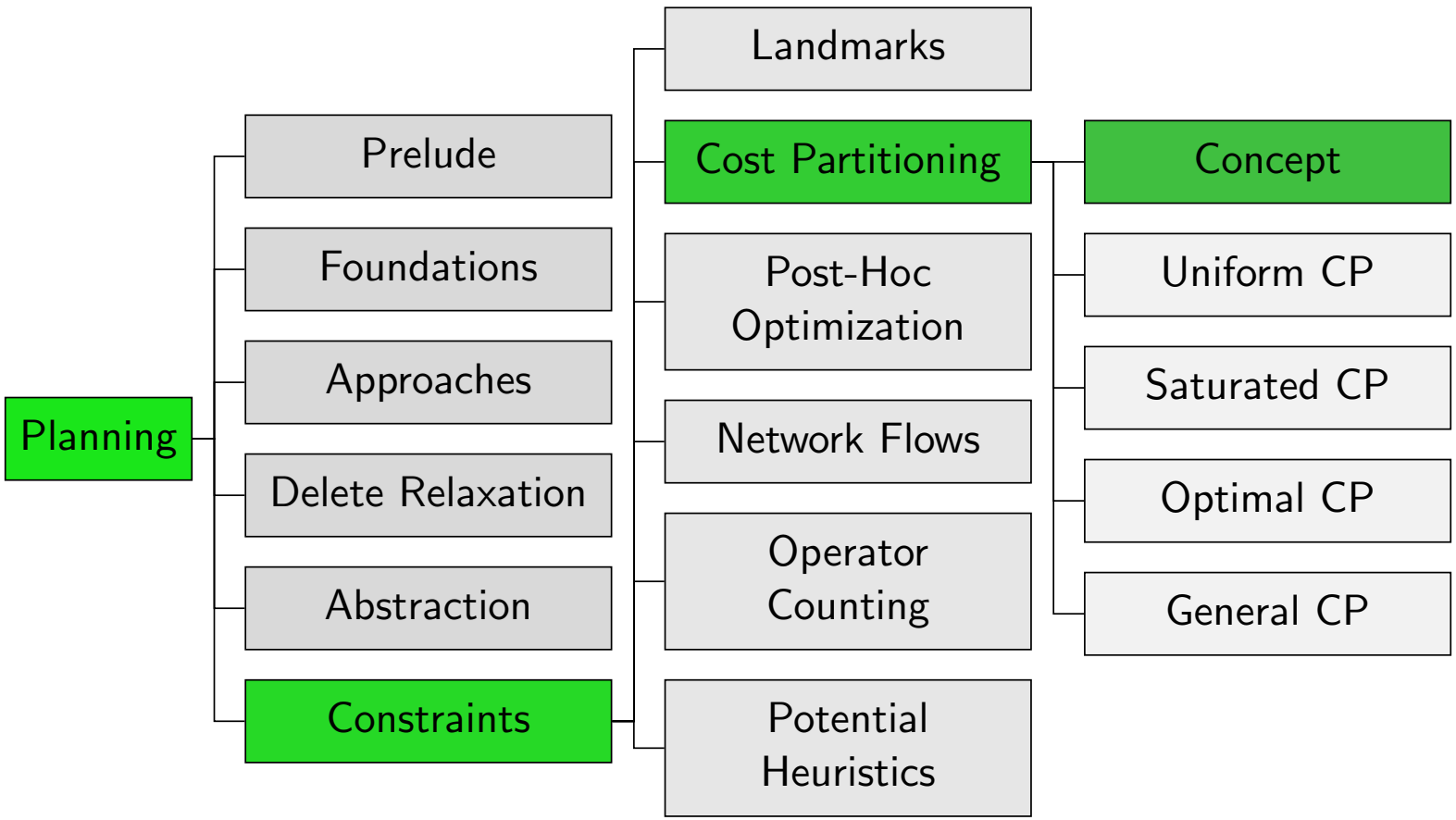
General solution: satisfy cost partitioning constraint

$$\sum_{i=1}^{n} cost_i(o) \leq cost(o) \text{ for all } o \in O$$

What about o_1, o_3 and o_4?

F7.2 Cost Partitioning

Content of the Course



Cost Partitioning

Definition (Cost Partitioning)

Let Π be a planning task with operators O.

A cost partitioning for Π is a tuple $\langle cost_1, \ldots, cost_n \rangle$, where

- $cost_i : O \to \mathbb{R}_0^+$ for $1 \leq i \leq n$ and
- $\sum_{i=1}^{n} cost_i(o) \leq cost(o)$ for all $o \in O$.

The cost partitioning induces a tuple $\langle \Pi_1, \ldots, \Pi_n \rangle$ of planning tasks, where each Π_i is identical to Π except that the cost of each operator o is $cost_i(o)$.

Cost Partitioning: Admissibility (1)

Theorem (Sum of Solution Costs is Admissible)

Let Π be a planning task, $\langle cost_1, \ldots, cost_n \rangle$ be a cost partitioning and $\langle \Pi_1, \ldots, \Pi_n \rangle$ be the tuple of induced tasks.

Then the sum of the solution costs of the induced tasks is an admissible heuristic for Π, i.e., $\sum_{i=1}^{n} h^*_{\Pi_i} \leq h^*_{\Pi}$.

Cost Partitioning: Admissibility (2)

Proof of Theorem.

If there is no plan for state s of Π, both sides are ∞. Otherwise, let $\pi = \langle o_1, \ldots, o_m \rangle$ be an optimal plan for s. Then

$$\begin{aligned}
\sum_{i=1}^{n} h^*_{\Pi_i}(s) &\leq \sum_{i=1}^{n} \sum_{j=1}^{m} cost_i(o_j) && (\pi \text{ plan in each } \Pi_i) \\
&= \sum_{j=1}^{m} \sum_{i=1}^{n} cost_i(o_j) && (\text{comm./ass. of sum}) \\
&\leq \sum_{j=1}^{m} cost(o_j) && (\text{cost partitioning}) \\
&= h^*_{\Pi}(s) && (\pi \text{ optimal plan in } \Pi)
\end{aligned}$$

□

Cost Partitioning Preserves Admissibility

In the rest of the chapter, we write h_Π to denote heuristic h evaluated on task Π.

Corollary (Sum of Admissible Estimates is Admissible)

Let Π be a planning task and let $\langle \Pi_1, \ldots, \Pi_n \rangle$ be induced by a cost partitioning.

For admissible heuristics $h_1, \ldots, h_n$, the sum $h(s) = \sum_{i=1}^{n} h_{i,\Pi_i}(s)$ is an admissible estimate for s in Π.

Cost Partitioning Preserves Consistency

Theorem (Cost Partitioning Preserves Consistency)

Let Π be a planning task and let $\langle \Pi_1, \ldots, \Pi_n \rangle$ be induced by a cost partitioning $\langle cost_1, \ldots, cost_n \rangle$.

If $h_1, \ldots, h_n$ are consistent heuristics then $h = \sum_{i=1}^{n} h_{i,\Pi_i}$ is a consistent heuristic for Π.

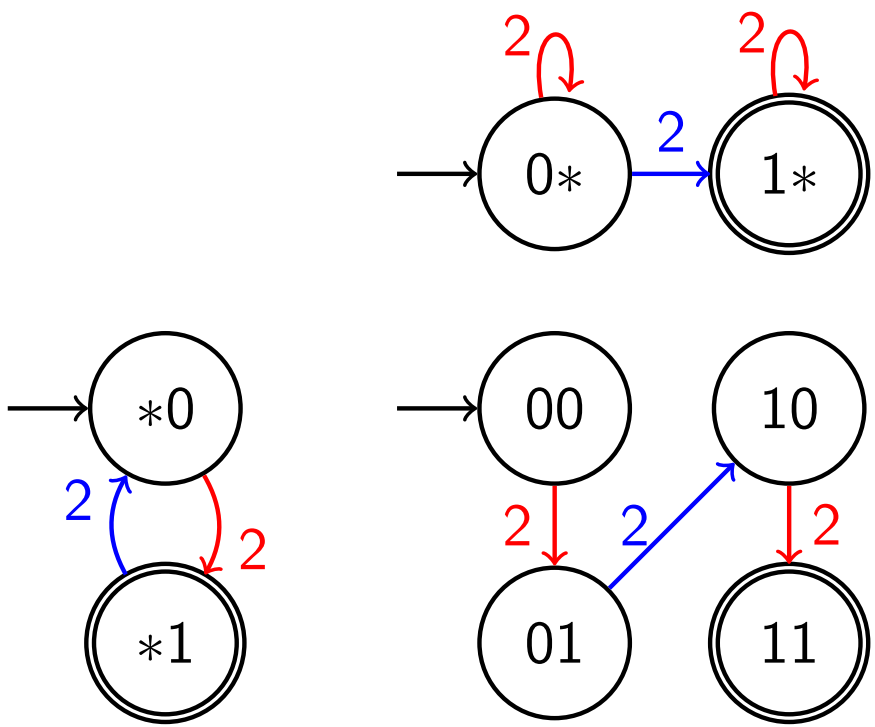
Proof.

Let o be an operator that is applicable in state s.

$$h(s) = \sum_{i=1}^{n} h_{i,\Pi_i}(s) \leq \sum_{i=1}^{n} (cost_i(o) + h_{i,\Pi_i}(s[\![o]\!]))$$
$$= \sum_{i=1}^{n} cost_i(o) + \sum_{i=1}^{n} h_{i,\Pi_i}(s[\![o]\!]) \leq cost(o) + h(s[\![o]\!])$$

Cost Partitioning: Example

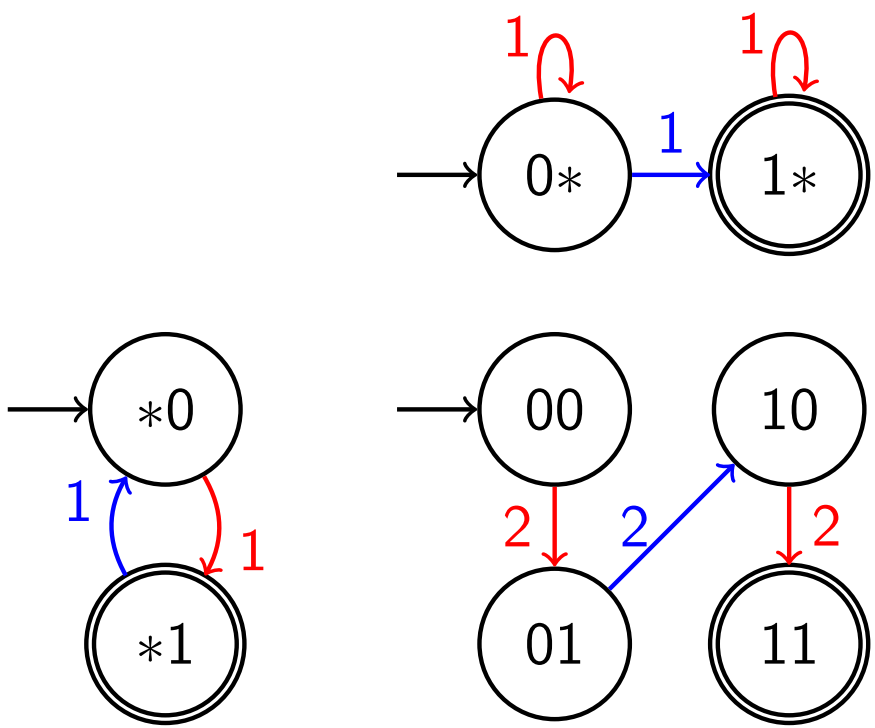
Example (No Cost Partitioning)



Heuristic value: $\max\{2, 2\} = 2$

Cost Partitioning: Example

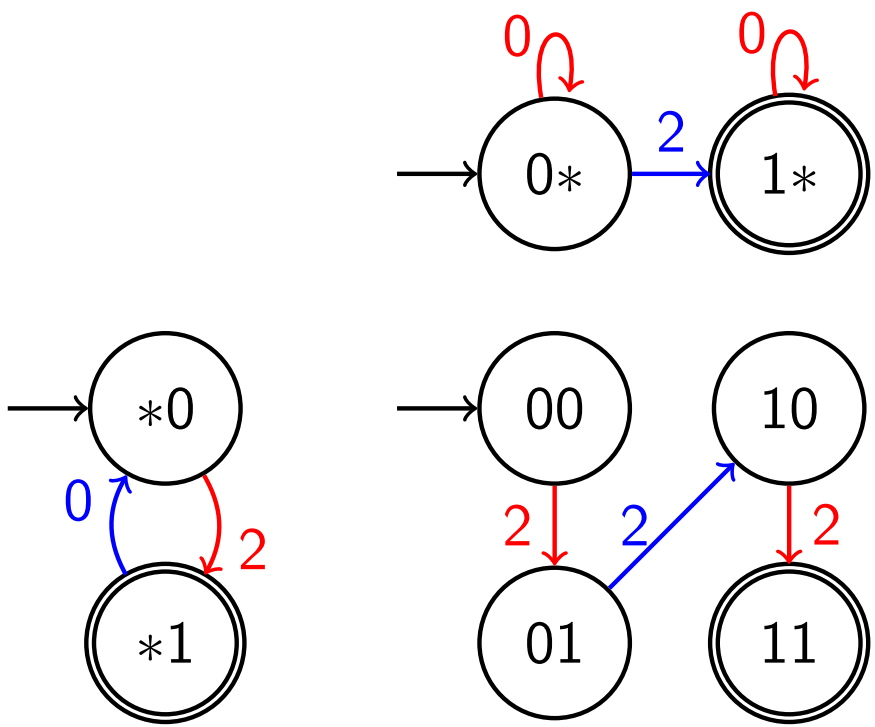
Example (Cost Partitioning 1)



Heuristic value: $1 + 1 = 2$

Cost Partitioning: Example

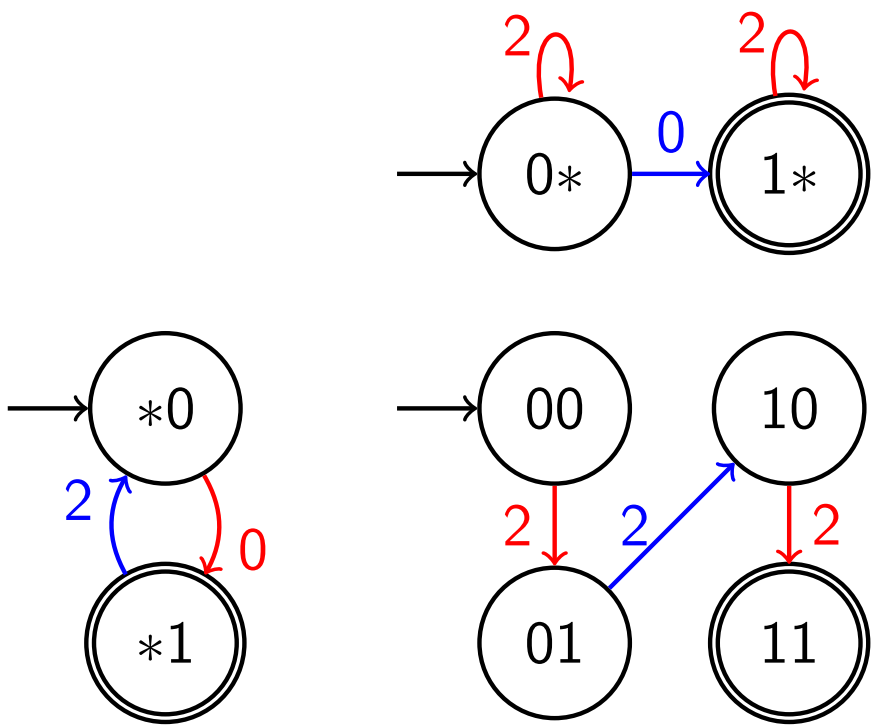
Example (Cost Partitioning 2)



Heuristic value: $2 + 2 = 4$

Cost Partitioning: Example

Example (Cost Partitioning 3)



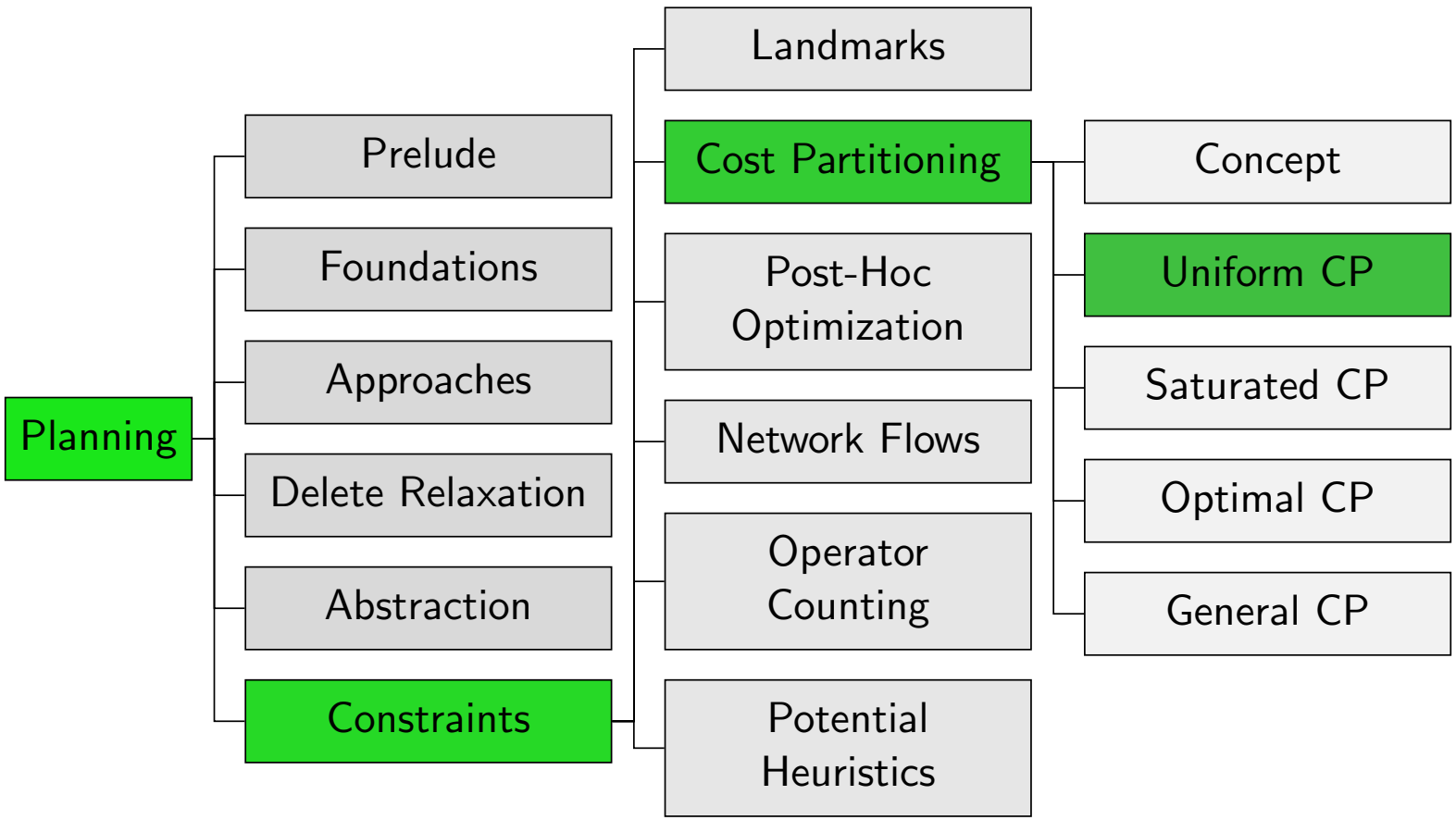
Heuristic value: $0 + 0 = 0$

Cost Partitioning: Quality

- $h(s) = h_{1,\Pi_1}(s) + \cdots + h_{n,\Pi_n}(s)$
  can be better or worse than any $h_{i,\Pi}(s)$
  $\rightarrow$ depending on cost partitioning
- strategies for defining cost-functions
  - uniform (now)
  - zero-one
  - saturated (afterwards)
  - optimal (next chapter)

F7.3 Uniform Cost Partitioning

Content of the Course



Idea

- Principal idea: Distribute the cost of each operator equally (= uniformly) among all heuristics.
- But: Some heuristics do only account for the cost of certain operators and the cost of other operators does not affect the heuristic estimate. For example:
  - a disjunctive action landmark accounts for the contained operators,
  - a PDB heuristic accounts for all operators affecting the variables in the pattern.

