Foundations of Artificial Intelligence

A1. Organizational Matters

Malte Helmert

University of Basel

February 17, 2025

Introduction: Overview

Chapter overview: introduction

- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

People

Teaching Staff: Lecturer

Lecturer

Prof. Dr. Malte Helmert

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

Teaching Staff: Assistant

Assistant

Dr. Florian Pommerening

- email: florian.pommerening@unibas.ch
- office: room 04.005, Spiegelgasse 1

Teaching Staff: Tutors

Tutors

Remo Christen

- email: remo.christen@unibas.ch
- office: room 04.001, Spiegelgasse 5

Simon Dold

- email: simon.dold@unibas.ch
- office: room 04.001, Spiegelgasse 5

Claudia Grundke

- email: claudia.grundke@unibas.ch
- office: room 04.001, Spiegelgasse 5

Students

target audience:

- Bachelor Computer Science, ∼3rd year
- Bachelor Computational Sciences, ∼3rd year
- Master Data Science
- other students welcome

prerequisites:

- algorithms and data structures
- basic mathematical concepts
  (formal proofs; sets, functions, relations, graphs)
- complexity theory
- programming skills (mainly for exercises)

Format

Structure Overview

Foundations of AI week structure:

- Monday: release of exercise sheet
- Monday and Wednesday: lectures
- Wednesday: exercise session
- Sunday: exercise sheet due
- exceptions due to holidays

Time & Place

Lectures

- Mon 16:15–18:00 in Biozentrum, lecture hall U1.141
- Wed 14:15–16:00 in Biozentrum, lecture hall U1.141

Exercise Sessions

- Wed 16:15–18:00 in Biozentrum, SR U1.195
- Fri 10:15–12:00 in Spiegelgasse 1, room U1.001 (changed)

first exercise session: February 19 (this week)

Exercises

exercise sheets (homework assignments):

- mostly theoretical exercises
- occasional programming exercises

exercise sessions:

- initial part:
  - discuss common mistakes in previous exercise sheet
  - answer questions on previous exercise sheet
- main part:
  - we support you solving the current exercise sheet
  - we answer your questions
  - we assist you comprehend the course content

Theoretical Exercises

theoretical exercises:

- exercises on ADAM every Monday
- covers material of that week (Monday and Wednesday)
- due Sunday of the same week (23:59) via ADAM
- solved in groups of at most two ($2 = 2$)
- support in exercise session of current week
- discussed in exercise session of following week

Programming Exercises

programming exercises (project):

- project with 3–4 parts over the duration of the semester
- additional one-off programming exercises (not on every sheet)
- integrated into the exercise sheets (no special treatment)
- solved in groups of at most two ($2 < 3$)
- implemented in Java; need working Linux system for some
- solutions that obviously do not work: 0 marks

Assessment

Course Material

course material that is relevant for the exam:

- slides
- content of lecture
- exercise sheets

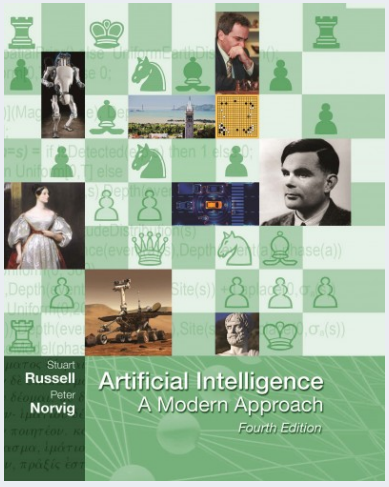
additional (optional) course material:

- textbook
- bonus material

Textbook

Artificial Intelligence: A Modern Approach
by Stuart Russell and Peter Norvig
(4th edition, Global edition)

- covers large parts of the course (and much more), but not everything



Exam

- written exam on Wednesday, July 2
  - 14:00-16:00
  - 105 minutes for working on the exam
  - location: Biozentrum, lecture hall U1.131
- 8 ECTS credits
- admission to exam: 50% of the exercise marks
- class participation not required but highly recommended
- no repeat exam

Plagiarism

Plagiarism (Wikipedia)

Plagiarism is the "wrongful appropriation" and "stealing and publication" of another author's "language, thoughts, ideas, or expressions" and the representation of them as one's own original work.

consequences:

- 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? Join the exercise session!

Tools

Course Homepage and Enrolment

Course Homepage

`https://dmi.unibas.ch/en/studium/computer-science-informatik/lehrangebot-fs25/13548-lecture-foundations-of-artificial-intelligence/`

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

enrolment:

- `https://services.unibas.ch/`

Communication Channels

Communication Channels

- lectures and exercise sessions
- ADAM workspace (linked from course homepage)
  - link to Discord server
  - exercise sheets and submission
  - exercise FAQ
  - bonus material that we cannot share publicly
- Discord server (linked from ADAM workspace)
  - opportunity for Q&A and informal interactions
- contact us by email
- meet us in person (by arrangement)
- meet us on Zoom (by arrangement)

About this Course

Classical AI Curriculum

"Classical" AI Curriculum

1. introduction
2. rational agents
3. uninformed search
4. informed search
5. constraint satisfaction
6. board games
7. propositional logic
8. predicate logic
9. modeling with logic
10. classical planning
11. probabilistic reasoning
12. decisions under uncertainty
13. acting under uncertainty
14. machine learning
15. deep learning
16. reinforcement learning

$\rightsquigarrow$ wide coverage, but somewhat superficial

Our AI Curriculum

Our AI Curriculum

1. introduction
2. rational agents
3. uninformed search
4. informed search
5. constraint satisfaction
6. board games
7. propositional logic
8. ~~predicate logic~~
9. ~~modeling with logic~~
10. classical planning
11. ~~probabilistic reasoning~~
12. ~~decisions under uncertainty~~
13. acting under uncertainty
14. ~~machine learning~~
15. ~~deep learning~~
16. ~~reinforcement learning~~

Topic Selection

guidelines for topic selection:

- fewer topics, more depth
- more emphasis on programming projects
- connections between topics
- avoiding overlap with other courses
  - Pattern Recognition (B.Sc.)
  - Machine Learning (M.Sc.)
- focus on algorithmic core of model-based AI

Under Construction...

- A course is never “done”.
- We are always happy about feedback, corrections and suggestions!

Foundations of Artificial Intelligence

A2. Introduction: What is Artificial Intelligence?

Malte Helmert

University of Basel

February 17, 2025

Introduction: Overview

Chapter overview: introduction

- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

What is AI?

What is AI?

What do we mean by artificial intelligence?

$\rightsquigarrow$ no generally accepted definition!

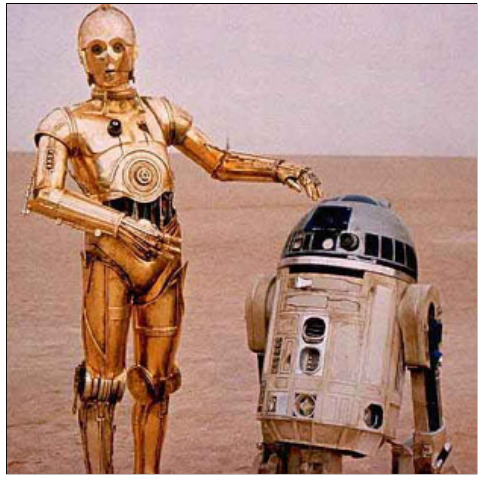
often pragmatic definitions:

- "AI is what AI researchers do."
- "AI is the solution of hard problems."

in this chapter: some common attempts at defining AI

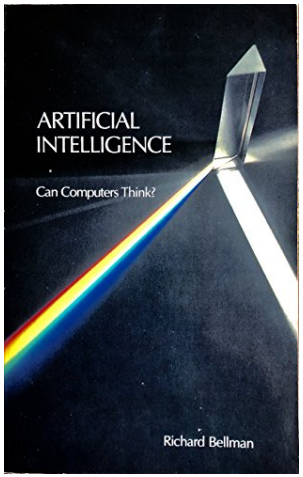
What Do We Mean by Artificial Intelligence?

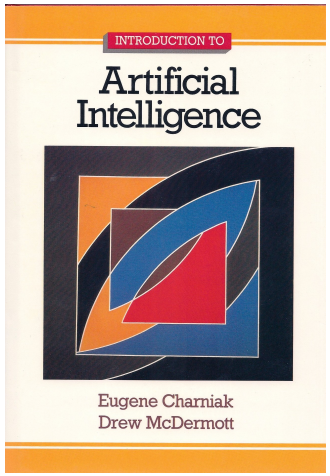
what pop culture tells us:

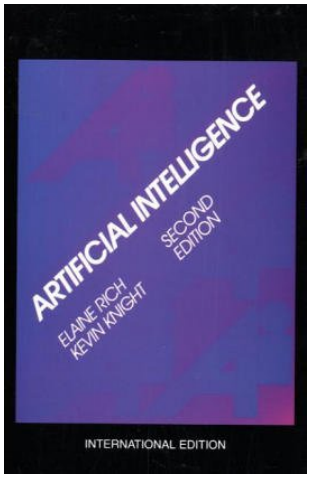


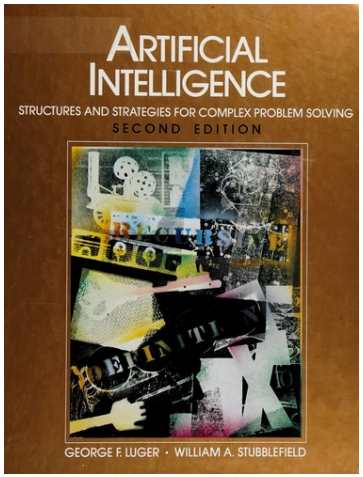
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:





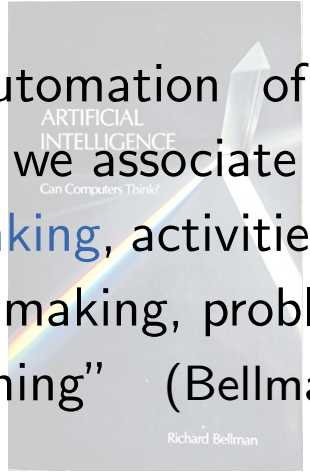


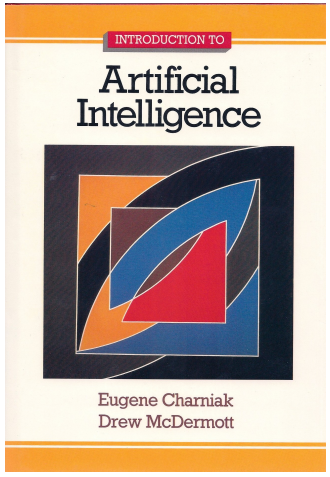


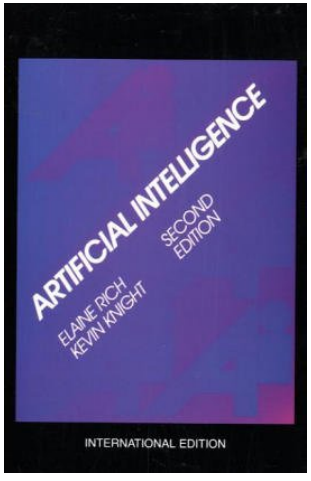
What is AI: Humanly vs. Rationally; Thinking vs. Acting

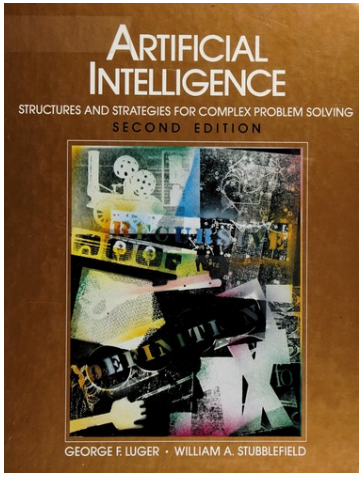
what scientists tell us:

"[the automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning" (Bellman, 1978)



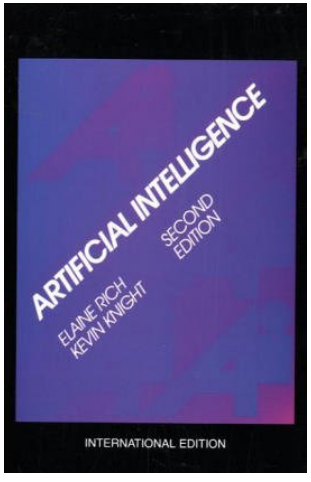
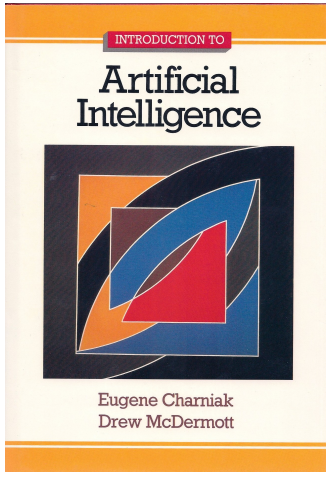
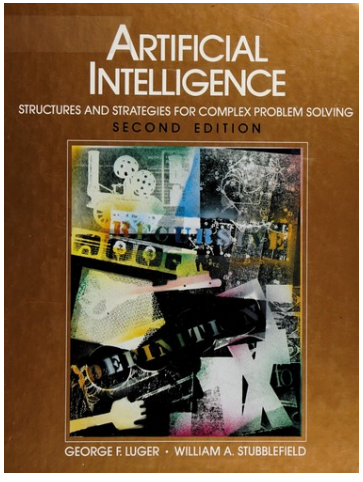






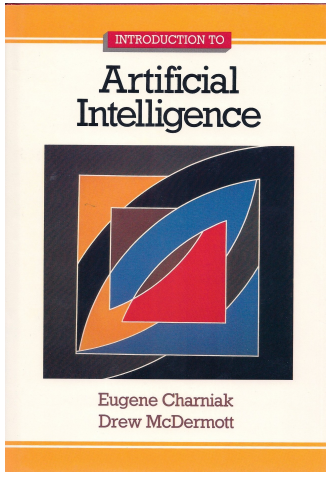
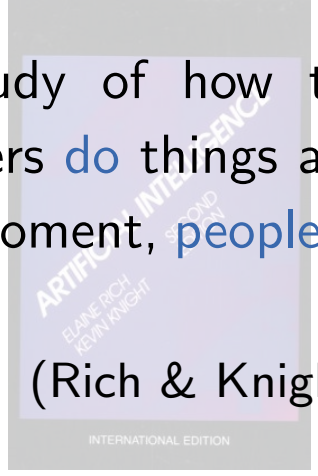
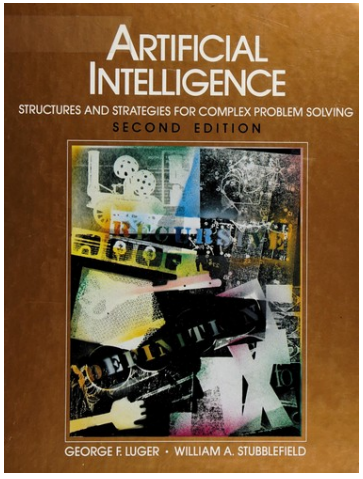
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans	
	
	

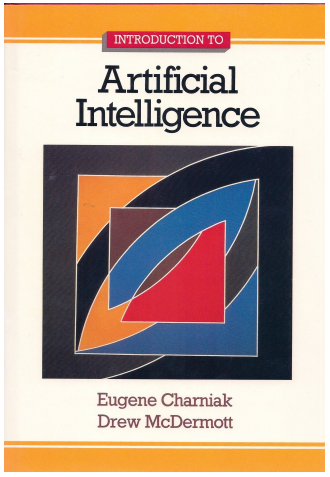
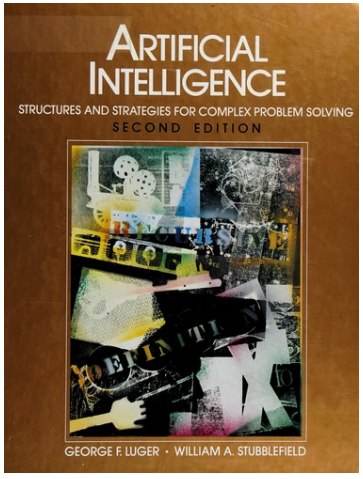
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans 	
"the study of how to make computers do things at which, at the moment, people are better" (Rich & Knight, 1991) 	

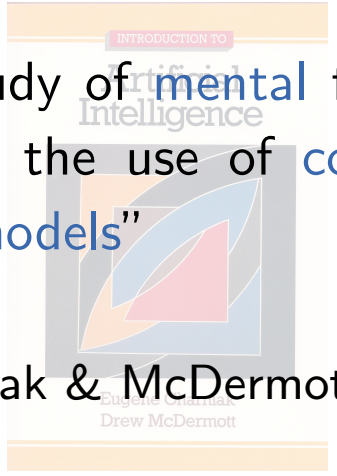
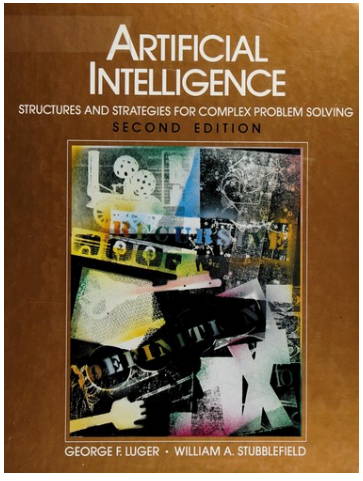
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans	 
acting like humans	 

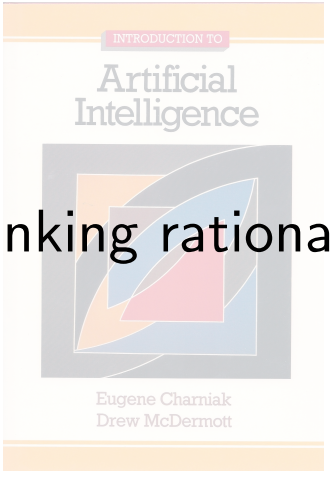
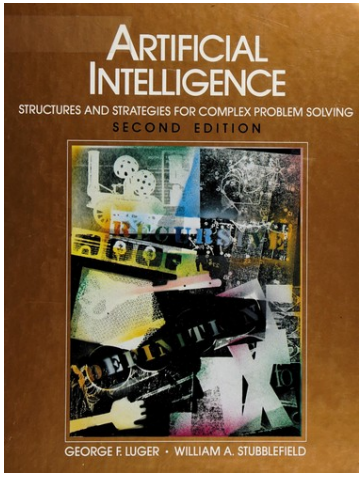
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans	
“the study of mental faculties through the use of computational models” (Charniak & McDermott, 1985)	
acting like humans	
	

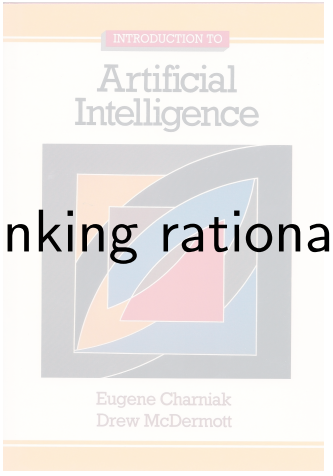
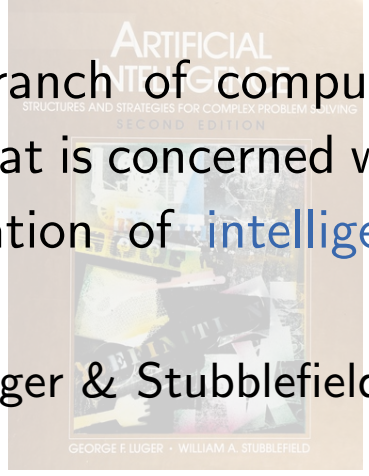
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans 	thinking rationally 
acting like humans 	

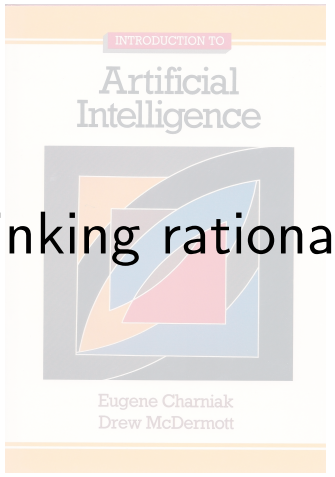
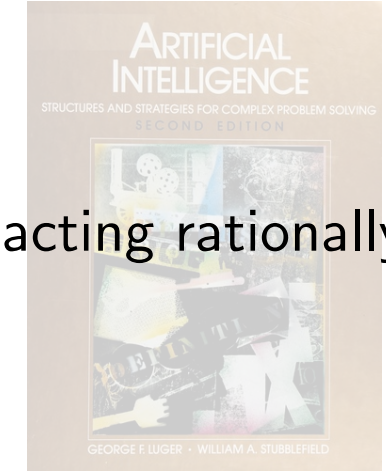
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

thinking like humans 	thinking rationally 
acting like humans 	"the branch of computer science that is concerned with the automation of intelligent behavior" (Luger & Stubblefield, 1993) 

What is AI: Humanly vs. Rationally; Thinking vs. Acting

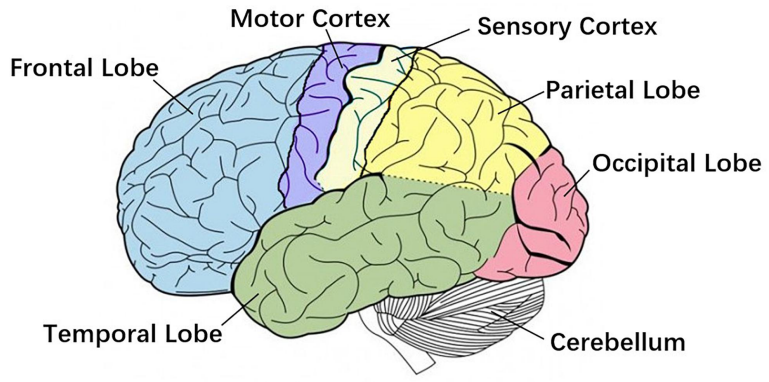
what scientists tell us:

thinking like humans 	thinking rationally 
acting like humans 	acting rationally 

Thinking Like Humans

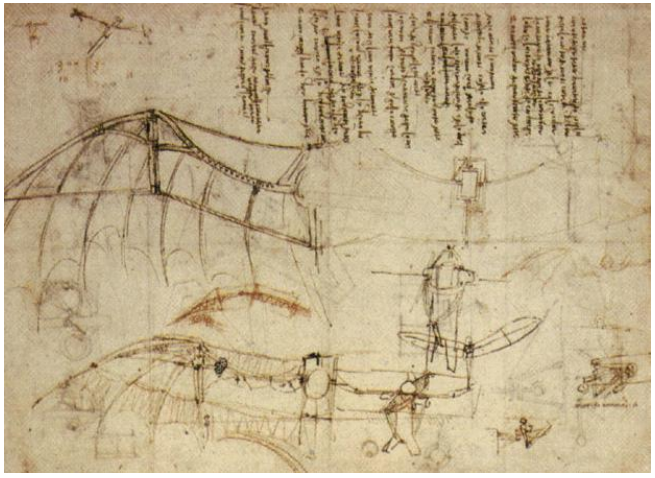
Cognitive (Neuro-) Science

- requires knowledge of how humans think
- two ways to a scientific theory of brain activity:
  - psychological: observation of human behavior
  - neurological: observation of brain activity



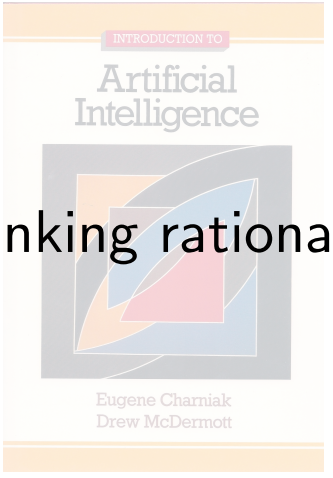
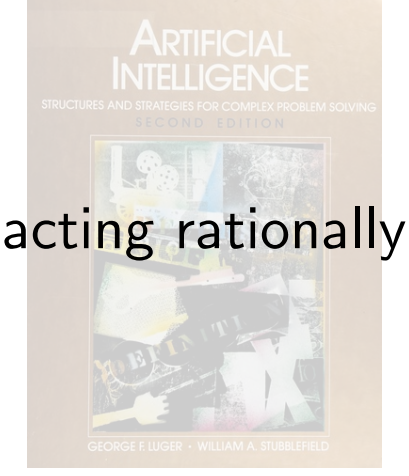
- roughly corresponds to cognitive science and cognitive neuroscience
- today separate research areas from AI

Machines that Think Like Humans



"brains are to intelligence as wings are to flight"

What Do We Mean by Artificial Intelligence?

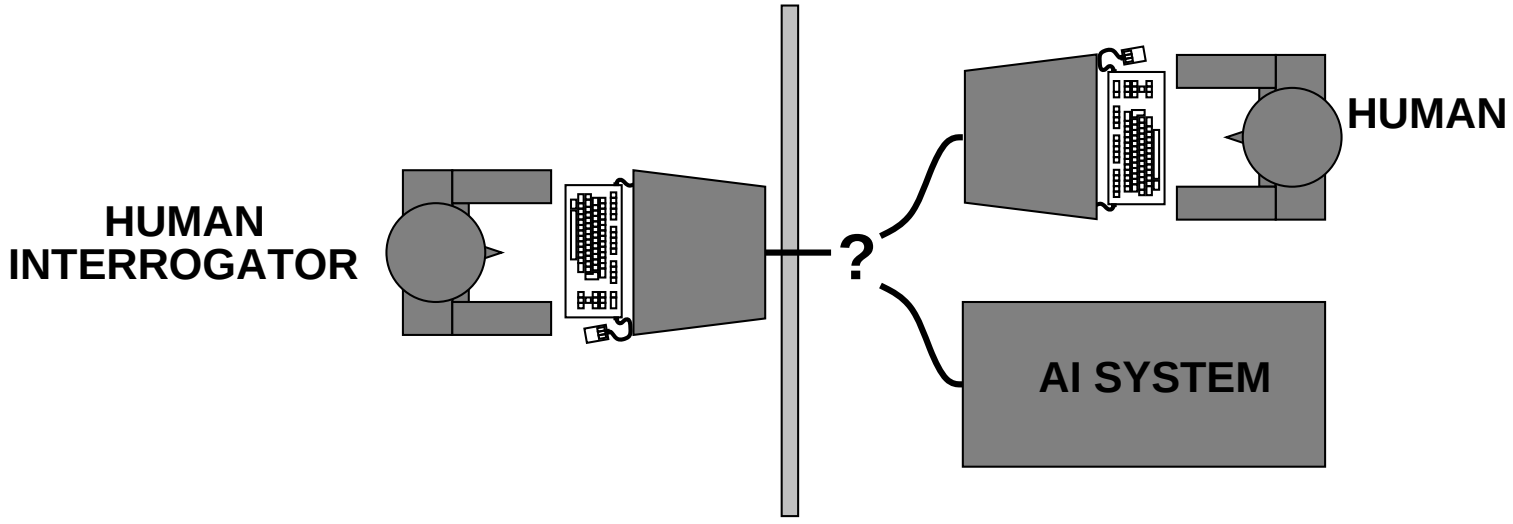
thinking like humans 	thinking rationally 
acting like humans 	acting rationally 

Acting Like Humans

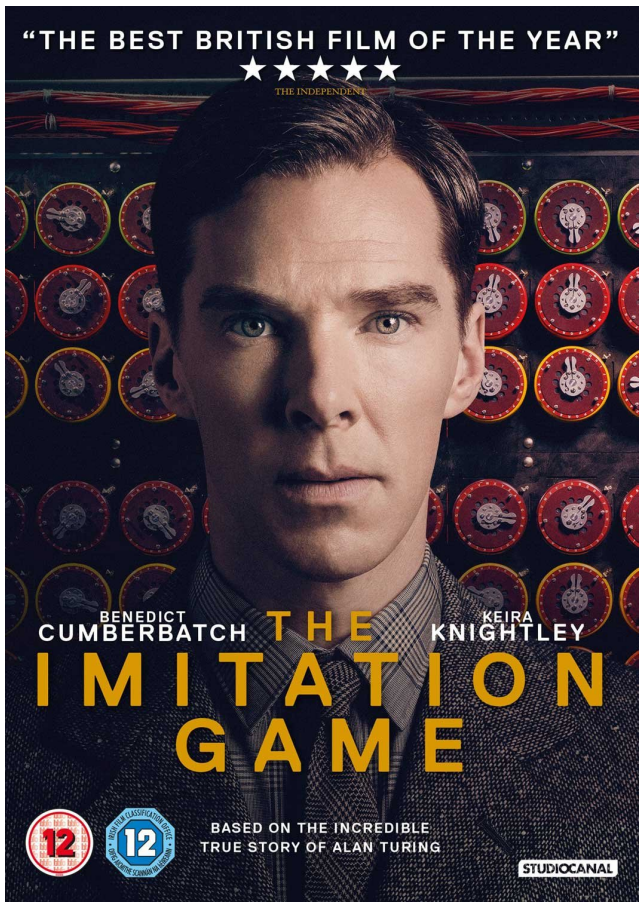
The Turing Test

Alan Turing, Computing Machinery and Intelligence (1950):

- central question: Can machines think?
- hypothesis: yes, if they can act like humans
- operationalization: the imitation game

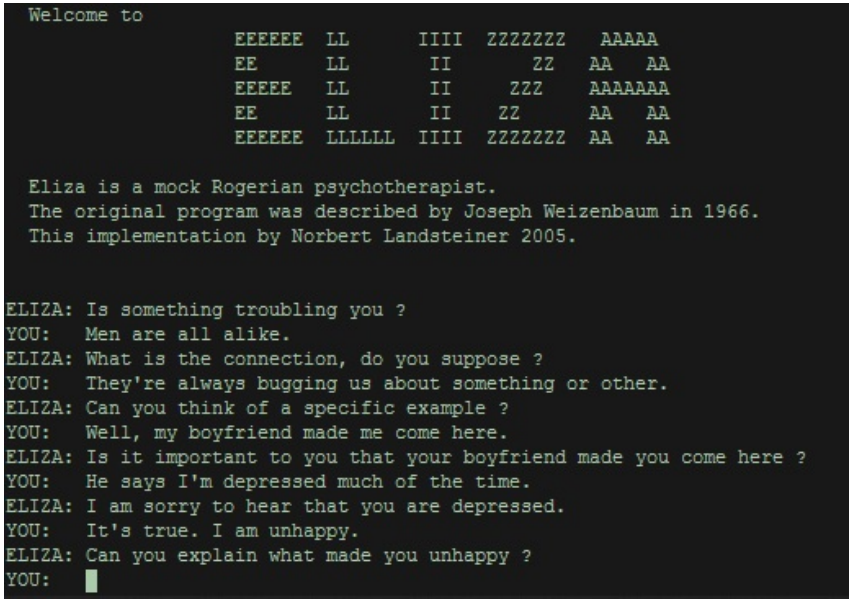


Turing Test in Cinema



Turing Test: Brief History

- Eliza



- developed in 1966 by J. Weizenbaum
- uses combination of pattern matching and scripted rules
- most famous script mimics a psychologist $\rightsquigarrow$ many questions
- fooled early users

Turing Test: Brief History

- Eliza
- Loebner Prize

- annual competition between 1991–2019
- most human-like AI is awarded
- highly controversial

Turing Test: Brief History

- Eliza
- Loebner Prize
- Eugene Goostman

- mimics a 13-year-old boy from Odessa, Ukraine with a guinea pig
- "not too old to know everything and not too young to know nothing"
- 33% of judges were convinced it was human in 2014
  $\rightsquigarrow$ first system that passed the Turing test (?)

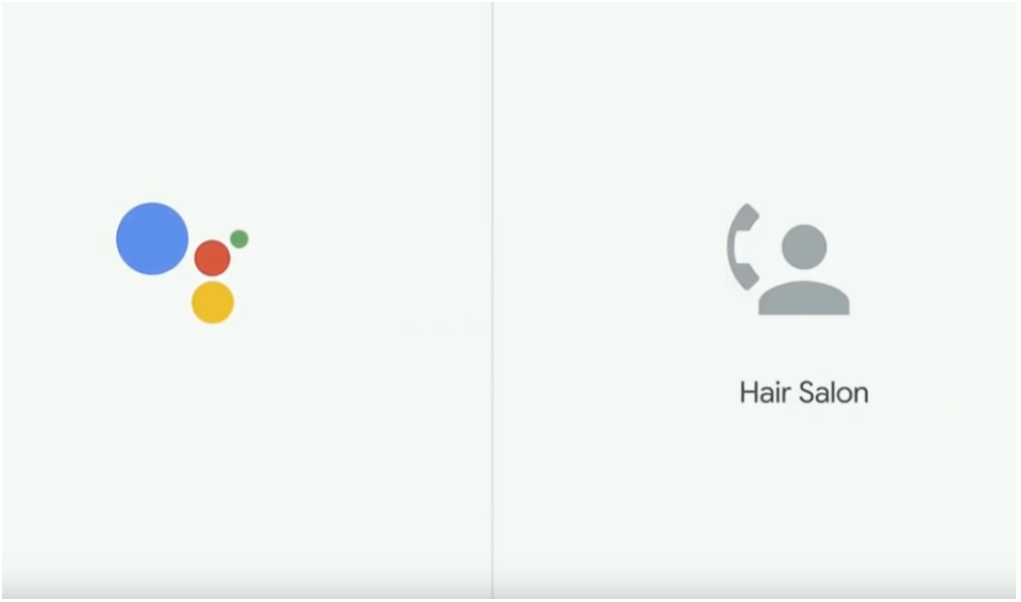
Turing Test: Brief History

- Eliza
- Loebner Prize
- Eugene Goostman
- Kuki (formerly Mitsuku)

- five times winner of Loebner prize competitions (2015-2019)
- winner of "bot battle" versus Facebook's Blenderbot
  ⇝ `https://youtu.be/RBK5j0yXDT8`

Turing Test: Brief History

- Eliza
- Loebner Prize
- Eugene Goostman
- Kuki (formerly Mitsuku)
- Google Duplex



- commercial product announced in 2018
- performs phone calls (making appointments) fully autonomously
- after criticism, it now starts conversation by identifying as a robot

Turing Test: Brief History

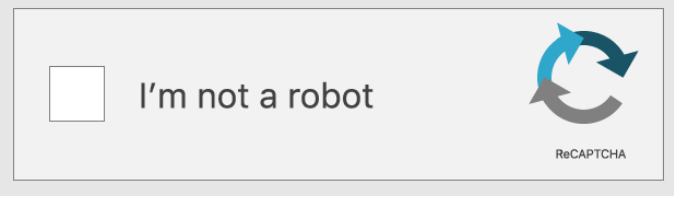
- Eliza
- Loebner Prize
- Eugene Goostman
- Kuki (formerly Mitsuku)
- Google Duplex
- LaMDA & ChatGPT

- systems like LaMDA and ChatGPT would likely pass the Turing test
- example conversation: `https://www.nytimes.com/2023/02/16/technology/bing-chatbot-transcript.html`
- ChatGPT even passed some exams (but failed on others)

Value of the Turing Test

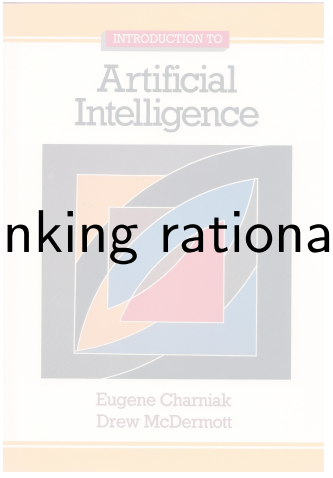
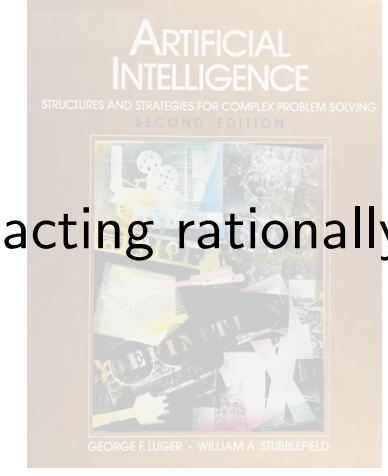
- human actions not always intelligent
- scientific value of Turing test questionable:
  - Test for AI or for interrogator?
  - results not reproducible
  - strategies to succeed $\neq$ intelligence:
    - deceive interrogator
    - mimic human behavior

$\rightsquigarrow$ not important in AI "mainstream"



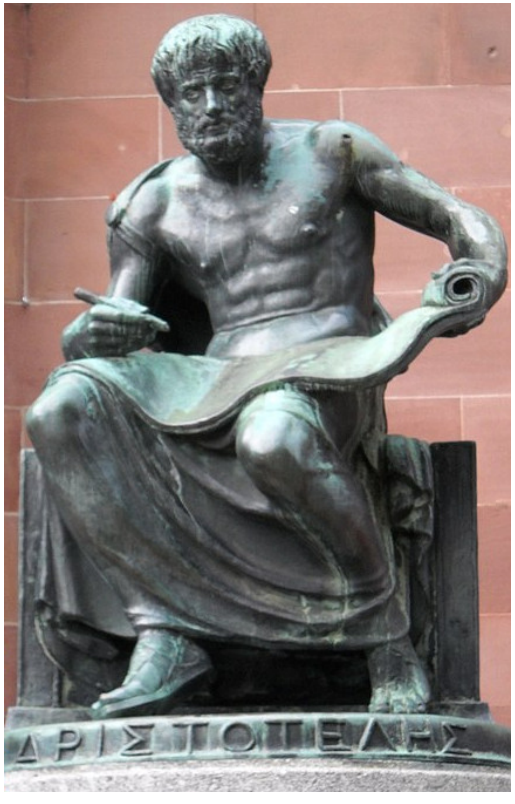
practical application: CAPTCHA ("**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part")

What Do We Mean by Artificial Intelligence?

thinking like humans 	thinking rationally 
acting like humans 	acting rationally 

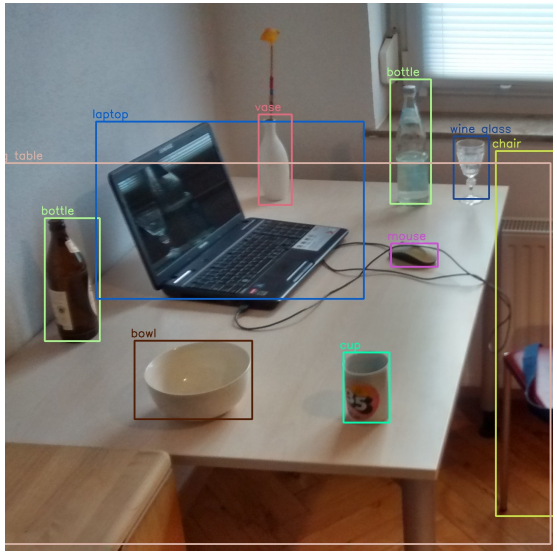
Thinking Rationally

Thinking Rationally: Laws of Thought

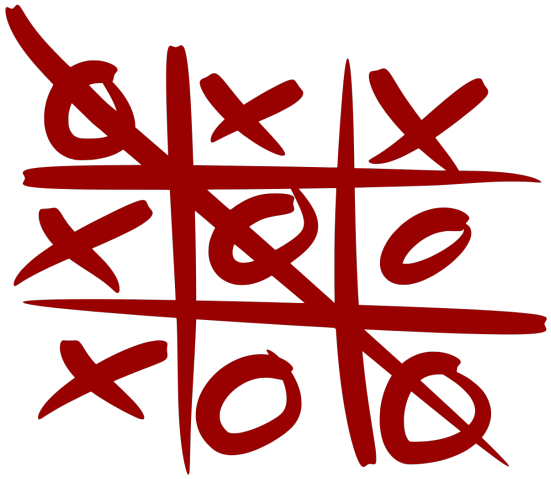


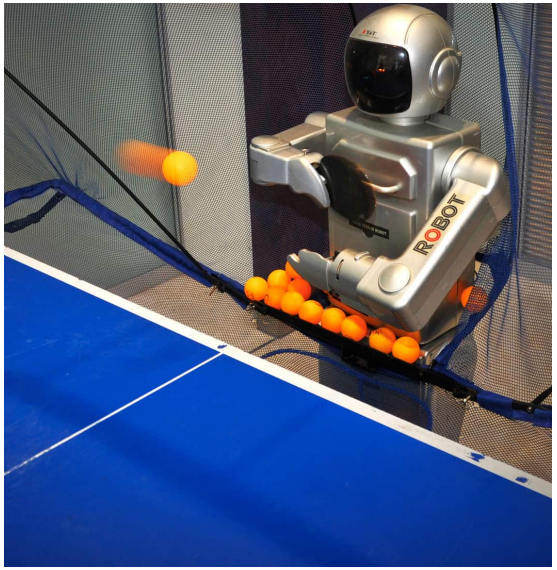
- Aristotle: What are correct arguments and modes of thought?
- syllogisms: structures for arguments that always yield correct conclusions given correct premises:
  - Socrates is a human.
  - All humans are mortal.
  - Therefore Socrates is mortal.
- direct connection to modern AI via mathematical logic

Problems of the Logical Approach



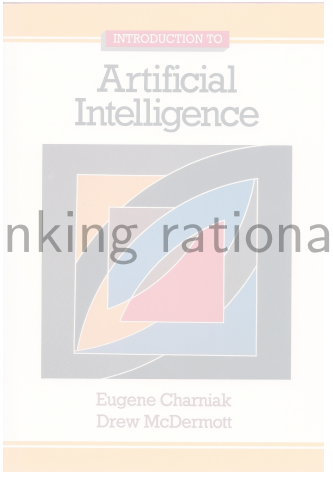
not all intelligent behavior stems from logical thinking and formal reasoning





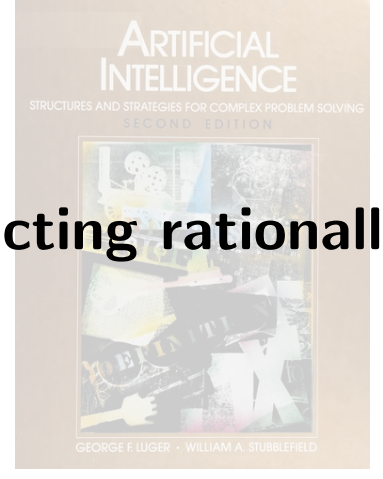
What Do We Mean by Artificial Intelligence?

thinking like humans



thinking rationally

acting like humans



acting rationally

Acting Rationally

Acting Rationally

acting rationally: "doing the right thing"

- the right thing: maximize utility
  given available information
- does not necessarily require "thought" (e.g., reflexes)

advantages of AI as development of rational agents:

- more general than thinking rationally
  (logical inference only one way to obtain rational behavior)
- better suited for scientific method
  than approaches based on human thinking and acting

$\rightsquigarrow$ most common view of AI scientists today
$\rightsquigarrow$ what we use in this course

Summary

Summary

What is AI? $\rightsquigarrow$ many possible definitions

- guided by humans vs. by utility (rationality)
- based on externally observable actions or inner thoughts?

$\rightsquigarrow$ four combinations:

- acting like humans: e.g., Turing test
- thinking like humans: cf. cognitive (neuro-)science
- thinking rationally: logic
- acting rationally: most common view today

  $\rightsquigarrow$ amenable to scientific method

  $\rightsquigarrow$ used in this course

Foundations of Artificial Intelligence

A3. Introduction: AI Past and Present

Malte Helmert

University of Basel

February 19, 2025

Introduction: Overview

Chapter overview: introduction

- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

A Short History of AI

Precursors (Until ca. 1943)

Philosophy and mathematics ask similar questions that influence AI.

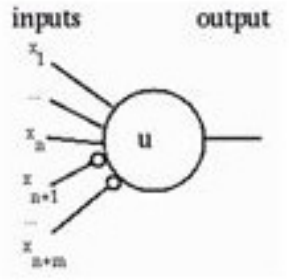
- Aristotle (384–322 BC)
- Leibniz (1646–1716)
- Hilbert program (1920s)

Gestation (1943–1956)

Invention of electrical computers raised question:
Can computers mimic the human mind?

Gestation (1943–1956)

Artificial Neurons



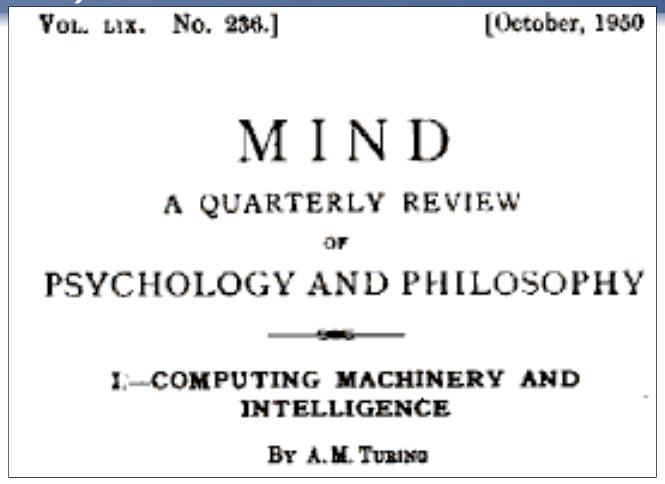
1950 1960 1970 1980 1990 2000 ...

W. McCulloch & W. Pitts (1943)

- first computational model of artificial neuron
- network of neurons can compute any computable function
- basis of deep learning

Gestation (1943–1956)

Artificial Neurons


Vol. lix. No. 236.] [October, 1950

MIND

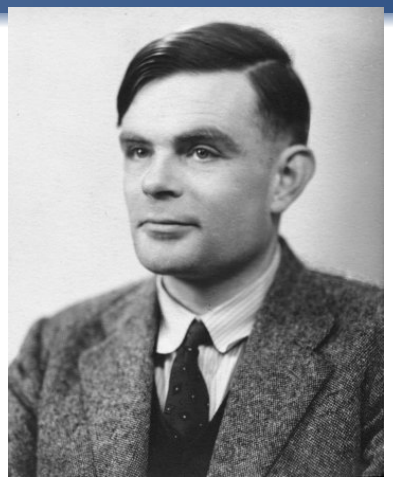
A QUARTERLY REVIEW

OF

PSYCHOLOGY AND PHILOSOPHY

I.—COMPUTING MACHINERY AND INTELLIGENCE

By A. M. Turing



1950 1960 1970 1980 1990 2000 ...

Turing Test

Computing Machinery and Intelligence (A. Turing, 1950)

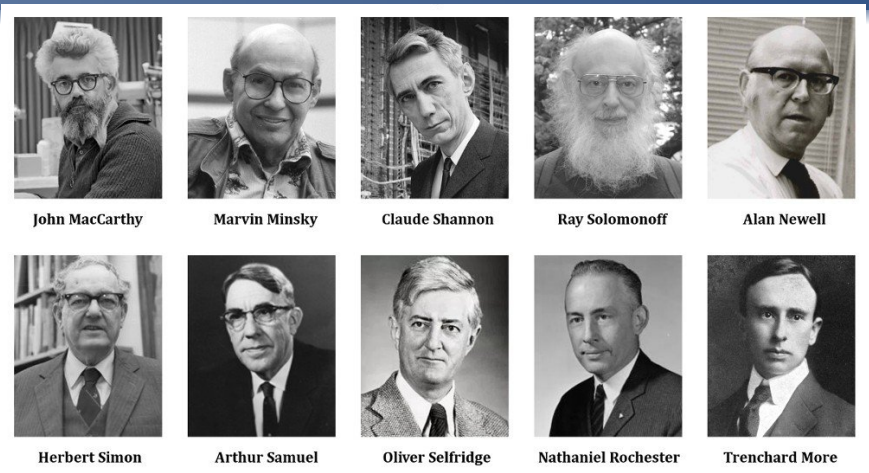
- famous for introducing Turing test
- (still) relevant discussion of AI potential and requirements
- suggests core AI aspects: knowledge representation, reasoning, language understanding, learning

Gestation (1943–1956)

Artificial Neurons

Dartmouth

Turing Test

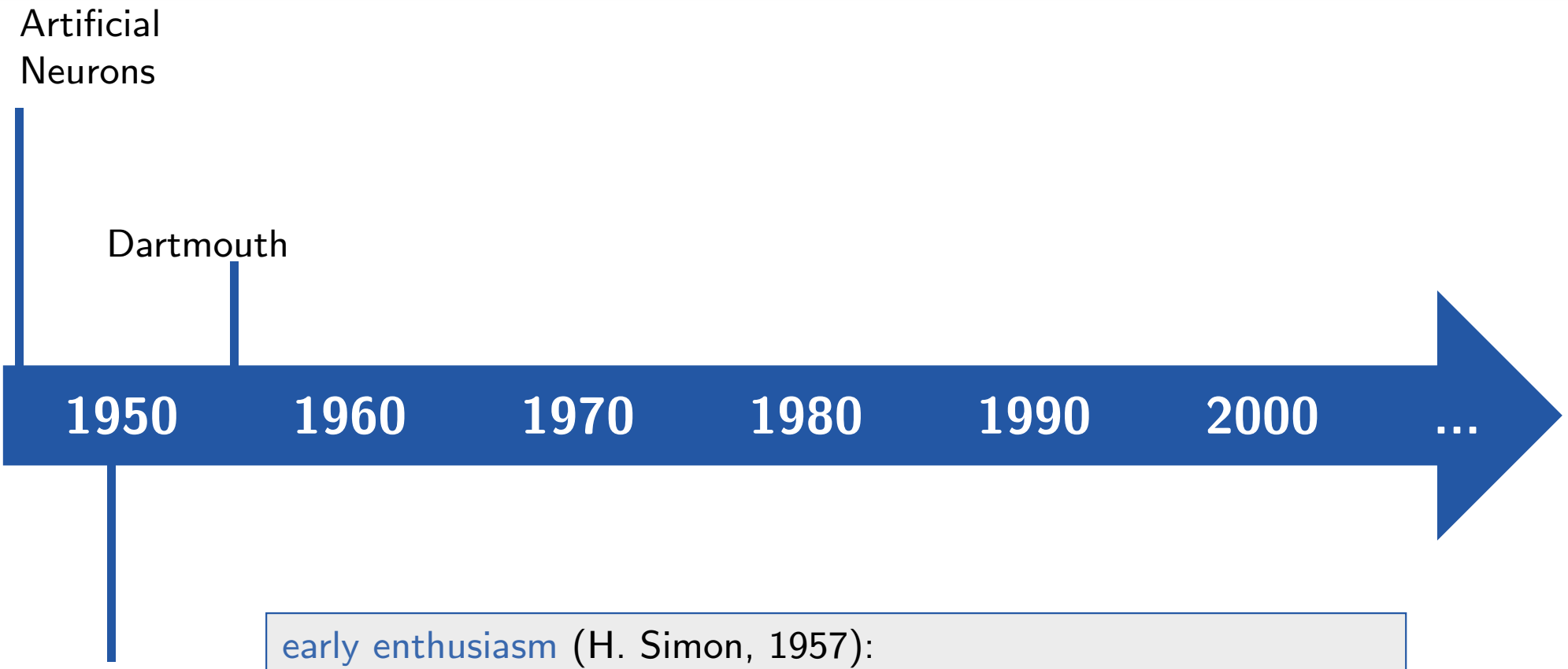


1950 1960 1970 1980 1990 2000 ...

Dartmouth workshop (1956)

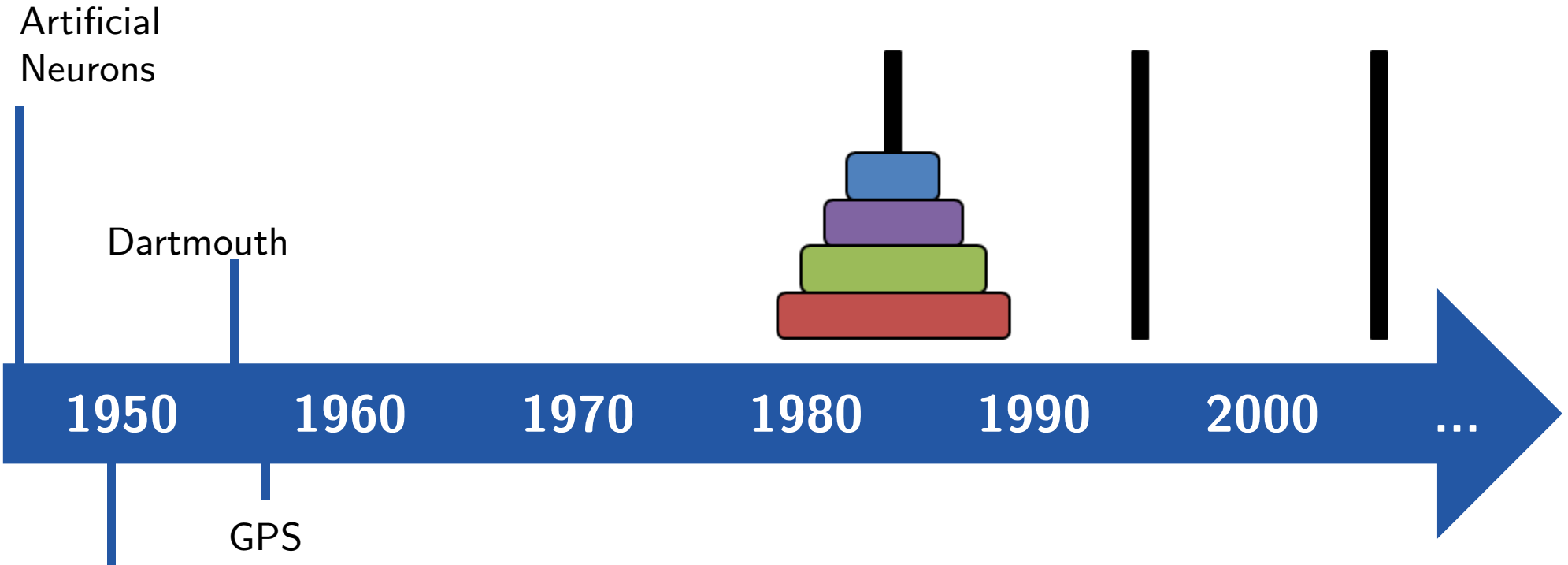
- ambitious proposal: "An attempt will be made to find how to make machines use language, [...] solve kinds of problems now reserved for humans, and improve themselves."
- J. McCarthy coins term artificial intelligence

Early Enthusiasm (1952–1969)



early enthusiasm (H. Simon, 1957):
"[...] there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until – in the visible future – the range of problems they can handle will be coextensive with the range to which the human mind has been applied."

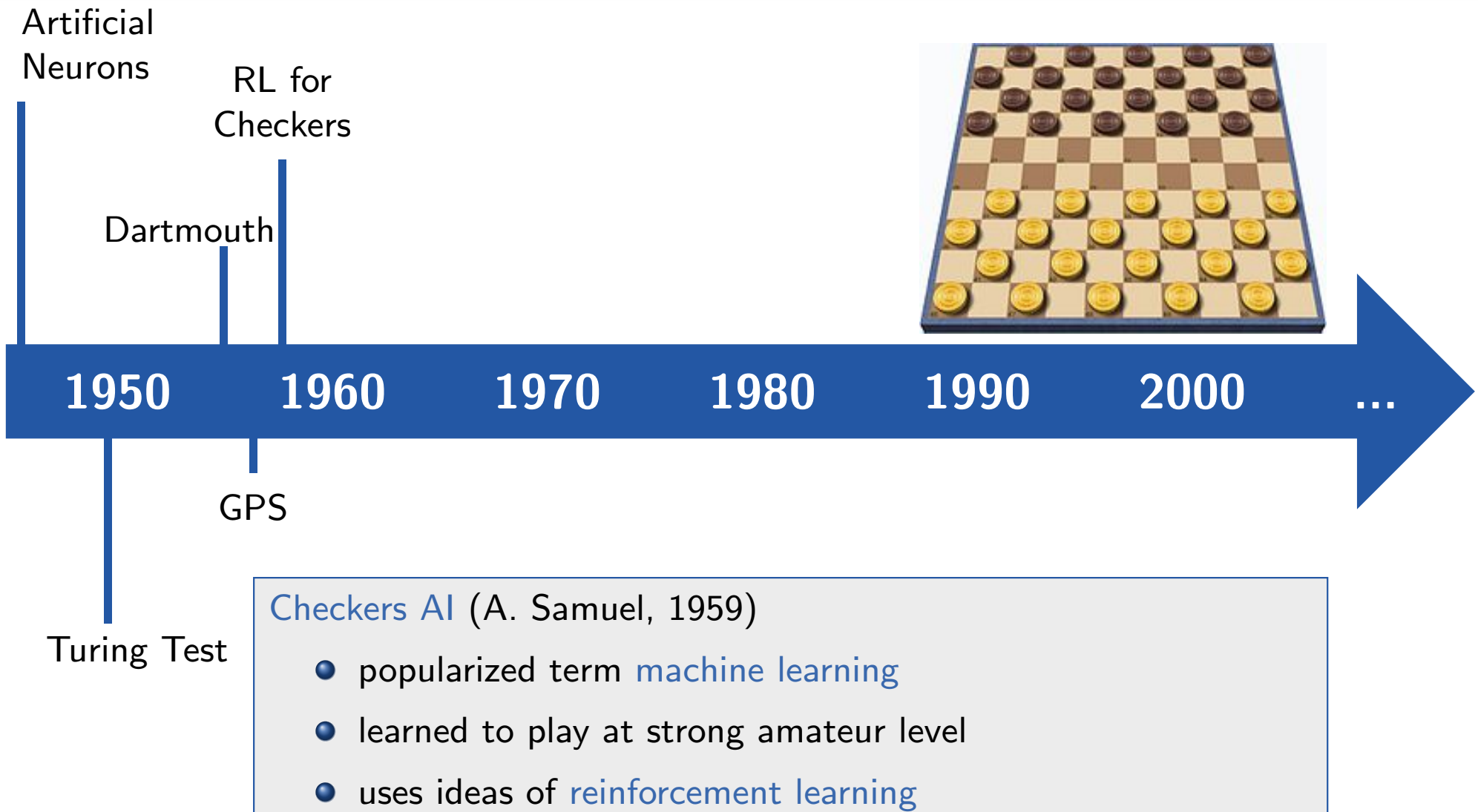
Early Enthusiasm (1952–1969)



General Problem Solver (H. Simon & A. Newell, 1957)

- universal problem solving machine
- imitates human problem solving strategies
- in principle able to solve every formalized symbolic problem
- in practice, GPS solves simple tasks like towers of Hanoi

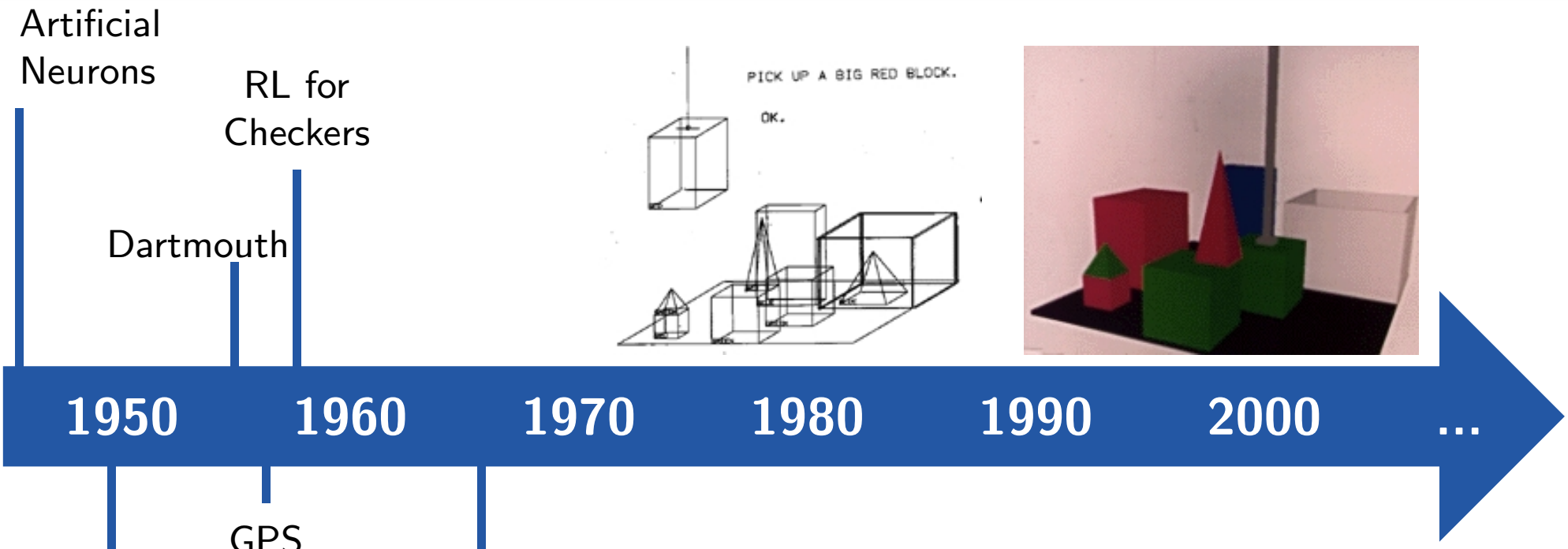
Early Enthusiasm (1952–1969)



Checkers AI (A. Samuel, 1959)

- popularized term **machine learning**
- learned to play at strong amateur level
- uses ideas of **reinforcement learning**

Early Enthusiasm (1952–1969)

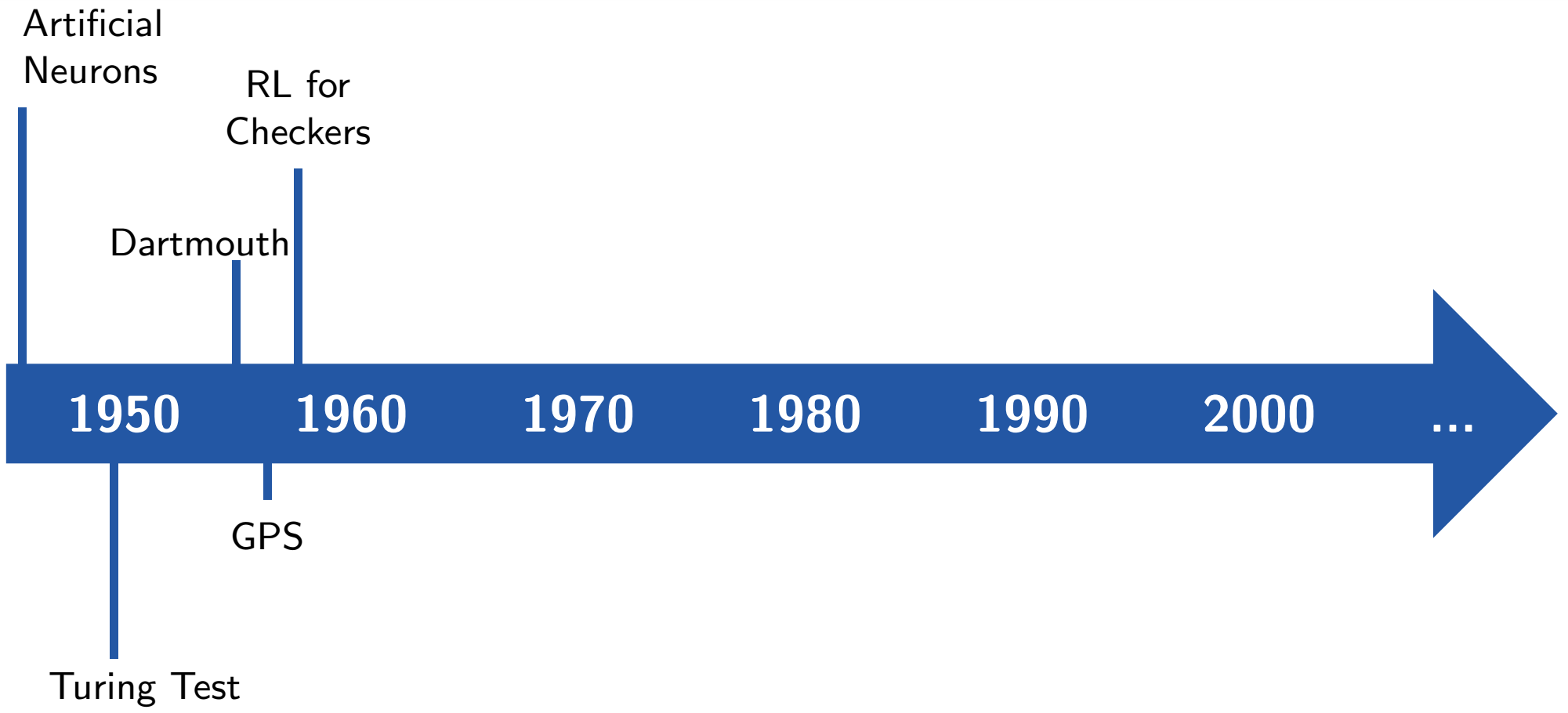


intelligence in microworlds, e.g. SHRDLU (T. Winograd, 1968)

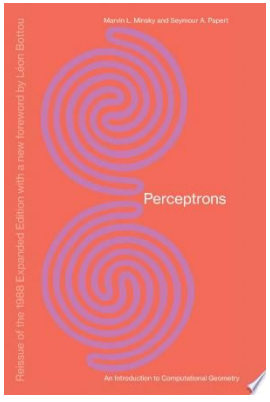
- understands natural language
- communicates with user via teletype on blocks world
- graphical representation

⇝ `https://hci.stanford.edu/winograd/shrdlu/`

Early Enthusiasm (1952–1969)

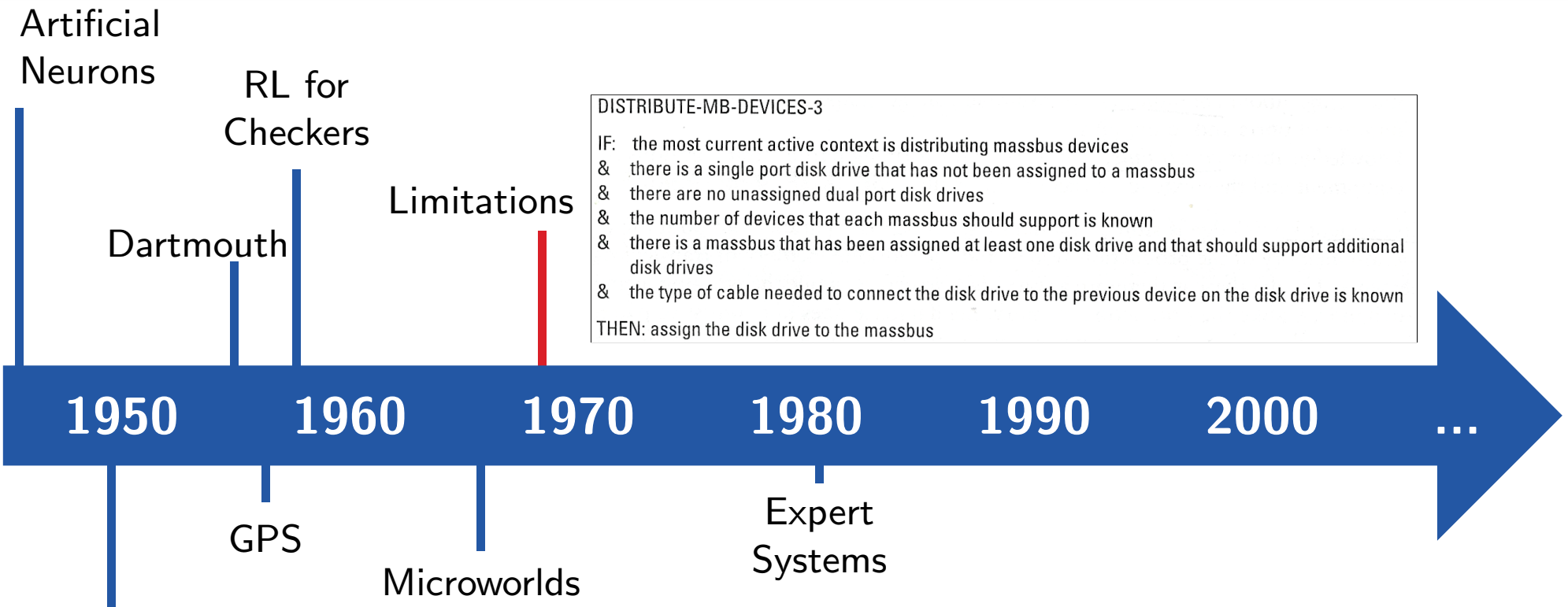


A Dose of Reality (1966–1973)



- realization that unlimited computational power is illusion (birth of complexity theory, NP-completeness)
- AI systems (e.g., GPS, systems for micro worlds) fail to scale
- fundamental limitations on basic structures
  e.g., XOR problem of perceptrons

Expert Systems (1969–1986)



DISTRIBUTE-MB-DEVICES-3

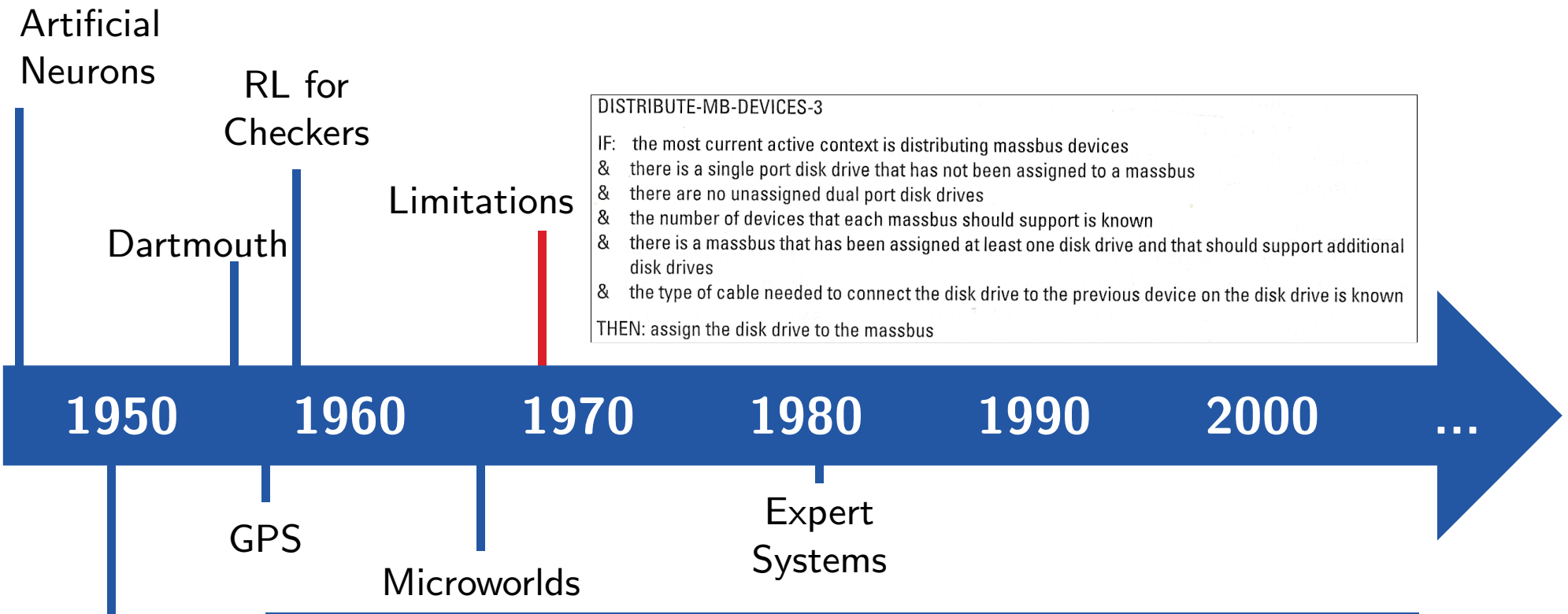
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned at least one disk drive and that should support additional disk drives
& the type of cable needed to connect the disk drive to the previous device on the disk drive is known

THEN: assign the disk drive to the massbus

1980s: AI gold rush

- rule-based expert systems commercially successful
- (human) expert knowledge as input
- allows automatic reasoning on larger problems in narrower applications
- also: second heyday of neural networks

Expert Systems (1969–1986)



DISTRIBUTE-MB-DEVICES-3

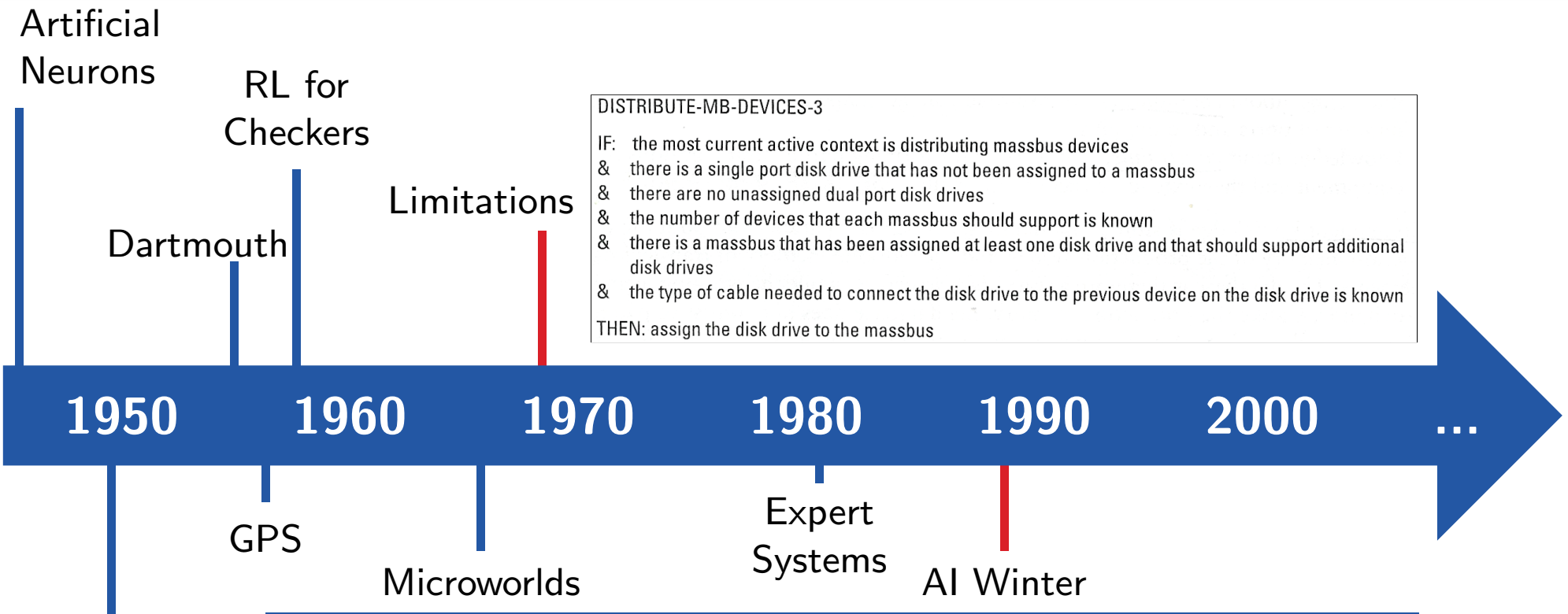
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned at least one disk drive and that should support additional disk drives
& the type of cable needed to connect the disk drive to the previous device on the disk drive is known