$\Rightarrow$ Distribute the cost of each operator uniformly among all heuristics that account for this operator.

Example: Uniform Cost Partitioning for Landmarks

- For disjunctive action landmark L of state s in task Π', let $h_{L,\Pi'}(s)$ be the cost of L in Π'.
- Then $h_{L,\Pi'}(s)$ is admissible (in Π').
- Consider set $\mathcal{L} = \{L_1, \ldots, L_n\}$ of disjunctive action landmarks for state s of task Π.
- Use cost partitioning $\langle cost_{L_1}, \ldots, cost_{L_n} \rangle$, where

$$cost_{L_i}(o) = \begin{cases} cost(o)/|\{L \in \mathcal{L} \mid o \in L\}| & \text{if } o \in L_i \\ 0 & \text{otherwise} \end{cases}$$

- Let $\langle \Pi_{L_1}, \ldots, \Pi_{L_n} \rangle$ be the tuple of induced tasks.
- $h(s) = \sum_{i=1}^{n} h_{L_i,\Pi_{L_i}}(s)$ is an admissible estimate for s in Π.
- h is the uniform cost partitioning heuristic for landmarks.

Example: Uniform Cost Partitioning for Landmarks

Definition (Uniform Cost Partitioning Heuristic for Landmarks)

Let $\mathcal{L}$ be a set of disjunctive action landmarks.

The uniform cost partitioning heuristic $h^{\mathrm{UCP}}(\mathcal{L})$ is defined as

$$h^{\mathrm{UCP}}(\mathcal{L}) = \sum_{L \in \mathcal{L}} \min_{o \in L} c'(o) \text{ with}$$

$$c'(o) = cost(o)/|\{L \in \mathcal{L} \mid o \in L\}|.$$

Example: Uniform Cost Partitioning for Landmarks

Example

Given disjunctive action landmarks
$L_1 = \{o_1, o_3\}$, $L_2 = \{o_1, o_2, o_4\}$, $L_3 = \{o_1, o_4, o_5\}$

with operator cost function
$c(o_1) = 6$, $c(o_2) = 4$, $c(o_3) = 1$, $c(o_4) = 6$, $c(o_5) = 3$

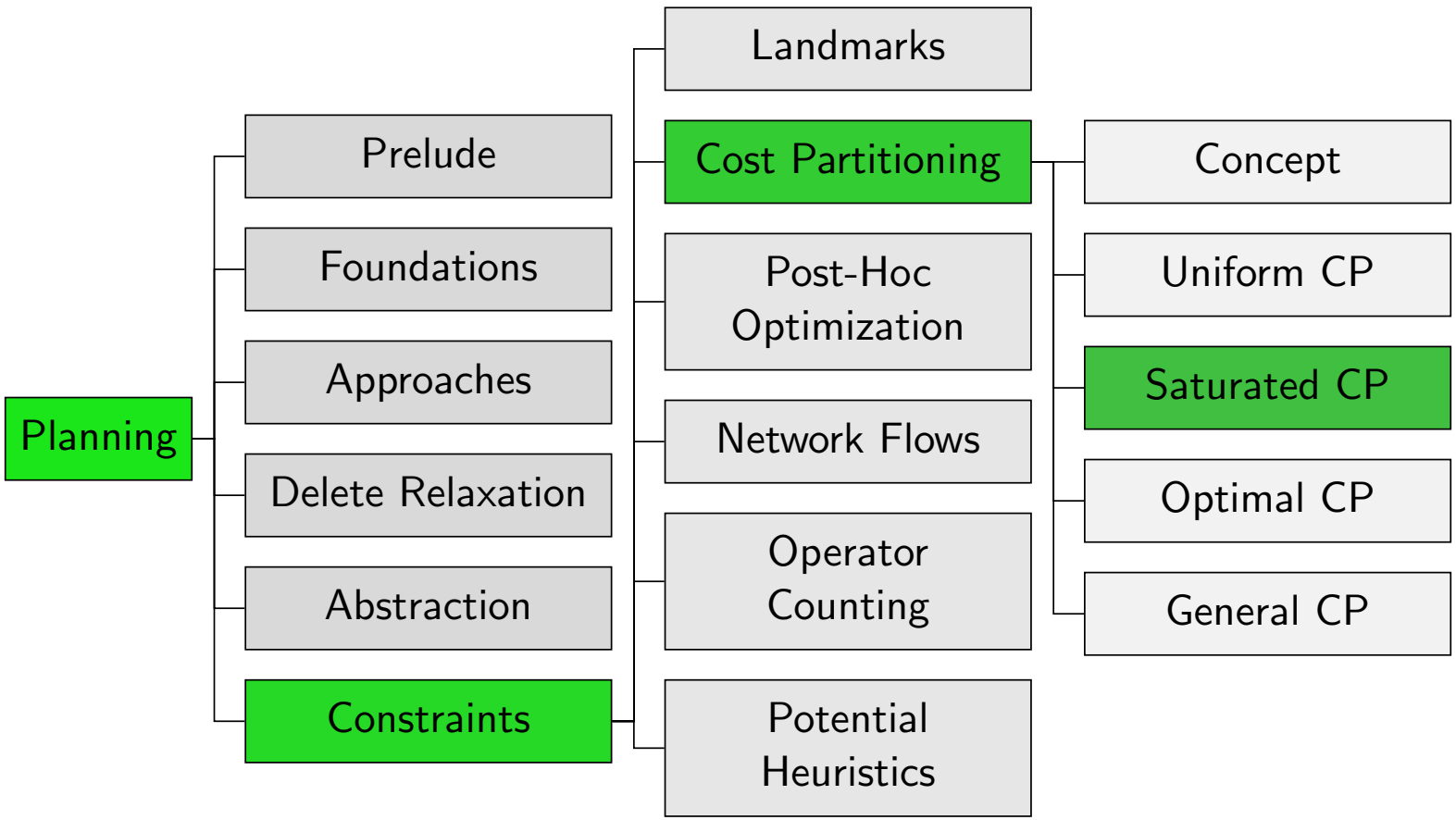
UCP for landmarks uses adapted costs
$c'(o_1) = 2$, $c'(o_2) = 4$, $c'(o_3) = 1$, $c'(o_4) = 3$, $c'(o_5) = 3$

with resulting heuristic estimate
$h^{\mathrm{UCP}}(\{L_1, L_2, L_3\}) = 1 + 2 + 2 = 5.$

(MHS heuristic estimate: 6)

F7.4 Saturated Cost Partitioning

Content of the Course



Idea

Heuristics do not always "need" all operator costs

- Pick a heuristic and use minimum costs preserving all estimates
- Continue with remaining cost until all heuristics were picked

Saturated cost partitioning (SCP) currently offers the best tradeoff between computation time and heuristic guidance in practice.

Saturated Cost Function

Definition (Saturated Cost Function)

Let Π be a planning task and h be a heuristic.
A cost function scf is saturated for h and *cost* if

1. $\text{scf}(o) \leq \mathit{cost}(o)$ for all operators o and
2. $h_{\Pi_{\text{scf}}}(s) = h_{\Pi}(s)$ for all states s,
   where Π_{scf} is Π with cost function scf.

Minimal Saturated Cost Function

For abstractions, there exists a unique minimal saturated cost function (MSCF).

Definition (MSCF for Abstractions)

Let Π be a planning task and α be an abstraction heuristic.
The minimal saturated cost function for α is

$$\mathrm{mscf}(o) = \max\Big(\max_{\alpha(s) \xrightarrow{o} \alpha(t)} h^\alpha(s) - h^\alpha(t), 0 \Big)$$

Algorithm

Saturated Cost Partitioning: Seipp & Helmert (2014)

Iterate:

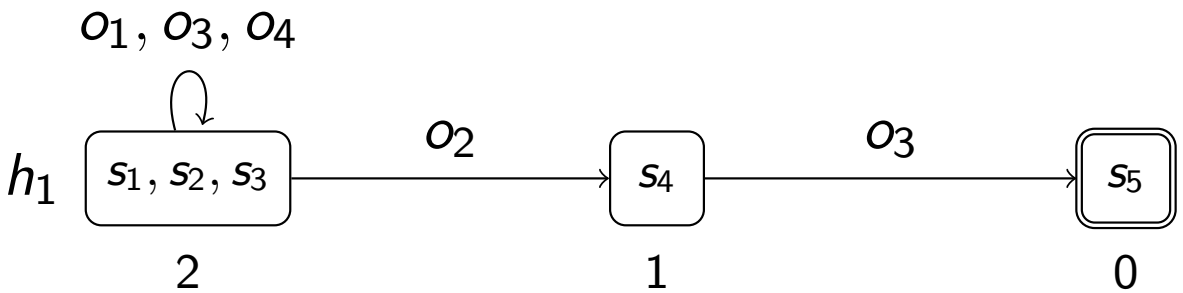
1. Pick a heuristic h_i that hasn't been picked before. Terminate if none is left.
2. Compute h_i given current *cost*
3. Compute an (ideally minimal) saturated cost function scf_i for h_i
4. Decrease $\mathit{cost}(o)$ by $\text{scf}_i(o)$ for all operators o

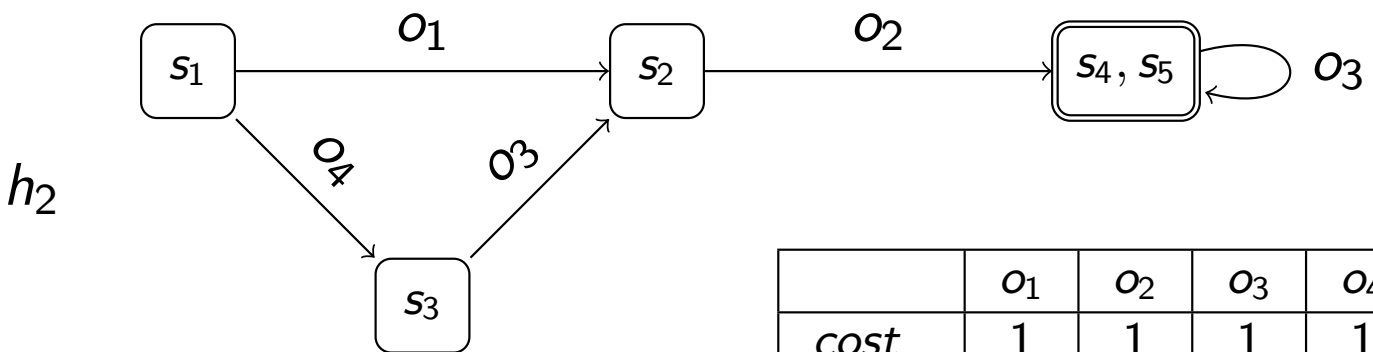
$\langle \text{scf}_1, \ldots, \text{scf}_n \rangle$ is a saturated cost partitioning (SCP) for $\langle h_1, \ldots, h_n \rangle$ (in pick order)

Example

Consider the abstraction heuristics h_1 and h_2

3. Compute minimal saturated cost function mscf_i for h_i



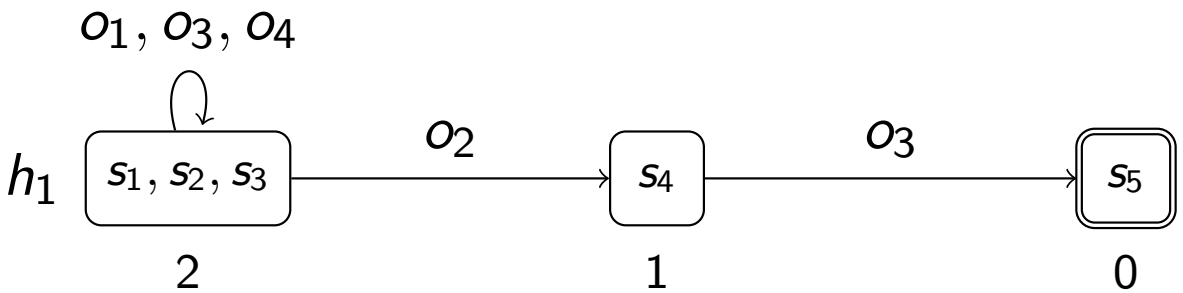


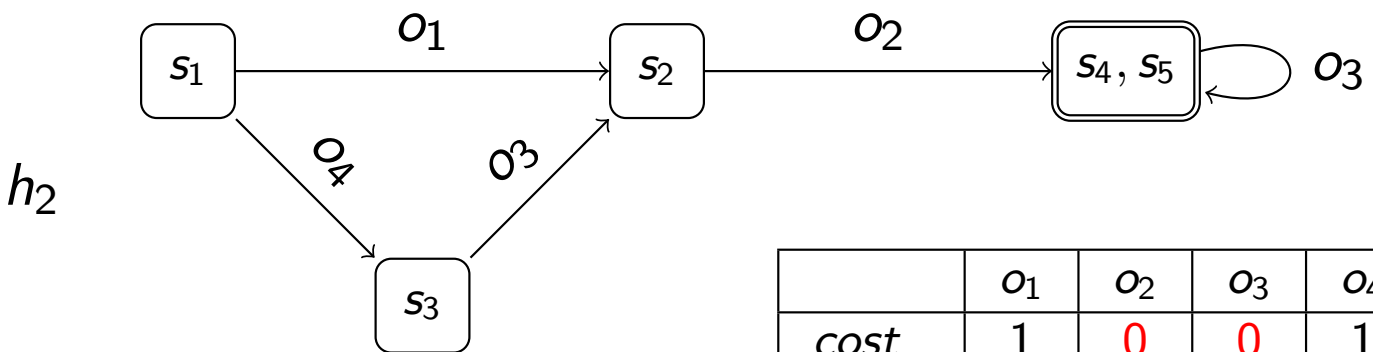
	o_1	o_2	o_3	o_4
cost	1	1	1	1
mscf_1	0	1	1	0