THEN: assign the disk drive to the massbus

example: R1/XCON (J. McDermott, 1978)

- input: desired properties of a VAX computer system according to customer specifications
- output: specification of the computer system
- inference engine: simple forward chaining of rules

Expert Systems (1969–1986)



DISTRIBUTE-MB-DEVICES-3

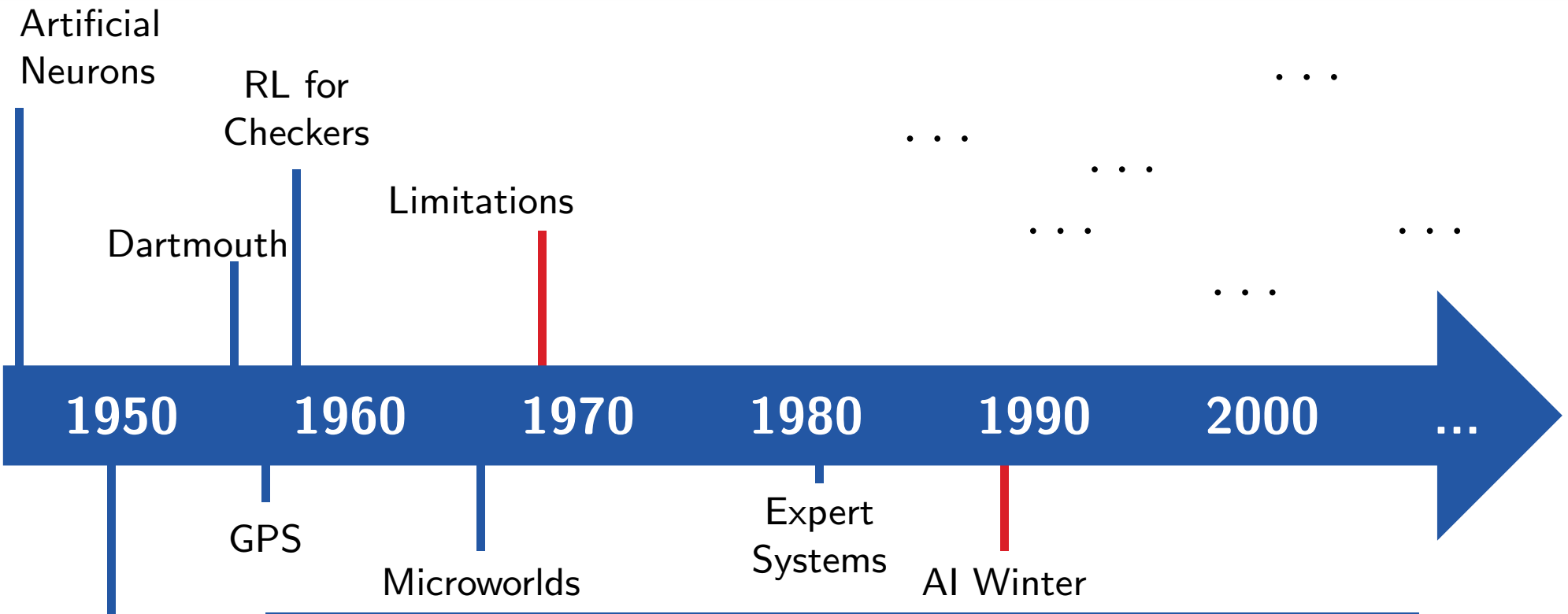
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned at least one disk drive and that should support additional disk drives
& the type of cable needed to connect the disk drive to the previous device on the disk drive is known

THEN: assign the disk drive to the massbus

end of 1980s: AI Winter

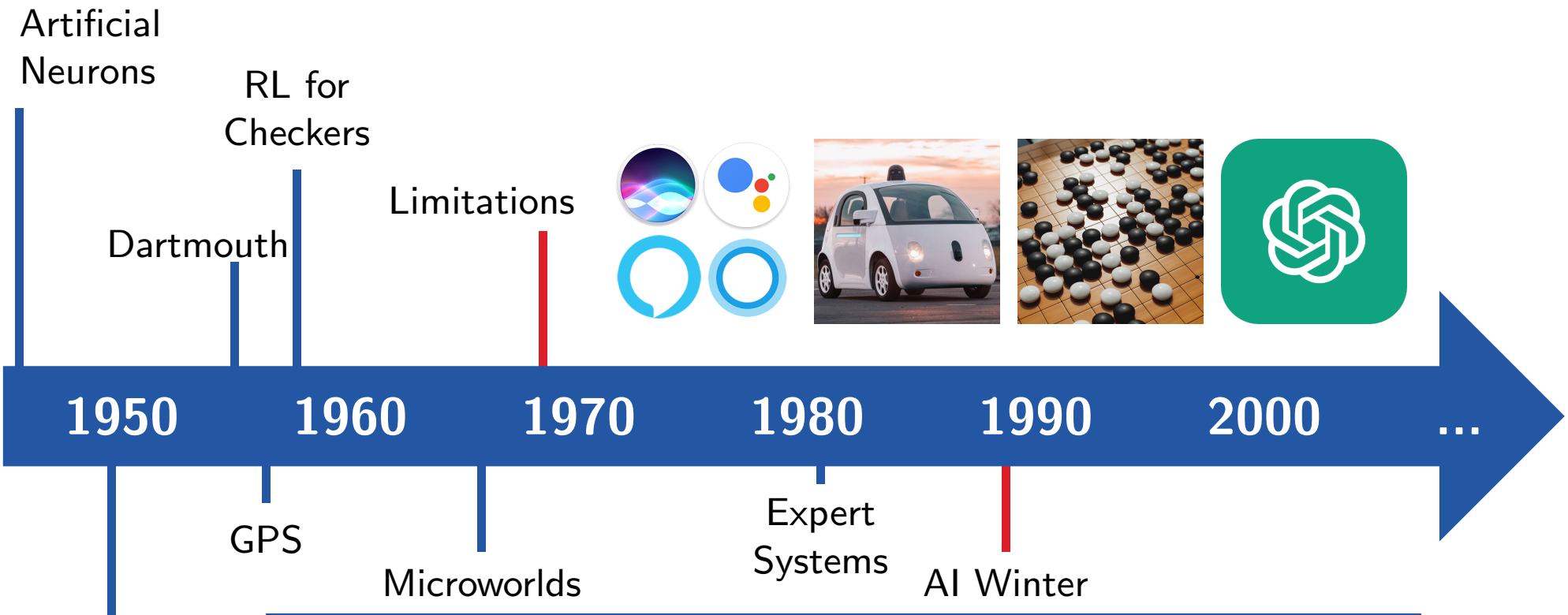
- companies failed to deliver promises
- expert systems difficult to maintain
- expert systems susceptible to uncertainty

Coming of Age (1990s and 2000s)



- advent of probabilistic methods
- formalization of AI techniques
- better understanding of theoretical complexity
- increased use of mathematical methods
- exploitation of large data sets (big data)

Broad Visibility in Society (Since 2010s)



well known systems and famous breakthroughs, e.g.,

- broadly used systems (e.g., virtual assistants)
- AI systems act in real-world (e.g., self-driving cars)
- systems outperform humans in hard tasks (e.g., AlphaGo)
- AI and human-written text hard to distinguish (ChatGPT)

Where are We Today?

AI Approaching Maturity

Russell & Norvig (1995)

Gentle revolutions have occurred in robotics, computer vision, machine learning, and knowledge representation.
A better understanding of the problems and their complexity properties, combined with increased mathematical sophistication, has led to workable research agendas and robust methods.

Where are We Today?

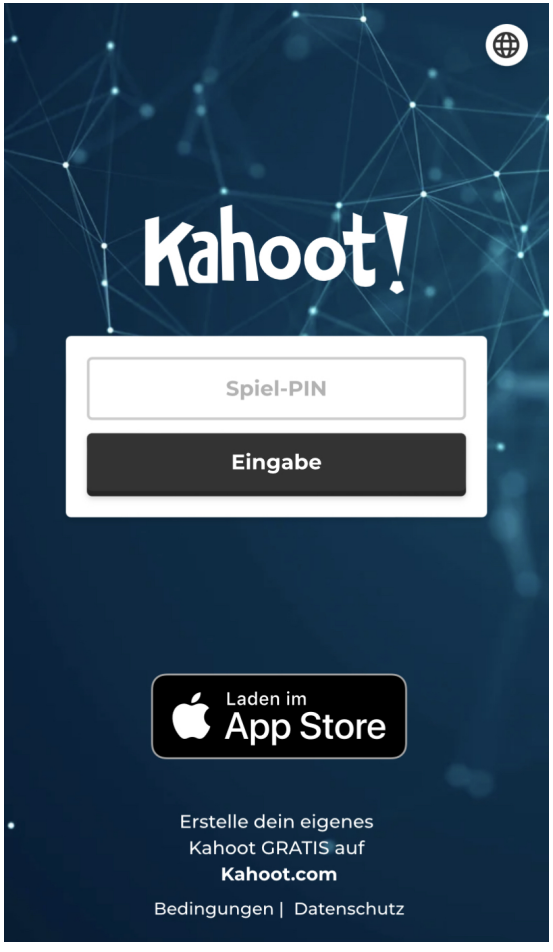
- many coexisting paradigms
  - reactive vs. deliberative
  - data-driven vs. model-driven
  - often hybrid approaches
- many methods, often borrowing from other research areas
  - logic, decision theory, statistics, ...
- different approaches
  - theoretical
  - algorithmic/experimental
  - application-oriented

Focus on Algorithms and Experiments

Many AI problems are inherently difficult (NP-hard), but strong search techniques and heuristics often solve large problem instances regardless:

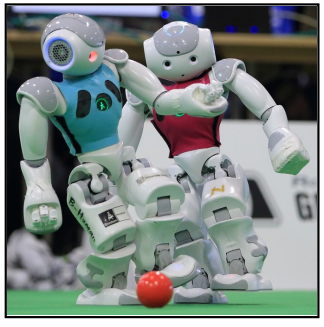
- satisfiability in propositional logic
  - 10,000 propositional variables or more via conflict-directed clause learning
- constraint solvers
  - good scalability via constraint propagation and automatic exploitation of problem structure
- action planning
  - 10^{100} search states and more by search using automatically inferred heuristics

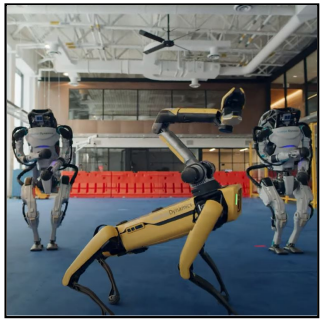
What Can AI Do Today?

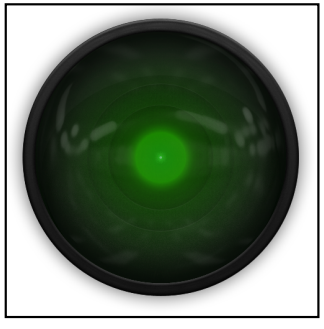


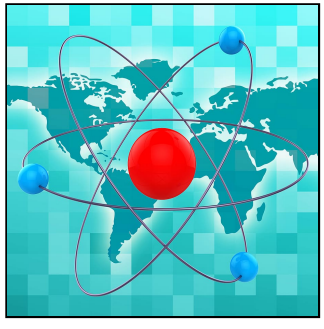
https://kahoot.it/

What Can AI Do Today? – Videos, Articles and AIs









What Can AI Do Today?

results of our classroom poll:

- ✓ successfully complete an off-road car race
- ✗ beat a world champion table tennis player
- ✓ play guitar in a robot band
- ✓ do and fold the laundry
- ✓ drive safely in downtown Basel
- ✗ win a football match against a human team
- ✓ dance synchronously in a group of robots
- ✓ write code on the level of a CS student
- ✓ beat a world champion Chess, Go or Poker player
- ✓ create inspiring quotes
- ✓ compose music
- ✓ engage in a scientific conversation

Summary

Summary

- 1950s/1960s: beginnings of AI; early enthusiasm
- 1970s: micro worlds and knowledge-based systems
- 1980s: gold rush of expert systems followed by "AI winter"
- 1990s/2000s: AI comes of age; research becomes more rigorous and mathematical; mature methods
- 2010s: AI systems enter mainstream

Foundations of Artificial Intelligence

A4. Introduction: Rational Agents

Malte Helmert

University of Basel

February 19, 2025

Introduction: Overview

Chapter overview: introduction

- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

Systematic AI Framework

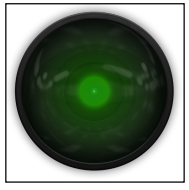
Systematic AI Framework

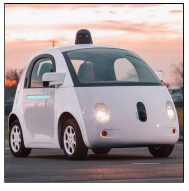
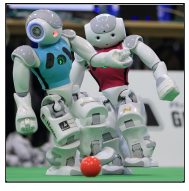
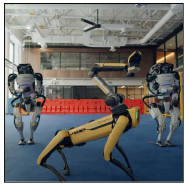
so far we have seen that:

- AI systems applied to wide variety of challenges





Systematic AI Framework

so far we have seen that:

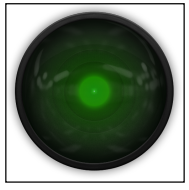
- AI systems act rationally

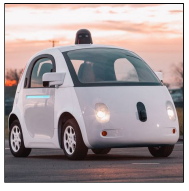
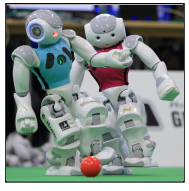
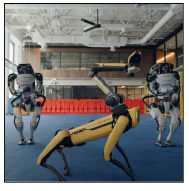
thinking like humans	thinking rationally
acting like humans	**acting rationally**

- AI systems applied to wide variety of challenges





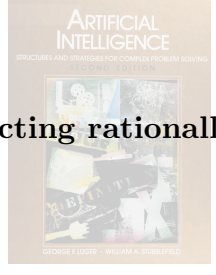



Systematic AI Framework

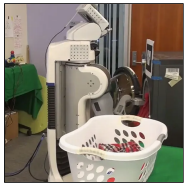
so far we have seen that:

- AI systems act rationally

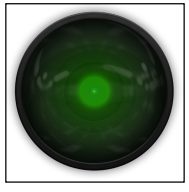
 thinking like humans	 thinking rationally
 acting like humans	 **acting rationally**

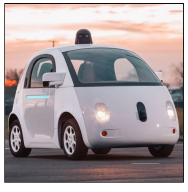
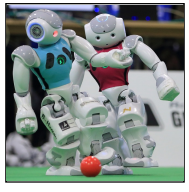
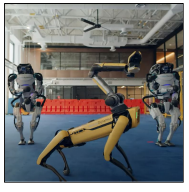
- AI systems applied to wide variety of challenges







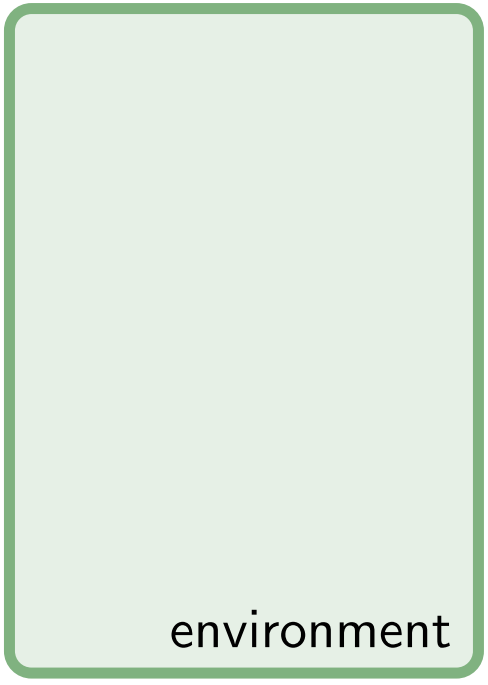





now: describe a systematic framework that

Systematic AI Framework

so far we have seen that:

- AI systems act rationally
- AI systems applied to wide variety of challenges



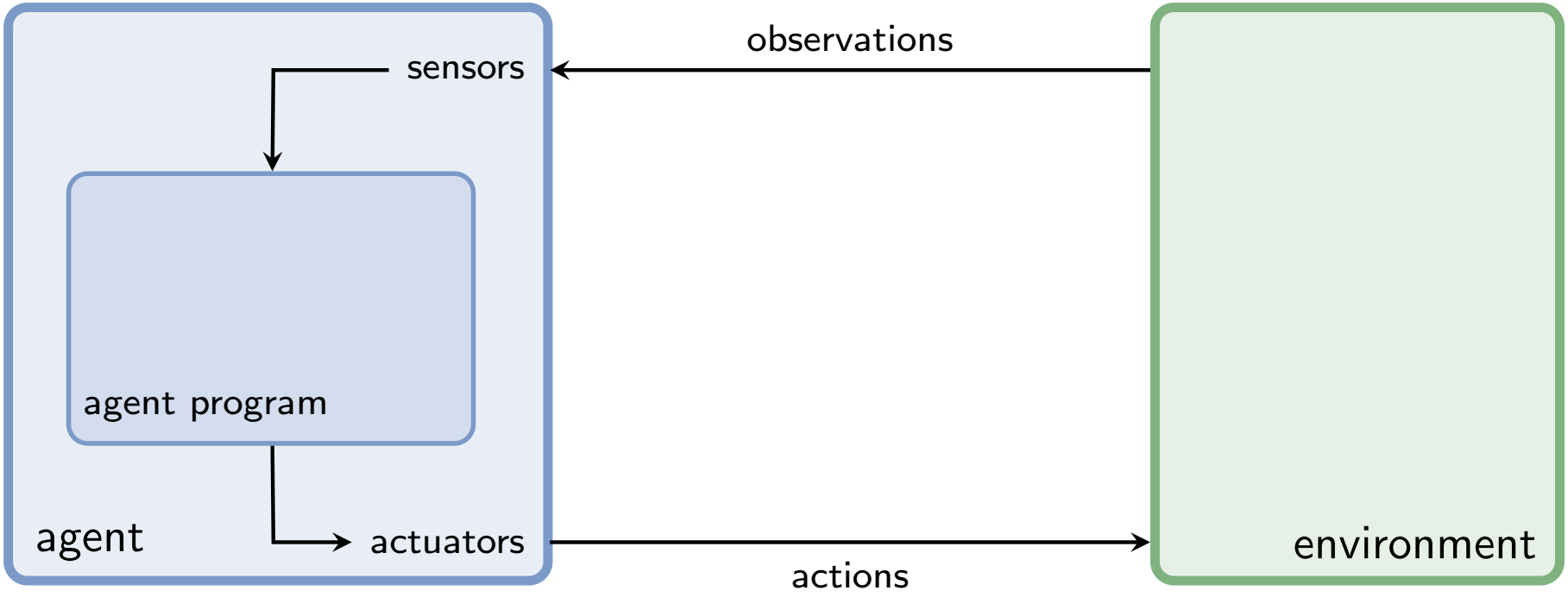
now: describe a systematic framework that

- captures this diversity of challenges

Systematic AI Framework

so far we have seen that:

- AI systems act rationally
- AI systems applied to wide variety of challenges



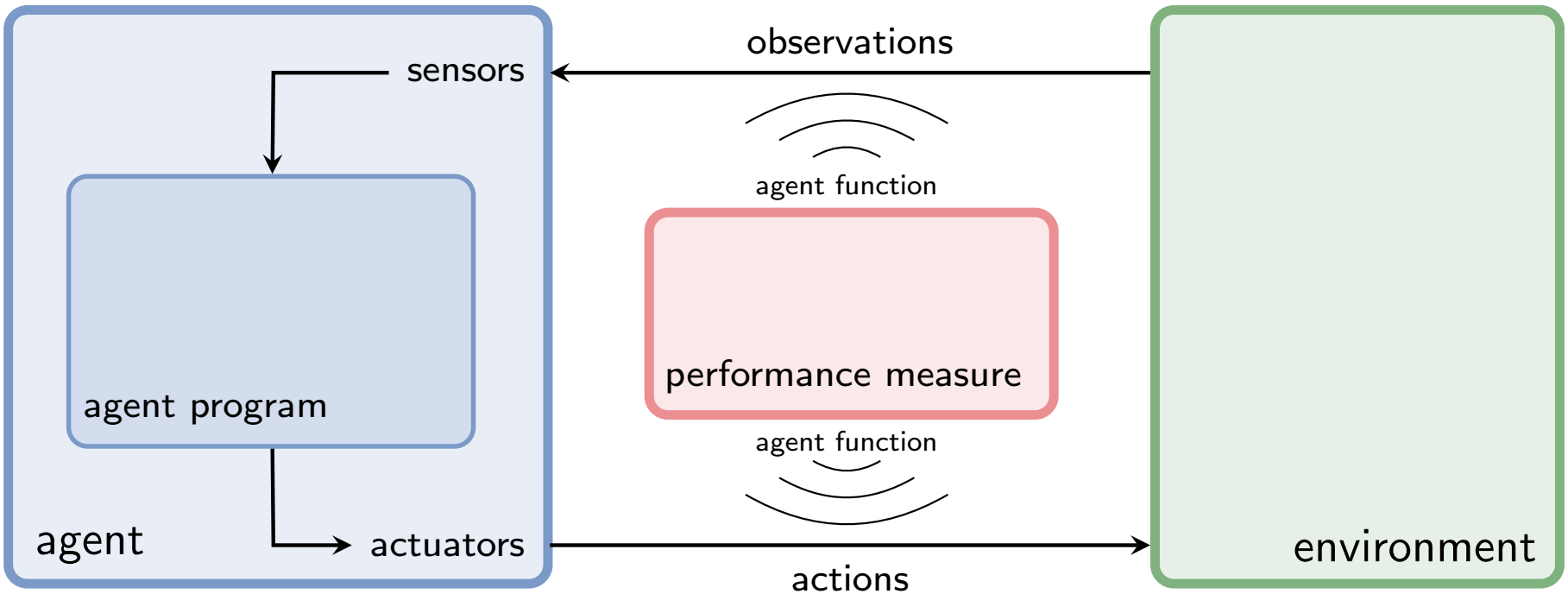
now: describe a systematic framework that

- captures this diversity of challenges
- includes an entity that acts in the environment

Systematic AI Framework

so far we have seen that:

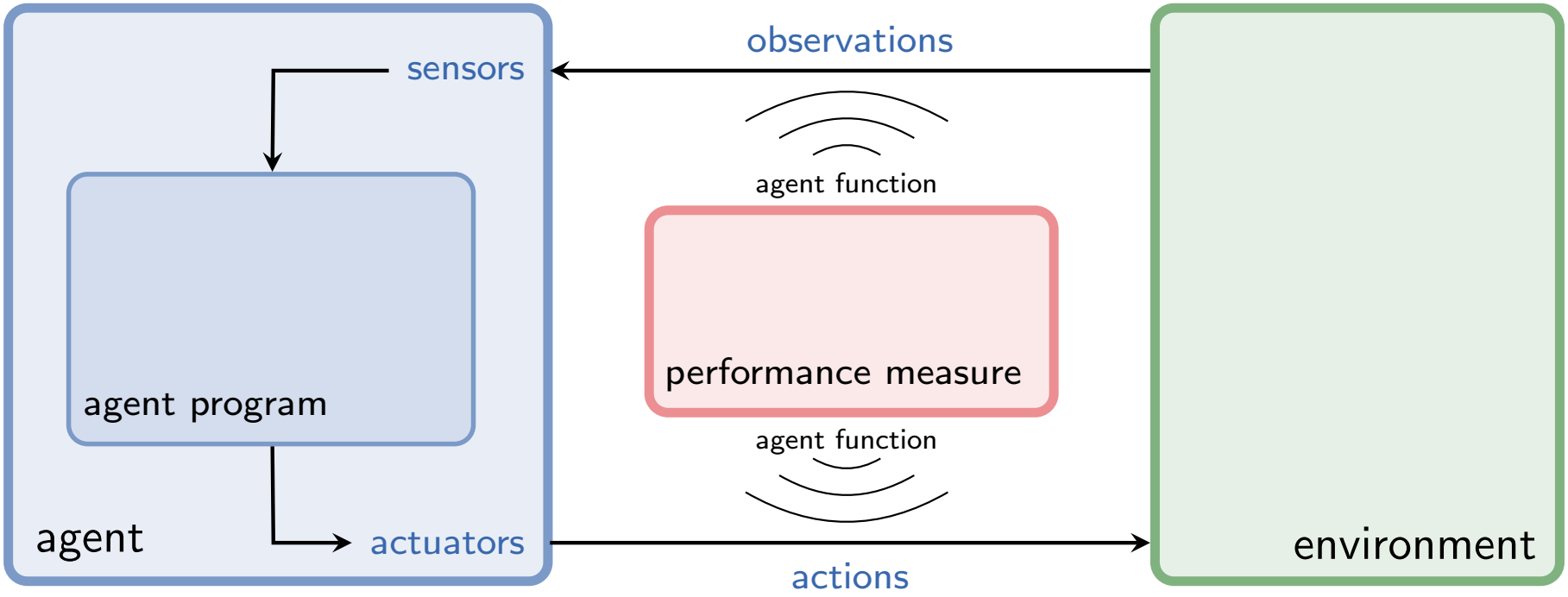
- AI systems act rationally
- AI systems applied to wide variety of challenges



now: describe a systematic framework that

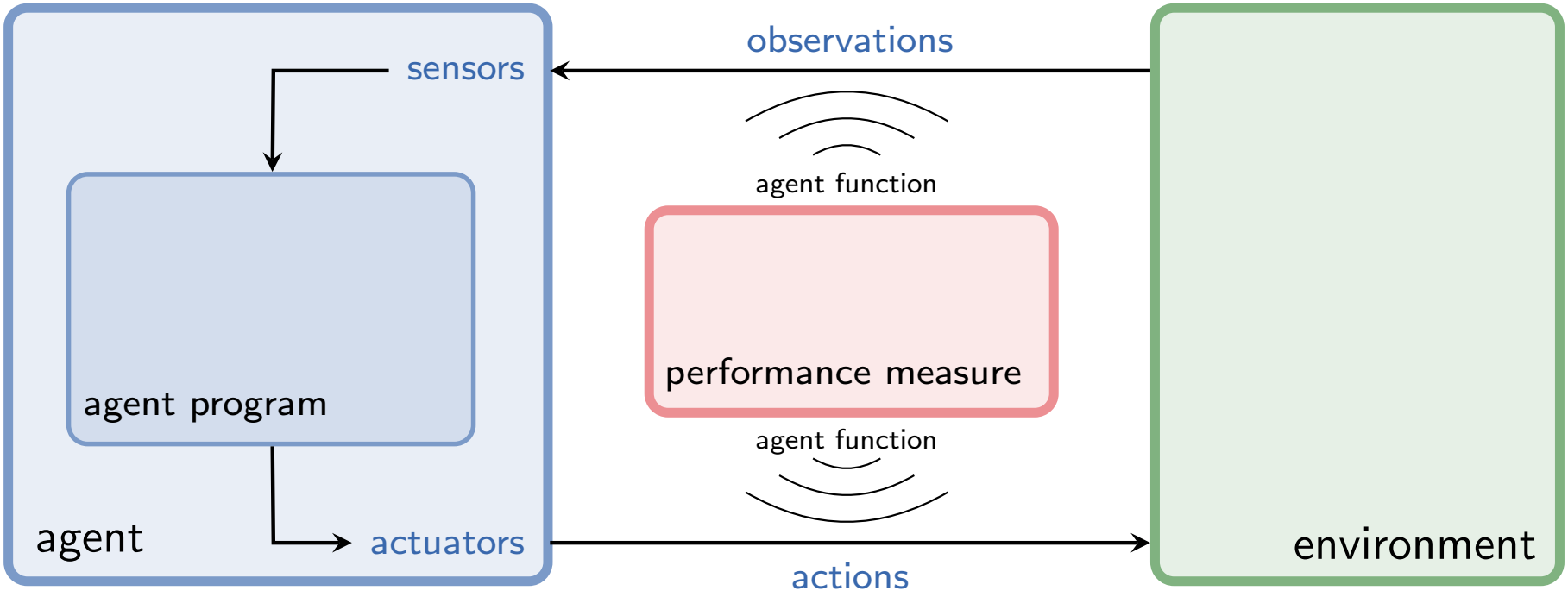
- captures this diversity of challenges
- includes an entity that acts in the environment
- determines if the agent acts rationally in the environment

Agent-Environment Interaction



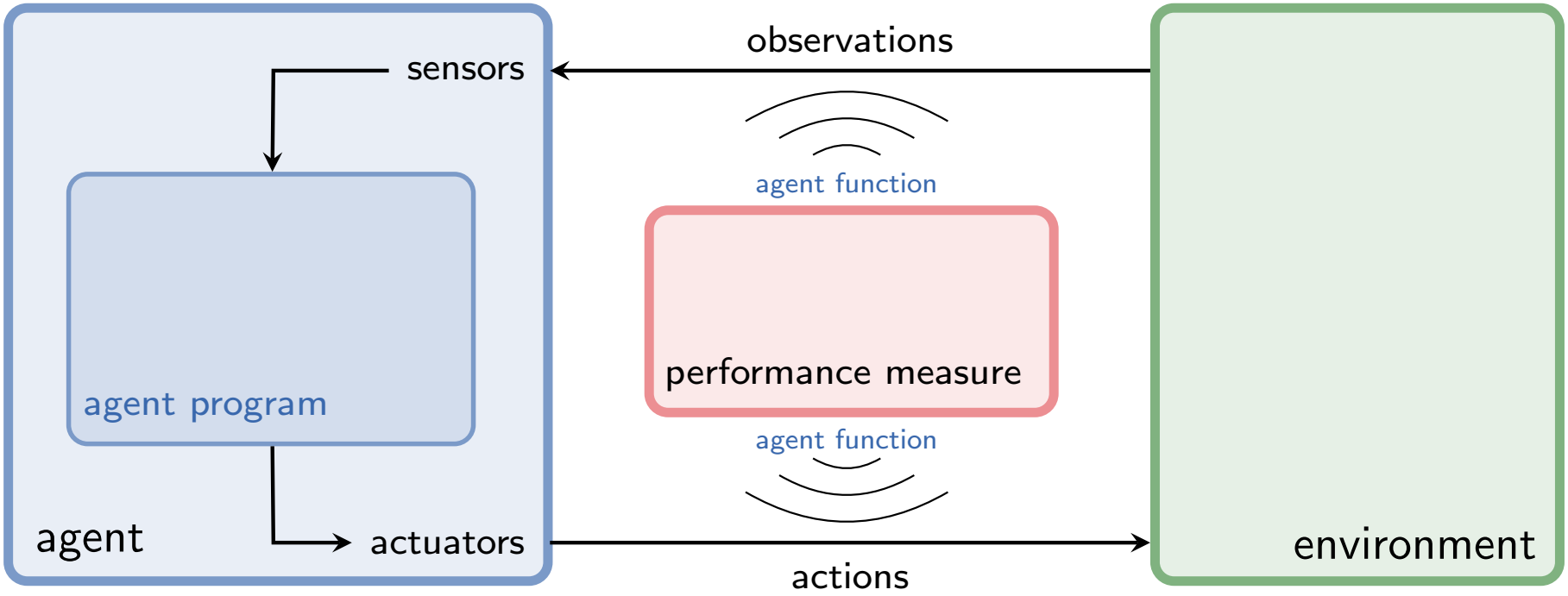
- sensors: physical entities that allow the agent to observe
- observation: data perceived by the agent's sensors
- actuators: physical entities that allow the agent to act
- action: abstract concept that affects the state of the environment

Agent-Environment Interaction



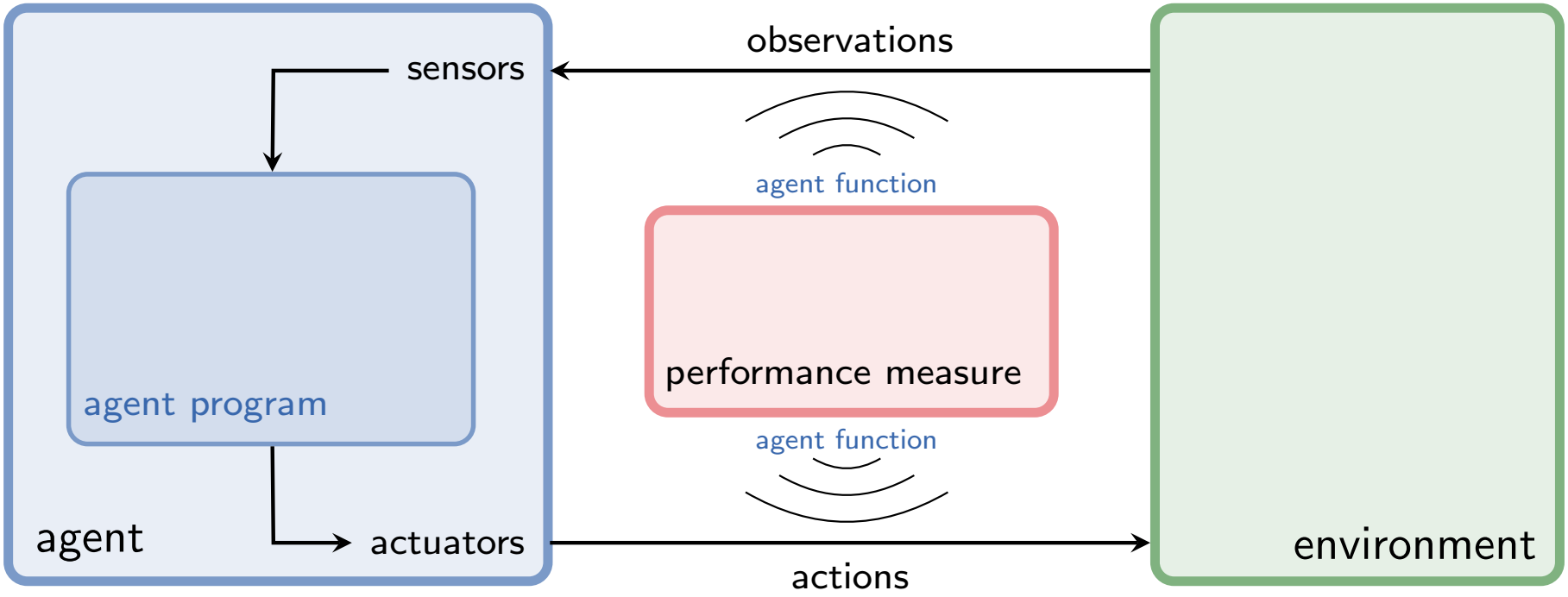
- sensors and actuators are not relevant for the course
  ($\rightsquigarrow$ typically covered in courses on robotics)
- observations and actions describe the agent's capabilities
  (the agent model)

Formalizing an Agent's Behavior



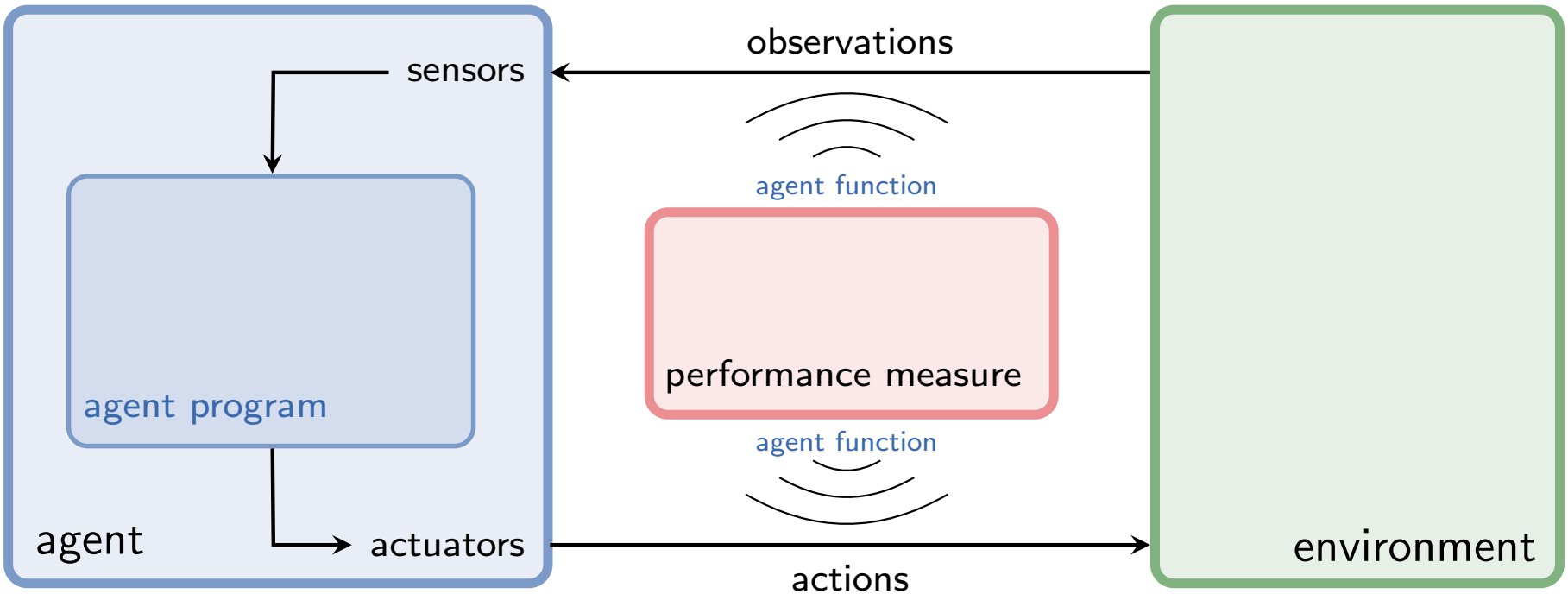
1. as agent program:
   - internal representation
   - specifics possibly unknown to outside
2. as agent function:
   - external characterization