Example

Consider the abstraction heuristics h_1 and h_2

4. Decrease $cost(o)$ by $\text{mscf}_i(o)$ for all operators o



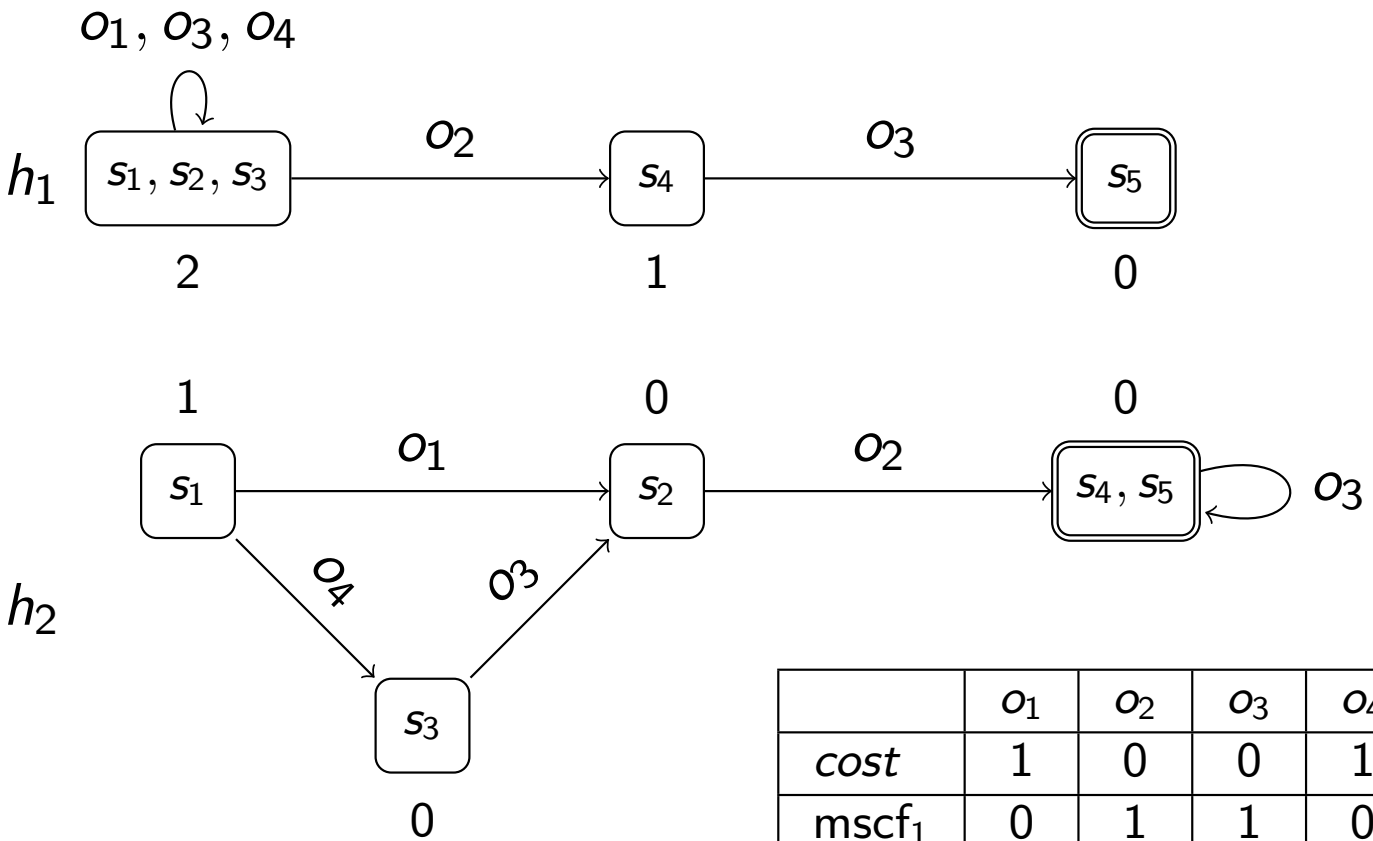


	o_1	o_2	o_3	o_4
$cost$	1	0	0	1
mscf_1	0	1	1	0

Example

Consider the abstraction heuristics h_1 and h_2

3. Compute minimal saturated cost function mscf_i for h_i

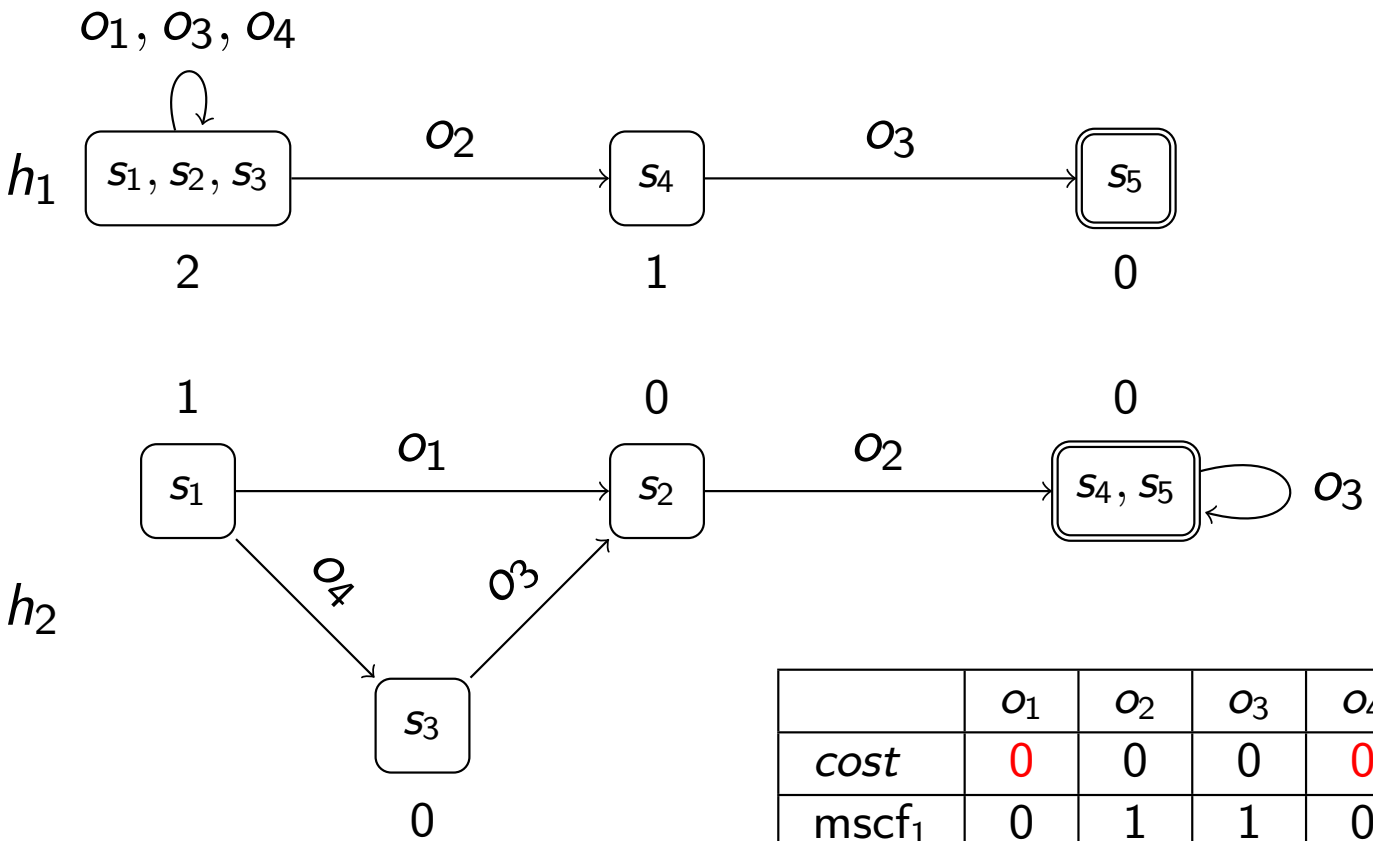


	o_1	o_2	o_3	o_4
cost	1	0	0	1
mscf_1	0	1	1	0
mscf_2	1	0	0	1

Example

Consider the abstraction heuristics h_1 and h_2

4. Decrease $cost(o)$ by $\text{mscf}_i(o)$ for all operators o

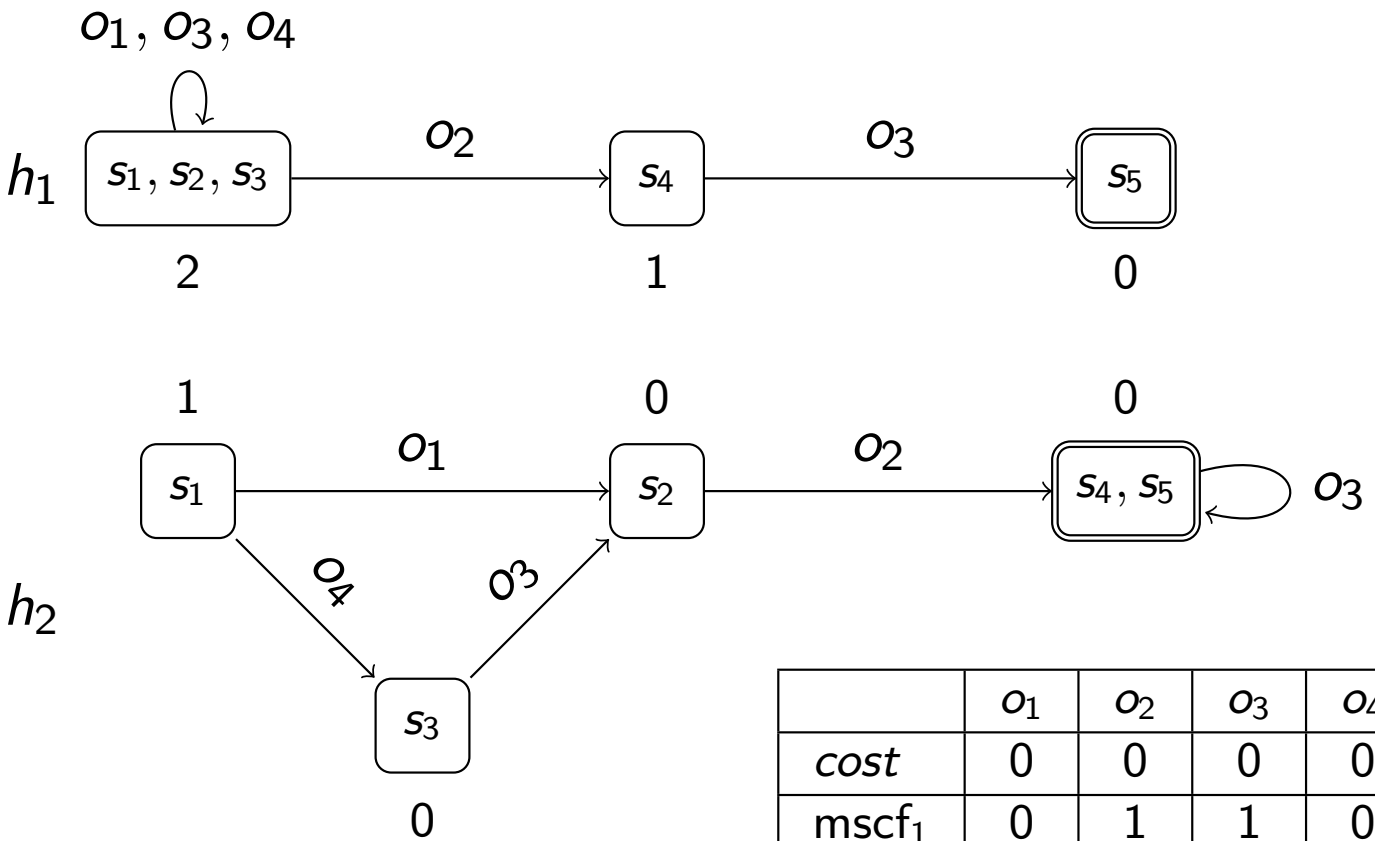


	o_1	o_2	o_3	o_4
$cost$	0	0	0	0
mscf_1	0	1	1	0
mscf_2	1	0	0	1

Example

Consider the abstraction heuristics h_1 and h_2

1. Pick a heuristic h_i. Terminate if none is left.



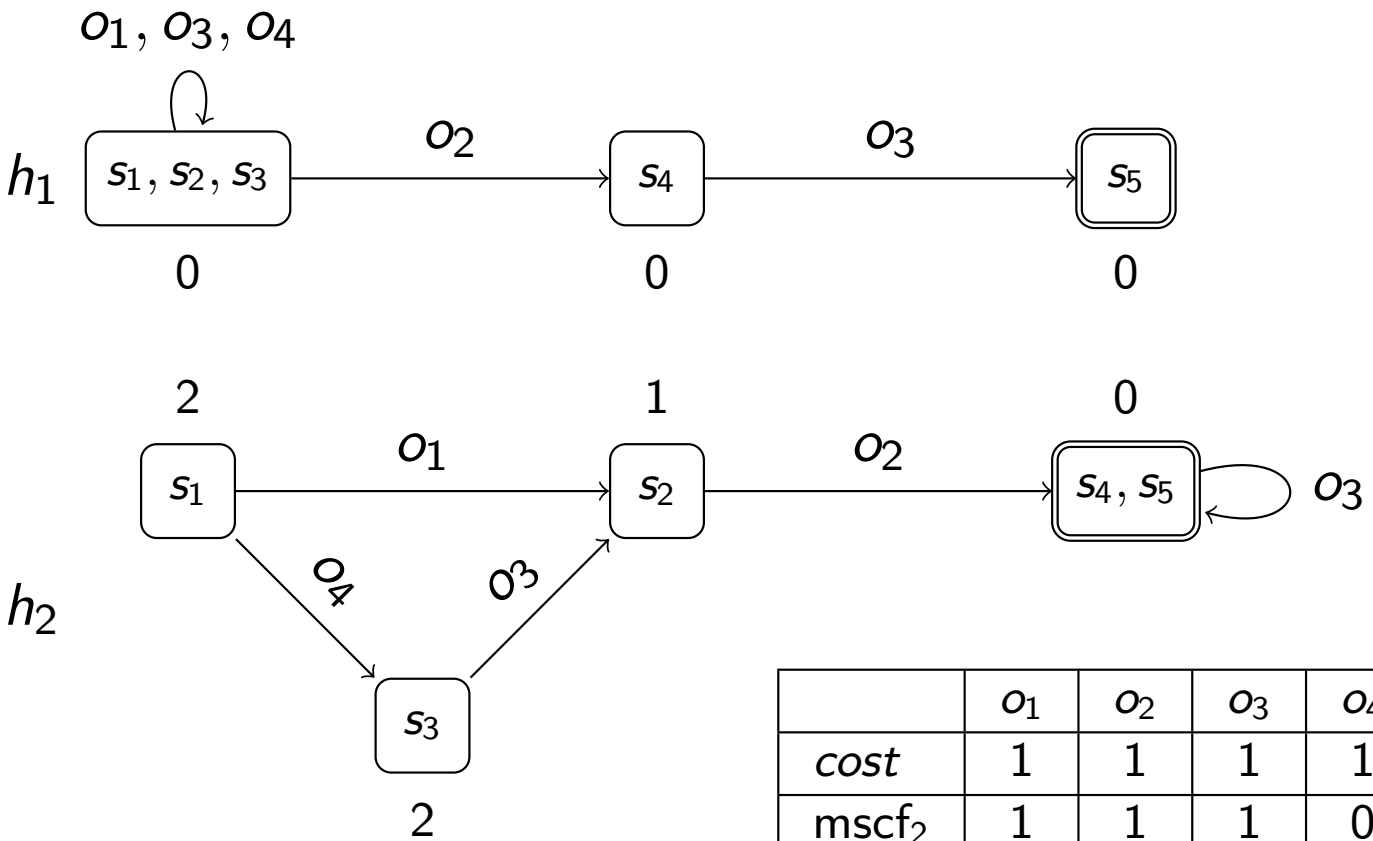
	o_1	o_2	o_3	o_4
cost	0	0	0	0
mscf_1	0	1	1	0
mscf_2	1	0	0	1

Influence of Selected Order

- quality highly susceptible to selected order
- there are almost always orders where SCP performs much better than uniform or zero-one cost partitioning
- but there are also often orders where SCP performs worse

Saturated Cost Partitioning: Order

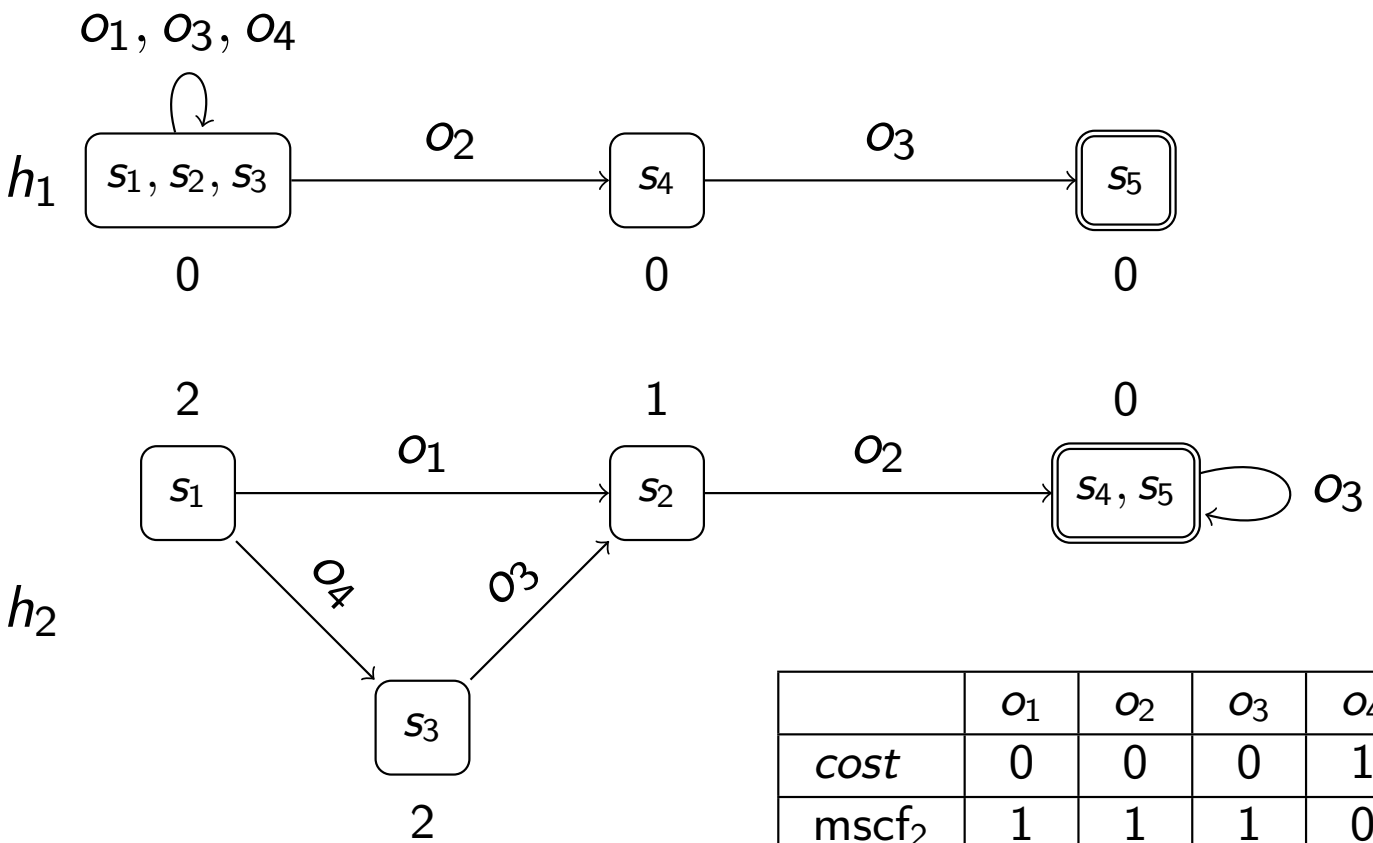
Consider the abstraction heuristics h_1 and h_2



	o_1	o_2	o_3	o_4
cost	1	1	1	1
mscf_2	1	1	1	0

Saturated Cost Partitioning: Order

Consider the abstraction heuristics h_1 and h_2



	o_1	o_2	o_3	o_4
cost	0	0	0	1
mscf_2	1	1	1	0
mscf_1	0	0	0	0

Influence of Selected Order

- quality highly susceptible to selected order
- there are almost always orders where SCP performs much better than uniform or zero-one cost partitioning
- but there are also often orders where SCP performs worse

Maximizing over multiple orders good solution in practice

SCP for Disjunctive Action Landmarks

For disjunctive action landmarks we also know how to compute a minimal saturated cost function:

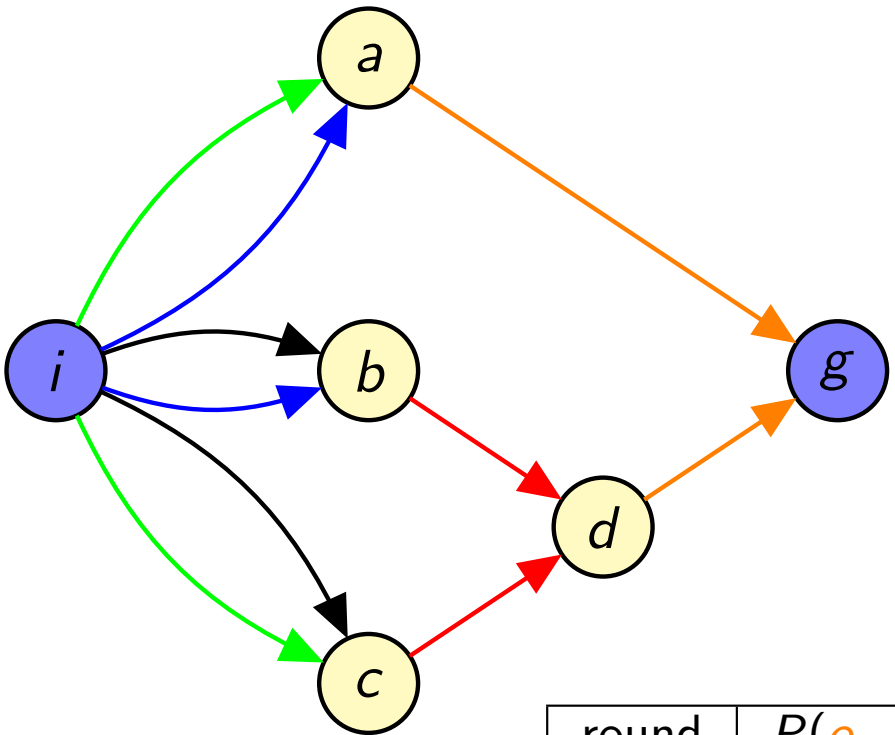
Definition (MSCF for Disjunctive Action Landmark)

Let Π be a planning task and $\mathcal{L}$ be a disjunctive action landmark. The minimal saturated cost function for $\mathcal{L}$ is

$$\text{mscf}(o) = \begin{cases} \min_{o \in \mathcal{L}} cost(o) & \text{if } o \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

Does this look familiar?

Reminder: LM-Cut



$$o_{\text{blue}} = \langle \{i\}, \{a, b\}, \{\}, 4 \rangle$$
$$o_{\text{green}} = \langle \{i\}, \{a, c\}, \{\}, 5 \rangle$$
$$o_{\text{black}} = \langle \{i\}, \{b, c\}, \{\}, 3 \rangle$$
$$o_{\text{red}} = \langle \{b, c\}, \{d\}, \{\}, 2 \rangle$$
$$o_{\text{orange}} = \langle \{a, d\}, \{g\}, \{\}, 0 \rangle$$

round	$P(o_{\text{orange}})$	$P(o_{\text{red}})$	landmark	cost
1	d	b	$\{o_{\text{red}}\}$	2
2	a	b	$\{o_{\text{green}}, o_{\text{blue}}\}$	4
3	d	c	$\{o_{\text{green}}, o_{\text{black}}\}$	1
			$h^{\text{LM-cut}}(I)$	7

SCP for Disjunctive Action Landmarks

Same algorithm can be used for disjunctive action landmarks, where we also have a minimal saturated cost function.

Definition (MSCF for Disjunctive Action Landmark)

Let Π be a planning task and $\mathcal{L}$ be a disjunctive action landmark. The minimal saturated cost function for $\mathcal{L}$ is

$$\text{mscf}(o) = \begin{cases} \min_{o \in \mathcal{L}} cost(o) & \text{if } o \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

Does this look familiar?

LM-Cut computes SCP over disjunctive action landmarks

F7.5 Summary

Summary

- Cost partitioning allows to admissibly add up estimates of several heuristics.
- This can be better or worse than the best individual heuristic on the original problem, depending on the cost partitioning.
- Uniform cost partitioning distributes the cost of each operator uniformly among all heuristics that account for it.
- Saturated cost partitioning offers a good tradeoff between computation time and heuristic guidance.
- LM-Cut computes a SCP over disjunctive action landmarks.

Planning and Optimization

F8. Optimal and General Cost-Partitioning

Malte Helmert and Gabriele Röger

Universität Basel

December 8, 2025

Planning and Optimization

December 8, 2025 — F8. Optimal and General Cost-Partitioning

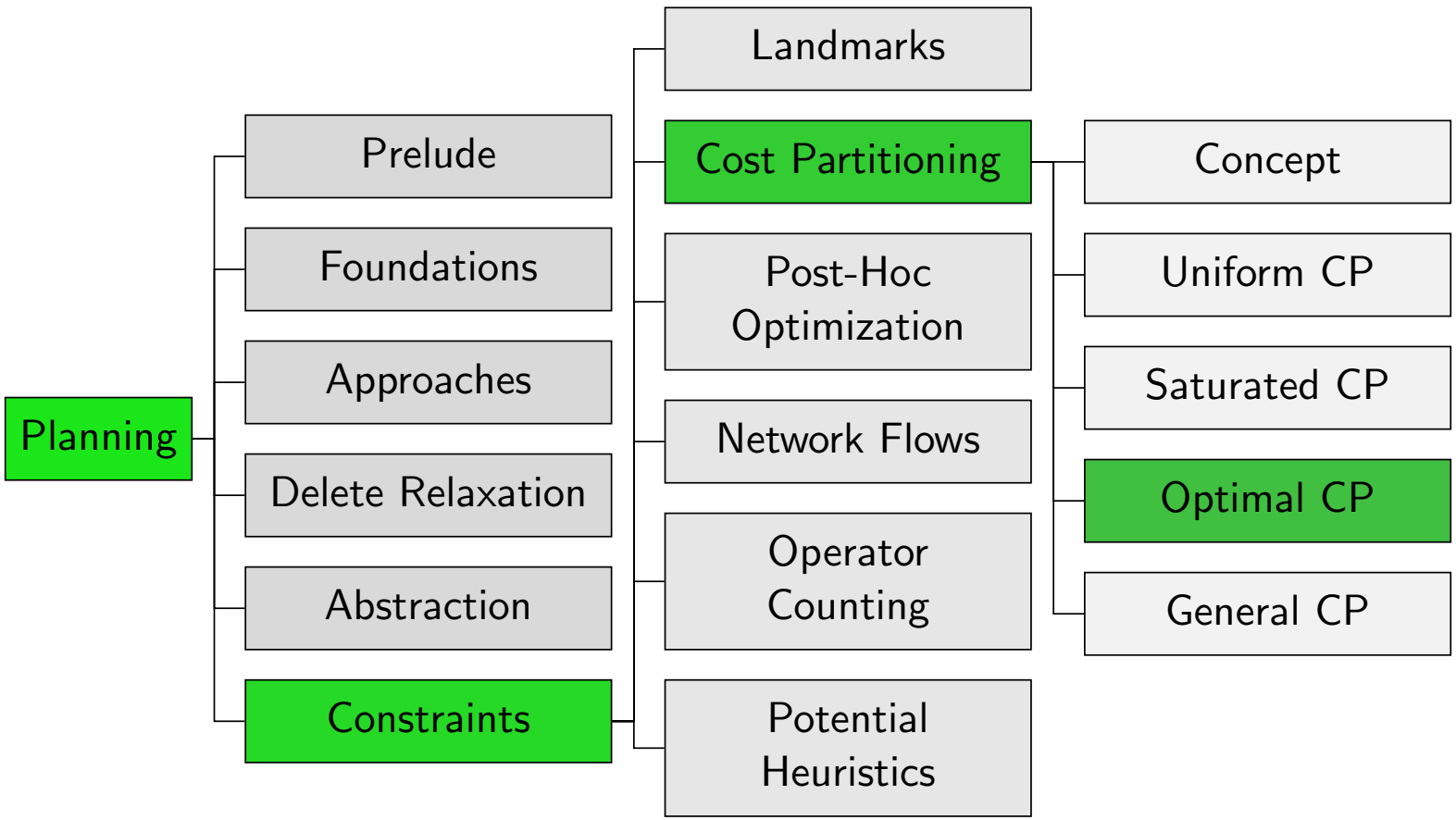
F8.1 Optimal Cost Partitioning

F8.2 General Cost Partitioning

F8.3 Summary

F8.1 Optimal Cost Partitioning

Content of the Course



Optimal Cost Partitioning: General Approach

- Can we find a better cost partitioning than with the uniform or saturation strategy? Even an optimal one?
- Idea: exploit linear programming
  - Use variables for cost of each operator in each task copy
  - Express heuristic values with linear constraints
  - Maximize sum of heuristic values subject to these constraints

LPs known for

- abstraction heuristics (not covered in this course)
- disjunctive action landmarks (now)

Optimal Cost Partitioning for Landmarks: Basic Version

- Use an LP that covers the heuristic computation and the cost partitioning.
- LP variable $C_{L,o}$ for cost of operator o in induced task for disjunctive action landmark L (cost partitioning)
- LP variable Cost_L for cost of disjunctive action landmark L in induced task (value of individual heuristics)

Optimal Cost Partitioning for Landmarks: Basic LP

Variables

Non-negative variable Cost_L for each disj. action landmark $L \in \mathcal{L}$
Non-negative variable $\text{C}_{L,o}$ for each $L \in \mathcal{L}$ and operator o

Objective

Maximize $\sum_{L \in \mathcal{L}} \text{Cost}_L$

Subject to

$$\sum_{L \in \mathcal{L}} \text{C}_{L,o} \leq \mathit{cost}(o) \qquad \text{for all operators } o$$

$$\text{Cost}_L \leq \text{C}_{L,o} \qquad \text{for all } L \in \mathcal{L} \text{ and } o \in L$$

Optimal Cost Partitioning for Landmarks: Improved

- Observation: Explicit variables for cost partitioning not necessary.
- Use implicitly $cost_L(o) = \text{Cost}_L$ for all $o \in L$ and 0 otherwise.

Optimal Cost Partitioning for Landmarks: Improved LP

Variables

Non-negative variable Cost_L for each disj. action landmark $L \in \mathcal{L}$

Objective

Maximize $\sum_{L \in \mathcal{L}} \text{Cost}_L$

Subject to

$$\sum_{L \in \mathcal{L}: o \in L} \text{Cost}_L \leq cost(o) \quad \text{for all operators } o$$

Example (1)

Example

Let Π be a planning task with operators $o_1, \ldots, o_4$ and $cost(o_1) = 3, cost(o_2) = 4, cost(o_3) = 5$ and $cost(o_4) = 0$.
Let the following be disjunctive action landmarks for Π:

$$\mathcal{L}_1 = \{o_4\}$$
$$\mathcal{L}_2 = \{o_1, o_2\}$$
$$\mathcal{L}_3 = \{o_1, o_3\}$$
$$\mathcal{L}_4 = \{o_2, o_3\}$$

Example (2)

Example

Maximize $\text{Cost}_{\mathcal{L}_1} + \text{Cost}_{\mathcal{L}_2} + \text{Cost}_{\mathcal{L}_3} + \text{Cost}_{\mathcal{L}_4}$ subject to

$$
\begin{aligned}
[o_1] \quad & \text{Cost}_{\mathcal{L}_2} + \text{Cost}_{\mathcal{L}_3} \leq 3 \\
[o_2] \quad & \text{Cost}_{\mathcal{L}_2} + \text{Cost}_{\mathcal{L}_4} \leq 4 \\
[o_3] \quad & \text{Cost}_{\mathcal{L}_3} + \text{Cost}_{\mathcal{L}_4} \leq 5 \\
[o_4] \quad & \text{Cost}_{\mathcal{L}_1} \leq 0 \\
& \text{Cost}_{\mathcal{L}_i} \geq 0 \quad \text{for } i \in \{1, 2, 3, 4\}
\end{aligned}
$$

Optimal Cost Partitioning for Landmarks (Dual view)

Variables

Non-negative variable Applied_o for each operator o

Objective

Minimize $\sum_o \text{Applied}_o \cdot cost(o)$

Subject to

$$\sum_{o \in L} \text{Applied}_o \geq 1 \text{ for all landmarks } L$$

Minimize "plan cost" with all landmarks satisfied.

Example: Dual View

Example (Optimal Cost Partitioning: Dual View)

Minimize $3\text{Applied}_{o_1} + 4\text{Applied}_{o_2} + 5\text{Applied}_{o_3}$ subject to

$$\begin{aligned}
\text{Applied}_{o_4} &\geq 1 \\
\text{Applied}_{o_1} + \text{Applied}_{o_2} &\geq 1 \\
\text{Applied}_{o_1} + \text{Applied}_{o_3} &\geq 1 \\
\text{Applied}_{o_2} + \text{Applied}_{o_3} &\geq 1 \\
\text{Applied}_{o_i} &\geq 0 \quad \text{for } i \in \{1, 2, 3, 4\}
\end{aligned}$$

This is equal to the LP relaxation of the MHS heuristic

Reminder: LP Relaxation of MHS heuristic

Example (Minimum Hitting Set)

$$\text{minimize} \quad 3X_{o_1} + 4X_{o_2} + 5X_{o_3} \quad \text{subject to}$$

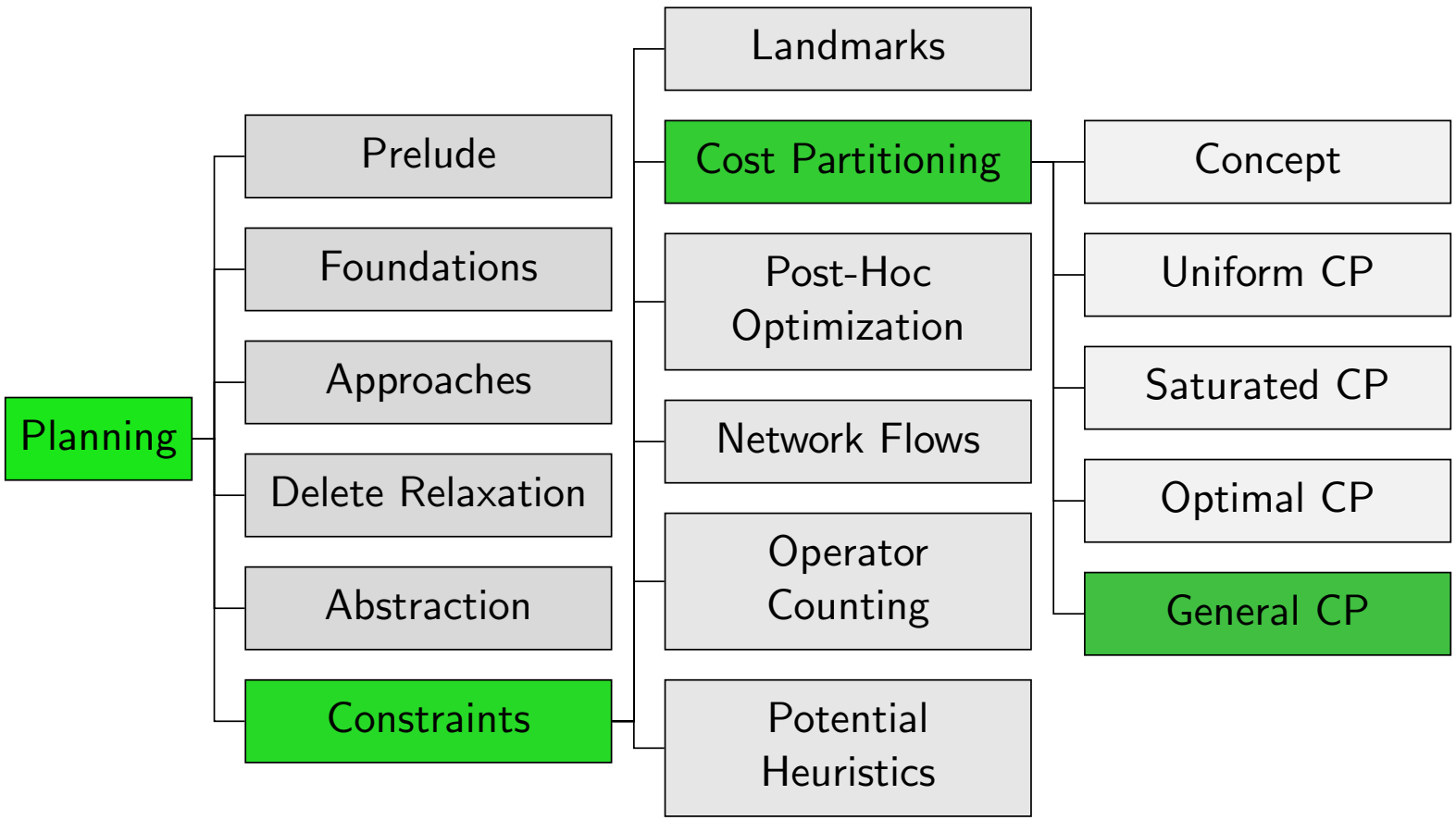
$$\begin{aligned} X_{o_4} &\geq 1 \\ X_{o_1} + X_{o_2} &\geq 1 \\ X_{o_1} + X_{o_3} &\geq 1 \\ X_{o_2} + X_{o_3} &\geq 1 \end{aligned}$$

$$X_{o_1} \geq 0, \quad X_{o_2} \geq 0, \quad X_{o_3} \geq 0, \quad X_{o_4} \geq 0$$

- $\rightsquigarrow$ optimal solution of LP relaxation:
  $X_{o_4} = 1$ and $X_{o_1} = X_{o_2} = X_{o_3} = 0.5$ with objective value 6
- $\rightsquigarrow$ LP relaxation of MHS heuristic is admissible
  and can be computed polynomial time

F8.2 General Cost Partitioning

Content of the Course



General Cost Partitioning

Cost functions are usually non-negative.

- We tacitly also required this for task copies
- Makes intuitively sense: original costs are non-negative
- But: not necessary for cost-partitioning!

General Cost Partitioning

Definition (General Cost Partitioning)

Let Π be a planning task with operators O.

A general cost partitioning for Π is a tuple $\langle cost_1, \ldots, cost_n \rangle$, where

- $cost_i : O \to \mathbb{R}$ for $1 \leq i \leq n$ and
- $\sum_{i=1}^{n} cost_i(o) \leq cost(o)$ for all $o \in O$.

General Cost Partitioning: Admissibility

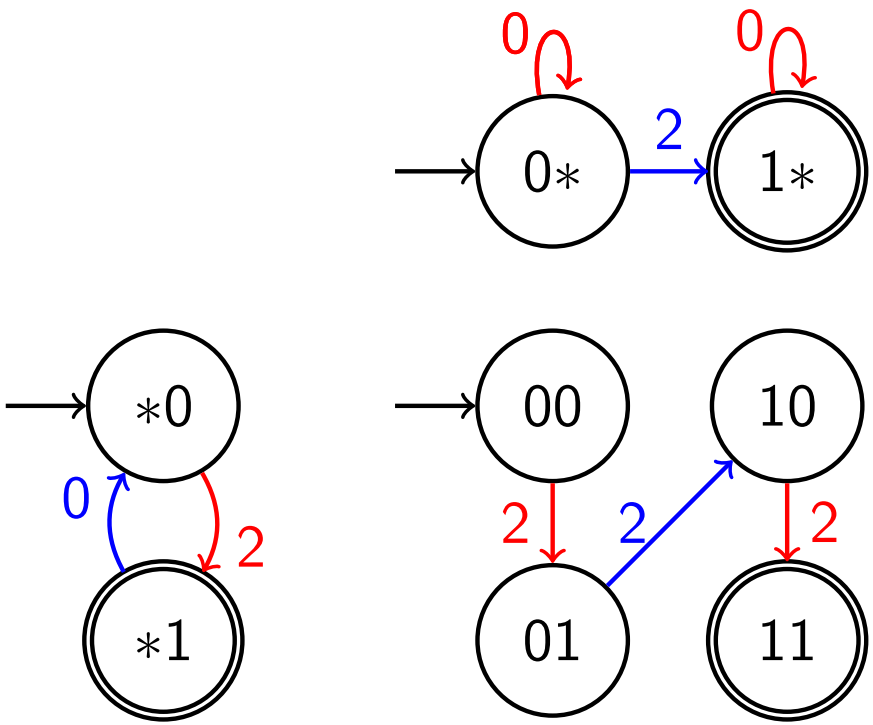
Theorem (Sum of Solution Costs is Admissible)

Let Π be a planning task, $\langle cost_1, \ldots, cost_n \rangle$ be a general cost partitioning and $\langle \Pi_1, \ldots, \Pi_n \rangle$ be the tuple of induced tasks.

Then the sum of the solution costs of the induced tasks is an admissible heuristic for Π, i.e., $\sum_{i=1}^{n} h^*_{\Pi_i} \leq h^*_{\Pi}$.