Formalizing an Agent's Behavior



1. as agent program:
   - internal representation
   - specifics possibly unknown to outside
   - takes observation as input
   - outputs an action
2. as agent function:
   - external characterization
   - maps sequence of observations to (probability distribution over) actions

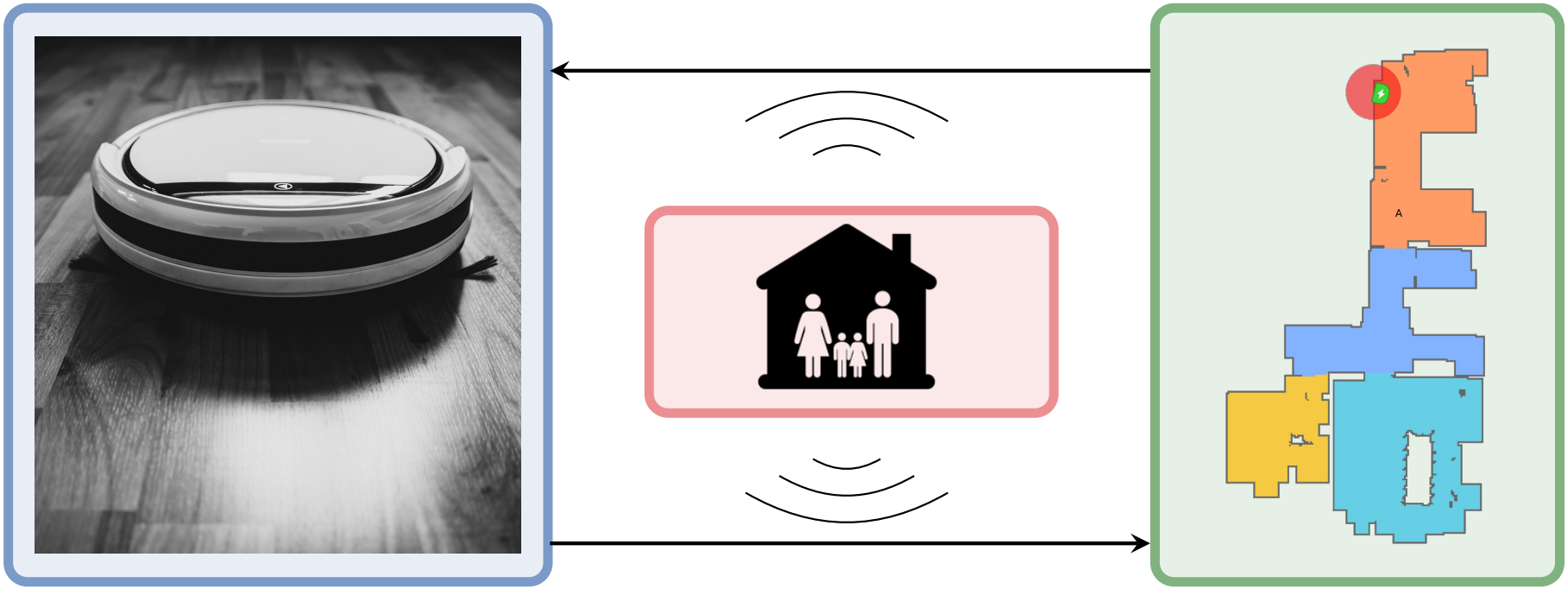
Formalizing an Agent's Behavior



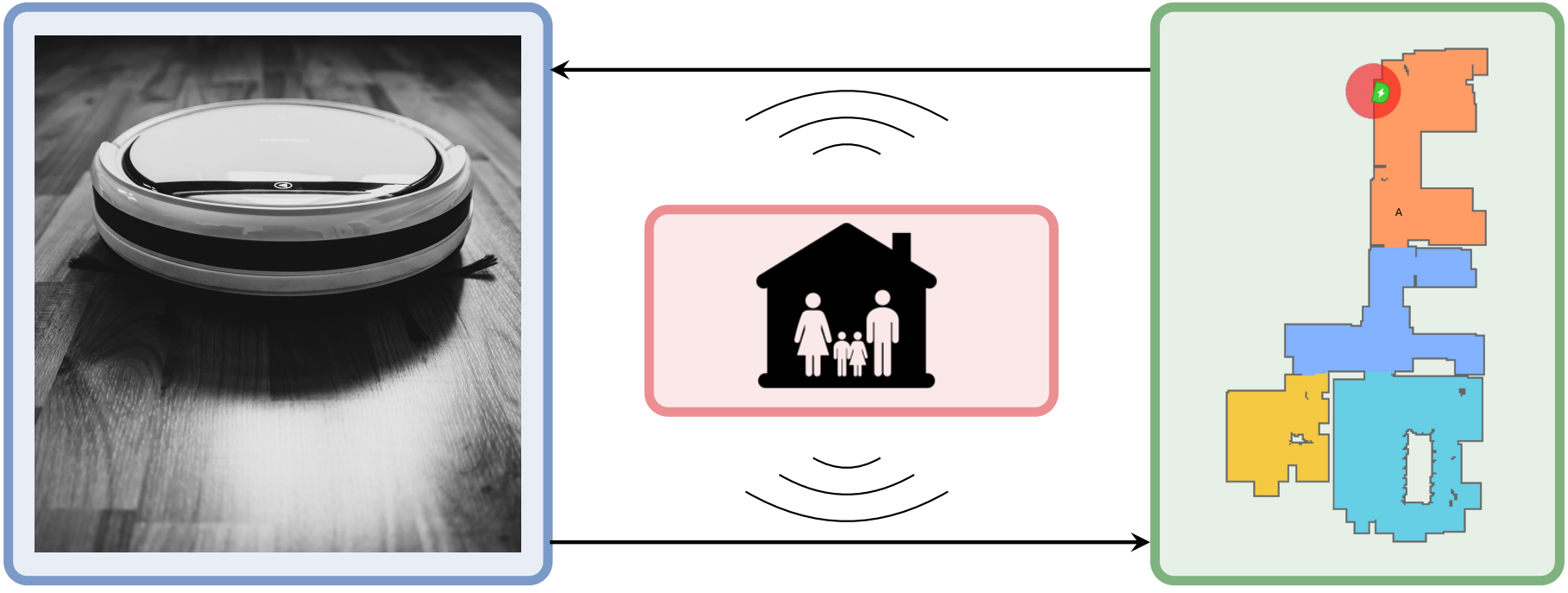
1. as agent program:
   - internal representation
   - specifics possibly unknown to outside
   - takes observation as input
   - outputs an action
   - computed on physical machine (the agent architecture)
2. as agent function:
   - external characterization
   - maps sequence of observations to (probability distribution over) actions
   - abstract mathematical formalization

Example

Vacuum Domain

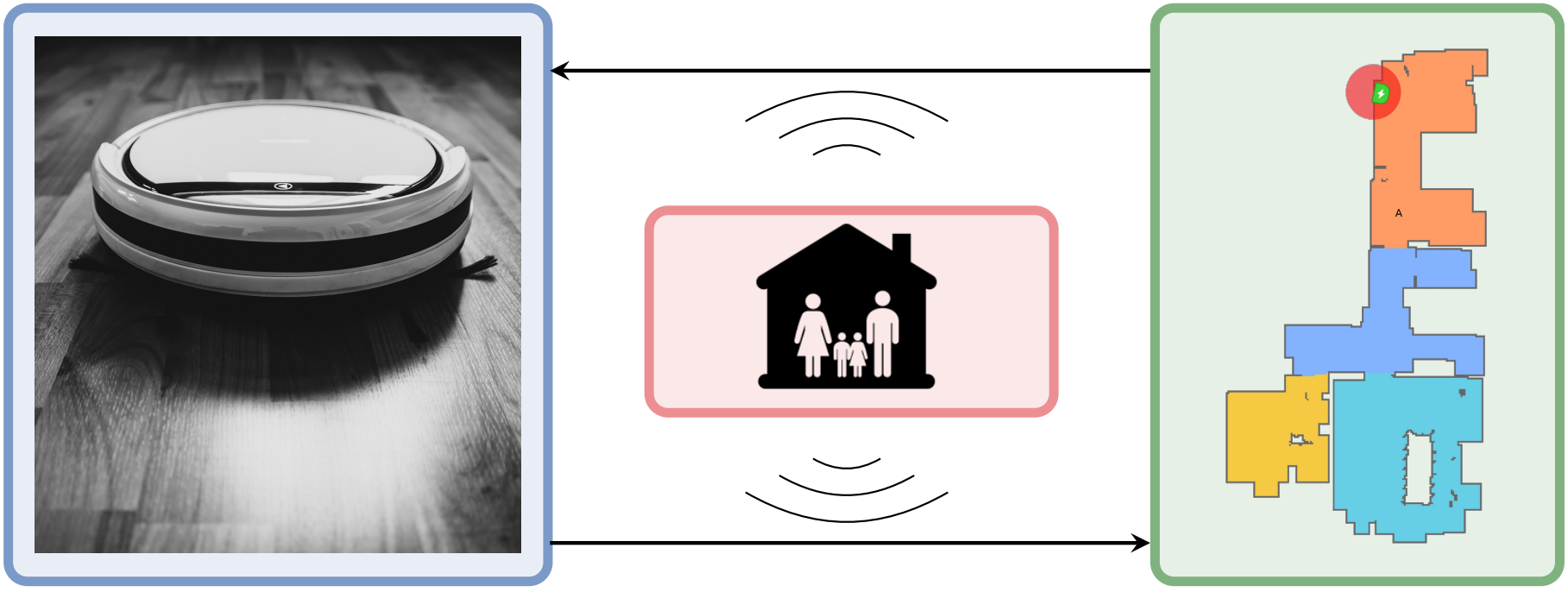


Vacuum Agent: Sensors and Actuators



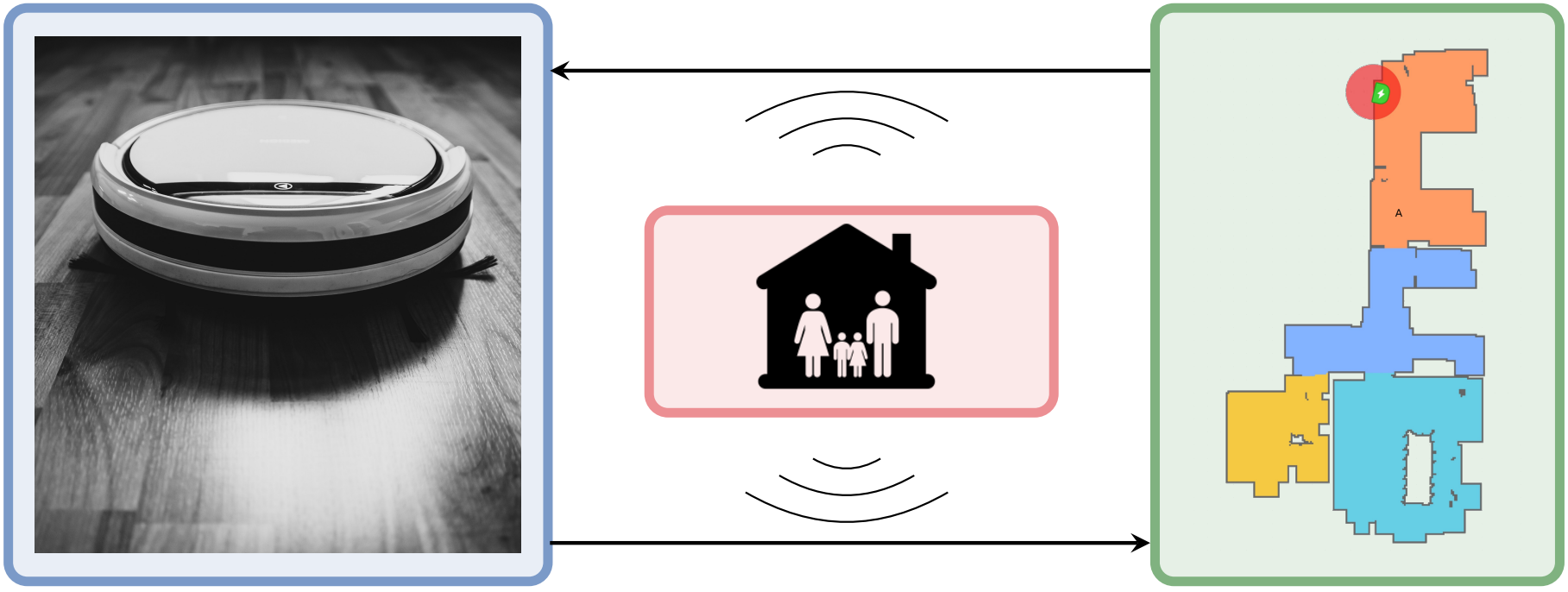
- sensors: cliff sensors, bump sensors, wall sensors, state of charge sensor, WiFi module
- actuators: wheels, cleaning system

Vacuum Agent: Observations and Actions



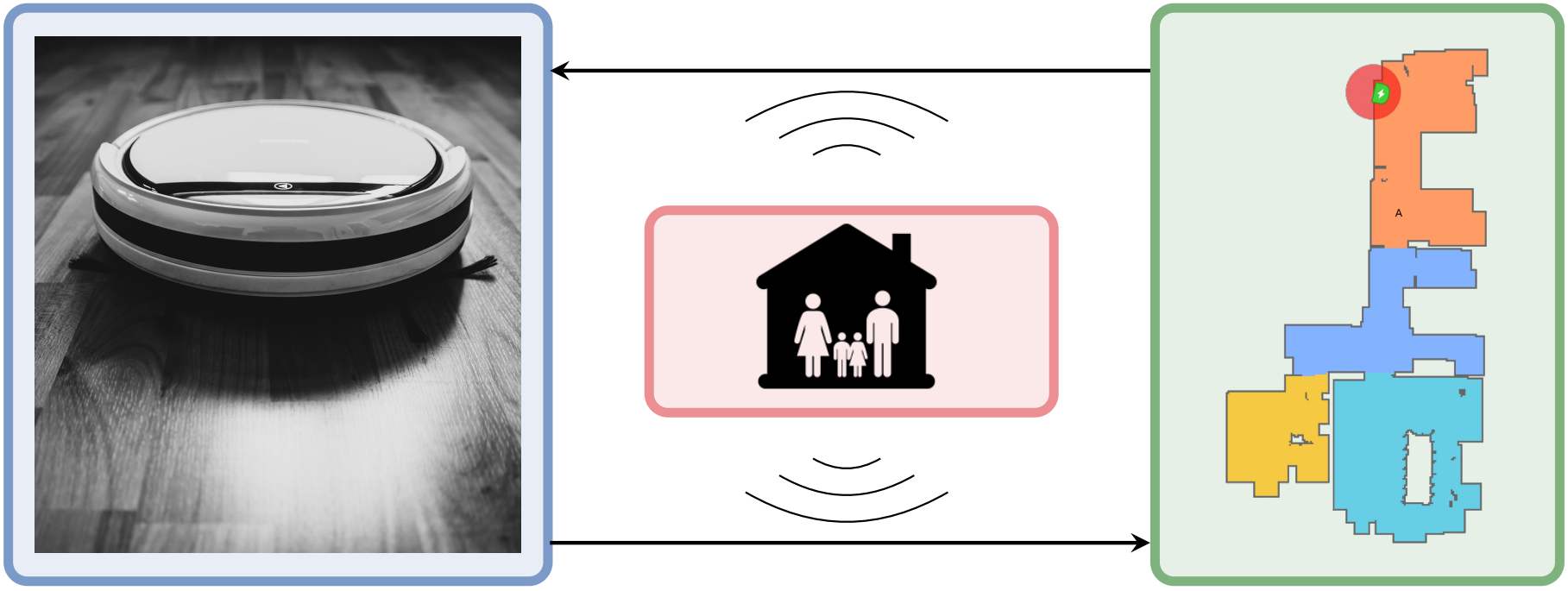
- observations: current location, dirt level of current room, presence of humans, battery charge
- actions: move-to-next-room, move-to-base, vacuum, wait

Vacuum Agent: Agent Program



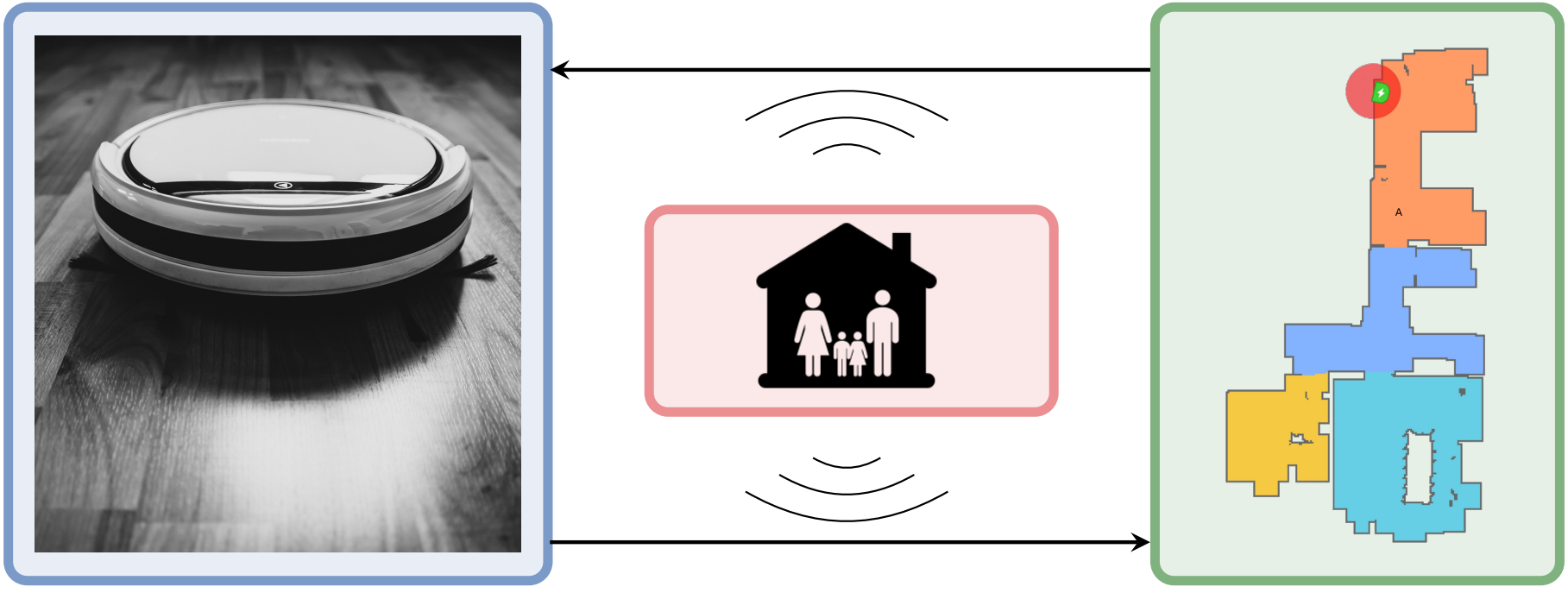
```
1 def vacuum-agent([location, dirt-level, owner-present, battery]):
2       if battery ≤ 10%: return move-to-base
3       else if owner-present = True: return move-to-next-room
4       else if dirt-level = dirty: return vacuum
5       else: return move-to-next-room
```

Vacuum Domain: Agent Function



observation sequence	action
⟨[blue, clean, False, 100%]⟩	*move-to-next-room*
⟨[blue, dirty, False, 100%]⟩	*vacuum*
⟨[blue, clean, True, 100%]⟩	*move-to-next-room*
...	...
⟨[blue, clean, False, 100%], [blue, clean, False, 90%]⟩	*move-to-next-room*
⟨[blue, clean, False, 100%], [blue, dirty, False, 90%]⟩	*vacuum*
...	...

Vacuum Domain: Performance Measure



potential influences on performance measure:

- dirt levels
- noise levels
- energy consumption
- safety

Rationality

Evaluating Agent Functions

What is the right agent function?

Rationality

rationality of an agent depends on performance measure (often: utility, reward, cost) and environment

Perfect Rationality

- for each possible observation sequence
- select an action which maximizes
- expected value of future performance
- given available information on observation history
- and environment

Perfect Rationality of Our Vacuum Agent

Is our vacuum agent perfectly rational?

Perfect Rationality of Our Vacuum Agent

Is our vacuum agent perfectly rational?

depends on performance measure and environment, e.g.:

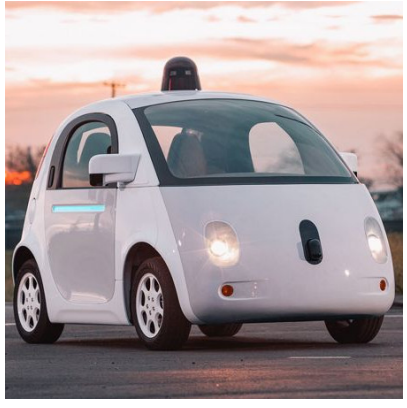
- Do actions reliably have the desired effect?
- Do we know the initial situation?
- Can new dirt be produced while the agent is acting?

Performance Measure

- specified by designer
- sometimes clear,
  sometimes not so clear
- significant impact on
  - desired behavior
  - difficulty of problem

Performance Measure

- specified by designer
- sometimes clear,
  sometimes not so clear
- significant impact on
  - desired behavior
  - difficulty of problem



Performance Measure

- specified by designer
- sometimes clear,
  sometimes not so clear
- significant impact on
  - desired behavior
  - difficulty of problem

Perfect Rationality of Our Vacuum Agent

consider performance measure:

- $+1$ utility for cleaning a dirty room

consider environment:

- actions and observations reliable
- world only changes through actions of the agent

our vacuum agent is perfectly rational

Perfect Rationality of Our Vacuum Agent

consider performance measure:

- -1 utility for each dirty room in each step

consider environment:

- actions and observations reliable
- world only changes through actions of the agent

our vacuum agent is not perfectly rational

Perfect Rationality of Our Vacuum Agent

consider performance measure:

- -1 utility for each dirty room in each step

consider environment:

- actions and observations reliable
- yellow room may spontaneously become dirty

our vacuum agent is not perfectly rational

Rationality: Discussion

- perfect rationality $\neq$ omniscience
  - incomplete information (due to limited observations) reduces achievable utility
- perfect rationality $\neq$ perfect prediction of future
  - uncertain behavior of environment (e.g., stochastic action effects) reduces achievable utility
- perfect rationality is rarely achievable
  - limited computational power $\rightsquigarrow$ bounded rationality

Summary

Summary (1)

common metaphor for AI systems: rational agents

agent interacts with environment:

- sensors perceive observations about state of the environment
- actuators perform actions modifying the environment
- formally: agent function maps observation sequences to actions

Summary (2)

rational agents:

- try to maximize performance measure (utility)
- perfect rationality: achieve maximal utility in expectation given available information
- for "interesting" problems rarely achievable
  $\rightsquigarrow$ bounded rationality

Foundations of Artificial Intelligence

A5. Introduction: Environments and Problem Solving Methods

Malte Helmert

University of Basel

February 24, 2025

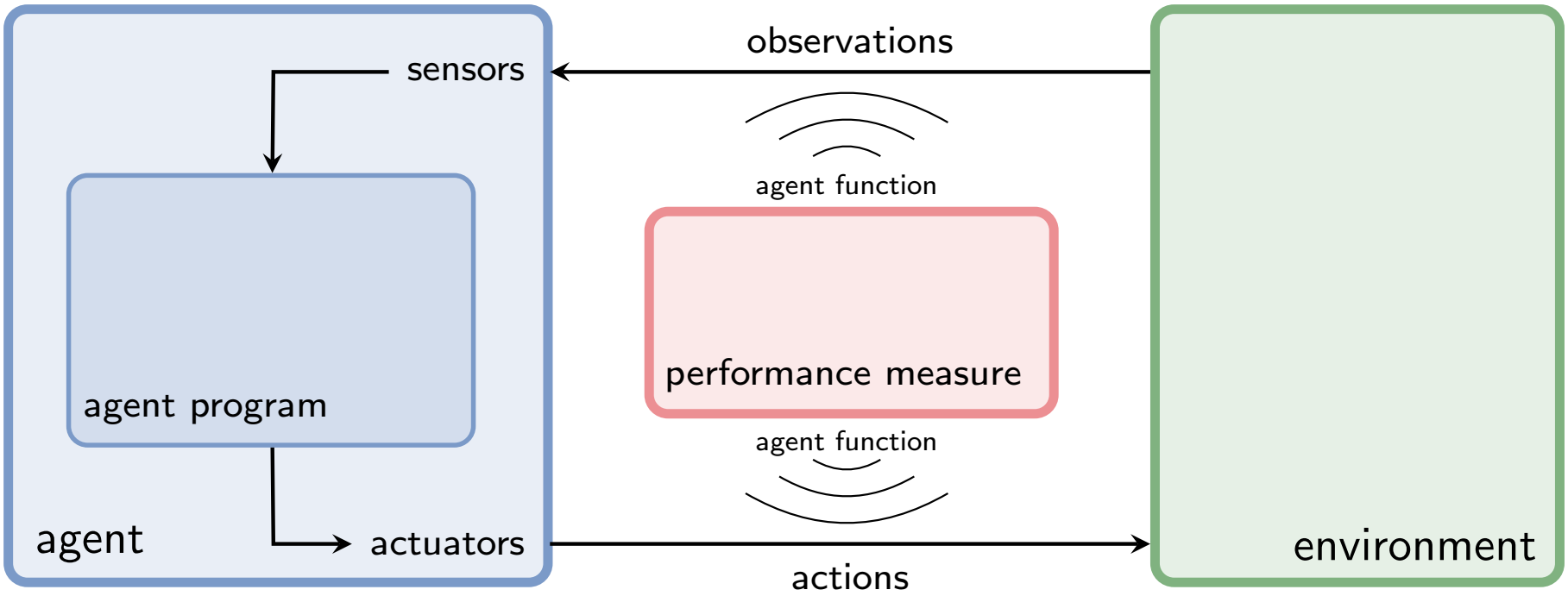
Introduction: Overview

Chapter overview: introduction

- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

Environments of Rational Agents

Environments of Rational Agents



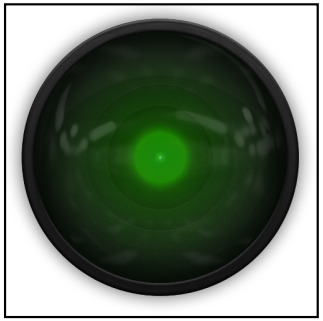
- Which environment aspects are relevant for the agent?
- How do the agent's actions change the environment?
- What does the agent observe?

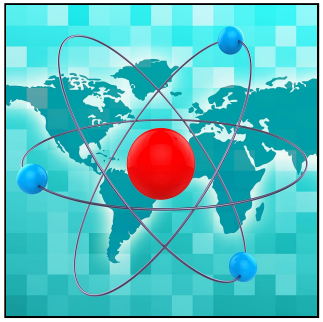
Properties of Environments

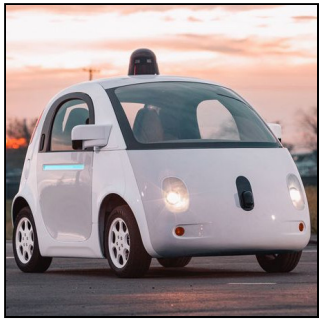
Environment properties determine character of AI problem.

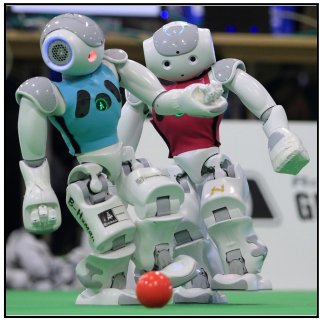
- fully observable vs. partially observable
- single-agent vs. multi-agent
- deterministic vs. nondeterministic vs. stochastic
- static vs. dynamic
- discrete vs. continuous

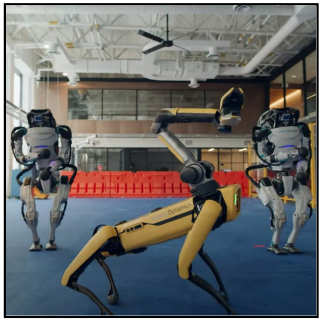
Properties of Environments











Properties of Environments

fully observable vs. partially observable

Can the agent fully observe the state of the environment at every decision step or not?

special case of partially observable: unobservable

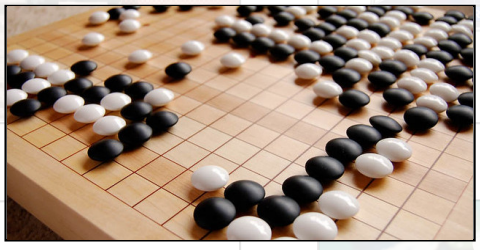
Properties of Environments

single-agent vs. multi-agent

Are other agents relevant for own performance?

subcases of multi-agent: are the other agents adversarial, cooperative, or selfish?

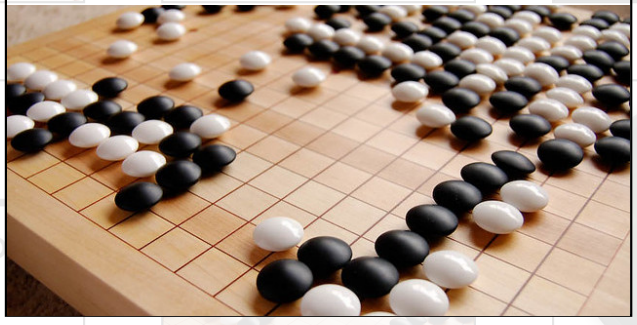
Properties of Environments



deterministic vs. nondeterministic vs. stochastic

Is the next state of the environment fully determined by the current state and the next action? Are probabilities involved?

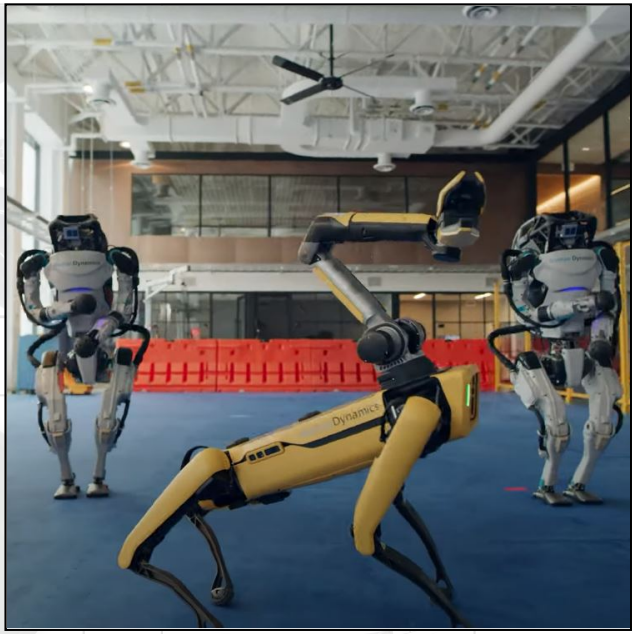
Properties of Environments



static vs. dynamic

Does the state of the environment remain the same while the agent is contemplating its next action?

Properties of Environments



discrete vs. continuous

Is the state of the environment (and actions, observations, time) given by discrete or by continuous quantities?

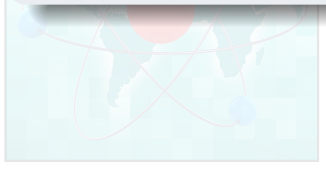
Properties of Environments

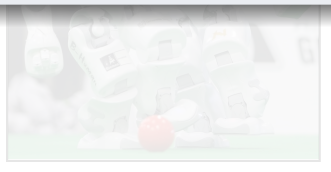
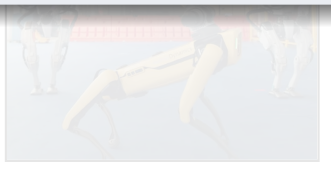
suitable problem-solving algorithms

Environments of different kinds (according to these criteria) usually require different algorithms.

real world

The "real world" combines all unpleasant (in the sense of: difficult to handle) properties.



Problem Solving Methods

Three Approaches to Solving AI Problems

We can solve a concrete AI problem (e.g., backgammon) in several ways:

Problem Solving Methods

1. problem-specific: implement algorithm tailored to problem

problem-specific algorithms:

- designed to solve a specific problem
- allow exploiting problem-specific knowledge
- solve just one (type of) problem

Three Approaches to Solving AI Problems

We can solve a concrete AI problem (e.g., backgammon) in several ways:

Problem Solving Methods

1. problem-specific: implement algorithm tailored to problem
2. general: create problem description as input for general solver

general problem solvers:

- user creates model of problem instance in formalism ("language")
- solver takes modeled instance as input
- solver implements general algorithm to compute solution

Three Approaches to Solving AI Problems

We can solve a concrete AI problem (e.g., backgammon) in several ways:

Problem Solving Methods

1. problem-specific: implement algorithm tailored to problem
2. general: create problem description as input for general solver
3. learning: learn (aspects of) algorithm from data

learners:

- general approach that learns to solve specific problem
- adapts via experience instead of via reasoning
- requires data and feedback instead of model of the AI problems

Three Approaches to Solving AI Problems

We can solve a concrete AI problem (e.g., backgammon) in several ways:

Problem Solving Methods

1. problem-specific: implement algorithm tailored to problem
2. general: create problem description as input for general solver
3. learning: learn (aspects of) algorithm from data

- all three approaches have strengths and weaknesses
- combinations are possible (and common in practice)
- we will mostly focus on general algorithms, but also consider other approaches

Classification of AI Topics

Classification of AI Topics

Many areas of AI are essentially characterized by

- the properties of environments they consider and
- which of the three problem solving approaches they use.

We conclude the introduction by giving some examples

- within this course and
- beyond the course ("advanced topics").