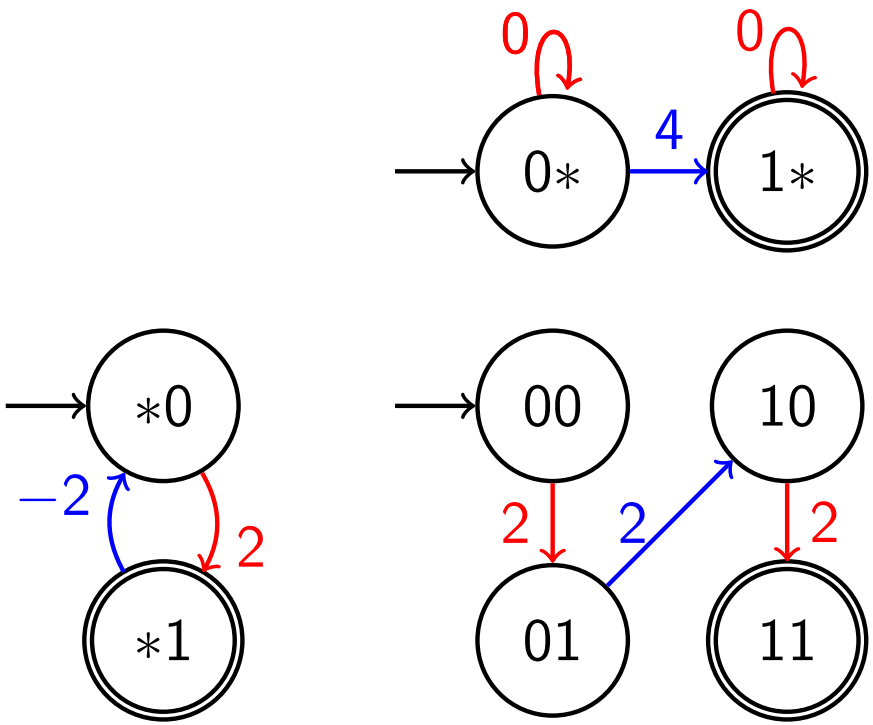
(Proof omitted.)

General Cost Partitioning: Example



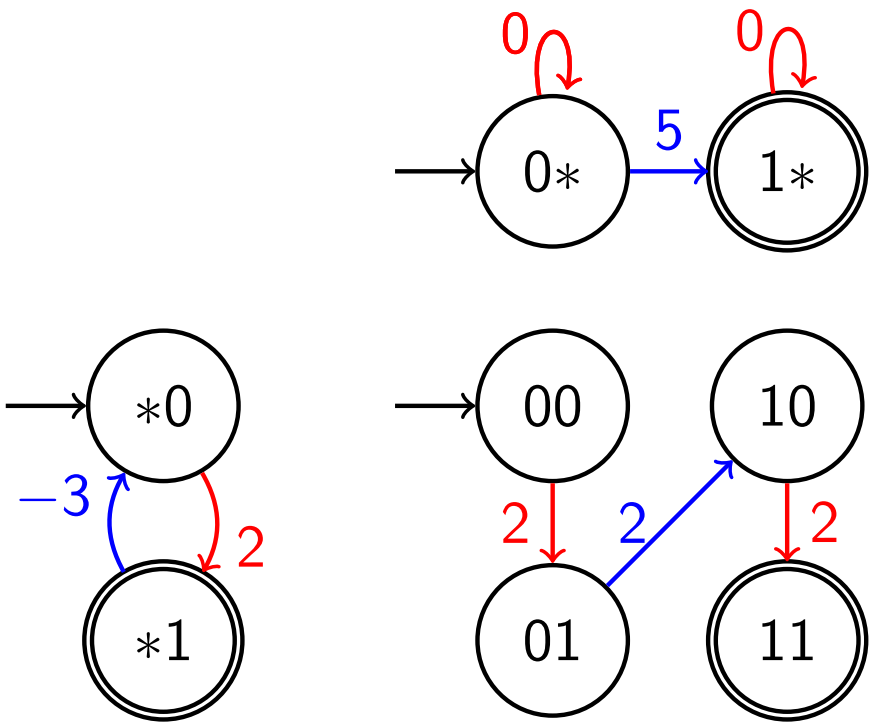
Heuristic value: $2 + 2 = 4$

General Cost Partitioning: Example



Heuristic value: $4 + 2 = 6$

General Cost Partitioning: Example



Heuristic value: $-\infty + 5 = -\infty$

F8.3 Summary

Summary

- For abstraction heuristics and disjunctive action landmarks, we know how to determine an optimal cost partitioning, using linear programming.
- Although solving a linear program is possible in polynomial time, the better heuristic guidance often does not outweigh the overhead (in particular for abstraction heuristics).
- In constrast to standard (non-negative) cost partitioning, general cost partitioning allows negative operators costs.
- General cost partitioning has the same relevant properties as non-negative cost partitioning but is more powerful.

Planning and Optimization

F9. Post-hoc Optimization

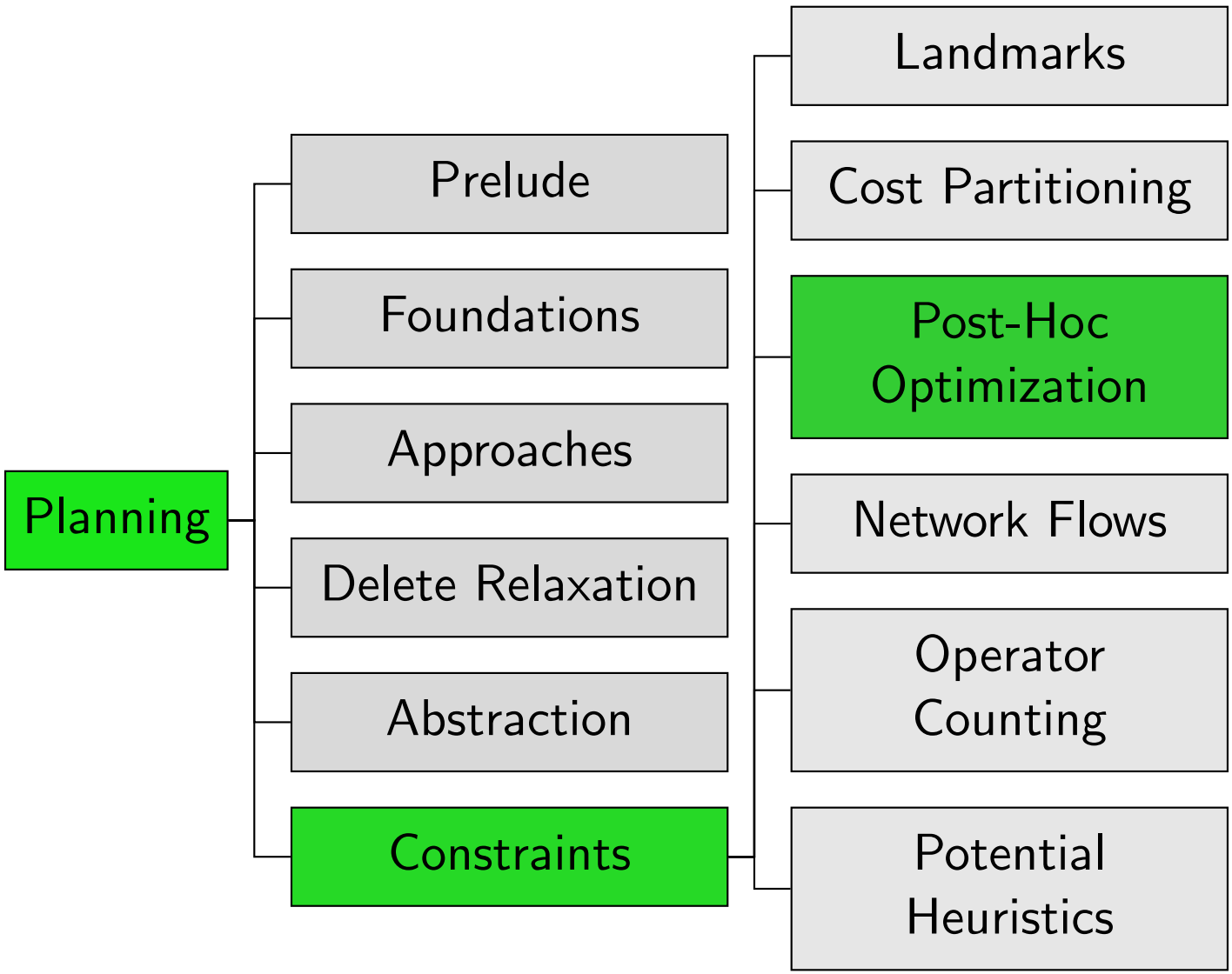
Malte Helmert and Gabriele Röger

Universität Basel

December 10, 2025

Planning and Optimization
December 10, 2025 — F9. Post-hoc Optimization

Content of the Course



F9.1 Introduction

Example Task (1)

Example (Example Task)

SAS$^+$ task $\Pi = \langle V, I, O, \gamma \rangle$ with

- $V = \{A, B, C\}$ with $\text{dom}(v) = \{0, 1, 2, 3, 4\}$ for all $v \in V$
- $I = \{A \mapsto 0, B \mapsto 0, C \mapsto 0\}$
- $O = \{inc_x^v \mid v \in V, x \in \{0, 1, 2\}\} \cup \{jump^v \mid v \in V\}$
  - $inc_x^v = \langle v = x, v := x + 1, 1 \rangle$
  - $jump^v = \langle \bigwedge_{v' \in V : v' \neq v} v' = 4, v := 3, 1 \rangle$
- $\gamma = A = 3 \wedge B = 3 \wedge C = 3$

- Each optimal plan consists of three increment operators for each variable $\rightsquigarrow h^*(I) = 9$
- Each operator affects only one variable.

Example Task (2)

- In projections on single variables we can reach the goal with a *jump* operator: $h^{\{A\}}(I) = h^{\{B\}}(I) = h^{\{C\}}(I) = 1$.
- In projections on more variables, we need for each variable three applications of increment operators to reach the abstract goal from the abstract initial state:
  $h^{\{A,B\}}(I) = h^{\{A,C\}}(I) = h^{\{B,C\}}(I) = 6$

Example (Canonical Heuristic)

$\mathcal{C} = \{\{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}\}$

$$h^{\mathcal{C}}(s) = \max\{h^{\{A\}}(s) + h^{\{B\}}(s) + h^{\{C\}}(s), h^{\{A\}}(s) + h^{\{B,C\}}(s), h^{\{B\}}(s) + h^{\{A,C\}}(s), h^{\{C\}}(s) + h^{\{A,B\}}(s)\}$$

$h^{\mathcal{C}}(I) = 7$

Post-hoc Optimization Heuristic: Idea

Consider the example task:

- *type-v operator:* operator modifying variable v
- $h^{\{A,B\}} = 6$

  $\Rightarrow$ in any plan operators of type A or B incur at least cost 6.
- $h^{\{A,C\}} = 6$

  $\Rightarrow$ in any plan operators of type A or C incur at least cost 6.
- $h^{\{B,C\}} = 6$

  $\Rightarrow$ in any plan operators of type B or C incur at least cost 6.
- $\Rightarrow$ any plan has at least cost ???.
- (let's use linear programming...)
- $\Rightarrow$ any plan has at least cost 9.

Can we generalize this kind of reasoning?

F9.2 Post-hoc Optimization

Post-hoc Optimization

The heuristic that generalizes this kind of reasoning
is the Post-hoc Optimization Heuristic (PhO)

- can be computed for any kind of heuristic ...
- ... as long as we are able to determine relevance of operators
- if in doubt, it's always safe to assume
  an operator is relevant for a heuristic
- but for PhO to work well, it's important that the set of
  relevant operators is as small as possible

Operator Relevance in Abstractions

Definition (Reminder: Affecting Transition Labels)

Let $\mathcal{T}$ be a transition system, and let ℓ be one of its labels.

We say that ℓ affects $\mathcal{T}$ if $\mathcal{T}$ has a transition $s \xrightarrow{\ell} t$ with $s \neq t$.

Definition (Operator Relevance in Abstractions)

An operator o is relevant for an abstraction α if o affects $\mathcal{T}^\alpha$.

We can efficiently determine operator relevance for abstractions.

Linear Program (1)

For a given set of abstractions $\{\alpha_1, \ldots, \alpha_n\}$, we construct a linear program:

- variable X_o for each operator $o \in O$
- intuitively, X_o is cost incurred by operator o
- abstraction heuristics are admissible

$$\sum_{o \in O} X_o \geq h^\alpha(s) \quad \text{for } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

- can tighten these constraints to

$$\sum_{o \in O : o \text{ relevant for } \alpha} X_o \geq h^\alpha(s) \quad \text{for } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

Linear Program (2)

For set of abstractions $\{\alpha_1, \ldots, \alpha_n\}$:

Variables

Non-negative variables X_o for all operators $o \in O$

Objective

Minimize $\sum_{o \in O} X_o$

Subject to

$$\sum_{o \in O : o \text{ relevant for } \alpha} X_o \geq h^\alpha(s) \quad \text{for } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

$$X_o \geq 0 \quad \text{for all } o \in O$$

Simplifying the LP

- Reduce the size of the LP by aggregating variables which always occur together in constraints.
- Happens if several operators are relevant for exactly the same heuristics.
- Partitioning $O/\sim$ induced by this equivalence relation
- One variable $X_{[o]}$ for each $[o] \in O/\sim$

Example

Example

- only operators o_1, o_2, o_3 and o_4 are relevant for h_1 and $h_1(s_0) = 11$
- only operators o_3, o_4, o_5 and o_6 are relevant for h_2 and $h_2(s_0) = 11$
- only operators o_1, o_2 and o_6 are relevant for h_3 and $h_3(s_0) = 8$

Which operators are relevant for exactly the same heuristics?
What is the resulting partitioning?

Answer: $o_1 \sim o_2$ and $o_3 \sim o_4$
$\Rightarrow O/\!\sim = \{[o_1], [o_3], [o_5], [o_6]\}$

Simplifying the LP: Example

LP before aggregation

Variables

Non-negative variable $X_1, \ldots, X_6$
for operators $o_1, \ldots, o_6$

$$\text{Minimize} \quad X_1 + X_2 + X_3 + X_4 + X_5 + X_6 \quad \text{subject to}$$

$$\begin{aligned}
X_1 + X_2 + X_3 + X_4 &\geq 11 \\
X_3 + X_4 + X_5 + X_6 &\geq 11 \\
X_1 + X_2 + X_6 &\geq 8 \\
X_i &\geq 0 \quad \text{for } i \in \{1, \ldots, 6\}
\end{aligned}$$

Simplifying the LP: Example

LP after aggregation

Variables

Non-negative variable $X_{[1]}, X_{[3]}, X_{[5]}, X_{[6]}$
for equivalence classes $[o_1], [o_3], [o_5], [o_6]$

$$\text{Minimize} \quad X_{[1]} + X_{[3]} + X_{[5]} + X_{[6]} \quad \text{subject to}$$

$$\begin{aligned}
X_{[1]} + X_{[3]} &\geq 11 \\
X_{[3]} + X_{[5]} + X_{[6]} &\geq 11 \\
X_{[1]} + X_{[6]} &\geq 8 \\
X_i &\geq 0 \quad \text{for } i \in \{[1], [3], [5], [6]\}
\end{aligned}$$

PhO Heuristic

Definition (Post-hoc Optimization Heuristic)

The post-hoc optimization heuristic $h^{\text{PhO}}_{\{\alpha_1,\ldots,\alpha_n\}}$ for abstractions $\alpha_1, \ldots, \alpha_n$ is the objective value of the following linear program:

$$\text{Minimize} \sum_{[o]\in O/\sim} X_{[o]} \text{ subject to}$$

$$\sum_{[o]\in O/\sim : o \text{ relevant for } \alpha} X_{[o]} \geq h^{\alpha}(s) \quad \text{for all } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

$$X_{[o]} \geq 0 \quad \text{for all } [o] \in O/\sim,$$

where $o \sim o'$ iff o and o' are relevant for exactly the same abstractions in $\alpha_1, \ldots, \alpha_n$.

PhO Heuristic

h^{PhO}

1. Precompute all abstraction heuristics $h^{\alpha_1}, \ldots, h^{\alpha_n}$.
2. Create LP for initial state s_0.
3. For each new state s:
   - Look up $h^{\alpha}(s)$ for all $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$.
   - Adjust LP by replacing bounds with the $h^{\alpha}(s)$ values.

Post-hoc Optimization Heuristic: Admissibility

Theorem (Admissibility)

The post-hoc optimization heuristic is admissible.

Proof.

Let Π be a planning task and $\{\alpha_1, \ldots, \alpha_n\}$ be a set of abstractions. We show that there is a feasible variable assignment with objective value equal to the cost of an optimal plan.

Let π be an optimal plan for state s and let $cost_\pi(O')$ be the cost incurred by operators from $O' \subseteq O$ in π.

Setting each $X_{[o]}$ to $cost_\pi([o])$ is a feasible variable assignment:
Constraints $X_{[o]} \geq 0$ are satisfied. ...

Post-hoc Optimization Heuristic: Admissibility

Theorem (Admissibility)

The post-hoc optimization heuristic is admissible.

Proof (continued).

For each $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$, π is a solution in the abstract transition system and the sum in the corresponding constraint equals the cost of the state-changing abstract state transitions (i.e.. not accounting for self-loops). As $h^\alpha(s)$ corresponds to the cost of an optimal solution in the abstraction, the inequality holds.

For this assignment, the objective function has value $h^*(s)$ (cost of π), so the objective value of the LP is admissible. □

F9.3 Comparison

Combining Estimates from Abstraction Heuristics

- Post-Hoc optimization combines multiple admissible heuristic estimates into one.
- We have already heard of two other such approaches for abstraction heuristics,
  - the canonical heuristic (for PDBs), and
  - optimal cost partitioning (not covered in detail).
- How does PhO compare to these?

What about Optimal Cost Partitioning for Abstractions?

Optimal cost partitioning for abstractions...

- ...uses a state-specific LP to find the best possible cost partitioning, and sums up the heuristic estimates.
- ...dominates the canonical heuristic, i.e. for the same pattern collection, it never gives lower estimates than $h^{\mathcal{C}}$.
- ...is very expensive to compute (recomputing all abstract goal distances in every state).

PhO: Dual Linear Program

For set of abstractions $\{\alpha_1, \ldots, \alpha_n\}$:

Variables

Y_α for each abstraction $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$

Objective

Maximize $\sum_{\alpha \in \{\alpha_1, \ldots, \alpha_n\}} h^\alpha(s) Y_\alpha$

Subject to

$$\sum_{\alpha \in \{\alpha_1, \ldots, \alpha_n\}: o \text{ relevant for } \alpha} Y_\alpha \leq 1 \quad \text{for all } [o] \in O/\sim$$

$$Y_\alpha \geq 0 \quad \text{for all } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

We compute a state-specific cost partitioning that can only scale the operator costs within each heuristic by a factor $0 \leq Y_\alpha \leq 1$.

Relation to Optimal Cost Partitioning

Theorem

Optimal cost partitioning dominates post-hoc optimization.

Proof Sketch.