Examples: Classification of AI Topics

Course Topic: Informed Search Algorithms

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Course Topic: Constraint Satisfaction Problems

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Course Topic: Board Games

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent (adversarial)

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Advanced Topic: General Game Playing

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. (stochastic)
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent (adversarial)

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Course Topic: Classical Planning

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Course Topic: Acting under Uncertainty

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Examples: Classification of AI Topics

Advanced Topic: Reinforcement Learning

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Summary

Summary (1)

AI problem: performance measure + agent model + environment

Properties of environment critical for choice of suitable algorithm:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

Summary (2)

Three problem solving methods:

- problem-specific
- general
- learning

general problem solvers:

- models characterize problem instances mathematically
- formalisms/languages describe models compactly
- algorithms use languages as problem description and to exploit problem structure

Foundations of Artificial Intelligence

B1. State-Space Search: State Spaces

Malte Helmert

University of Basel

February 24, 2025

State-Space Search: Overview

Chapter overview: state-space search

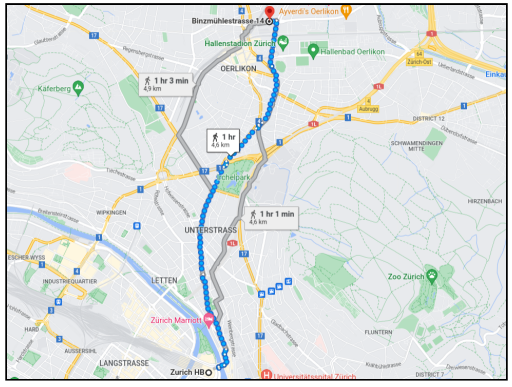
- B1–B3. Foundations
  - B1. State Spaces
  - B2. Representation of State Spaces
  - B3. Examples of State Spaces
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms

State-Space Search Problems

State-Space Search Applications

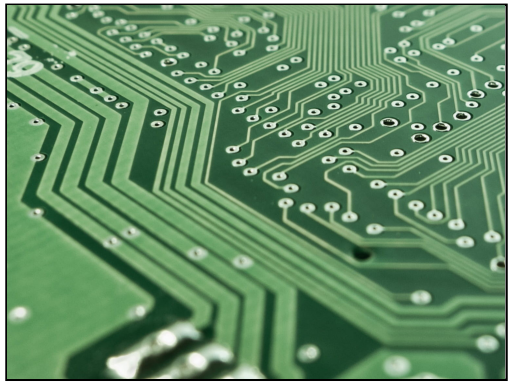
Mario AI competition

route planning



multi-agent path finding

scheduling



software/hardware verification

NPC behaviour

Classical Assumptions

"classical" assumptions considered in this part of the course:

- no other agents in the environment (single-agent)
- always knows state of the world (fully observable)
- state only changed by the agent (static)
- finite number of states/actions (in particular discrete)
- actions have deterministic effect on the state

$\rightsquigarrow$ can all be generalized (but not in this part of the course)

Classification

classification:

State-Space Search

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Informal Description

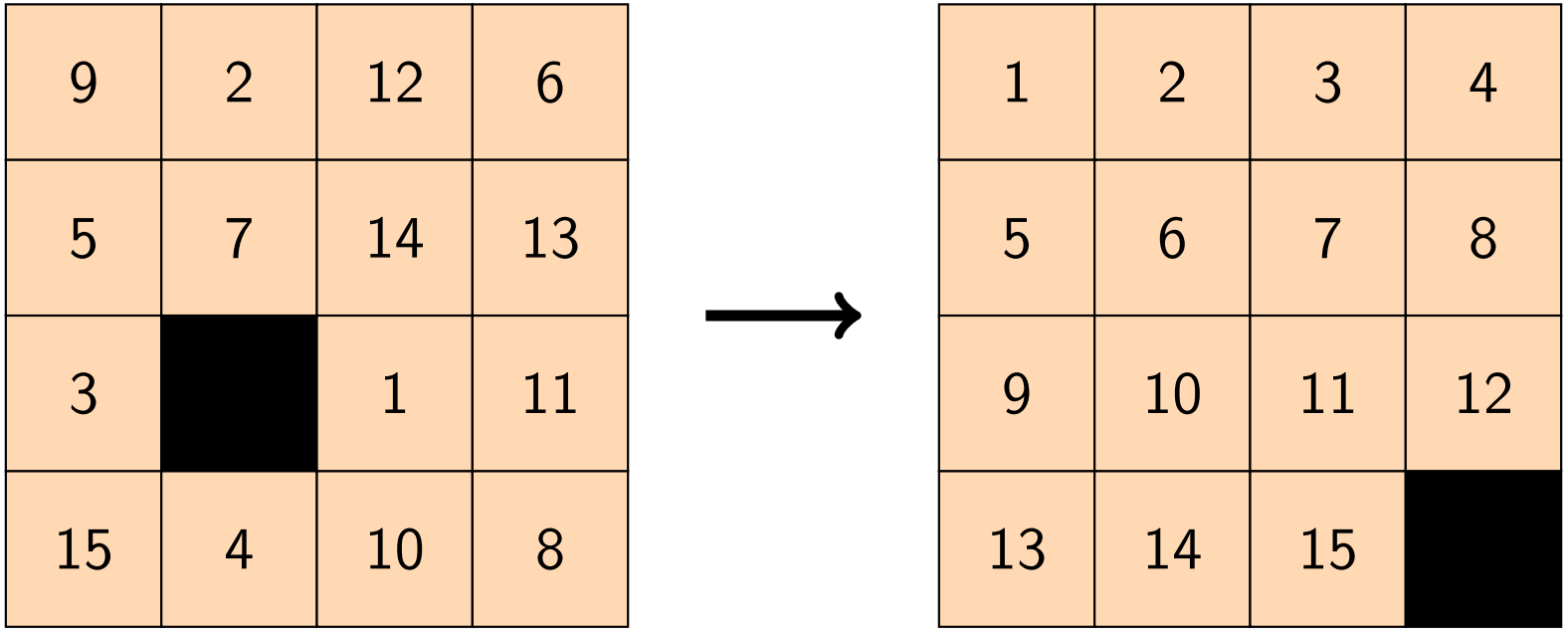
State-space search problems are among the "simplest" and most important classes of AI problems.

objective of the agent:

- apply a sequence of actions
- that reaches a goal state
- from a given initial state

performance measure: minimize total action cost

Motivating Example: 15-Puzzle



Formalization

State Spaces

Definition (state space)

A state space or transition system is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ with

- finite set of states S
- finite set of actions A
- action costs $cost : A \to \mathbb{R}_0^+$
- transition relation $T \subseteq S \times A \times S$ that is deterministic in $\langle s, a \rangle$ (see next slide)
- initial state $s_I \in S$
- set of goal states $S_G \subseteq S$

German: Zustandsraum, Transitionssystem, Zustände, Aktionen, Aktionskosten, Transitions-/Übergangsrelation, deterministisch, Anfangszustand, Zielzustände

State Spaces: Terminology & Notation

Definition (transition, deterministic)

Let $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_\mathsf{I}, S_\mathsf{G} \rangle$ be a state space.

The triples $\langle s, a, s' \rangle \in T$ are called (state) transitions.

We say $\mathcal{S}$ has the transition $\langle s, a, s' \rangle$ if $\langle s, a, s' \rangle \in T$.
We write this as $s \xrightarrow{a} s'$, or $s \to s'$ when a does not matter.

Transitions are deterministic in $\langle s, a \rangle$: it is forbidden to have both $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ with $s_1 \neq s_2$.

State Space: Running Example

Consider the bounded inc-and-square search problem.

informal description:

- find a sequence of
  - increment-mod10 (*inc*) and
  - square-mod10 (*sqr*) actions
- on the natural numbers from 0 to 9
- that reaches the number 6 or 7
- starting from the number 1
- assuming each action costs 1.

State Space: Running Example

Consider the bounded inc-and-square search problem.

informal description:

- find a sequence of
  - increment-mod10 (*inc*) and
  - square-mod10 (*sqr*) actions
- on the natural numbers from 0 to 9
- that reaches the number 6 or 7
- starting from the number 1
- assuming each action costs 1.

formal model:

- $S = \{0, 1, \ldots, 9\}$
- $A = \{inc, sqr\}$
- $cost(inc) = cost(sqr) = 1$
- T s.t. for $i = 0, \ldots, 9$:
  - $\langle i, inc, (i + 1) \bmod 10 \rangle \in T$
  - $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- $s_I = 1$
- $S_G = \{6, 7\}$

Graph Interpretation

state spaces are often depicted as directed, labeled graphs

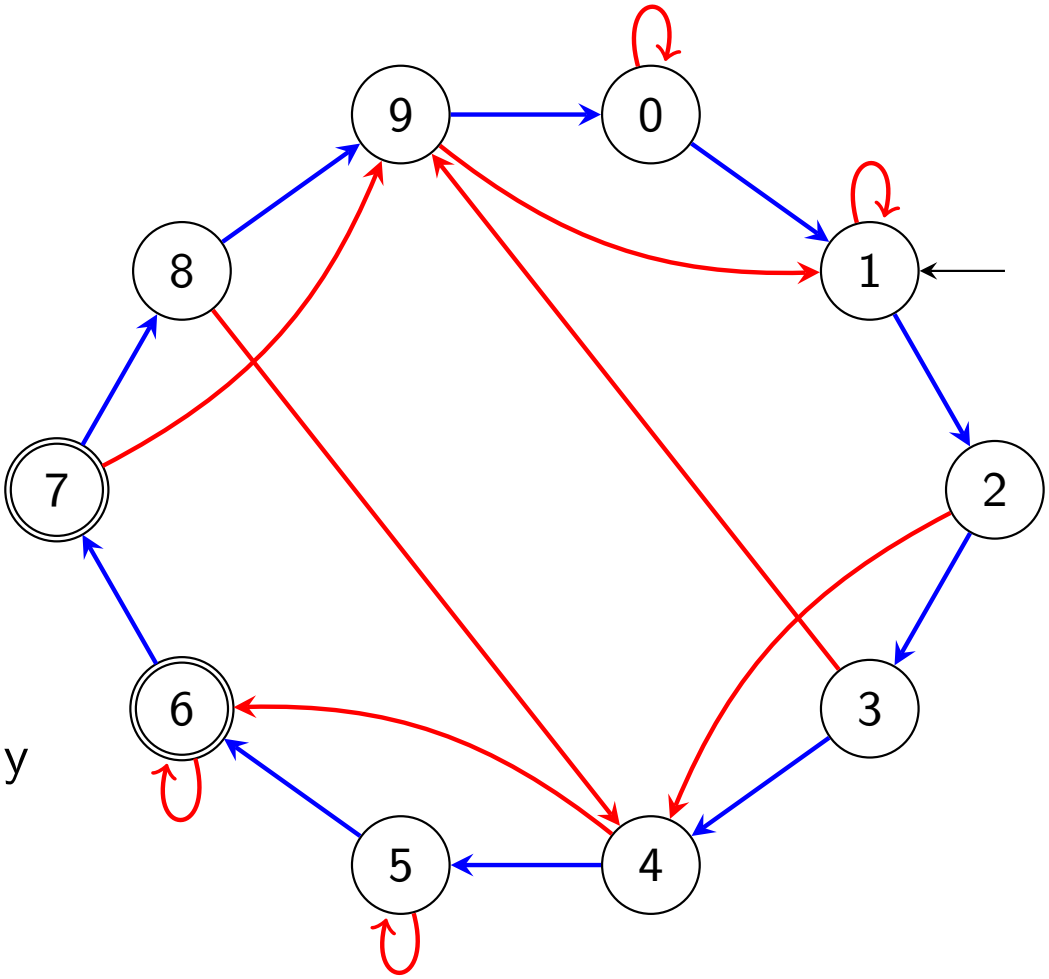
- states: graph vertices
- transitions: labeled arcs

- initial state: incoming arrow
- goal states: double circles
- actions: the arc labels
- action costs: described separately
  (or implicitly $= 1$)

Graph Interpretation

state spaces are often depicted as directed, labeled graphs

- states: graph vertices
- transitions: labeled arcs
  (here: colors instead of labels)
- initial state: incoming arrow
- goal states: double circles
- actions: the arc labels
- action costs: described separately
  (or implicitly $= 1$)



State Spaces: More Terminology (1)

We use common terminology from graph theory.

Definition (predecessor, successor, applicable action)

Let $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ be a state space.

Let $s, s' \in S$ be states with $s \rightarrow s'$.

- s is a predecessor of s'
- s' is a successor of s

If $s \xrightarrow{a} s'$, then action a is applicable in s.

German: Vorgänger, Nachfolger, anwendbar

State Spaces: More Terminology (2)

Definition (path)

Let $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ be a state space.

Let $s_0, \ldots, s_n \in S$ be states and $a_1, \ldots, a_n \in A$ be actions such that $s_0 \xrightarrow{a_1} s_1, \ldots, s_{(n-1)} \xrightarrow{a_n} s_n$.

- $\pi = \langle a_1, \ldots, a_n \rangle$ is a path from s_0 to s_n
- length of π: $|\pi| = n$
- cost of π: $cost(\pi) = \sum_{i=1}^{n} cost(a_i)$

German: Pfad, Länge, Kosten

- paths may have length 0
- sometimes "path" is used for state sequence $\langle s_0, \ldots, s_n \rangle$ or sequence $\langle s_0, a_1, s_1, \ldots, s_{(n-1)}, a_n, s_n \rangle$

State Spaces: More Terminology (3)

More terminology:

Definition (reachable, solution, optimal)

Let $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_I, S_G \rangle$ be a state space.

- state s is reachable if a path from s_I to s exists
- paths from $s \in S$ to some state $s_G \in S_G$ are solutions for/from s
- solutions for s_I are called solutions for $\mathcal{S}$
- optimal solutions (for s) have minimal costs among all solutions (for s)

German: erreichbar, Lösung für/von s, optimale Lösung

State-Space Search

Solving Search Problems

Consider again the running example.

How do you solve this?

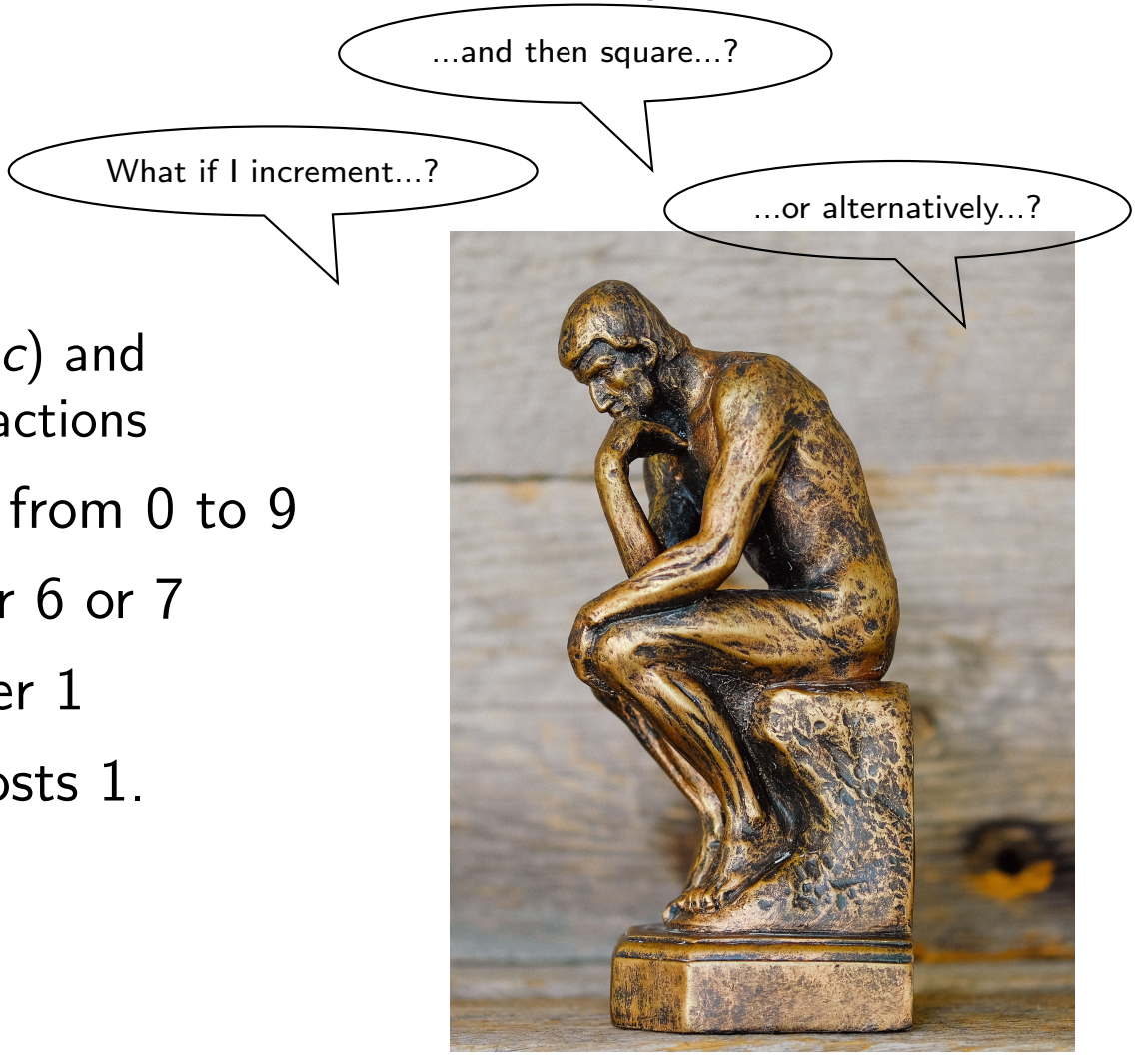
informal description:

- find a sequence of
  - increment-mod10 (*inc*) and
  - square-mod10 (*sqr*) actions
- on the natural numbers from 0 to 9
- that reaches the number 6 or 7
- starting from the number 1
- assuming each action costs 1.

Solving Search Problems

Consider again the running example.

How do you solve this?



informal description:

- find a sequence of
  - increment-mod10 (*inc*) and
  - square-mod10 (*sqr*) actions
- on the natural numbers from 0 to 9
- that reaches the number 6 or 7
- starting from the number 1
- assuming each action costs 1.

State-Space Search

State-Space Search

State-space search is the algorithmic problem
of finding solutions in state spaces
or proving that no solution exists.

In optimal state-space search, only optimal solutions
may be returned.

German: Zustandsraumsuche, optimale Zustandsraumsuche

Learning Objectives for State-Space Search

Learning Objectives for the Topic of State-Space Search

- understanding state-space search:
  What is the problem and how can we formalize it?
- evaluate search algorithms:
  completeness, optimality, time/space complexity
- get to know search algorithms:
  uninformed vs. informed; tree and graph search
- evaluate heuristics for search algorithms:
  goal-awareness, safety, admissibility, consistency
- efficient implementation of search algorithms
- experimental evaluation of search algorithms
- design and comparison of heuristics for search algorithms

Summary

Summary

- state-space search problems:
  find action sequence leading from initial state to a goal state
- performance measure: sum of action costs
- formalization via state spaces:
  - states, actions, action costs, transitions,
    initial state, goal states
- terminology for transitions, paths, solutions
- definition of (optimal) state-space search

Foundations of Artificial Intelligence

B2. State-Space Search: Representation of State Spaces

Malte Helmert

University of Basel

February 26, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
  - B1. State Spaces
  - B2. Representation of State Spaces
  - B3. Examples of State Spaces
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms

Representation of State Spaces

Representation of State Spaces

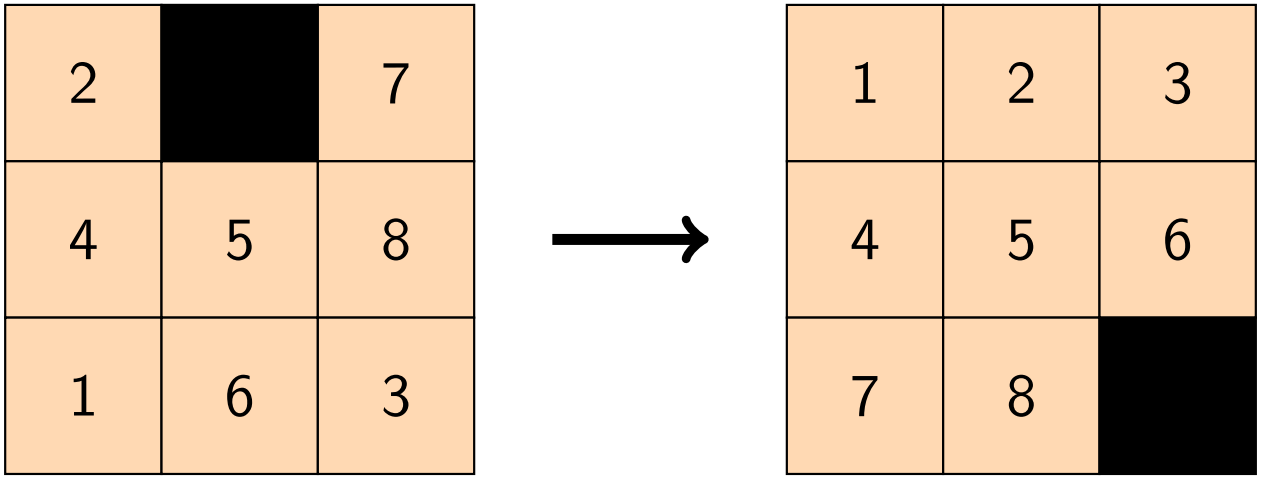
- practically interesting state spaces are often huge (10^{10}, 10^{20}, 10^{100} states)
- How do we represent them, so that we can efficiently deal with them algorithmically?

three main options:

1. as explicit (directed) graphs
2. with declarative representations
3. as a black box

German: explizit, deklarativ, Black Box

Example: 8-Puzzle



Explicit Graphs

State Spaces as Explicit Graphs

State Spaces as Explicit Graphs

represent state spaces as explicit directed graphs:

- vertices = states
- directed arcs = transitions

$\rightsquigarrow$ represented as adjacency list or adjacency matrix

German: Adjazenzliste, Adjazenzmatrix

Example (explicit graph for bounded inc-and-square)

`ai-b02-bounded-inc-and-square.graph`

State Spaces as Explicit Graphs

State Spaces as Explicit Graphs

represent state spaces as explicit directed graphs:

- vertices = states
- directed arcs = transitions

$\rightsquigarrow$ represented as adjacency list or adjacency matrix

German: Adjazenzliste, Adjazenzmatrix

Example (explicit graph for 8-puzzle)

`ai-b02-puzzle8.graph`

State Spaces as Explicit Graphs: Discussion

discussion:

- impossible for large state spaces (too much space required)
- if spaces small enough for explicit representations, solutions easy to compute: Dijkstra's algorithm $O(|S| \log |S| + |T|)$
- interesting for time-critical all-pairs-shortest-path queries (examples: route planning, path planning in video games)

Declarative Representations

State Spaces with Declarative Representations

State Spaces with Declarative Representations

represent state spaces declaratively:

- compact description of state space as input to algorithms
  $\rightsquigarrow$ state spaces exponentially larger than the input
- algorithms directly operate on compact description

$\rightsquigarrow$ allows automatic reasoning about problem:
reformulation, simplification, abstraction, etc.

Example (declarative representation for 8-puzzle)

`puzzle8-domain.pddl` + `puzzle8-problem.pddl`

Black Box

State Spaces as Black Boxes

State Spaces as Black Boxes

Define an abstract interface for state spaces.

For state space $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$
we need these methods:

- init(): generate initial state
  result: state s_I
- is_goal(s): test if s is a goal state
  result: **true** if $s \in S_G$; **false** otherwise
- succ(s): generate applicable actions and successors of s
  result: sequence of pairs $\langle a, s' \rangle$ with $s \xrightarrow{a} s'$
- cost(a): gives cost of action a
  result: $cost(a)$ $(\in \mathbb{N}_0)$

Remark: we will extend the interface later
in a small but important way

State Spaces as Black Boxes: Example and Discussion

Example (Black Box Representation for 8-Puzzle)

demo: `puzzle8.py`

- in the following: focus on black box model
- explicit graphs only as illustrating examples
- near end of semester: declarative state spaces (classical planning)

Summary

Summary

- state spaces often huge ($> 10^{10}$ states)
  $\rightsquigarrow$ how to represent?
- explicit graphs: adjacency lists or matrices; only suitable for small problems
- declaratively: compact description as input to search algorithms
- black box: implement an abstract interface

Foundations of Artificial Intelligence

B3. State-Space Search: Examples of State Spaces

Malte Helmert

University of Basel

February 26, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
  - B1. State Spaces
  - B2. Representation of State Spaces
  - B3. Examples of State Spaces
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms

Three Examples

In this chapter we introduce three state spaces that we will use as illustrating examples:

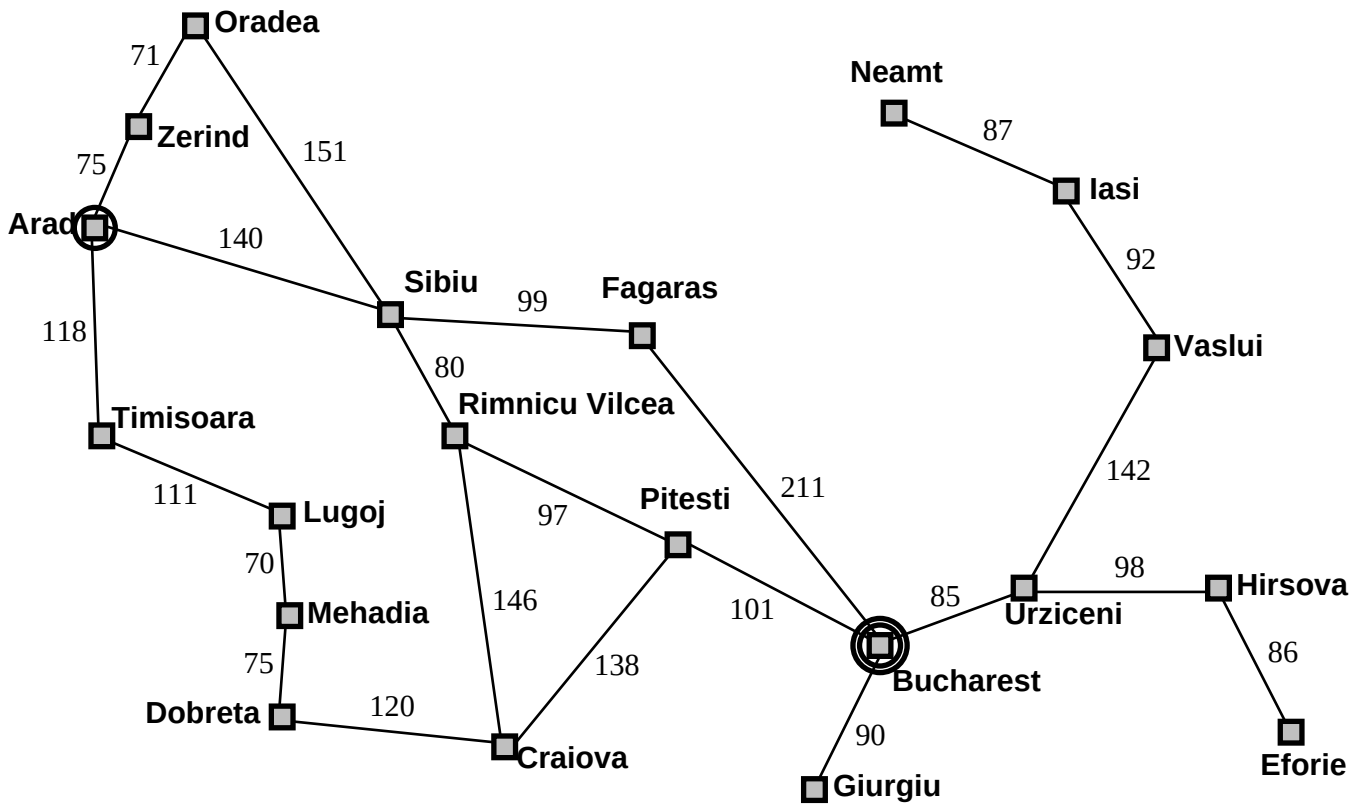
1. route planning in Romania
2. blocks world
3. missionaries and cannibals

Route Planning in Romania

Route Planning in Romania

Setting: Route Planning in Romania

We are on holiday in Romania and are currently located in Arad. Our flight home leaves from Bucharest. How to get there?



Romania Formally

State Space Route Planning in Romania

- states S: $\{\text{arad}, \text{bucharest}, \text{craiova}, \ldots, \text{zerind}\}$
- actions A: $move_{c,c'}$ for any two cities c and c' connected by a single road segment
- action costs $cost$: see figure, e.g., $cost(move_{\text{iasi,vaslui}}) = 92$
- transitions T: $s \xrightarrow{a} s'$ iff $a = move_{s,s'}$
- initial state: $s_{\text{I}} = \text{arad}$
- goal states: $S_{\text{G}} = \{\text{bucharest}\}$

Blocks World

Blocks World

Blocks world is a traditional example problem in AI.

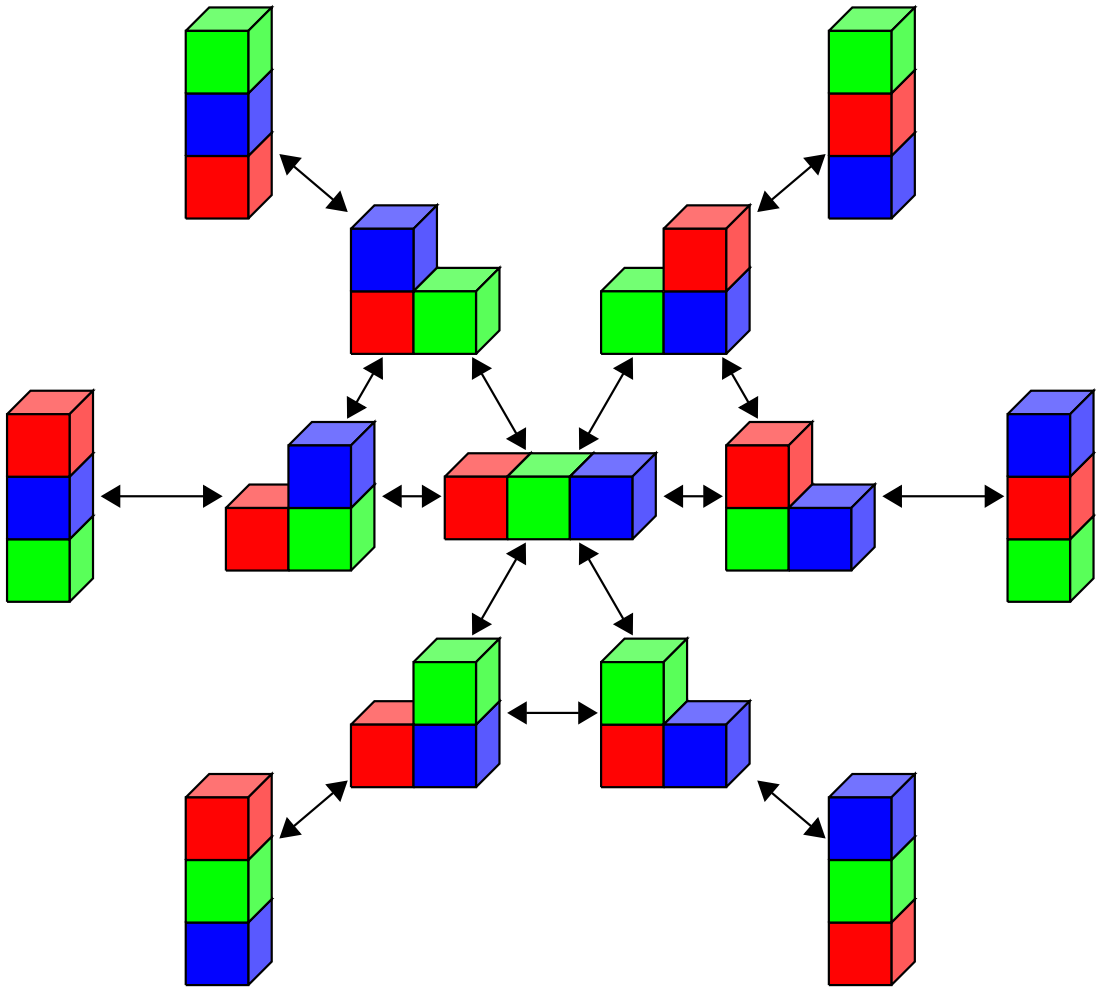
Setting: Blocks World

- Colored blocks lie on a table.
- They can be stacked into towers, moving one block at a time.
- Our task is to create a given goal configuration.

Example: Blocks World with Three Blocks

Action names omitted for readability. All actions cost 1.

Initial state and goal can be arbitrary.



Blocks World: Formal Definition

state space $\langle S, A, \mathit{cost}, T, s_I, S_G \rangle$ for blocks world with n blocks

State Space Blocks World

states S:

partitions of $\{1, 2, \ldots, n\}$ into nonempty ordered lists

example $n = 3$:

- $\{\langle 1,2,3\rangle\}$, $\{\langle 1,3,2\rangle\}$, $\{\langle 2,1,3\rangle\}$, $\{\langle 2,3,1\rangle\}$, $\{\langle 3,1,2\rangle\}$, $\{\langle 3,2,1\rangle\}$
- $\{\langle 1,2\rangle, \langle 3\rangle\}$, $\{\langle 2,1\rangle, \langle 3\rangle\}$, $\{\langle 1,3\rangle, \langle 2\rangle\}$, $\{\langle 3,1\rangle, \langle 2\rangle\}$, $\{\langle 2,3\rangle, \langle 1\rangle\}$, $\{\langle 3,2\rangle, \langle 1\rangle\}$
- $\{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$

Blocks World: Formal Definition

state space $\langle S, A, cost, T, s_I, S_G \rangle$ for blocks world with n blocks

State Space Blocks World

actions A:

- $\{move_{u,v} \mid u, v \in \{1, \ldots, n\}$ with $u \neq v\}$
  - move block u onto block v.
  - both must be uppermost blocks in their towers
- $\{to\text{-}table_u \mid u \in \{1, \ldots, n\}\}$
  - move block u onto the table ($\rightsquigarrow$ forming a new tower)
  - must be uppermost block in its tower

action costs $cost$:

$cost(a) = 1$ for all actions $a \in A$

Blocks World: Formal Definition

state space $\langle S, A, cost, T, s_I, S_G \rangle$ for blocks world with n blocks

State Space Blocks World

transitions:

- transition $s \xrightarrow{a} s'$ with $a = move_{u,v}$ exists iff
  - $s = \{\langle b_1, \ldots, b_k, u \rangle, \langle c_1, \ldots, c_m, v \rangle\} \cup X$ and
  - if $k > 0$: $s' = \{\langle b_1, \ldots, b_k \rangle, \langle c_1, \ldots, c_m, v, u \rangle\} \cup X$
  - if $k = 0$: $s' = \{\langle c_1, \ldots, c_m, v, u \rangle\} \cup X$
- transition $s \xrightarrow{a} s'$ with $a = to\text{-}table_u$ exists iff
  - $s = \{\langle b_1, \ldots, b_k, u \rangle\} \cup X$ with $k > 0$ and
  - $s' = \{\langle b_1, \ldots, b_k \rangle, \langle u \rangle\} \cup X$

Blocks World: Formal Definition

state space $\langle S, A, \mathit{cost}, T, s_I, S_G \rangle$ for blocks world with n blocks

State Space Blocks World

initial state s_I and goal states S_G:

one possible scenario for $n = 3$:

- $s_I = \{\langle 1, 3\rangle, \langle 2\rangle\}$
- $S_G = \{\{\langle 3, 2, 1\rangle\}\}$

(in general can have arbitrary scenarios)

Blocks World: 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

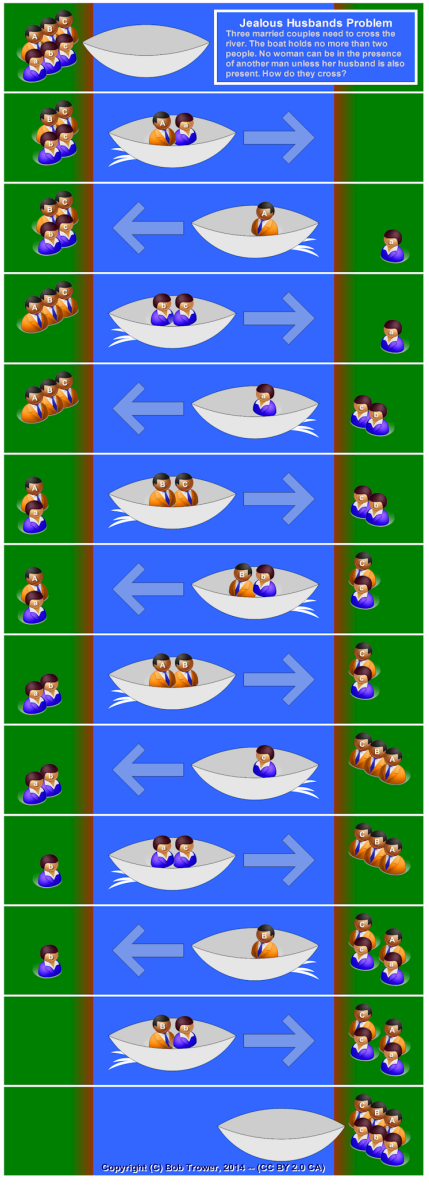
- For every given initial and goal state with n blocks, simple algorithms find a solution in time $O(n)$. (How?)
- Finding optimal solutions is NP-complete (with a compact problem description).

Missionaries and Cannibals

Missionaries and Cannibals

Setting: Missionaries and Cannibals

- Six people must cross a river.
- Their rowing boat can carry one or two people across the river at a time.
  (It is too small for three.)
- Three people are missionaries,
  three are cannibals.
- Missionaries may never stay
  with a majority of cannibals.



Missionaries and Cannibals Formally

State Space Missionaries and Cannibals

states S:

triples of numbers $\langle m, c, b \rangle \in \{0, 1, 2, 3\} \times \{0, 1, 2, 3\} \times \{0, 1\}$:

- number of missionaries m,
- cannibals c and
- boats b

on the left river bank

initial state: $s_I = \langle 3, 3, 1 \rangle$

goal: $S_G = \{\langle 0, 0, 0 \rangle, \langle 0, 0, 1 \rangle\}$

actions, action costs, transitions: ?

Summary

Summary

illustrating examples for state spaces:

- route planning in Romania:
  - small example of explicitly representable state space
- blocks world:
  - family of tasks where n blocks on a table must be rearranged
  - traditional example problem in AI
  - number of states explodes quickly as n grows
- missionaries and cannibals:
  - traditional brain teaser with small state space (32 states, of which many unreachable)

Foundations of Artificial Intelligence

B4. State-Space Search: Data Structures for Search Algorithms

Malte Helmert

University of Basel

March 3, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
  - B4. Data Structures for Search Algorithms
  - B5. Tree Search and Graph Search
  - B6. Breadth-first Search
  - B7. Uniform Cost Search
  - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Introduction

Finding Solutions in State Spaces

How can we systematically find a solution?