Consider a feasible assignment $\langle Y_{\alpha_1}, \ldots, Y_{\alpha_n} \rangle$ for the variables of the dual LP for PhO.

Its objective value is equivalent to the cost-partitioning heuristic for the same abstractions with cost partitioning $\langle Y_{\alpha_1} \mathit{cost}, \ldots, Y_{\alpha_n} \mathit{cost} \rangle$.

Relation to Canonical Heuristic

Theorem

Consider the dual D of the LP solved by the post-hoc optimization heuristic in state s for a given set of abstractions. If we restrict the variables in D to integers, the objective value is the canonical heuristic value $h^{\mathcal{C}}(s)$.

Corollary

The post-hoc optimization heuristic dominates the canonical heuristic for the same set of abstractions.

h^{PhO} vs $h^{\mathcal{C}}$

- For the canonical heuristic, we need to find all maximal cliques, which is an NP-hard problem.
- The post-hoc optimization heuristic dominates the canonical heuristic and can be computed in polynomial time.
- The post-hoc optimization heuristic solves an LP in each state.
- With post-hoc optimization, a large number of small patterns works well.

F9.4 Summary

Summary

- Post-hoc optimization heuristic constraints express admissibility of heuristics
- exploits (ir-)relevance of operators for heuristics
- explores the middle ground between canonical heuristic and optimal cost partitioning.
- For the same set of abstractions, the post-hoc optimization heuristic dominates the canonical heuristic.
- The computation can be done in polynomial time.

Planning and Optimization

F10. Network Flow Heuristics

Malte Helmert and Gabriele Röger

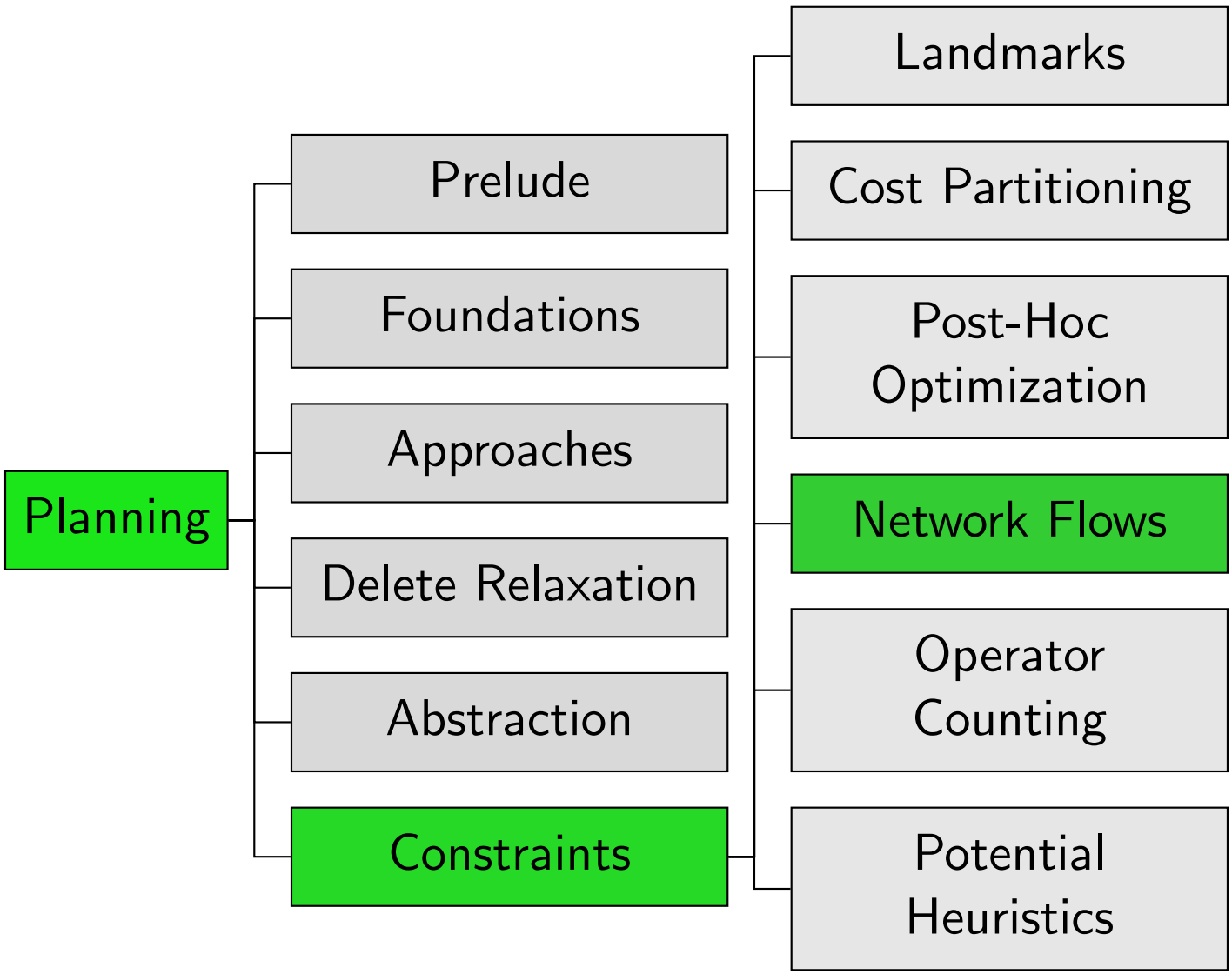
Universität Basel

December 10, 2025

Planning and Optimization

December 10, 2025 — F10. Network Flow Heuristics

Content of the Course



F10.1 Introduction

Reminder: SAS$^+$ Planning Tasks

For a SAS$^+$ planning task $\Pi = \langle V, I, O, \gamma \rangle$:

- V is a set of finite-domain state variables,
- Each atom has the form $v = d$ with $v \in V, d \in \mathrm{dom}(v)$.
- Operator preconditions and the goal formula γ are satisfiable conjunctions of atoms.
- Operator effects are conflict-free conjunctions of atomic effects of the form $v_1 := d_1 \wedge \cdots \wedge v_n := d_n$.

Example Task (1)

- One package, two trucks, two locations
- Variables:
  - *pos-p* with $\text{dom}(\textit{pos-p}) = \{loc_1, loc_2, t_1, t_2\}$
  - *pos-t-i* with $\text{dom}(\textit{pos-t-i}) = \{loc_1, loc_2\}$ for $i \in \{1, 2\}$
- The package is at location 1 and the trucks at location 2,
  - $I = \{\textit{pos-p} \mapsto loc_1, \textit{pos-t-1} \mapsto loc_2, \textit{pos-t-2} \mapsto loc_2)$
- The goal is to have the package at location 2 and truck 1 at location 1.
  - $\gamma = (\textit{pos-p} = loc_2) \wedge (\textit{pos-t-1} = loc_1)$

Example Task (2)

- Operators: for $i, j, k \in \{1, 2\}$:

$$\begin{aligned} load(t_i, loc_j) = \langle & pos\text{-}t\text{-}i = loc_j \wedge pos\text{-}p = loc_j, \\ & pos\text{-}p := t_i, 1 \rangle \\ unload(t_i, loc_j) = \langle & pos\text{-}t\text{-}i = loc_j \wedge pos\text{-}p = t_i, \\ & pos\text{-}p := loc_j, 1 \rangle \\ drive(t_i, loc_j, loc_k) = \langle & pos\text{-}t\text{-}i = loc_j, \\ & pos\text{-}t\text{-}i := loc_k, 1 \rangle \end{aligned}$$

Example Task: Observations

Consider some atoms of the example task:

- $pos\text{-}p = loc_1$ initially true and must be false in the goal
  - ▷ at location 1 the package must be loaded once more than it is unloaded.
- $pos\text{-}p = loc_2$ initially false and must be true in the goal
  - ▷ at location 2 the package must be unloaded once more than it is loaded.
- $pos\text{-}p = t_1$ initially false and must be false in the goal
  - ▷ same number of load and unload actions for truck 1.

Can we derive a heuristic from this kind of information?

Example: Flow Constraints

Let π be some arbitrary plan for the example task and let $\#o$ denote the number of occurrences of operator o in π.
Then the following holds:

- $\mathit{pos}\text{-}p = \mathit{loc}_1$ initially true and must be false in the goal
  - ▷ at location 1 the package must be loaded once more than it is unloaded.

  $\#\mathit{load}(t_1, \mathit{loc}_1) + \#\mathit{load}(t_2, \mathit{loc}_1) = 1 + \#\mathit{unload}(t_1, \mathit{loc}_1) + \#\mathit{unload}(t_2, \mathit{loc}_1)$
- $\mathit{pos}\text{-}p = t_1$ initially false and must be false in the goal
  - ▷ same number of load and unload actions for truck 1.

  $\#\mathit{unload}(t_1, \mathit{loc}_1) + \#\mathit{unload}(t_1, \mathit{loc}_2) = \#\mathit{load}(t_1, \mathit{loc}_1) + \#\mathit{load}(t_1, \mathit{loc}_2)$

Network Flow Heuristics: General Idea

- Formulate flow constraints for each atom.
- These are satisfied by every plan of the task.
- The cost of a plan is $\sum_{o \in O} cost(o) \# o$
- The objective value of an integer program that minimizes this cost subject to the flow constraints is a lower bound on the plan cost (i.e., an admissible heuristic estimate).
- As solving the IP is NP-hard, we solve the LP relaxation instead.

How do we get the flow constraints?

How to Derive Flow Constraints?

- The constraints formulate how often an atom can be produced or consumed.
- "Produced" (resp. "consumed") means that the atom is false (resp. true) before an operator application and true (resp. false) in the successor state.
- For general SAS^+ operators, this depends on the state where the operator is applied: effect $v := d$ only produces $v = d$ if the operator is applied in a state s with $s(v) \neq d$.
- For general SAS^+ tasks, the goal does not have to specify a value for every variable.
- All this makes the definition of flow constraints somewhat involved and in general such constraints are inequalitites.

Good news: easy for tasks in transition normal form

F10.2 Transition Normal Form

Variables Occurring in Conditions and Effects

- Many algorithmic problems for SAS^+ planning tasks become simpler when we can make two further restrictions.
- These are related to the variables that occur in conditions and effects of the task.

Definition ($vars(\varphi)$, $vars(e)$)

For a logical formula φ over finite-domain variables V, $vars(\varphi)$ denotes the set of finite-domain variables occurring in φ.

For an effect e over finite-domain variables V, $vars(e)$ denotes the set of finite-domain variables occurring in e.

Transition Normal Form

Definition (Transition Normal Form)

A SAS$^+$ planning task $\Pi = \langle V, I, O, \gamma \rangle$
is in transition normal form (TNF) if

- for all $o \in O$, $vars(pre(o)) = vars(eff(o))$, and
- $vars(\gamma) = V$.

In words, an operator in TNF must mention the same variables in the precondition and effect, and a goal in TNF must mention all variables (= specify exactly one goal state).

Converting Operators to TNF: Violations

There are two ways in which an operator o can violate TNF:

- There exists a variable $v \in \mathit{vars}(\mathit{pre}(o)) \setminus \mathit{vars}(\mathit{eff}(o))$.
- There exists a variable $v \in \mathit{vars}(\mathit{eff}(o)) \setminus \mathit{vars}(\mathit{pre}(o))$.

The first case is easy to address: if $v = d$ is a precondition with no effect on v, just add the effect $v := d$.

The second case is more difficult: if we have the effect $v := d$ but no precondition on v, how can we add a precondition on v without changing the meaning of the operator?

Converting Operators to TNF: Multiplying Out

Solution 1: multiplying out

1. While there exists an operator o and a variable $v \in \mathit{vars}(\mathit{eff}(o))$ with $v \notin \mathit{vars}(\mathit{pre}(o))$:
   - For each $d \in \mathrm{dom}(v)$, add a new operator that is like o but with the additional precondition $v = d$.
   - Remove the original operator.
2. Repeat the previous step until no more such variables exist.

Problem:

- If an operator o has n such variables, each with k values in its domain, this introduces k^n variants of o.
- Hence, this is an exponential transformation.

Converting Operators to TNF: Auxiliary Values

Solution 2: auxiliary values

1. For every variable v, add a new auxiliary value u to its domain.
2. For every variable v and value $d \in \mathrm{dom}(v) \setminus \{\mathrm{u}\}$, add a new operator to change the value of v from d to u at no cost: $\langle v = d, v := \mathrm{u}, 0 \rangle$.
3. For all operators o and all variables $v \in \mathit{vars}(\mathit{eff}(o)) \setminus \mathit{vars}(\mathit{pre}(o))$, add the precondition $v = \mathrm{u}$ to $\mathit{pre}(o)$.

Properties:

- Transformation can be computed in linear time.
- Due to the auxiliary values, there are new states and transitions in the induced transition system, but all path costs between original states remain the same.

Converting Goals to TNF

- The auxiliary value idea can also be used to convert the goal γ to TNF.
- For every variable $v \notin \mathit{vars}(\gamma)$, add the condition $v = \mathrm{u}$ to γ.

With these ideas, every SAS^+ planning task can be converted into transition normal form in linear time.

TNF for Example Task (1)

The example task is not in transition normal form:

- Load and unload operators have preconditions on the position of some truck but no effect on this variable.
- The goal does not specify a value for variable *pos-t-2*.

TNF for Example Task (2)

Operators in transition normal form: for $i, j, k \in \{1, 2\}$:

$$\begin{aligned}
load(t_i, loc_j) = \langle & pos\text{-}t\text{-}i = loc_j \wedge pos\text{-}p = loc_j, \\
& pos\text{-}p := t_i \wedge pos\text{-}t\text{-}i := loc_j, 1\rangle \\
unload(t_i, loc_j) = \langle & pos\text{-}t\text{-}i = loc_j \wedge pos\text{-}p = t_i, \\
& pos\text{-}p := loc_j \wedge pos\text{-}t\text{-}i := loc_j, 1\rangle \\
drive(t_i, loc_j, loc_k) = \langle & pos\text{-}t\text{-}i = loc_j, \\
& pos\text{-}t\text{-}i := loc_k, 1\rangle
\end{aligned}$$

TNF for Example Task (3)

To bring the goal in normal form,

- add an additional value $\mathbf{u}$ to dom(*pos-t-2*)
- add zero-cost operators
  $o_1 = \langle \mathit{pos\text{-}t\text{-}2} = \mathit{loc}_1, \mathit{pos\text{-}t\text{-}2} := \mathbf{u}, 0 \rangle$ and
  $o_2 = \langle \mathit{pos\text{-}t\text{-}2} = \mathit{loc}_2, \mathit{pos\text{-}t\text{-}2} := \mathbf{u}, 0 \rangle$
- Add $\mathit{pos\text{-}t\text{-}2} = \mathbf{u}$ to the goal:
  $\gamma = (\mathit{pos\text{-}p} = \mathit{loc}_2) \wedge (\mathit{pos\text{-}t\text{-}1} = \mathit{loc}_1) \wedge (\mathit{pos\text{-}t\text{-}2} = \mathbf{u})$

F10.3 Flow Heuristic

Notation

- In SAS$^+$ tasks, states are variable assignments, conditions are conjunctions over atoms, and effects are conjunctions of atomic effects.
- In the following, we use a unifying notation to express that an atom is true in a state/entailed by a condition/ made true by an effect.
- For state s, we write $(v = d) \in s$ to express that $s(v) = d$.
- For a conjunction of atoms φ, we write $(v = d) \in \varphi$ to express that φ has a conjunct $v = d$ (or alternatively $\varphi \models v = d$).
- For effect e, we write $(v = d) \in e$ to express that e contains the atomic effect $v := d$.

Flow Constraints (1)

A flow constraint for an atom relates how often it can be produced to how often it can be consumed.

Let o be an operator in transition normal form. Then:

- o produces atom a iff $a \in \mathit{eff}(o)$ and $a \notin \mathit{pre}(o)$.
- o consumes atom a iff $a \in \mathit{pre}(o)$ and $a \notin \mathit{eff}(o)$.
- Otherwise o is neutral wrt. atom a.

$\rightsquigarrow$ State-independent

Flow Constraints (2)

A flow constraint for an atom relates how often it can be produced to how often it can be consumed.

The constraint depends on the current state s and the goal γ.
If γ mentions all variables (as in TNF), the following holds:

- If $a \in s$ and $a \in \gamma$ then atom a must be equally often produced and consumed.
- Analogously for $a \notin s$ and $a \notin \gamma$.
- If $a \in s$ and $a \notin \gamma$ then a must be consumed once more than it is produced.
- If $a \notin s$ and $a \in \gamma$ then a must be produced once more than it is consumed.

Iverson Bracket

The dependency on the current state and the goal can concisely be expressed with Iverson brackets:

Definition (Iverson Bracket)

Let P be a logical proposition (= some statement that can be evaluated to true or false). Then

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{if } P \text{ is false.} \end{cases}$$

Example: $[2 \neq 3] = 1$

Flow Constraints (3)

Definition (Flow Constraint)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a task in transition normal form.
The flow constraint for atom a in state s is

$$[a \in s] + \sum_{o \in O : a \in \mathit{eff}(o)} \mathrm{Count}_o = [a \in \gamma] + \sum_{o \in O : a \in \mathit{pre}(o)} \mathrm{Count}_o$$

- Count_o is an LP variable for the number of occurrences of operator o.
- Neutral operators either appear on both sides or on none.

Flow Heuristic

Definition (Flow Heuristic)

Let $\Pi = \langle V, I, O, \gamma \rangle$ be a SAS$^+$ task in transition normal form and let $A = \{(v = d) \mid v \in V, d \in dom(v)\}$ be the set of atoms of Π.

The flow heuristic $h^{\text{flow}}(s)$ is the objective value of the following LP or ∞ if the LP is infeasible:

$$\text{minimize} \quad \sum_{o \in O} cost(o) \cdot \text{Count}_o \quad \text{subject to}$$

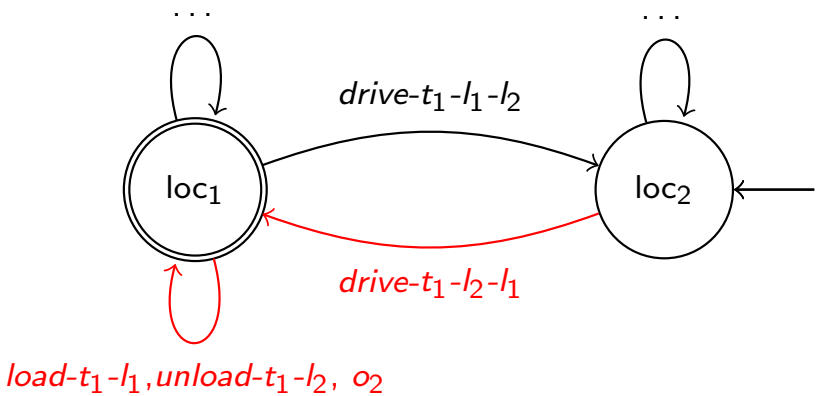
$$[a \in s] + \sum_{o \in O : a \in eff(o)} \text{Count}_o = [a \in \gamma] + \sum_{o \in O : a \in pre(o)} \text{Count}_o \quad \text{for all } a \in A$$

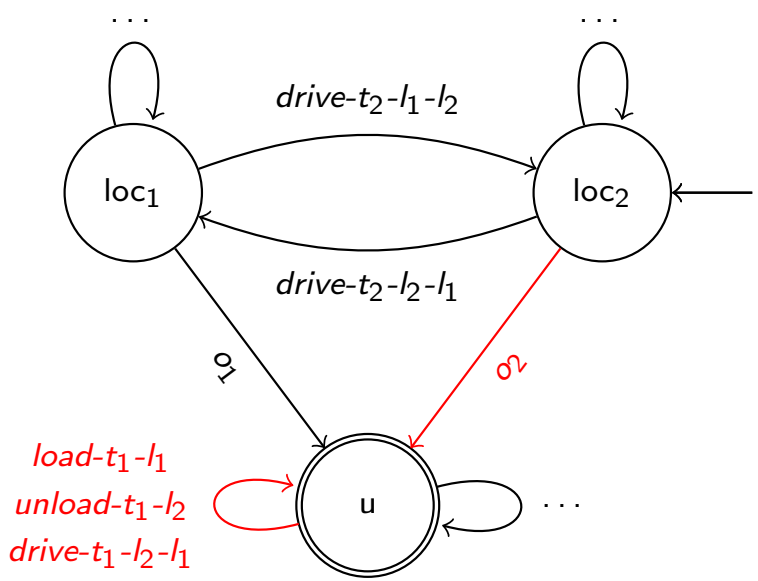
$$\text{Count}_o \geq 0 \quad \text{for all } o \in O$$

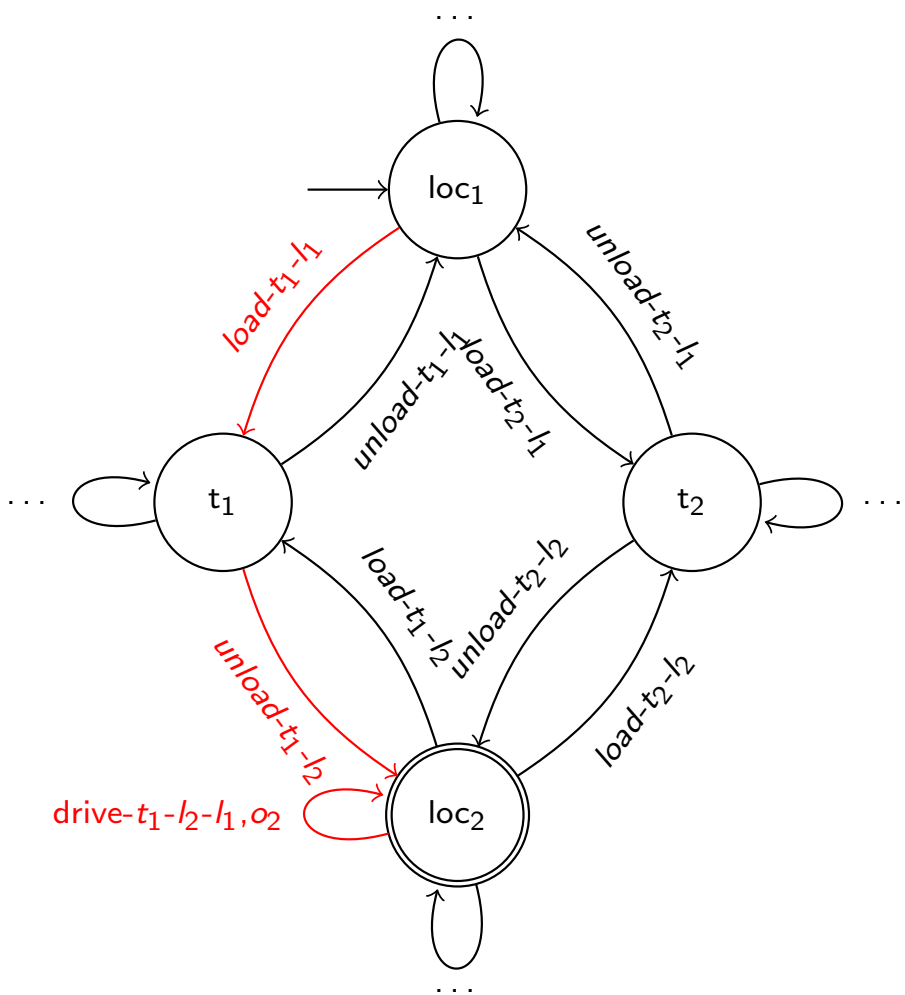
Flow Heuristic on Example Task

$\rightsquigarrow$ Demo

Visualization of Flow in Example Task







Flow Heuristic: Properties (1)

Theorem

The flow heuristic h^{flow} is goal-aware, safe, consistent and admissible.

Proof Sketch.

It suffices to prove goal-awareness and consistency.

Goal-awareness: If $s \models \gamma$ then $\text{Count}_o = 0$ for all $o \in O$ is feasible and the objective function has value 0. As $\text{Count}_o \geq 0$ for all variables and operator costs are nonnegative, the objective value cannot be smaller. ...

Flow Heuristic: Properties (2)

Proof Sketch (continued).

Consistency: Let o be an operator that is applicable in state s and let $s' = s[\![o]\!]$.

Increasing Count_o by one in an optimal feasible assignment for the LP for state s' yields a feasible assignment for the LP for state s, where the objective function is $h^{\text{flow}}(s') + cost(o)$.

This is an upper bound on $h^{\text{flow}}(s)$, so in total $h^{\text{flow}}(s) \leq h^{\text{flow}}(s') + cost(o)$. □

F10.4 Summary

Summary

- A flow constraint for an atom describes how the number of producing operator applications is linked to the number of consuming operator applications.
- The flow heuristic computes a lower bound on the cost of each operator sequence that satisfies these constraints for all atoms.
- The flow heuristic only considers the number of occurrences of each operator, but ignores their order.

Planning and Optimization

F11. Operator Counting

Malte Helmert and Gabriele Röger

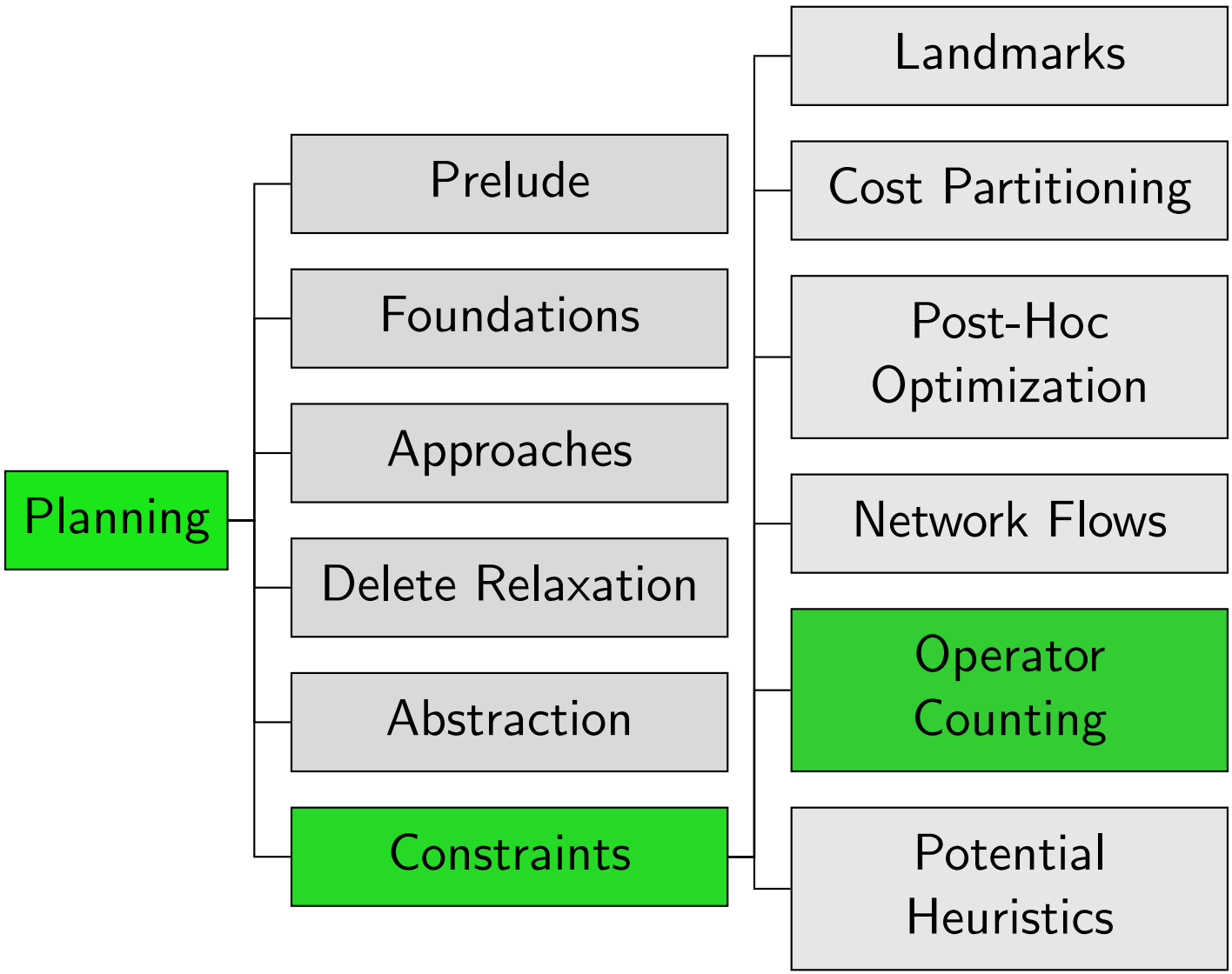
Universität Basel

December 15, 2025

Planning and Optimization

December 15, 2025 — F11. Operator Counting

Content of the Course



F11.1 Introduction

Reminder: Flow Heuristic

In the previous chapter, we used flow constraints to describe how often operators must be used in each plan.

Example (Flow Constraints)

Let Π be a planning problem with operators $\{o_{\text{red}}, o_{\text{green}}, o_{\text{blue}}\}$.
The flow constraint for some atom a is the constraint

$$1 + \mathit{Count}_{o_{\text{green}}} = \mathit{Count}_{o_{\text{red}}} \text{ if}$$

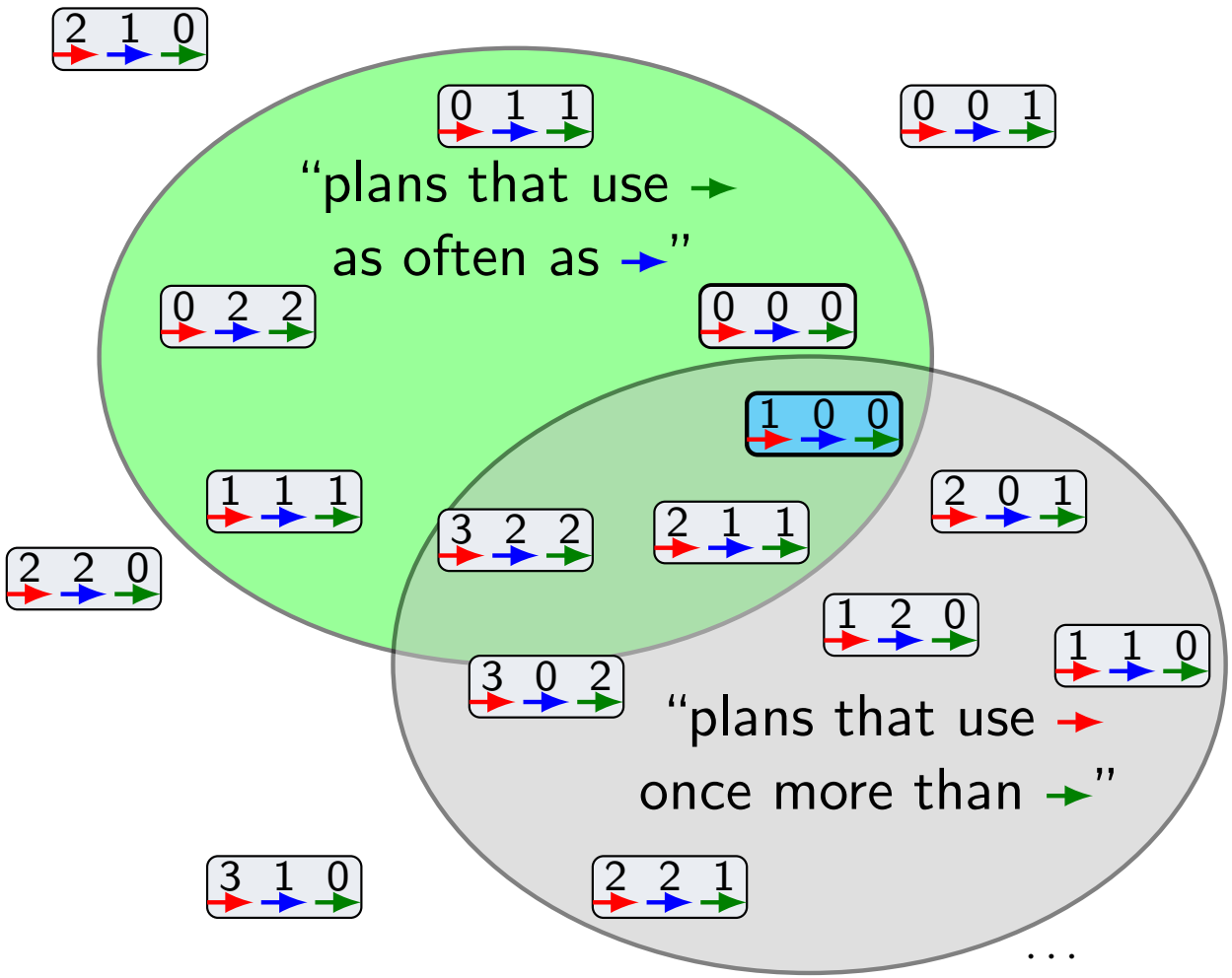
- a is true in the initial state
- a is false in the goal
- o_{green} produces a
- o_{red} consumes a

In natural language, the flow constraint expresses that

every plan uses o_{red} once more than o_{green}.

Reminder: Flow Heuristic

Let us now observe how each flow constraint alters the operator count solution space.



F11.2 Operator-counting Framework

Operator Counting

Operator counting

- generalizes this idea to a framework that allows to admissibly combine different heuristics.
- uses linear constraints ...
- ... that describe number of occurrences of an operator ...
- ... and must be satisfied by every plan.
- provides declarative way to describe knowledge about solutions.
- allows reasoning about solutions to derive heuristic estimates.

Operator-counting Constraint

Definition (Operator-counting Constraints)

Let Π be a planning task with operators O and let s be a state.
Let $\mathcal{V}$ be the set of integer variables Count_o for each $o \in O$.

A linear inequality over $\mathcal{V}$ is called an operator-counting constraint for s if for every plan π for s setting each Count_o to the number of occurrences of o in π is a feasible variable assignment.

Operator-counting Heuristics

Definition (Operator-counting IP/LP Heuristic)

The operator-counting integer program IP_C for a set C of operator-counting constraints for state s is

$$\text{Minimize} \quad \sum_{o \in O} cost(o) \cdot \text{Count}_o \quad \text{subject to}$$

$$C \text{ and } \text{Count}_o \geq 0 \text{ for all } o \in O,$$

where O is the set of operators.

The IP heuristic h^{IP}_C is the objective value of IP_C,
the LP heuristic h^{LP}_C is the objective value of its LP-relaxation.

If the IP/LP is infeasible, the heuristic estimate is ∞.

Operator-counting Constraints

- Adding more constraints can only remove feasible solutions.
- Fewer feasible solutions can only increase the objective value.
- Higher objective value means better informed heuristic

$\Rightarrow$ Have we already seen other operator-counting constraints?

Reminder: Minimum Hitting Set for Landmarks

Variables

Non-negative variable Applied_o for each operator o

Objective

Minimize $\sum_o \mathit{cost}(o) \cdot \text{Applied}_o$

Subject to

$$\sum_{o \in L} \text{Applied}_o \geq 1 \text{ for all landmarks } L$$

Operator Counting with Disjunctive Action Landmarks

Variables

Non-negative variable Count_o for each operator o

Objective

Minimize $\sum_o cost(o) \cdot \text{Count}_o$

Subject to

$$\sum_{o \in L} \text{Count}_o \geq 1 \text{ for all landmarks } L$$

Reminder: Post-hoc Optimization Heuristic

For set of abstractions $\{\alpha_1, \ldots, \alpha_n\}$:

Variables

Non-negative variables X_o for all operators $o \in O$

X_o is cost incurred by operator o

Objective

Minimize $\sum_{o \in O} X_o$

Subject to

$$\sum_{o \in O: o \text{ relev. for } \alpha} X_o \geq h^\alpha(s) \quad \text{for } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

$$X_o \geq 0 \quad \text{for all } o \in O$$

Operator Counting with Post-hoc Optimization Constraints

For set of abstractions $\{\alpha_1, \ldots, \alpha_n\}$:

Variables

Non-negative variables Count_o for all operators $o \in O$

$\text{Count}_o \cdot \mathit{cost}(o)$ is cost incurred by operator o

Objective

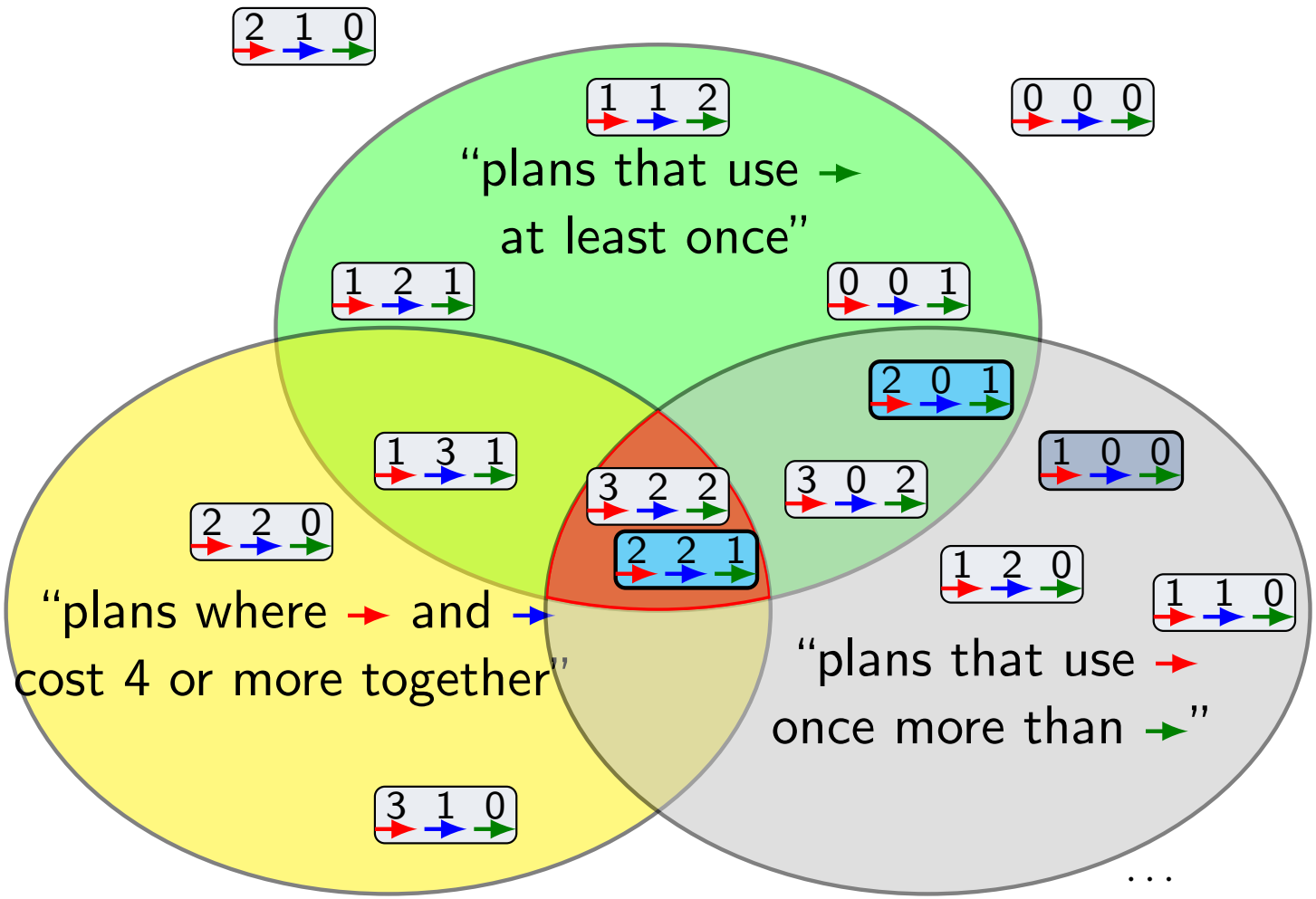
Minimize $\sum_{o \in O} \mathit{cost}(o) \cdot \text{Count}_o$

Subject to

$$\sum_{o \in O : o \text{ relev. for } \alpha} \mathit{cost}(o) \cdot \text{Count}_o \geq h^\alpha(s) \quad \text{for } \alpha \in \{\alpha_1, \ldots, \alpha_n\}$$

$$\mathit{cost}(o) \cdot \text{Count}_o \geq 0 \quad \text{for all } o \in O$$

Example


2 1 0
1 1 2
0 0 0
"plans that use → at least once"
1 2 1
0 0 1
2 0 1
1 3 1
1 0 0
3 2 2
3 0 2
2 2 0
2 2 1
1 2 0
1 1 0
"plans where → and → cost 4 or more together"
"plans that use → once more than →"
3 1 0
...