Search Algorithms

- We now move to search algorithms.
- As everywhere in computer science, suitable data structures are a key to good performance.
  - ⇝ common operations must be fast
- Well-implemented search algorithms process up to ∼30,000,000 states/second on a single CPU core.
  - ⇝ bonus materials (Burns et al. paper)

this chapter: some fundamental data structures for search

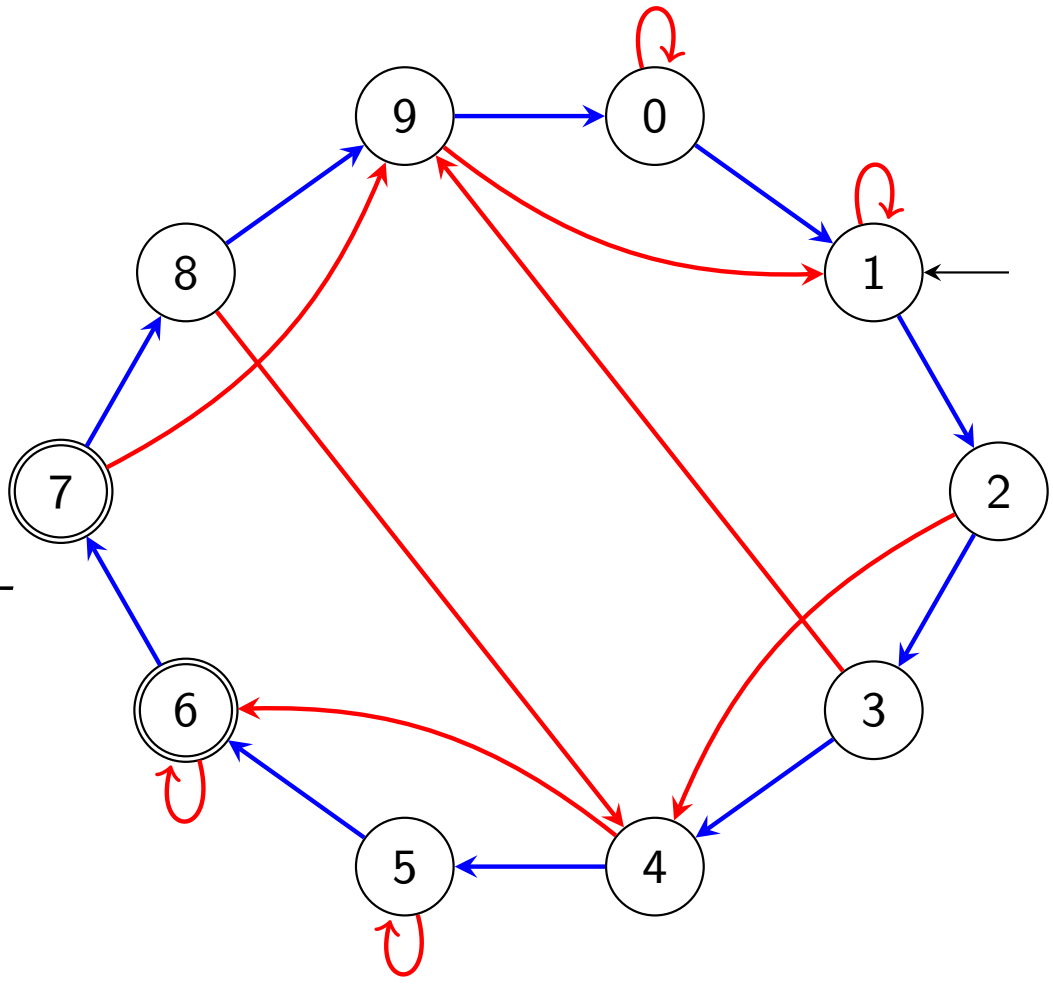
Preview: Search Algorithms

- next chapter: we introduce search algorithms
- now: short preview to motivate data structures for search

Running Example: Reminder

bounded inc-and-square:

- $S = \{0, 1, \ldots, 9\}$
- $A = \{inc, sqr\}$
- $cost(inc) = cost(sqr) = 1$
- T s.t. for $i = 0, \ldots, 9$:
  - $\langle i, inc, (i + 1) \bmod 10 \rangle \in T$
  - $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- $s_I = 1$
- $S_G = \{6, 7\}$



Search Algorithms: Idea

iteratively create a search tree:

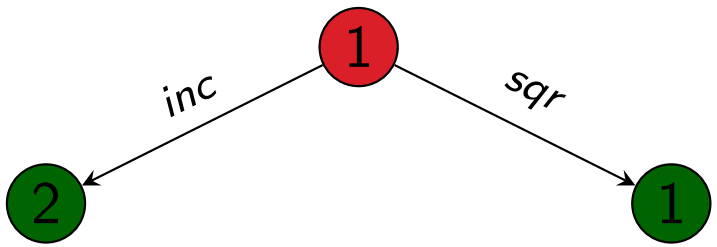
- starting with the initial state,

Search Algorithms: Idea

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)

German: expandieren, erzeugen

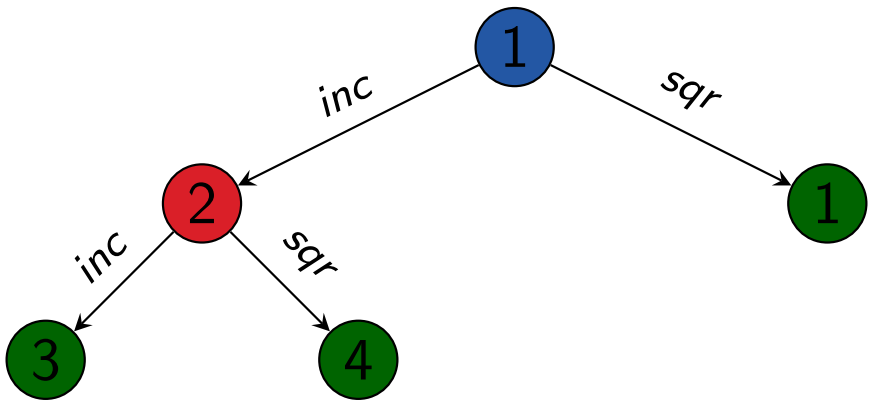


Search Algorithms: Idea

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)

German: expandieren, erzeugen

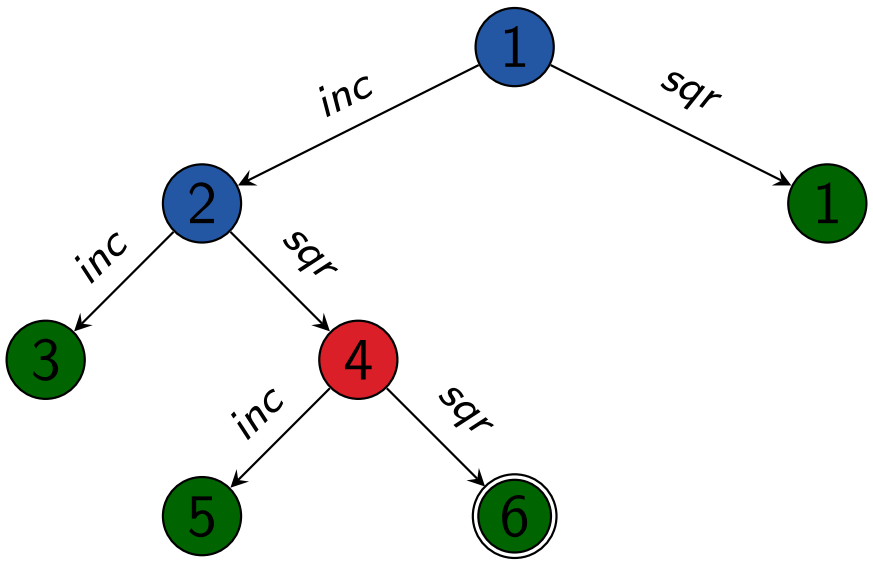


Search Algorithms: Idea

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)

German: expandieren, erzeugen

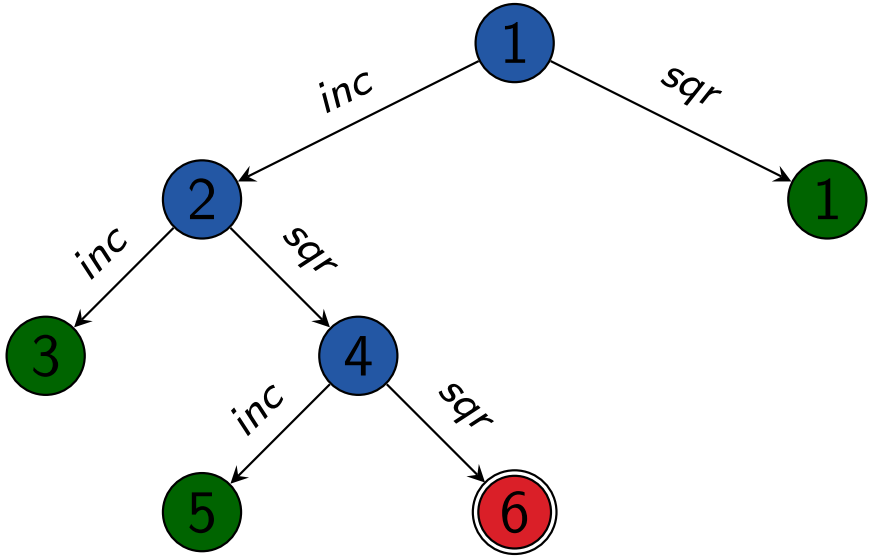


Search Algorithms: Idea

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)
- stop when a goal state is expanded (sometimes: generated)
- or all reachable states have been considered

German: expandieren, erzeugen



Fundamental Data Structures for Search

We consider three abstract data structures for search:

- search node: stores a state that has been reached, how it was reached, and at which cost
  - ⇝ nodes of the example search tree
- open list: efficiently organizes leaves of search tree
  - ⇝ set of leaves of example search tree
- closed list: remembers expanded states to avoid duplicated expansions of the same state
  - ⇝ inner nodes of a search tree

German: Suchknoten, Open-Liste, Closed-Liste

Not all algorithms use all three data structures,
and they are sometimes implicit (e.g., on the CPU stack)

Search Nodes

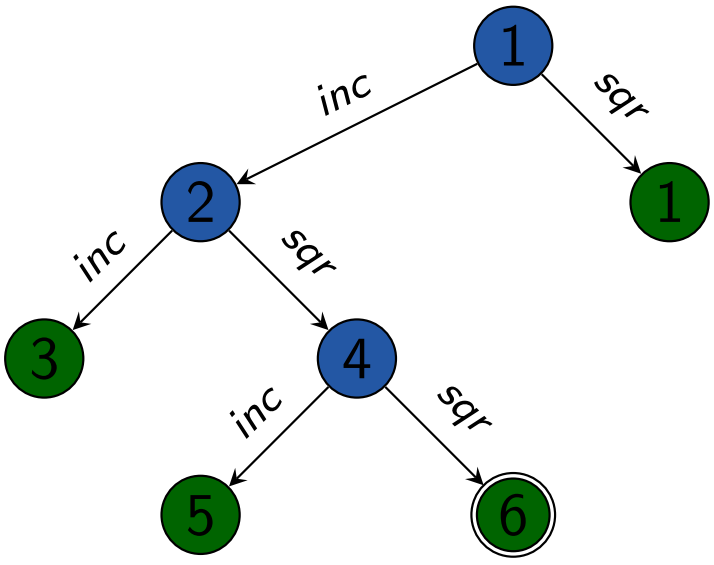
Search Nodes

Search Node

A search node (node for short) stores a state
that has been reached, how it was reached, and at which cost.

Collectively they form the so-called search tree (Suchbaum).

Data Structure: Search Nodes

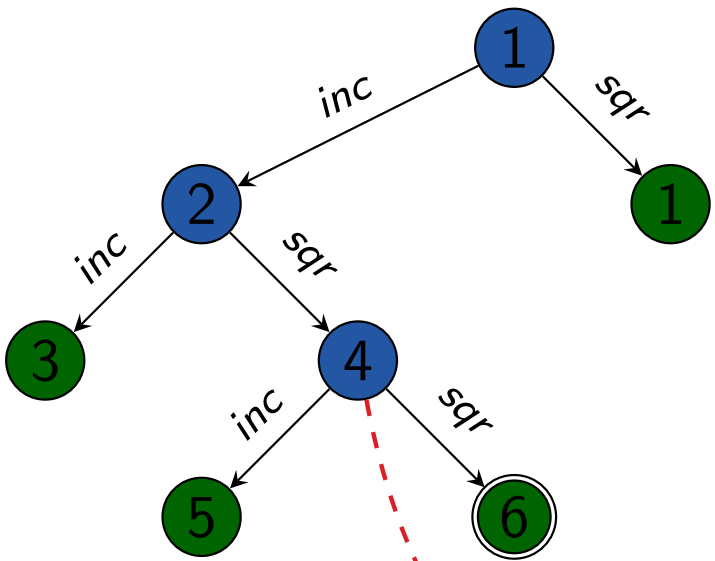


attributes of search node n:

n.state	state associated with n
n.parent	search node that generated n (**none** for the root node)
n.action	action leading from n.parent to n (**none** for the root node)
n.path_cost	cost of path from s_I to n.state that results from following parent references (traditionally denoted by $g(n)$)

. . . and sometimes additional attributes

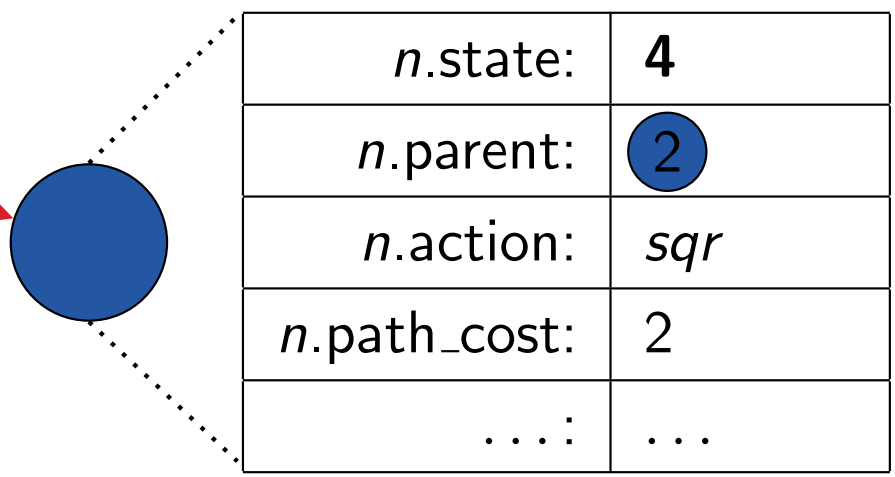
Data Structure: Search Nodes



attributes of search node n:

n.state	state associated with n
n.parent	search node that generated n (**none** for the root node)
n.action	action leading from n.parent to n (**none** for the root node)
n.path_cost	cost of path from s_I to n.state that results from following parent references (traditionally denoted by $g(n)$)

. . . and sometimes additional attributes



Search Nodes: Java

Search Nodes (Java Syntax)

```
public interface State {
}

public interface Action {
}

public class SearchNode {
    State state;
    SearchNode parent;
    Action action;
    int pathCost;
}
```

Implementing Search Nodes

- reasonable implementation of search nodes is easy
- advanced aspects:
  - Do we need explicit nodes at all?
  - Can we use lazy evaluation?
  - Should we manually manage memory?
  - Can we compress information?

Operations on Search Nodes: make_root_node

Generate root node of a search tree:

function make_root_node()

node := **new** SearchNode
node.state := init()
node.parent := **none**
node.action := **none**
node.path_cost := 0
return *node*

Operations on Search Nodes: make_node

Generate child node of a search node:

function make_node(*parent*, *action*, *state*)

node := **new** SearchNode
node.state := *state*
node.parent := *parent*
node.action := *action*
node.path_cost := *parent*.path_cost + cost(*action*)
return *node*

Operations on Search Nodes: extract_path

Extract the path to a search node:

function extract_path(*node*)

```
path := ⟨⟩
while node.parent ≠ none:
    path.append(node.action)
    node := node.parent
path.reverse()
return path
```

Open Lists

Open Lists

Open List

The open list (also: frontier) organizes the leaves of a search tree.

It must support two operations efficiently:

- determine and remove the next node to expand
- insert a new node that is a candidate node for expansion

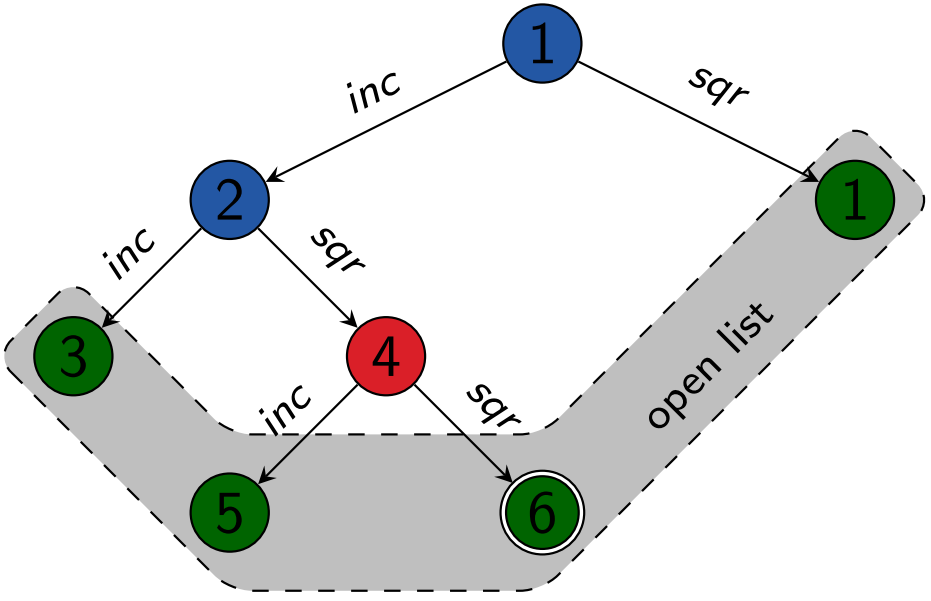
Remark: despite the name, it is usually a very bad idea to implement open lists as simple lists.

Open Lists: Modify Entries

- Some implementations support modifying an open list entry when a shorter path to the corresponding state is found.
- This complicates the implementation.

$\rightsquigarrow$ We do not consider such modifications and instead use delayed duplicate elimination ($\rightsquigarrow$ later).

Interface of Open Lists

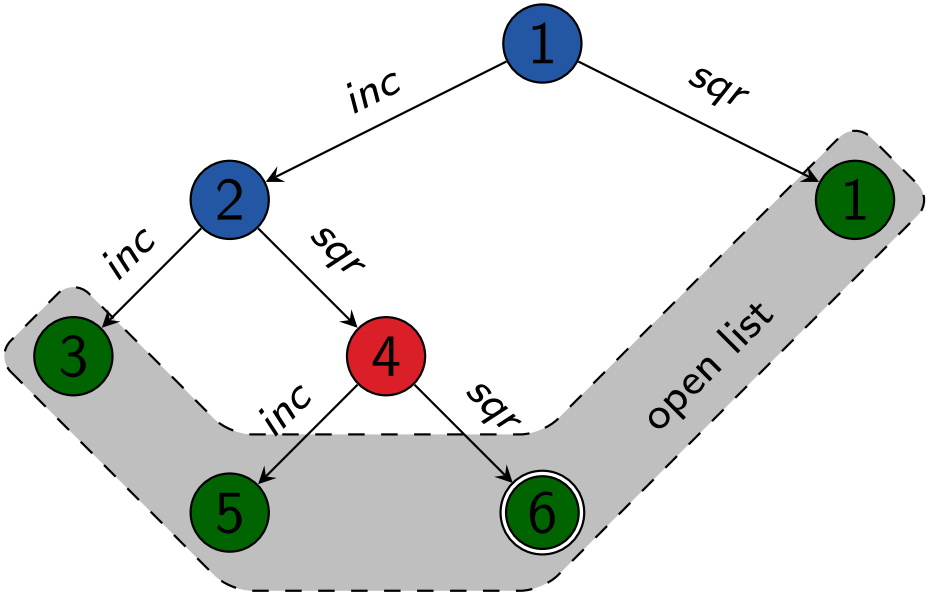


- open list *open* organizes leaves of search tree with the methods:

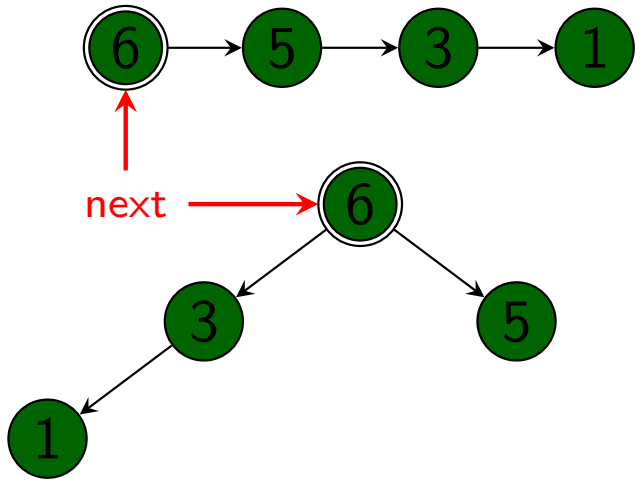
open.is_empty()	test if the open list is empty
open.pop()	remove and return the next node to expand
open.insert(n)	insert node n into the open list

- *open* determines strategy which node to expand next (depends on algorithm)
- underlying data structure choice depends on this strategy

Interface of Open Lists



examples: deque, min-heap



- open list *open* organizes leaves of search tree with the methods:

open.is_empty()	test if the open list is empty
open.pop()	remove and return the next node to expand
open.insert(n)	insert node n into the open list

- *open* determines strategy which node to expand next (depends on algorithm)
- underlying data structure choice depends on this strategy

Closed Lists

Closed Lists

Closed List

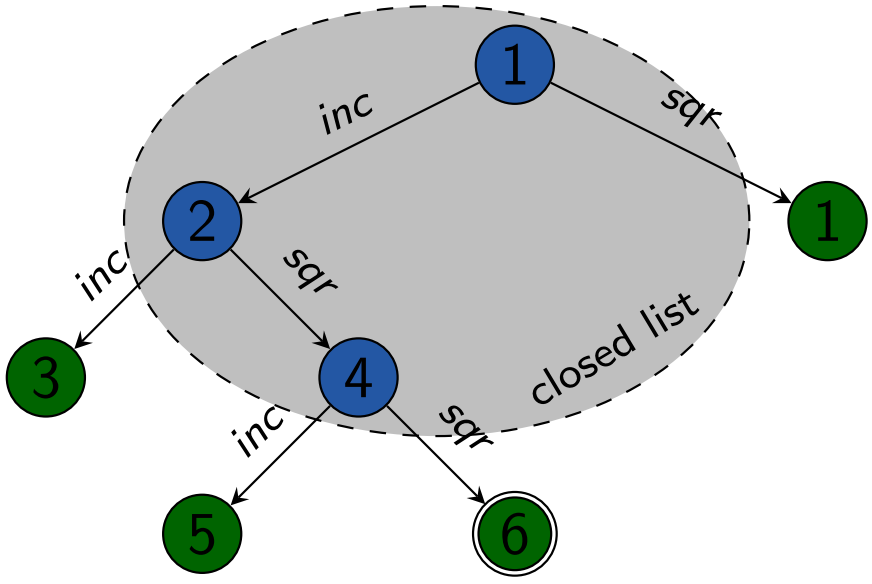
The closed list remembers expanded states
to avoid duplicated expansions of the same state.

It must support two operations efficiently:

- insert a node whose state is not yet in the closed list
- test if a node with a given state is in the closed list;
  if yes, return it

Remark: despite the name, it is usually a very bad idea
to implement closed lists as simple lists. (Why?)

Interface and Implementation of Closed Lists



- closed list *closed* keeps track of expanded states with the methods:

  closed.insert(*n*) insert node *n* into *closed*;
  if a node with this state already exists in *closed*, replace it

  closed.lookup(*s*) test if a node with state *s* exists in the closed list;
  if yes, return it; otherwise, return **none**

- efficient implementation often as hash table with states as keys

Summary

Summary

- search node:
  represents states reached during search
  and associated information
- node expansion:
  generate successor nodes of a node by applying all actions
  applicable in the state belonging to the node
- open list or frontier:
  set of nodes that are currently candidates for expansion
- closed list:
  set of already expanded nodes (and their states)

Foundations of Artificial Intelligence

B5. State-Space Search: Tree Search and Graph Search

Malte Helmert

University of Basel

March 3, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
  - B4. Data Structures for Search Algorithms
  - B5. Tree Search and Graph Search
  - B6. Breadth-first Search
  - B7. Uniform Cost Search
  - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Introduction

Search Algorithms

General Search Algorithm

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)
- stop when a goal state is expanded (sometimes: generated)
- or all reachable states have been considered

Search Algorithms

General Search Algorithm

iteratively create a search tree:

- starting with the initial state,
- repeatedly expand a state by generating its successors (which state depends on the used search algorithm)
- stop when a goal state is expanded (sometimes: generated)
- or all reachable states have been considered

In this chapter, we study two essential classes of search algorithms:

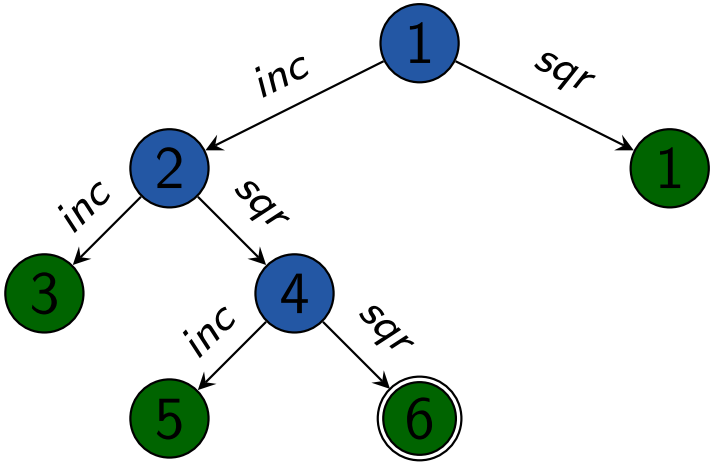
- tree search
- graph search

Each class consists of a large number of concrete algorithms.

German: expandieren, erzeugen, Baumsuche, Graphensuche

Tree Search

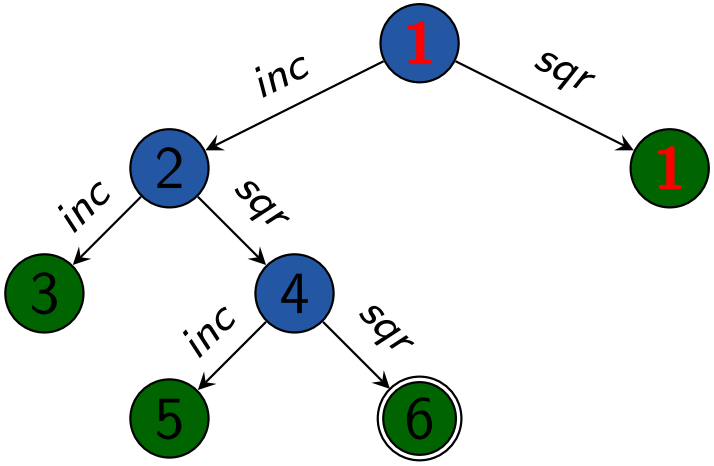
Tree Search: General Idea



- possible paths to be explored organized in a tree (search tree)
- search nodes correspond 1:1 to paths from initial state
- duplicates a.k.a. transpositions (i.e., multiple nodes with identical state) possible
- search tree can have unbounded depth

German: Suchbaum, Duplikate, Transpositionen

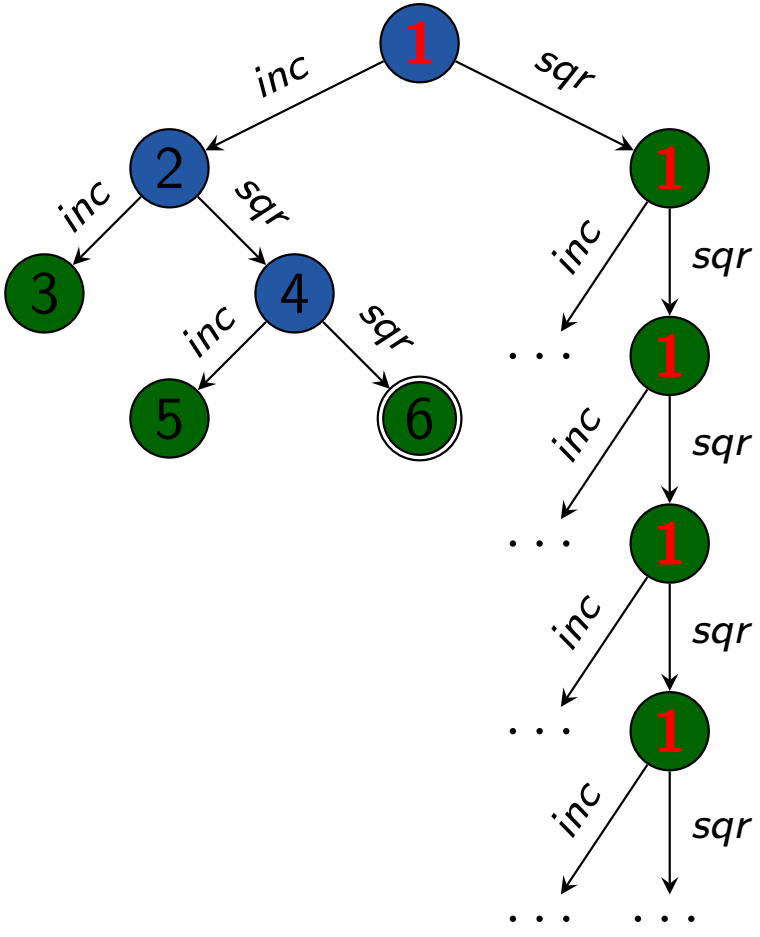
Tree Search: General Idea



- possible paths to be explored organized in a tree (search tree)
- search nodes correspond 1:1 to paths from initial state
- duplicates a.k.a. transpositions (i.e., multiple nodes with identical state) possible
- search tree can have unbounded depth

German: Suchbaum, Duplikate, Transpositionen

Tree Search: General Idea



- possible paths to be explored organized in a tree (search tree)
- search nodes correspond 1:1 to paths from initial state
- duplicates a.k.a. transpositions (i.e., multiple nodes with identical state) possible
- search tree can have unbounded depth