Further Examples?

- The definition of operator-counting constraints can be extended to groups of constraints and auxiliary variables.
- With this extended definition we could also cover more heuristics, e.g., the perfect relaxation heuristic h^+

F11.3 Properties

Admissibility

Theorem (Operator-counting Heuristics are Admissible)

The IP and the LP heuristic are admissible.

Proof.

Let C be a set of operator-counting constraints for state s and π be an optimal plan for s. The number of operator occurrences of π are a feasible solution for C. As the IP/LP minimizes the total plan cost, the objective value cannot exceed the cost of π and is therefore an admissible estimate. □

Dominance

Theorem

Let C and C' be sets of operator-counting constraints for s and let $C \subseteq C'$. Then $\mathrm{IP}_C \leq \mathrm{IP}_{C'}$ and $\mathrm{LP}_C \leq \mathrm{LP}_{C'}$.

Proof.

Every feasible solution of C' is also feasible for C. As the LP/IP is a minimization problem, the objective value subject to C can therefore not be larger than the one subject to C'. □

Adding more constraints can only improve the heuristic estimate.

Heuristic Combination

Operator counting as heuristic combination

- Multiple operator-counting heuristics can be combined by computing $h^{\mathrm{LP}}_{\mathcal{C}}$ / $h^{\mathrm{IP}}_{\mathcal{C}}$ for the union of their constraints.
- This is an admissible combination.
  - Never worse than maximum of individual heuristics
  - Sometimes even better than their sum
- We already know a way of admissibly combining heuristics: cost partitioning.

  $\Rightarrow$ How are they related?

Connection to Cost Partitioning

Theorem

Let $C_1, \ldots, C_n$ be sets of operator-counting constraints for s and $\mathcal{C} = \bigcup_{i=1}^{n} C_i$. Then $h^{\mathrm{LP}}_{\mathcal{C}}$ is the optimal general cost partitioning over the heuristics $h^{\mathrm{LP}}_{C_i}$.

Proof Sketch.

In $\mathrm{LP}_{\mathcal{C}}$, add variables Count^i_o and constraints $\mathrm{Count}_o = \mathrm{Count}^i_o$ for all operators o and $1 \leq i \leq n$. Then replace Count_o by Count^i_o in C_i.

Dualizing the resulting LP shows that $h^{\mathrm{LP}}_{\mathcal{C}}$ computes a cost partitioning. Dualizing the component heuristics of that cost partitioning shows that they are $h^{\mathrm{LP}}_{C_i}$.

Comparison to Optimal Cost Partitioning

- some heuristics are more compact if expressed as operator counting
- some heuristics cannot be expressed as operator counting
- operator counting IP even better than optimal cost partitioning
- Cost partitioning maximizes, so heuristics must be encoded perfectly to guarantee admissibility.
  Operator counting minimizes, so missing information just makes the heuristic weaker.

F11.4 Summary

Summary

- Many heuristics can be formulated in terms of operator-counting constraints.
- The operator counting heuristic framework allows to combine the constraints and to reason on the entire encoded declarative knowledge.
- The heuristic estimate for the combined constraints can be better than the one of the best ingredient heuristic but never worse.
- Operator counting is equivalent to optimal general cost partitioning over individual constraints.

Planning and Optimization

F12. Potential Heuristics

Malte Helmert and Gabriele Röger

Universität Basel

December 15, 2025

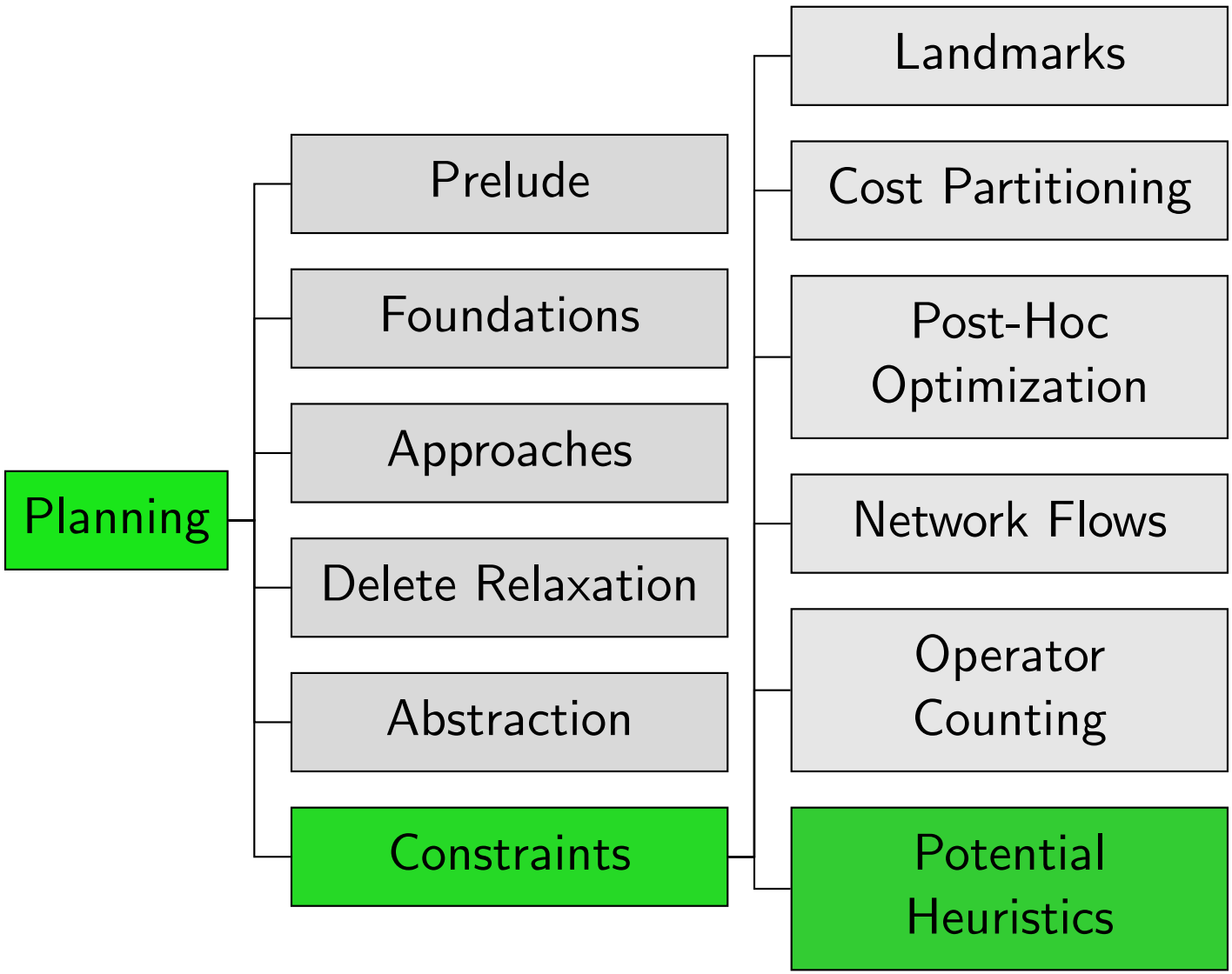
Planning and Optimization
December 15, 2025 — F12. Potential Heuristics

F12.1 Introduction

F12.2 Potential Heuristics

F12.3 Summary

Content of the Course



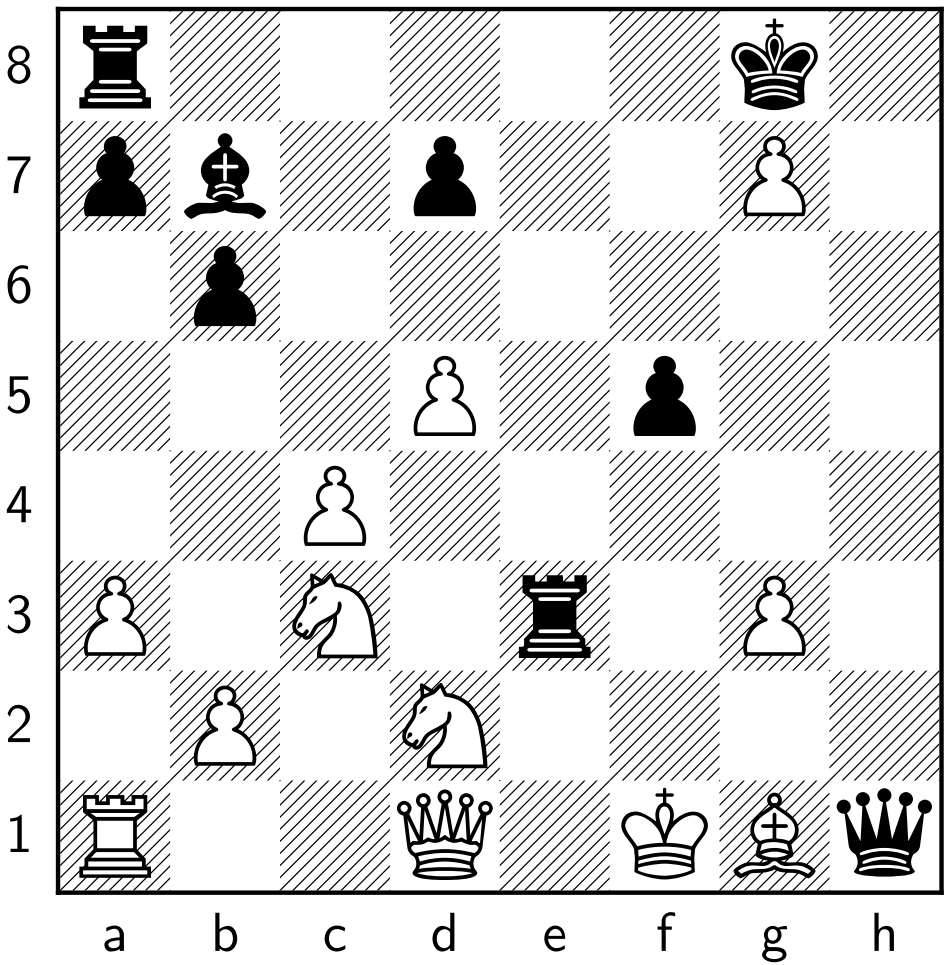
F12.1 Introduction

Reminder: Transition Normal Form

In this chapter, we consider SAS^+ tasks in transition normal form.

- A TNF operator mentions the same variables in the precondition and in the effect.
- A TNF goal specifies a value for every variable.

Material Value of a Chess Position



Material value for white:

$$
\begin{aligned}
&+ 1 \cdot 6 && \text{(white pawns)} \\
&- 1 \cdot 4 && \text{(black pawns)} \\
&+ 3 \cdot 2 && \text{(white knights)} \\
&- 3 \cdot 0 && \text{(black knights)} \\
&+ 3 \cdot 1 && \text{(white bishops)} \\
&- 3 \cdot 1 && \text{(black bishops)} \\
&+ 5 \cdot 1 && \text{(white rooks)} \\
&- 5 \cdot 2 && \text{(black rooks)} \\
&+ 9 \cdot 1 && \text{(white queen)} \\
&- 9 \cdot 1 && \text{(black queen)} \\
&= 3
\end{aligned}
$$

Idea

- Define simple numerical state features $f_1, \ldots, f_n$.
- Consider heuristics that are linear combinations of features:

$$h(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

  with weights (potentials) $w_i \in \mathbb{R}$
- heuristic very fast to compute if feature values are

F12.2 Potential Heuristics

Definition

Definition (Feature)

A (state) feature for a planning task is a numerical function defined on the states of the task: $f : S \to \mathbb{R}$.

Definition (Potential Heuristic)

A potential heuristic for a set of features $\mathcal{F} = \{f_1, \ldots, f_n\}$ is a heuristic function h defined as a linear combination of the features:

$$h(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

with weights (potentials) $w_i \in \mathbb{R}$.

Many possibilities $\Rightarrow$ need some restrictions

Features for SAS$^+$ Planning Tasks

Which features are good for planning?

Atomic features test if some atom is true in a state:

Definition (Atomic Feature)

Let $v = d$ be an atom of a FDR planning task.

The atomic feature $f_{v=d}$ is defined as:

$$f_{v=d}(s) = [(v = d) \in s] = \begin{cases} 1 & \text{if variable } v \text{ has value } d \text{ in state } s \\ 0 & \text{otherwise} \end{cases}$$

Offer good tradeoff between computation time and guidance

Example: Atomic Features

Example

Consider a planning task Π with state variables v_1 and v_2 and $\mathrm{dom}(v_1) = \mathrm{dom}(v_2) = \{d_1, d_2, d_3\}$. The set

$$\mathcal{F} = \{f_{v_i=d_j} \mid i \in \{1,2\}, j \in \{1,2,3\}\}$$

is the set of atomic features of Π and the function

$$h(s) = 3f_{v_1=d_1} + 0.5f_{v_1=d_2} - 2f_{v_1=d_3} + 2.5f_{v_2=d_1}$$

is a potential heuristic for $\mathcal{F}$.
The heuristic estimate for a state $s = \{v_1 \mapsto d_2, v_2 \mapsto d_1\}$ is

$$h(s) = 3 \cdot 0 + 0.5 \cdot 1 - 2 \cdot 0 + 2.5 \cdot 1 = 3.$$

Potentials for Optimal Planning

Which potentials are good for optimal planning
and how can we compute them?

- We seek potentials for which h is admissible and well-informed
  $\Rightarrow$ declarative approach to heuristic design
- We derive potentials for atomic features by solving an optimization problem

How to achieve this? Linear programming to the rescue!

Admissible and Consistent Potential Heuristics

We achieve admissibility through goal-awareness and consistency

Goal-awareness

$$\sum_{a \in \gamma} w_a = 0$$

Consistency

$$\sum_{a \in s} w_a - \sum_{a \in s'} w_a \leq cost(o) \quad \text{for all transitions } s \xrightarrow{o} s'$$

One constraint transition per transition.

Can we do this more compactly?

Admissible and Consistent Potential Heuristics

Consistency for a transition $s \xrightarrow{o} s'$

$$
\begin{aligned}
cost(o) &\geq \sum_{a \in s} w_a - \sum_{a \in s'} w_a \\
&= \sum_{a} w_a[a \in s] - \sum_{a} w_a[a \in s'] \\
&= \sum_{a} w_a([a \in s] - [a \in s']) \\
&= \sum_{a} w_a[a \in s \text{ but } a \notin s'] - \sum_{a} w_a[a \notin s \text{ but } a \in s'] \\
&= \sum_{\substack{a \text{ consumed} \\ \text{by } o}} w_a \; - \sum_{\substack{a \text{ produced} \\ \text{by } o}} w_a
\end{aligned}
$$

Admissible and Consistent Potential Heuristics

Goal-awareness and Consistency independent of s

Goal-awareness

$$\sum_{a \in \gamma} w_a = 0$$

Consistency

$$\sum_{\substack{a \text{ consumed} \\ \text{by } o}} w_a - \sum_{\substack{a \text{ produced} \\ \text{by } o}} w_a \leq cost(o) \quad \text{for all operators } o$$

Potential Heuristics

- All atomic potential heuristics that satisfy these constraints are admissible and consistent
- Furthermore, all admissible and consistent atomic potential heuristics satisfy these constraints

Constraints are a compact characterization of all admissible and consistent atomic potential heuristics.

LP can be used to find the best admissible and consistent potential heuristics by encoding a quality metric in the objective function

Well-Informed Potential Heuristics

What do we mean by the best potential heuristic?
Different possibilities, e.g., the potential heuristic that

- maximizes heuristic value of a given state s (e.g., initial state)
- maximizes average heuristic value of all states (including unreachable ones)
- maximizes average heuristic value of some sample states
- minimizes estimated search effort

Potential and Flow Heuristic

Theorem

For state s, let $h^{\text{maxpot}}(s)$ denote the maximal heuristic value of all admissible and consistent atomic potential heuristics in s.

Then $h^{\text{maxpot}}(s) = h^{flow}(s)$.

Proof idea: compare dual of $h^{\text{flow}}(s)$ LP to potential heuristic constraints optimized for state s.

If we optimize the potentials for a given state then for this state it equals the flow heuristic.

F12.3 Summary

Summary

- Potential heuristics are computed as a weighted sum of state features
- Admissibility and consistency can be encoded compactly in constraints
- With linear programming, we can efficiently compute the best potential heuristic wrt some objective
- Potential heuristics can be used as fast admissible approximations of h^{flow}.