German: Suchbaum, Duplikate, Transpositionen

Generic Tree Search Algorithm

Generic Tree Search Algorithm

```
open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n := open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

Generic Tree Search Algorithm: Discussion

discussion:

- generic template for tree search algorithms

$\rightsquigarrow$ for concrete algorithm, we must (at least) decide how to implement the open list

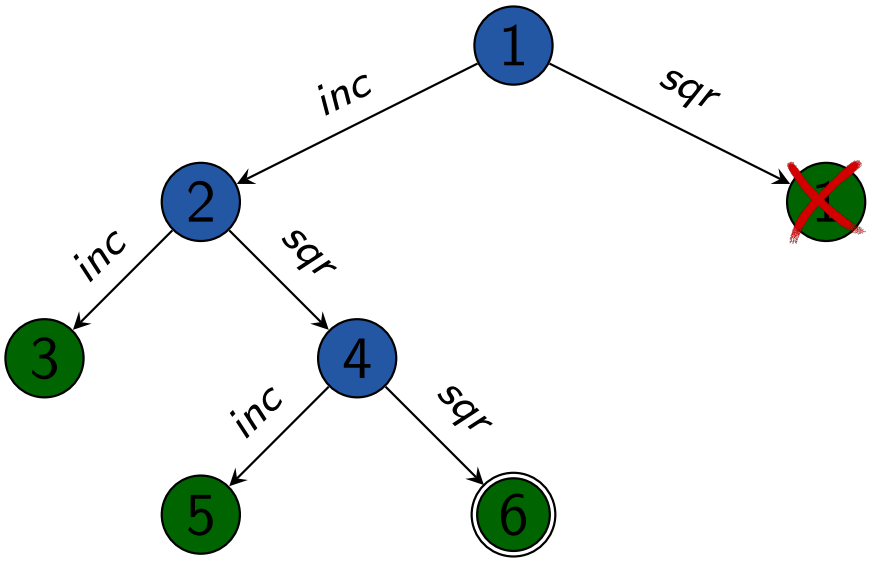
- concrete algorithms often conceptually follow template, (= generate the same search tree), but deviate from details for efficiency reasons

Graph Search

Graph Search

differences to tree search:

- recognize duplicates: when a state is reached on multiple paths, only keep one search node
- search nodes correspond 1:1 to reachable states
- depth of search tree bounded



remarks:

- some graph search algorithms do not immediately eliminate all duplicates ($\rightsquigarrow$ later)
- one possible reason: find optimal solutions when a path to state s found later is cheaper than one found earlier

Generic Graph Search Algorithm

Generic Graph Search Algorithm

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n := open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Generic Graph Search Algorithm: Discussion

discussion:

- same comments as for generic tree search apply
- in "pure" algorithm, closed list does not actually need to store the search nodes
  - sufficient to implement *closed* as set of states
  - advanced algorithms often need access to the nodes, hence we show this more general version here
- some variants perform goal and duplicate tests elsewhere (earlier) $\rightsquigarrow$ following chapters

Evaluating Search Algorithms

Criteria: Completeness

four criteria for evaluating search algorithms:

Completeness

Is the algorithm guaranteed to find a solution if one exists?

Does it terminate if no solution exists?

first property: semi-complete
both properties: complete

German: Vollständigkeit, semi-vollständig, vollständig

Criteria: Optimality

four criteria for evaluating search algorithms:

Optimality

Are the solutions returned by the algorithm always optimal?

German: Optimalität

Criteria: Time Complexity

four criteria for evaluating search algorithms:

Time Complexity

How much time does the algorithm need until termination?

- usually worst case analysis
- usually measured in generated nodes

often a function of the following quantities:

- b: (branching factor) of state space
  (max. number of successors of a state)
- d: search depth
  (length of longest path in generated search tree)

German: Zeitaufwand, Verzweigungsgrad, Suchtiefe

Criteria: Space Complexity

four criteria for evaluating search algorithms:

Space Complexity

How much memory does the algorithm use?

- usually worst case analysis
- usually measured in (concurrently) stored nodes

often a function of the following quantities:

- b: (branching factor) of state space
  (max. number of successors of a state)
- d: search depth
  (length of longest path in generated search tree)

German: Speicheraufwand

Analyzing the Generic Search Algorithms

Generic Tree Search Algorithm

- Is it complete? Is it semi-complete?
- Is it optimal?
- What is its worst-case time complexity?
- What is its worst-case space complexity?

Generic Graph Search Algorithm

- Is it complete? Is it semi-complete?
- Is it optimal?
- What is its worst-case time complexity?
- What is its worst-case space complexity?

Summary

Summary (1)

tree search:

- search nodes correspond 1:1 to paths from initial state

graph search:

- search nodes correspond 1:1 to reachable states

$\rightsquigarrow$ duplicate elimination

generic methods with many possible variants

Summary (2)

evaluating search algorithms:

- completeness and semi-completeness
- optimality
- time complexity and space complexity

Foundations of Artificial Intelligence

B6. State-Space Search: Breadth-first Search

Malte Helmert

University of Basel

March 5, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
  - B4. Data Structures for Search Algorithms
  - B5. Tree Search and Graph Search
  - B6. Breadth-first Search
  - B7. Uniform Cost Search
  - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Blind Search

Blind Search

In Chapters B6–B8 we consider blind search algorithms:

Blind Search Algorithms

Blind search algorithms use no information about state spaces apart from the black box interface.

They are also called uninformed search algorithms.

contrast: heuristic search algorithms (Chapters B9–B15)

Blind Search Algorithms: Examples

examples of blind search algorithms:

- breadth-first search
- uniform cost search
- depth-first search
- depth-limited search
- iterative deepening search

Blind Search Algorithms: Examples

examples of blind search algorithms:

- breadth-first search ($\rightsquigarrow$ this chapter)
- uniform cost search
- depth-first search
- depth-limited search
- iterative deepening search

Blind Search Algorithms: Examples

examples of blind search algorithms:

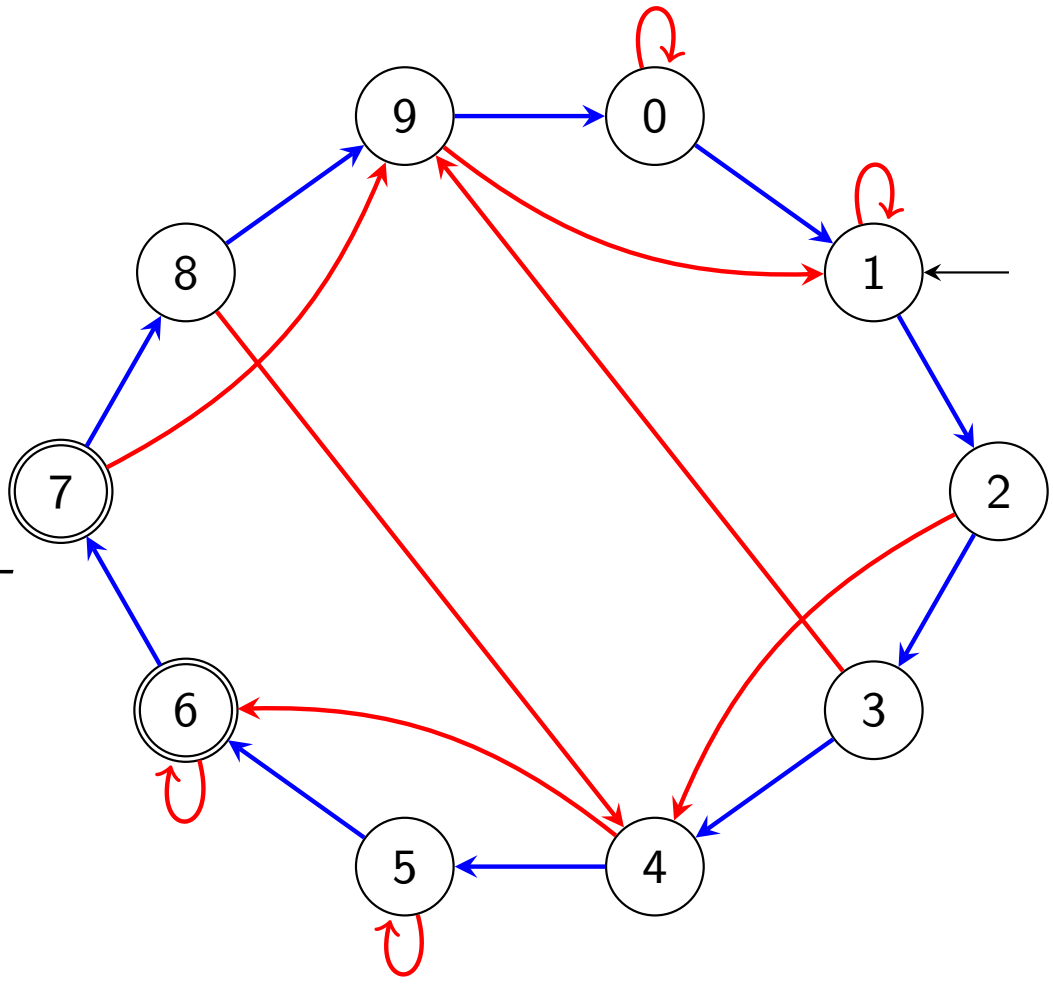
- breadth-first search ($\rightsquigarrow$ this chapter)
- uniform cost search ($\rightsquigarrow$ Chapter B7)
- depth-first search ($\rightsquigarrow$ Chapter B8)
- depth-limited search ($\rightsquigarrow$ Chapter B8)
- iterative deepening search ($\rightsquigarrow$ Chapter B8)

Breadth-first Search: Introduction

Running Example: Reminder

bounded inc-and-square:

- $S = \{0, 1, \ldots, 9\}$
- $A = \{inc, sqr\}$
- $cost(inc) = cost(sqr) = 1$
- T s.t. for $i = 0, \ldots, 9$:
  - $\langle i, inc, (i + 1) \bmod 10 \rangle \in T$
  - $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- $s_I = 1$
- $S_G = \{6, 7\}$



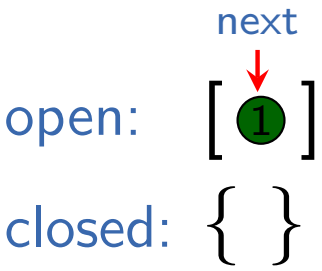
Idea

breadth-first search:

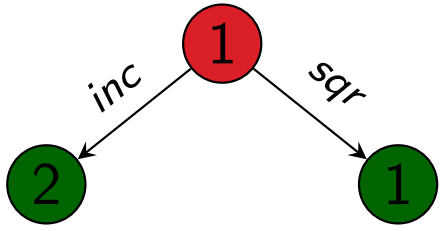
- expand nodes in order of generation (FIFO)
  - $\rightsquigarrow$ open list is linked list or deque
- we start with an example using graph search

German: Breitensuche

Example: Generic Graph Search with FIFO Expansion



Example: Generic Graph Search with FIFO Expansion

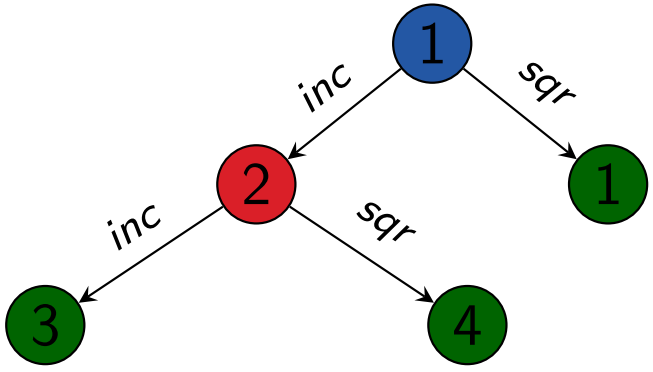


next

open: [2 1]

closed: {1}

Example: Generic Graph Search with FIFO Expansion

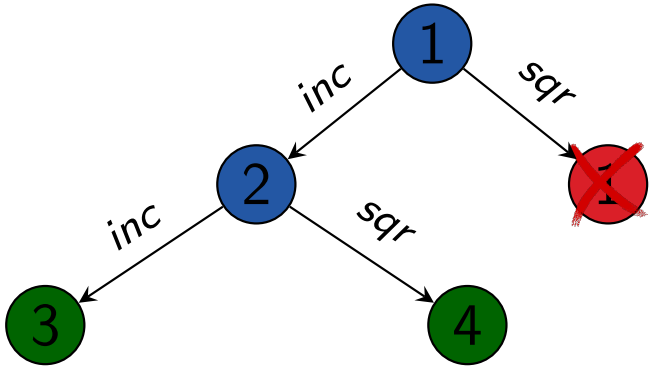


next

open: [1 3 4]

closed: {1, 2}

Example: Generic Graph Search with FIFO Expansion

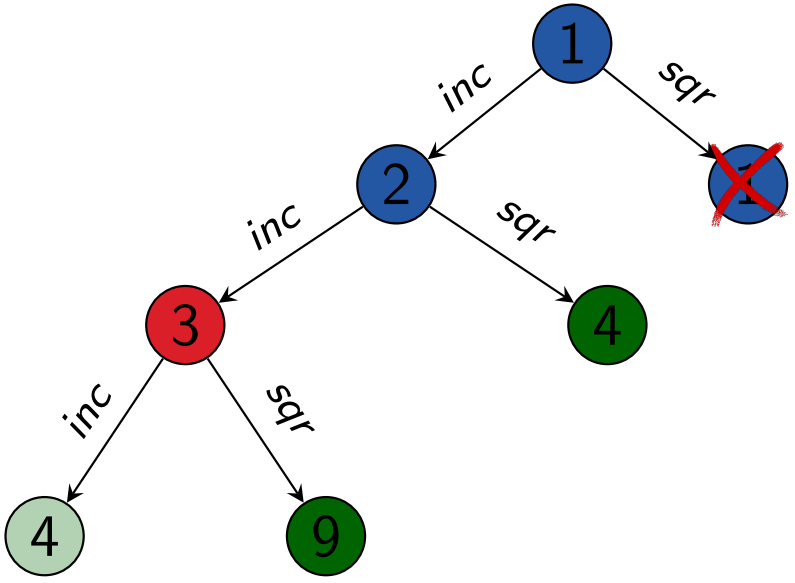


next

open: [3 4]

closed: $\{1, 2\}$

Example: Generic Graph Search with FIFO Expansion

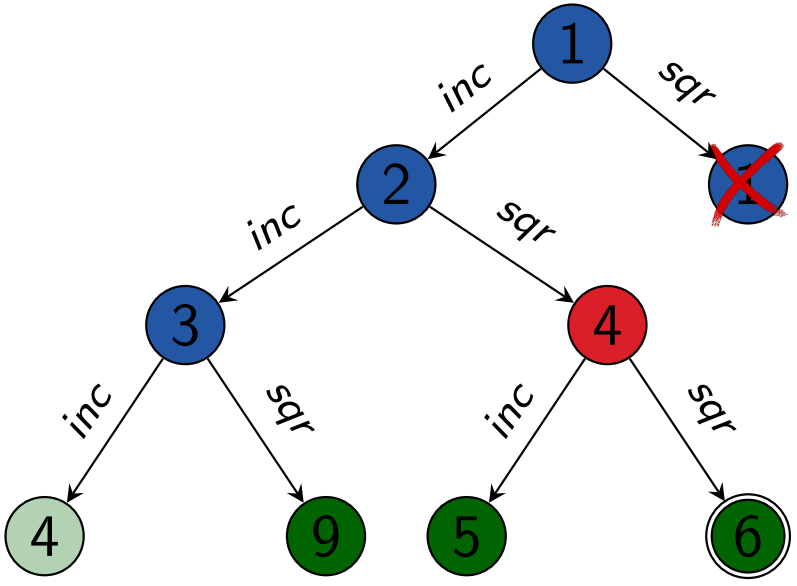


next

open: [4 4 9]

closed: {1, 2, 3}

Example: Generic Graph Search with FIFO Expansion

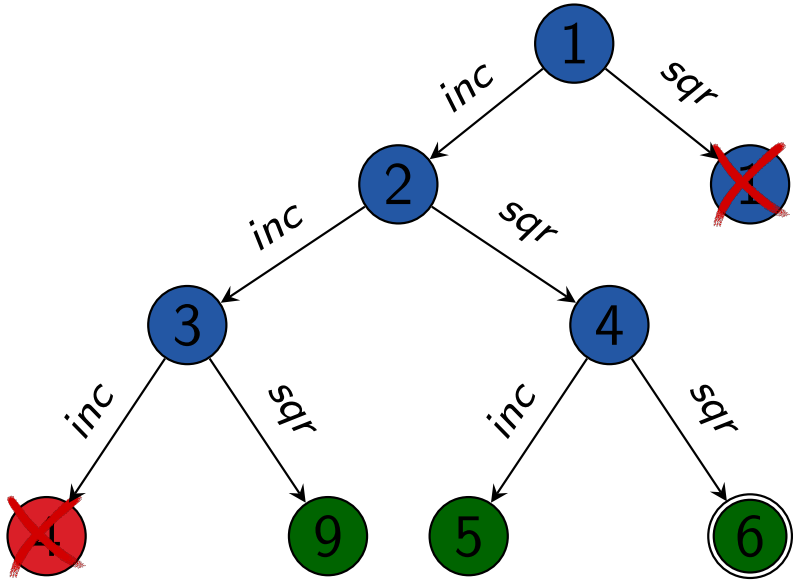


next

open: [4 9 5 6]

closed: {1, 2, 3, 4}

Example: Generic Graph Search with FIFO Expansion

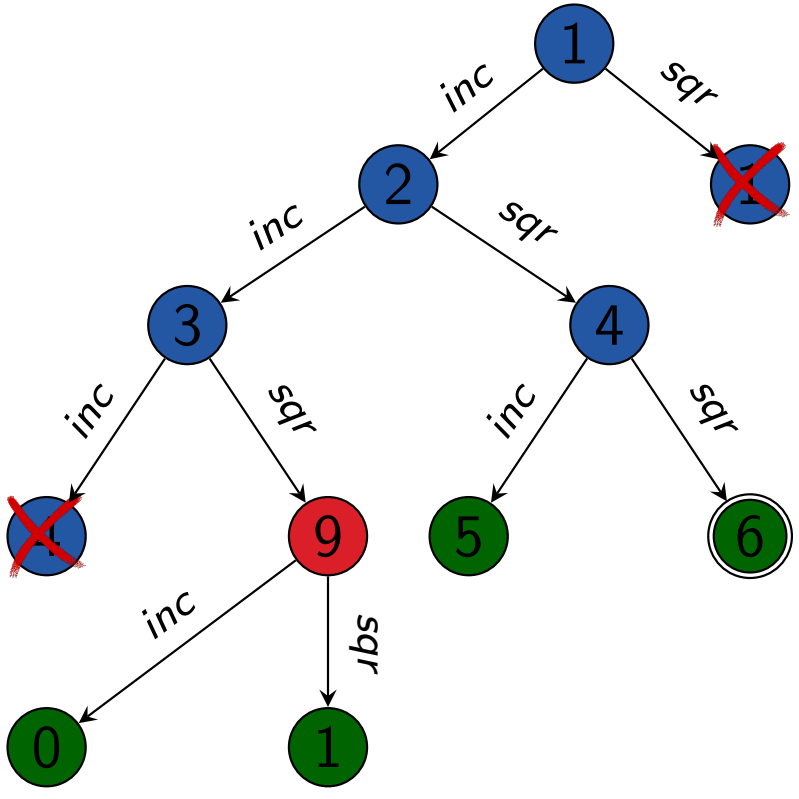


next

open: [9 5 6]

closed: {1, 2, 3, 4}

Example: Generic Graph Search with FIFO Expansion

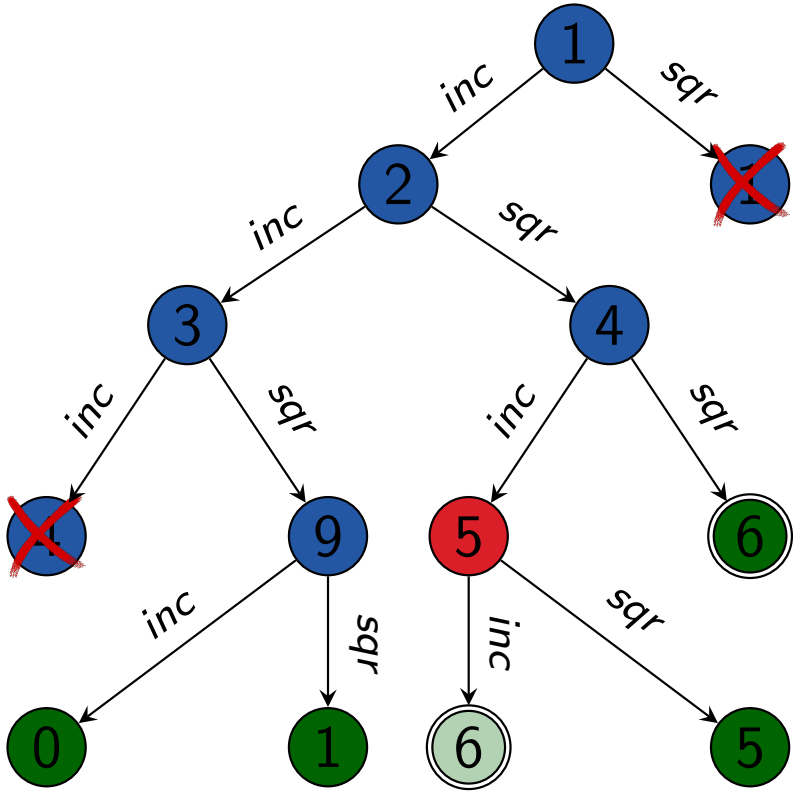


next

open: [5 6 0 1]

closed: {1, 2, 3, 4, 9}

Example: Generic Graph Search with FIFO Expansion

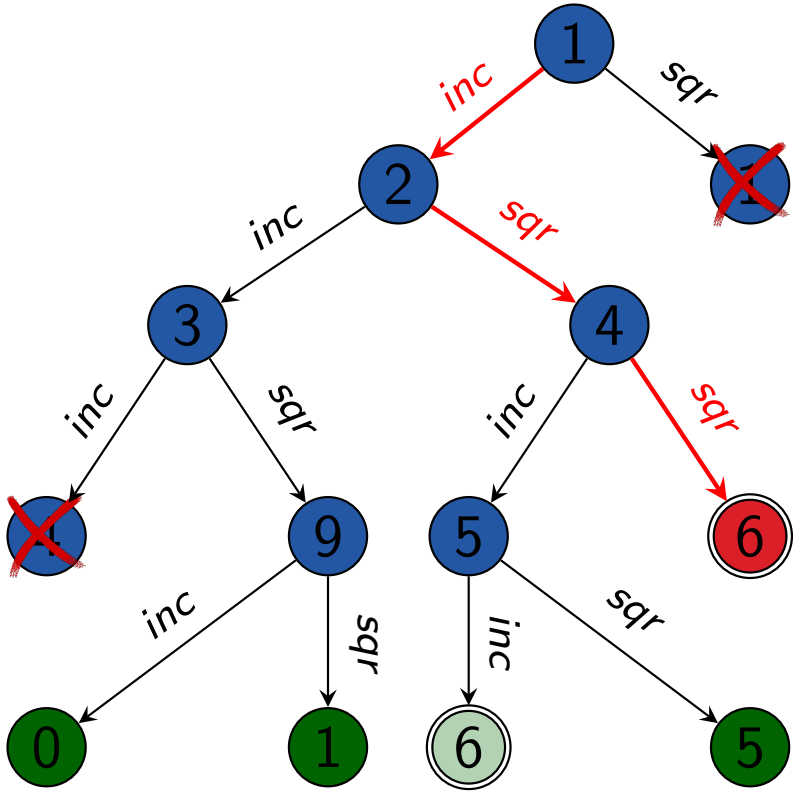


next

open: [6 0 1 6 5]

closed: {1, 2, 3, 4, 5, 9}

Example: Generic Graph Search with FIFO Expansion



next

open: [0 1 6 5]

closed: {1, 2, 3, 4, 5, 6, 9}

Observations from Example

breadth-first search behaviour:

- state space is searched layer by layer

$\rightsquigarrow$ shallowest goal node is always found first

Breadth-first Search: Tree Search or Graph Search?

Breadth-first search can be performed

- without duplicate elimination (as a tree search)
  $\rightsquigarrow$ BFS-Tree
- or with duplicate elimination (as a graph search)
  $\rightsquigarrow$ BFS-Graph

(BFS = breadth-first search).

$\rightsquigarrow$ We consider both variants.

BFS-Tree

Reminder: Generic Tree Search Algorithm

reminder from Chapter B5:

Generic Tree Search

```
open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n := open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

BFS-Tree (1st Attempt)

breadth-first search without duplicate elimination (1st attempt):

BFS-Tree (1st Attempt)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.push_back(n')
return unsolvable
```

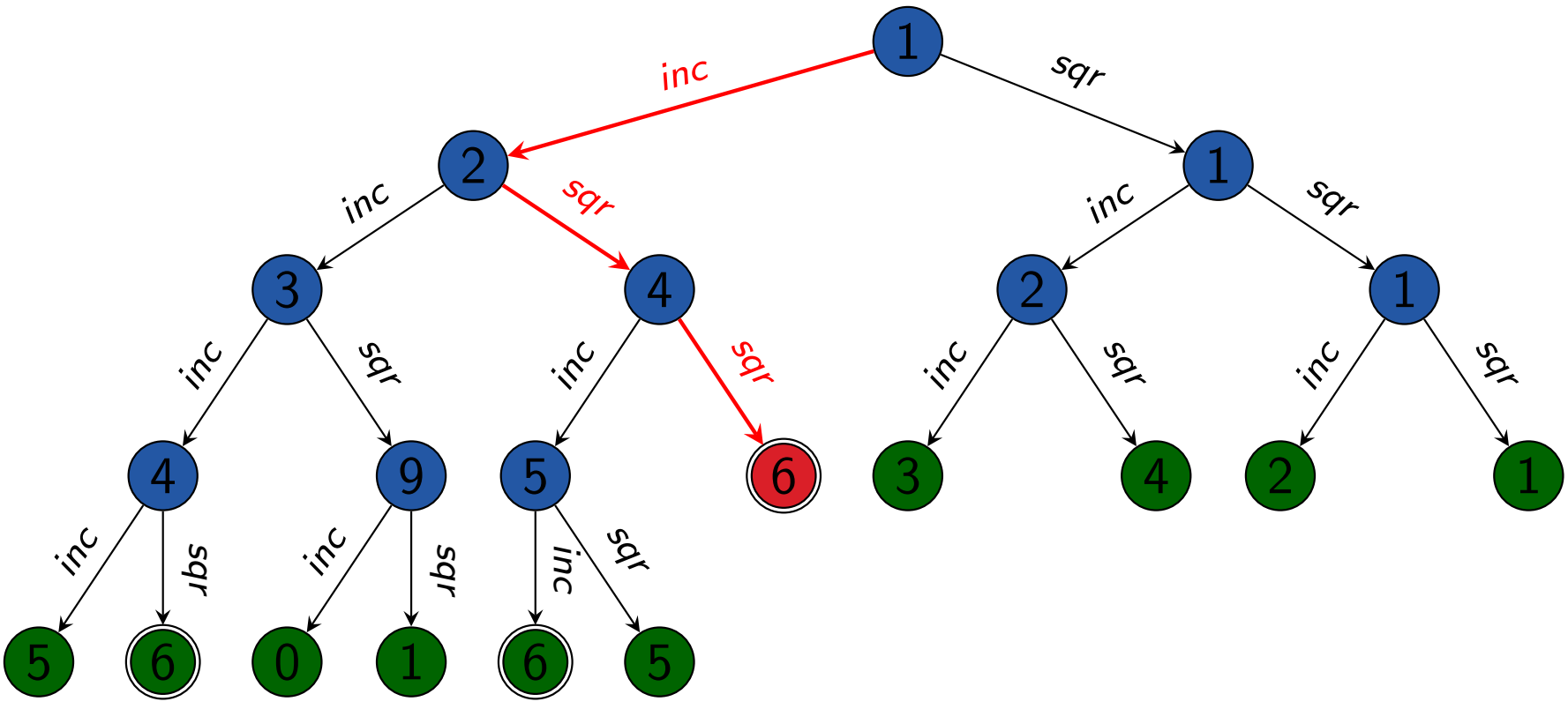
BFS-Tree (1st Attempt)

breadth-first search without duplicate elimination (1st attempt):

BFS-Tree (1st Attempt)

```
open := new Queue
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.push_back(n')
return unsolvable
```

Running Example: BFS-Tree (1st Attempt)



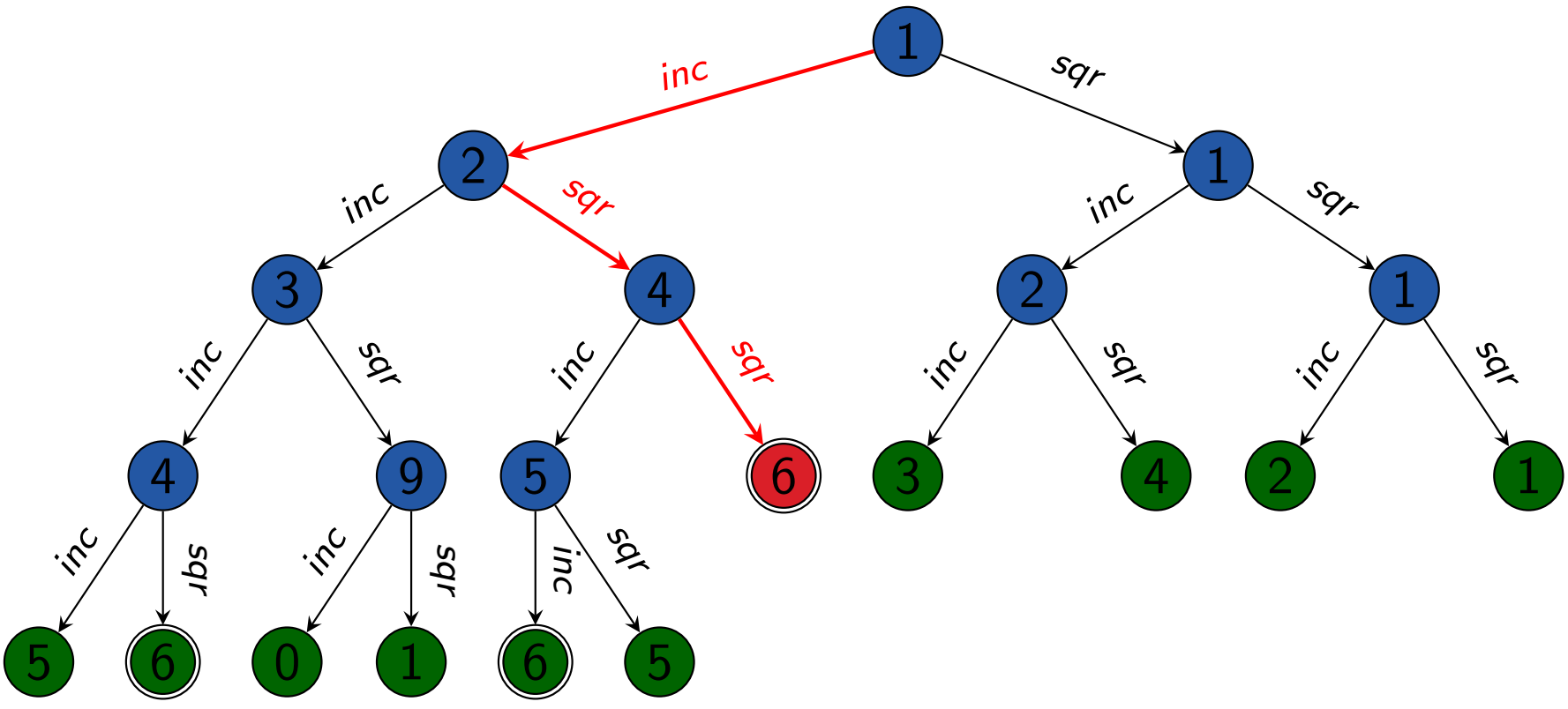
Opportunities for Improvement

- In a BFS, the first generated goal node is always the first expanded goal node. (Why?)

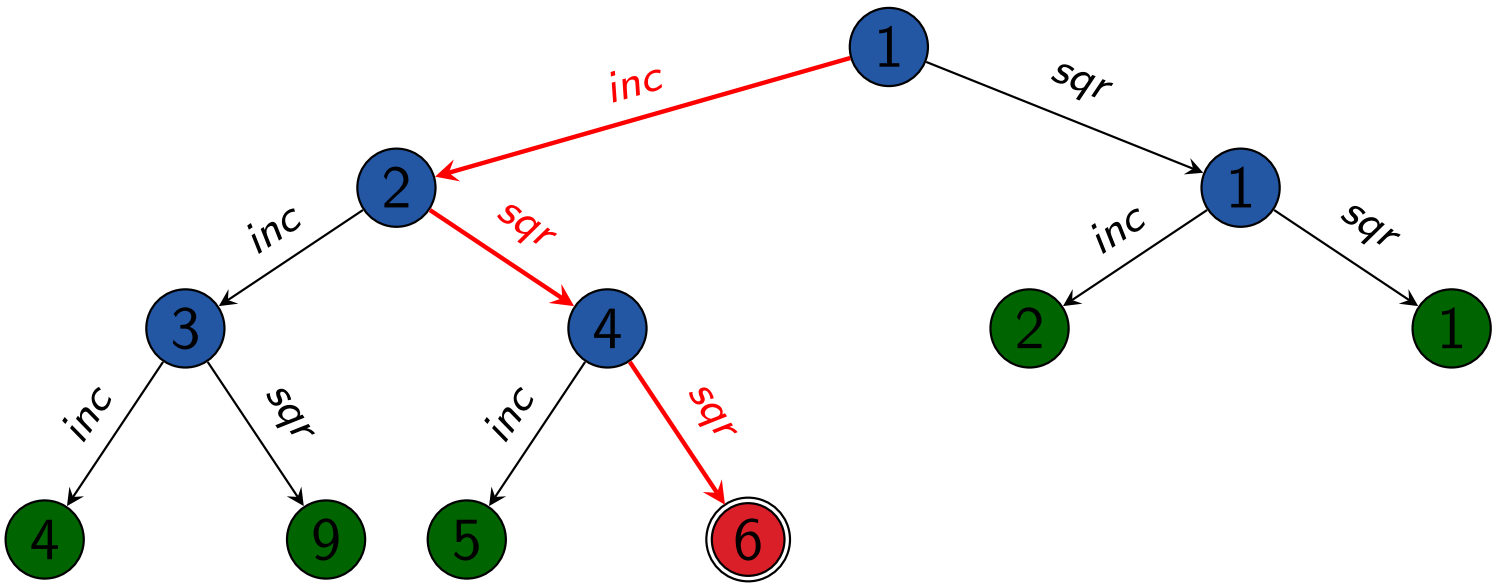
$\rightsquigarrow$ It is more efficient to perform the goal test upon generating a node (rather than upon expanding it).

$\rightsquigarrow$ How much effort does this save?

BFS-Tree without Early Goal Tests



BFS-Tree with Early Goal Tests



BFS-Tree (2nd Attempt)

breadth-first search without duplicate elimination (2nd attempt):

BFS-Tree (2nd Attempt)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (2nd Attempt)

breadth-first search without duplicate elimination (2nd attempt):

BFS-Tree (2nd Attempt)

```
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (2nd Attempt): Discussion

Where is the bug?

BFS-Tree (Final Version)

breadth-first search without duplicate elimination (final version):

BFS-Tree

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Tree (Final Version)

breadth-first search without duplicate elimination (final version):

BFS-Tree

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        open.push_back(n')
return unsolvable
```

BFS-Graph

Reminder: Generic Graph Search Algorithm

reminder from Chapter B5:

Generic Graph Search

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n := open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Adapting Generic Graph Search to Breadth-First Search

Adapting the generic algorithm to breadth-first search:

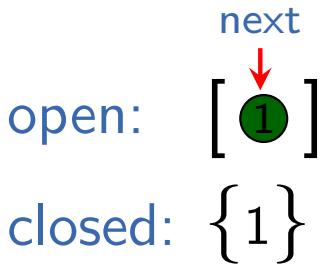
- similar adaptations to BFS-Tree
  (deque as open list, early goal tests)
- as closed list does not need to manage node information,
  a set data structure suffices
- for the same reasons why early goal tests are a good idea,
  we should perform duplicate tests against the closed list
  and updates of the closed lists as early as possible

BFS-Graph (Breadth-First Search with Duplicate Elim.)

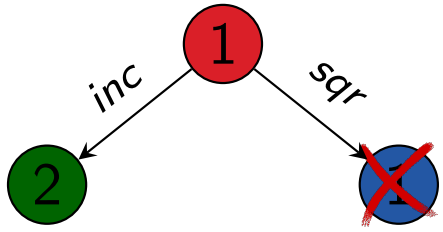
BFS-Graph

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
closed := new HashSet
closed.insert(init())
while not open.is_empty():
    n := open.pop_front()
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        if s' ∉ closed:
            closed.insert(s')
            open.push_back(n')
return unsolvable
```

BFS-Graph: Example



BFS-Graph: Example

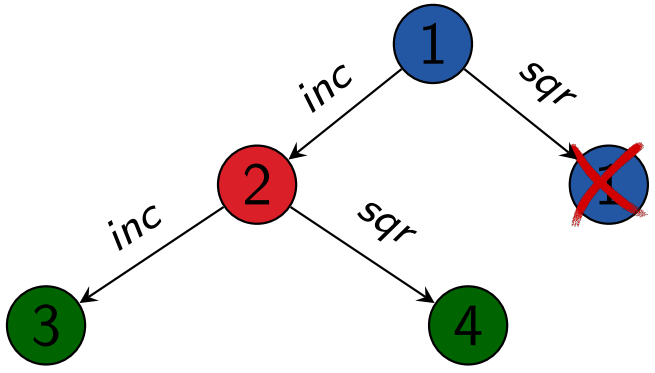


next

open: [2]

closed: {1,2}

BFS-Graph: Example

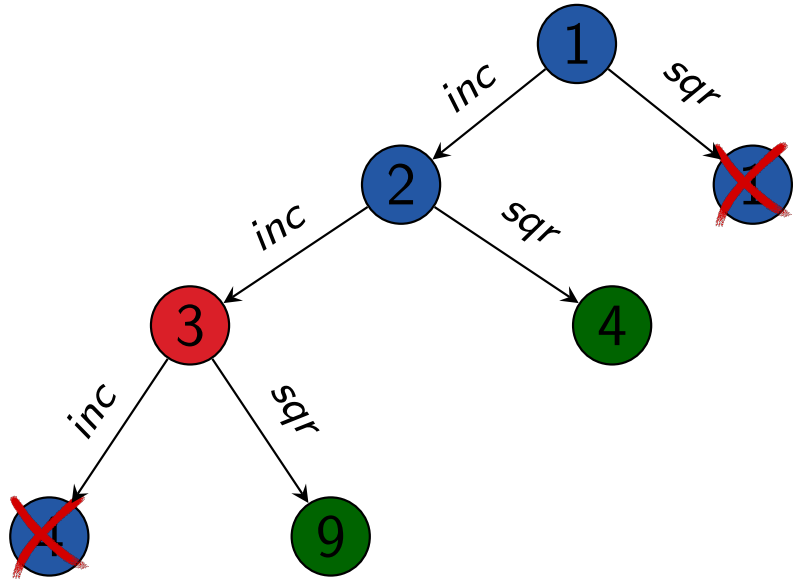


next

open: [3 4]

closed: {1, 2, 3, 4}

BFS-Graph: Example

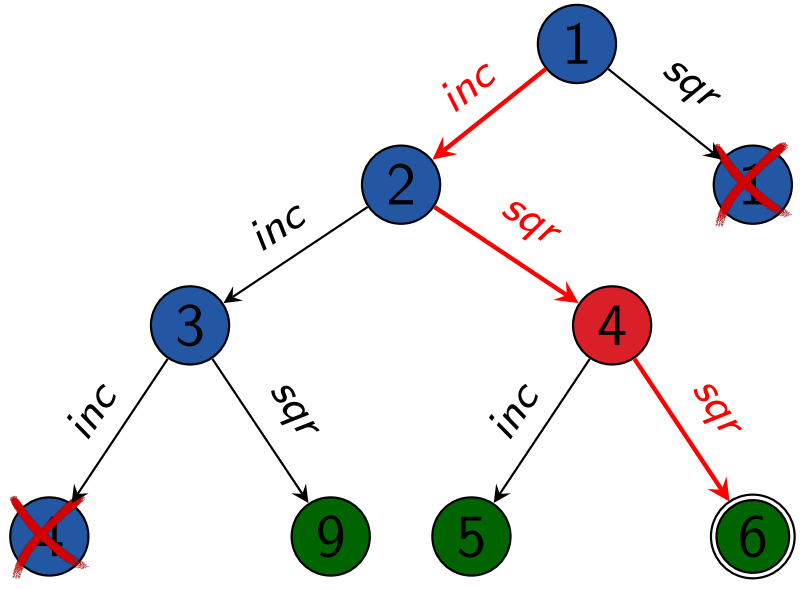


next

open: [4 9]

closed: {1, 2, 3, 4, 9}

BFS-Graph: Example



next

open: [9 5]

closed: {1, 2, 3, 4, 5, 9}

Properties of Breadth-first Search

Properties of Breadth-first Search

Properties of Breadth-first Search:

- BFS-Tree is semi-complete, but not complete. (Why?)
- BFS-Graph is complete. (Why?)
- BFS (both variants) is optimal
  if all actions have the same cost (Why?),
  but not in general (Why not?).
- complexity: next slides

Breadth-first Search: Complexity

The following result applies to both BFS variants:

Theorem (time complexity of breadth-first search)

Let b be the branching factor and d be the minimal solution length of the given state space. Let $b \geq 2$.

Then the time complexity of breadth-first search is

$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Reminder: we measure time complexity in generated nodes.

It follows that the space complexity of both BFS variants also is $O(b^d)$ (if $b \geq 2$). (Why?)

Breadth-first Search: Example of Complexity

example: $b = 13$; 100 000 nodes/second; 32 bytes/node

d	nodes	time	memory
4	30 940	0.3 s	966 KiB
6	$5.2 \cdot 10^6$	52 s	159 MiB
8	$8.8 \cdot 10^8$	147 min	26 GiB
10	10^{11}	17 days	4.3 TiB
12	10^{13}	8 years	734 TiB
14	10^{15}	1 352 years	121 PiB
16	10^{17}	$2.2 \cdot 10^5$ years	20 EiB
18	10^{20}	$38 \cdot 10^6$ years	3.3 ZiB

Breadth-first Search: Example of Complexity

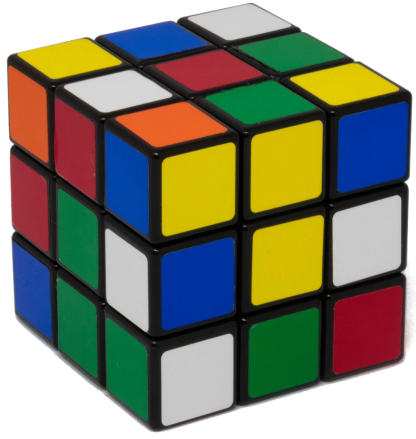
example: $b = 13$; 100 000 nodes/second; 32 bytes/node

Realistic numbers?

d	nodes	time	memory
4	30 940	0.3 s	966 KiB
6	$5.2 \cdot 10^6$	52 s	159 MiB
8	$8.8 \cdot 10^8$	147 min	26 GiB
10	10^{11}	17 days	4.3 TiB
12	10^{13}	8 years	734 TiB
14	10^{15}	1 352 years	121 PiB
16	10^{17}	$2.2 \cdot 10^5$ years	20 EiB
18	10^{20}	$38 \cdot 10^6$ years	3.3 ZiB

Breadth-first Search: Example of Complexity

example: $b = 13$; 100 000 nodes/second; 32 bytes/node



Rubik's cube:

- branching factor: ≈ 13
- typical solution length: 18

d	nodes	time	memory
4	30 940	0.3 s	966 KiB
6	$5.2 \cdot 10^6$	52 s	159 MiB
8	$8.8 \cdot 10^8$	147 min	26 GiB
10	10^{11}	17 days	4.3 TiB
12	10^{13}	8 years	734 TiB
14	10^{15}	1 352 years	121 PiB
16	10^{17}	$2.2 \cdot 10^5$ years	20 EiB
18	10^{20}	$38 \cdot 10^6$ years	3.3 ZiB

BFS-Tree or BFS-Graph?

Which is better, BFS-Tree or BFS-Graph?

BFS-Tree or BFS-Graph?

Which is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

BFS-Tree or BFS-Graph?

Which is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

advantages of BFS-Tree:

- simpler
- less overhead (time/space) if there are few duplicates

BFS-Tree or BFS-Graph?

Which is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

advantages of BFS-Tree:

- simpler
- less overhead (time/space) if there are few duplicates

Conclusion

BFS-Graph is usually preferable, unless we know that there is a negligible number of duplicates in the given state space.

Summary

Summary

- blind search algorithm: use no information
  except black box interface of state space
- breadth-first search: expand nodes in order of generation
  - search state space layer by layer
  - can be tree search or graph search
  - complexity $O(b^d)$ with branching factor b,
    minimal solution length d (if $b \geq 2$)
  - complete as a graph search; semi-complete as a tree search
  - optimal with uniform action costs

Foundations of Artificial Intelligence

B7. State-Space Search: Uniform Cost Search

Malte Helmert

University of Basel

March 5, 2025

State-Space Search: Overview

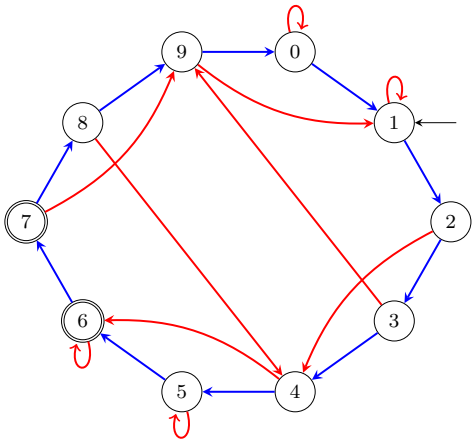
Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
  - B4. Data Structures for Search Algorithms
  - B5. Tree Search and Graph Search
  - B6. Breadth-first Search
  - B7. Uniform Cost Search
  - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Introduction

Uniform Cost Search

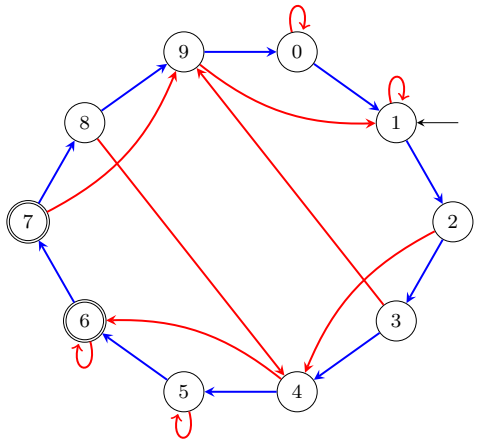
- breadth-first search optimal if all action costs equal
- otherwise no optimality guarantee $\rightsquigarrow$ example:



- consider bounded inc-and-square problem with $cost(inc) = 1$, $cost(sqr) = 3$
- solution of breadth-first search still $\langle inc, sqr, sqr \rangle$ (cost: 7)
- but: $\langle inc, inc, inc, inc, inc \rangle$ (cost: 5) is cheaper!

Uniform Cost Search

- breadth-first search optimal if all action costs equal
- otherwise no optimality guarantee $\rightsquigarrow$ example:



- consider bounded inc-and-square problem with $cost(inc) = 1$, $cost(sqr) = 3$
- solution of breadth-first search still $\langle inc, sqr, sqr \rangle$ (cost: 7)
- but: $\langle inc, inc, inc, inc, inc \rangle$ (cost: 5) is cheaper!

remedy: uniform cost search

- always expand a node with minimal path cost (n.path_cost a.k.a. $g(n)$)
- implementation: priority queue (min-heap) for open list

Algorithm

Reminder: Generic Graph Search Algorithm

reminder from Chapter B5:

Generic Graph Search

```
open := new OpenList
open.insert(make_root_node())
closed := new ClosedList
while not open.is_empty():
    n := open.pop()
    if closed.lookup(n.state) = none:
        closed.insert(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Uniform Cost Search

Uniform Cost Search

```
open := new MinHeap ordered by g
open.insert(make_root_node())
closed := new HashSet
while not open.is_empty():
    n := open.pop_min()
    if n.state ∉ closed:
        closed.insert(n.state)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Uniform Cost Search: Discussion

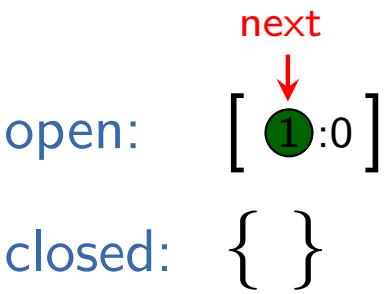
Adapting generic graph search to uniform cost search:

- here, early goal tests/early updates of the closed list not a good idea. (Why not?)
- as in BFS-Graph, a set is sufficient for the closed list
- a tree search variant is possible, but rare:
  has the same disadvantages as BFS-Tree
  and in general not even semi-complete (Why not?)

Remarks:

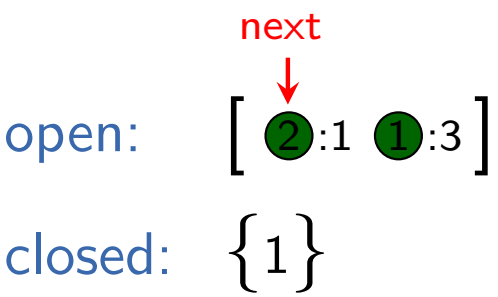
- identical to Dijkstra's algorithm for shortest paths
- for both: variants with/without delayed duplicate elimination

Example

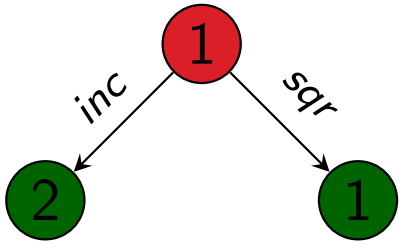


bounded inc-and-square variant: $\mathrm{cost}(sqr) = 3$

Example



bounded inc-and-square variant: $\mathrm{cost}(sqr) = 3$



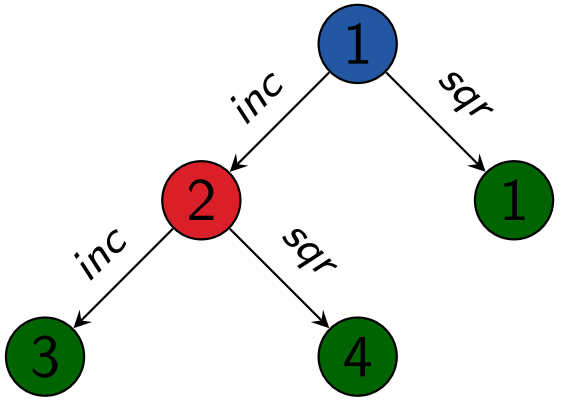
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [3:2 1:3 4:4]

closed: $\{1, 2\}$



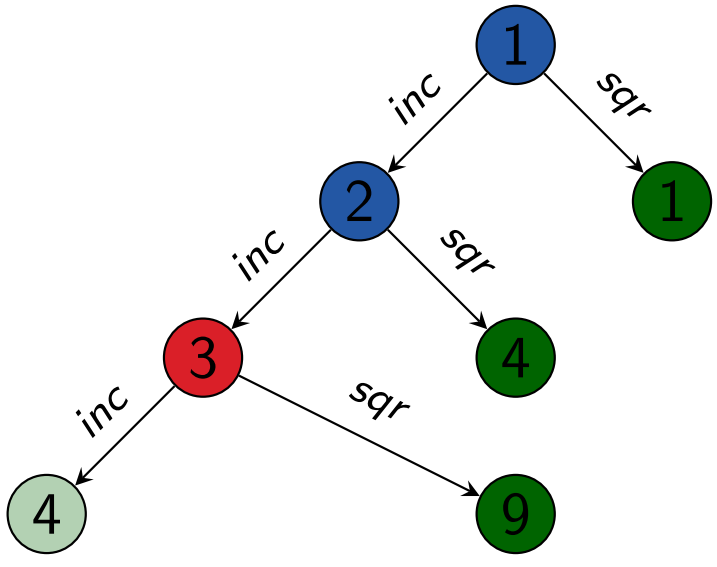
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [1:3 4:3 4:4 9:5]

closed: $\{1, 2, 3\}$



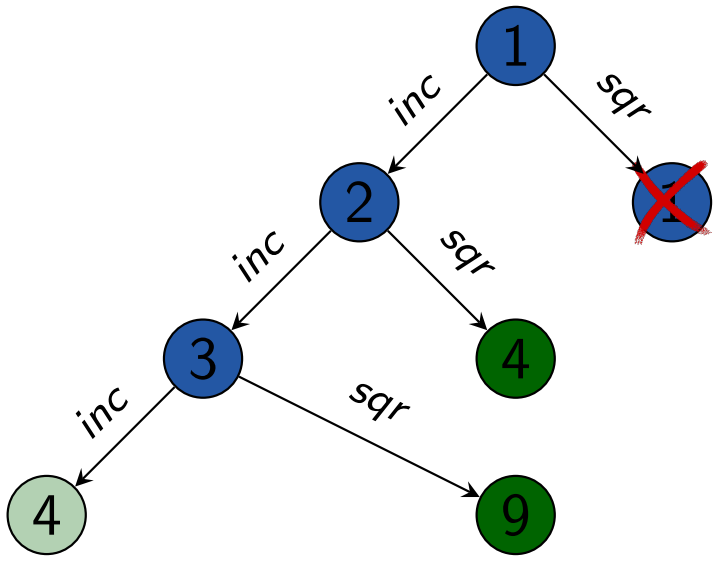
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [4:3 4:4 9:5]

closed: {1, 2, 3}



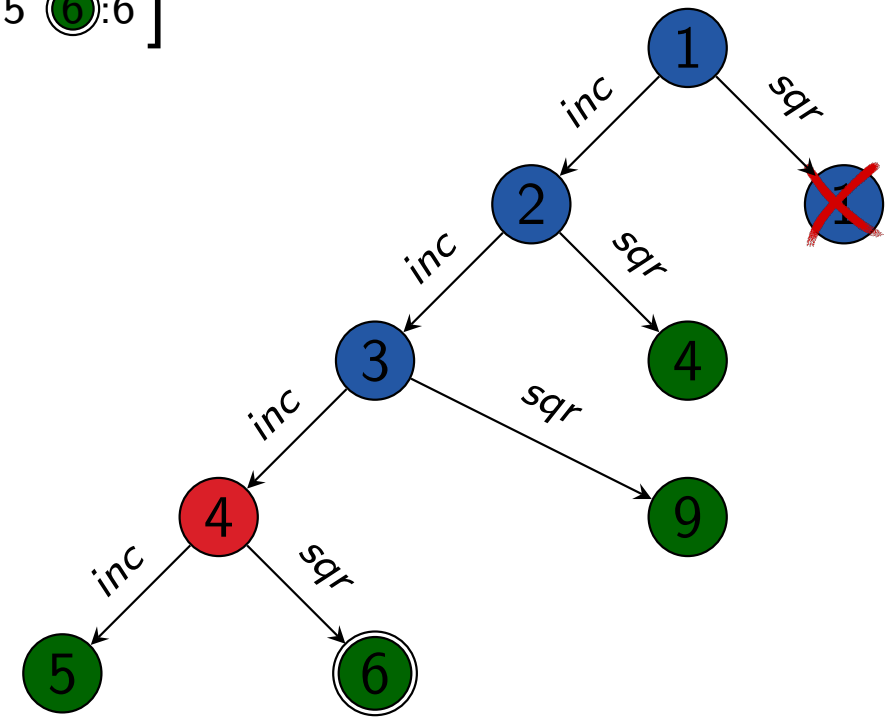
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [4:4 5:4 9:5 6:6]

closed: {1, 2, 3, 4}



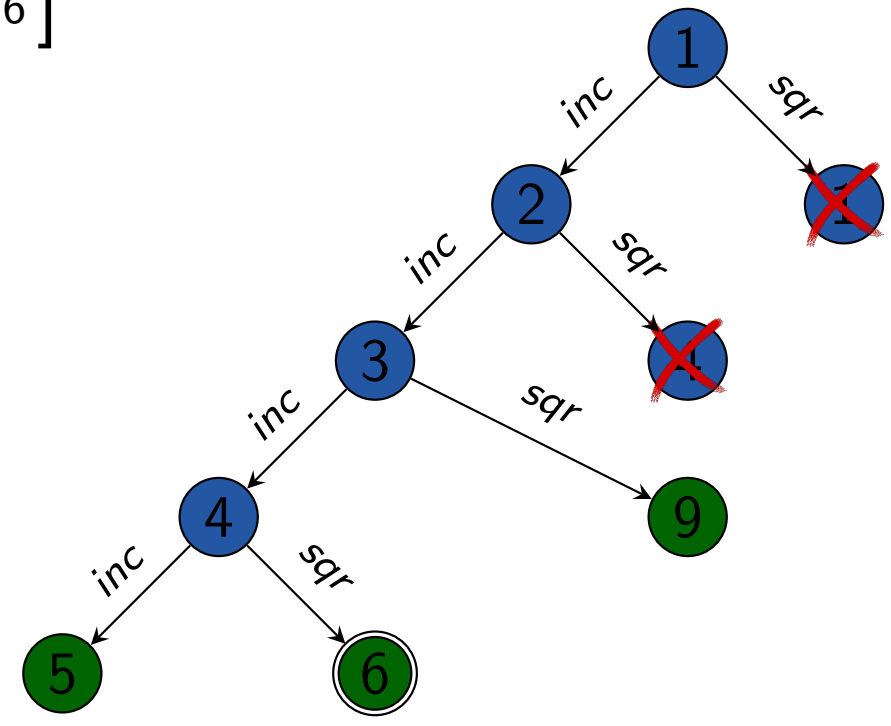
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [5:4 9:5 6:6]

closed: {1, 2, 3, 4}



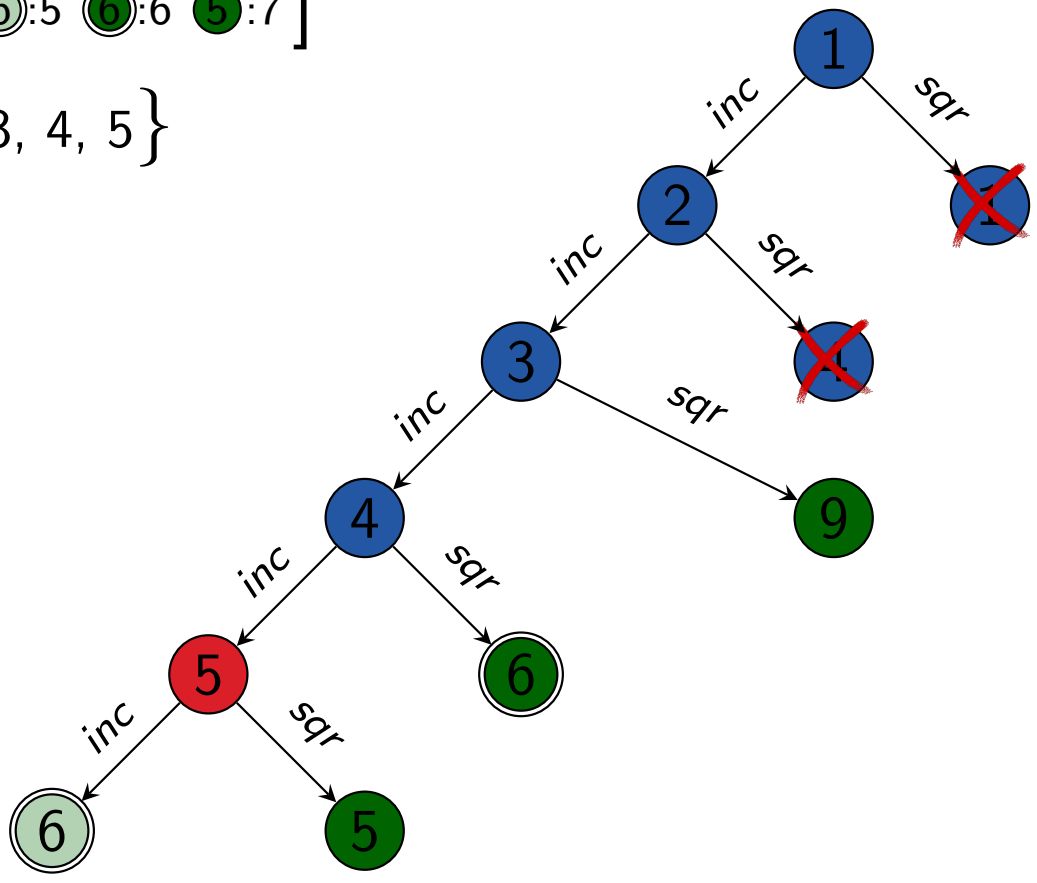
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [9:5 6:5 6:6 5:7]

closed: {1, 2, 3, 4, 5}



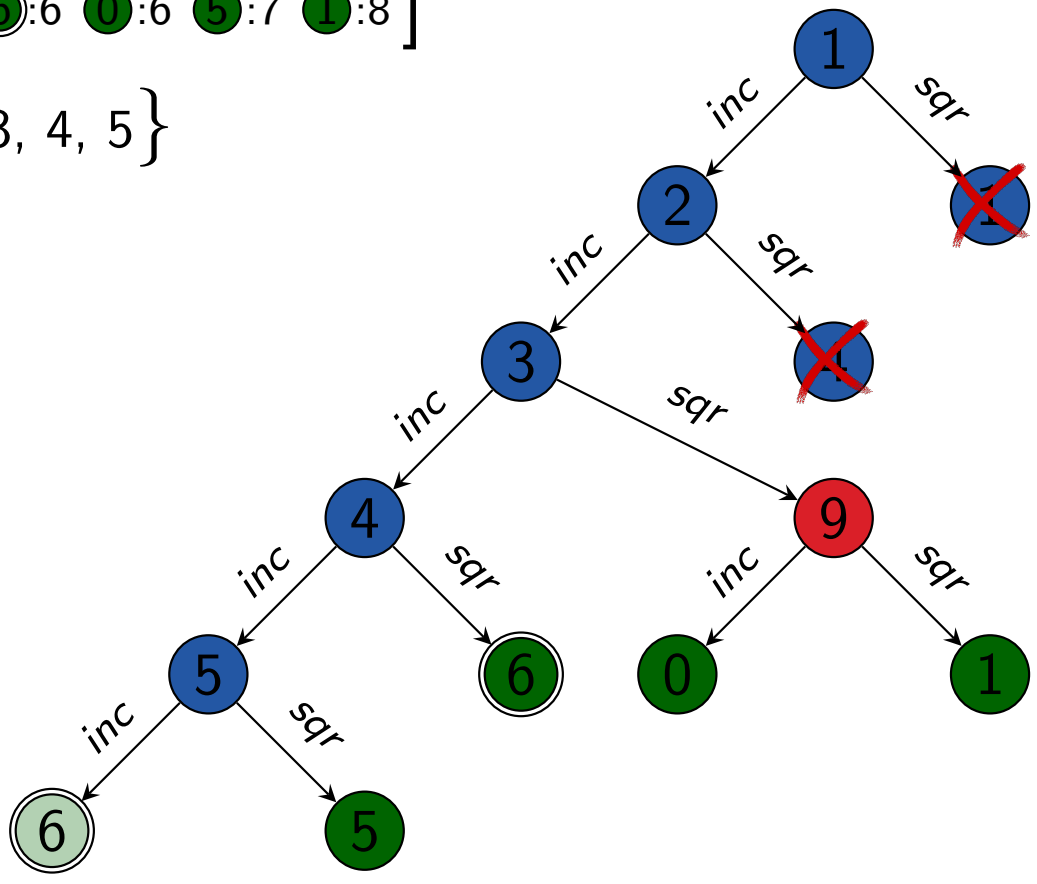
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [6:5 6:6 0:6 5:7 1:8]

closed: {1, 2, 3, 4, 5}



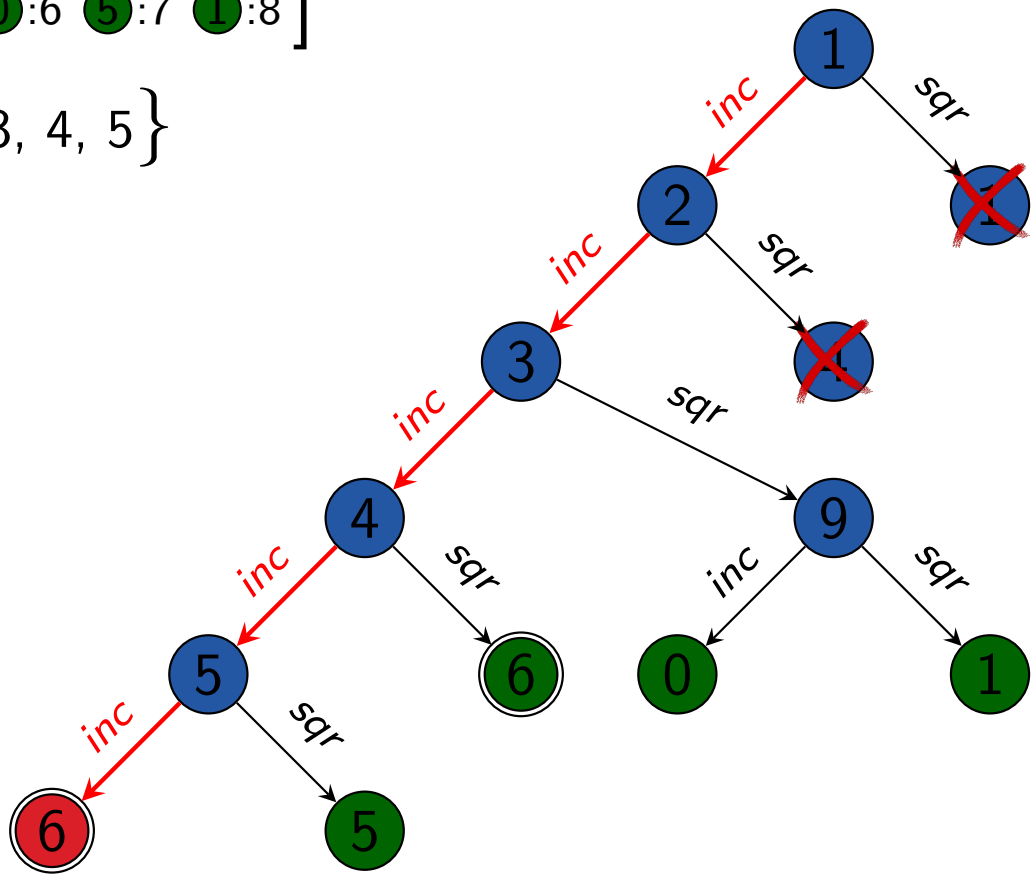
Example

bounded inc-and-square variant: $\text{cost}(sqr) = 3$

next

open: [6:6 0:6 5:7 1:8]

closed: {1, 2, 3, 4, 5}



Uniform Cost Search: Improvements

possible improvements:

- if action costs are small integers,
  bucket heaps often more efficient
- additional early duplicate tests for generated nodes
  can reduce memory requirements
  - can be beneficial or detrimental for runtime
  - must be careful to keep shorter path to duplicate state

Properties

Completeness and Optimality

properties of uniform cost search:

- uniform cost search is complete (Why?)
- uniform cost search is optimal (Why?)

Time and Space Complexity

properties of uniform cost search:

- Time complexity depends on distribution of action costs (no simple and accurate bounds).
  - Let $\varepsilon := \min_{a \in A} cost(a)$ and consider the case $\varepsilon > 0$.
  - Let c^* be the optimal solution cost.
  - Let b be the branching factor and consider the case $b \geq 2$.
  - Then the time complexity is at most $O(b^{\lfloor c^*/\varepsilon \rfloor + 1})$. (Why?)
  - often a very weak upper bound
- space complexity = time complexity

Summary

Summary

uniform cost search: expand nodes in order of ascending path costs

- usually as a graph search
- then corresponds to Dijkstra's algorithm
- complete and optimal

Foundations of Artificial Intelligence

B8. State-Space Search: Depth-first Search & Iterative Deepening

Malte Helmert

University of Basel

March 17, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
  - B4. Data Structures for Search Algorithms
  - B5. Tree Search and Graph Search
  - B6. Breadth-first Search
  - B7. Uniform Cost Search
  - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Depth-first Search

Idea of Depth-first Search

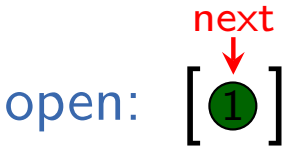
depth-first search:

- expands nodes in opposite order of generation (LIFO)
- open list implemented as stack

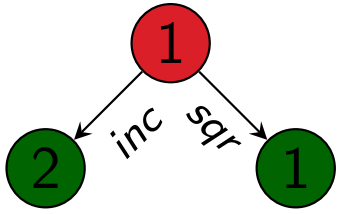
$\rightsquigarrow$ deepest node expanded first

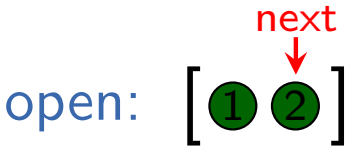
German: Tiefensuche

Depth-first Search Example

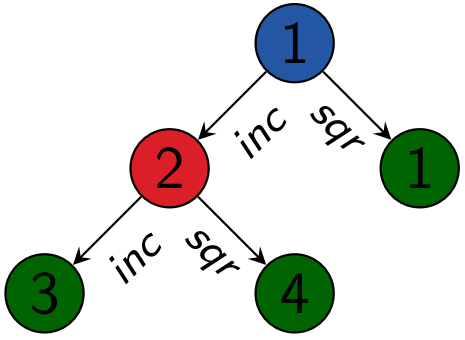


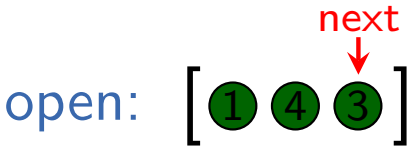
Depth-first Search Example



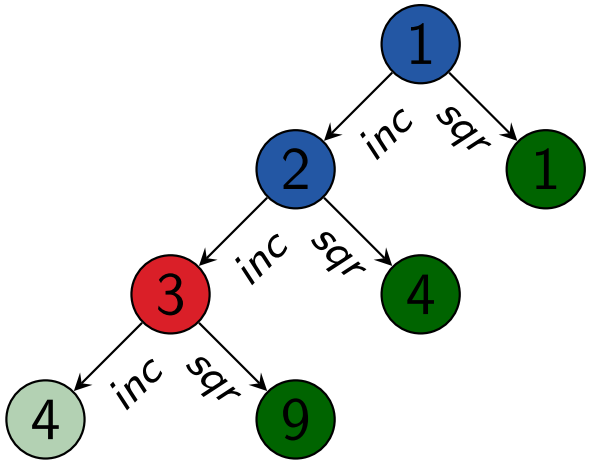


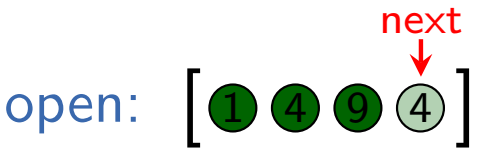
Depth-first Search Example



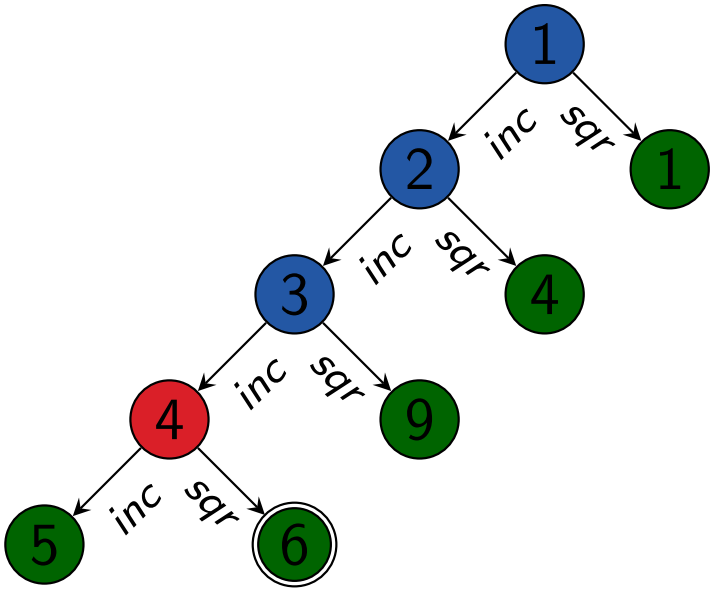


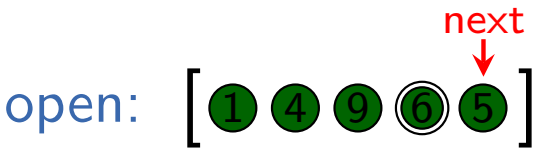
Depth-first Search Example



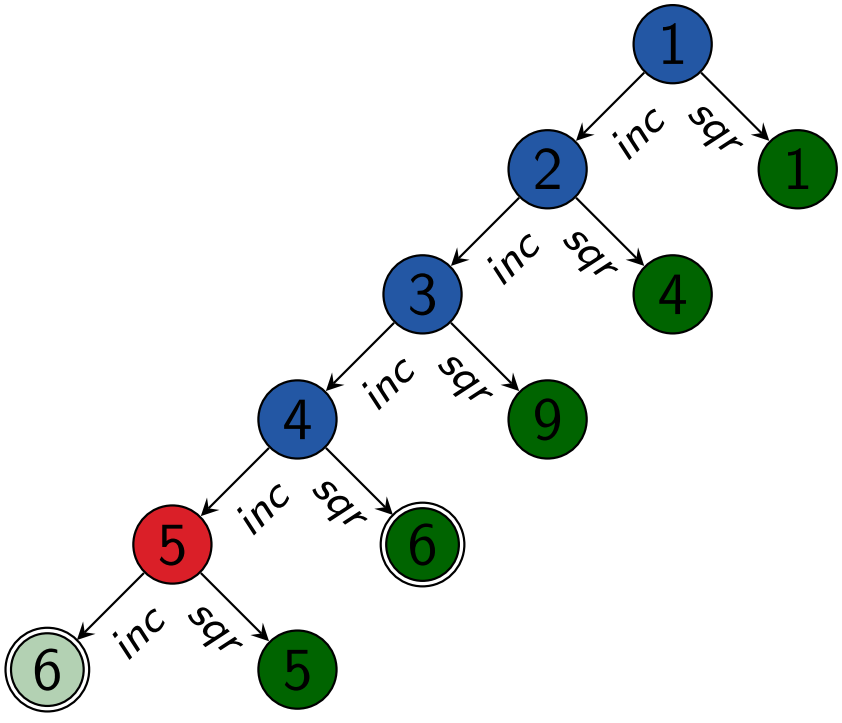


Depth-first Search Example

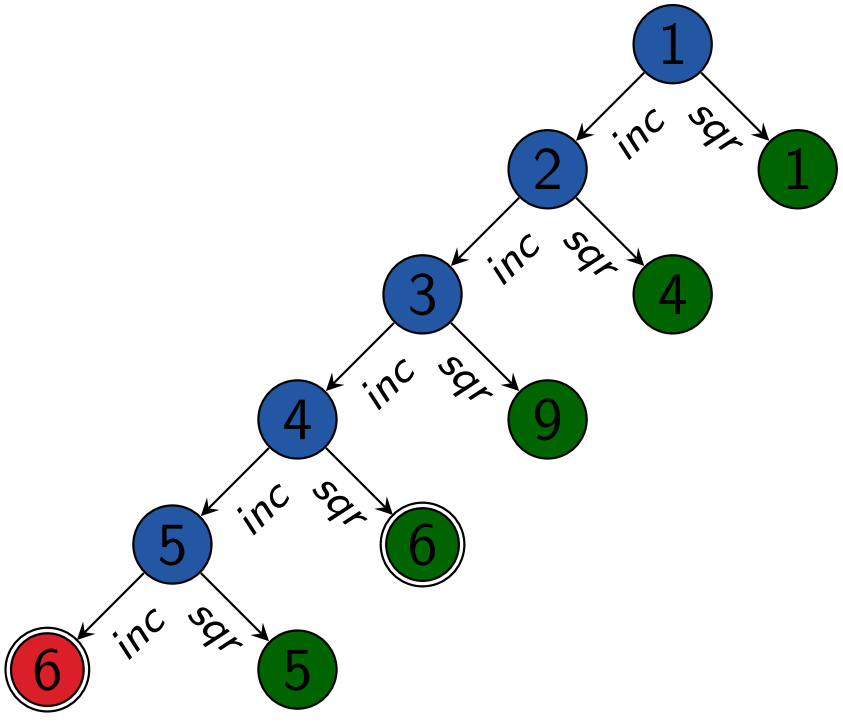




Depth-first Search Example

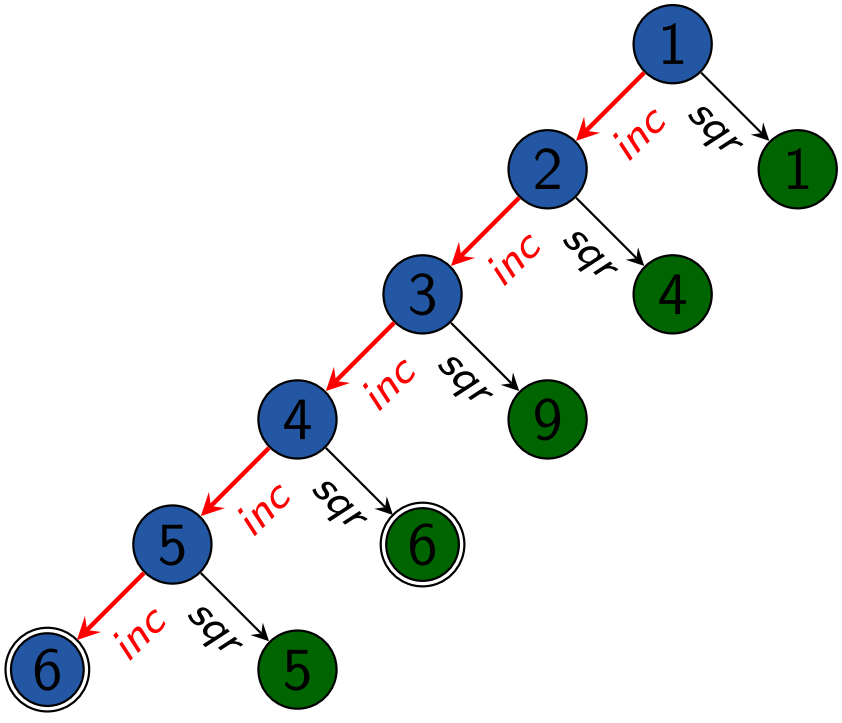


Depth-first Search Example



open: [1 4 9 6 5]

Depth-first Search Example



open: [1 4 9 6 5]

Depth-first Search: Some Properties

- almost always implemented as a tree search (we will see why)
- not complete, not semi-complete, not optimal (Why?)
- complete for acyclic state spaces,
  e.g., if state space directed tree

Reminder: Generic Tree Search Algorithm

reminder from Chapter B5:

Generic Tree Search

```
open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n := open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

Depth-first Search (Non-recursive Version)

depth-first search (non-recursive version):

Depth-first Search (Non-recursive Version)

```
open := new Stack
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_back()
    if is_goal(n.state):
        return extract_path(n)
    for each ⟨a, s'⟩ ∈ succ(n.state):
        n' := make_node(n, a, s')
        open.push_back(n')
return unsolvable
```

Non-recursive Depth-first Search: Discussion

discussion:

- there isn't much wrong with this pseudo-code
  (as long as we ensure to release nodes that are no longer required when using programming languages without garbage collection)
- however, depth-first search as a recursive algorithm is simpler and more efficient

$\rightsquigarrow$ CPU stack as implicit open list

$\rightsquigarrow$ no search node data structure needed

Depth-first Search (Recursive Version)

function depth_first_search(s)

if is_goal(s):
    return $\langle\rangle$
for each $\langle a, s' \rangle \in$ succ(s):
    solution := depth_first_search(s')
    if *solution* $\neq$ **none**:
        solution.push_front(a)
        return *solution*
return none

main function:

Depth-first Search (Recursive Version)

return depth_first_search(init())

Depth-first Search: Complexity

time complexity:

- If the state space includes paths of length m, depth-first search can generate $O(b^m)$ nodes, even if much shorter solutions (e.g., of length 1) exist.
- On the other hand: in the best case, solutions of length ℓ can be found with $O(b\ell)$ generated nodes. (Why?)
- improvable to $O(\ell)$ with incremental successor generation

Depth-first Search: Complexity

time complexity:

- If the state space includes paths of length m, depth-first search can generate $O(b^m)$ nodes, even if much shorter solutions (e.g., of length 1) exist.
- On the other hand: in the best case, solutions of length ℓ can be found with $O(b\ell)$ generated nodes. (Why?)
- improvable to $O(\ell)$ with incremental successor generation

space complexity:

- only need to store nodes along currently explored path ("along": nodes on path and their children)

$\rightsquigarrow$ space complexity $O(bm)$ if m maximal search depth reached

- low memory complexity main reason why depth-first search interesting despite its disadvantages

Iterative Deepening

Idea of Depth-limited Search

depth-limited search:

- parameterized with depth limit $\ell \in \mathbb{N}_0$
- behaves like depth-first search, but prunes (does not expand) search nodes at depth ℓ
- not very useful on its own, but important ingredient of more useful algorithms

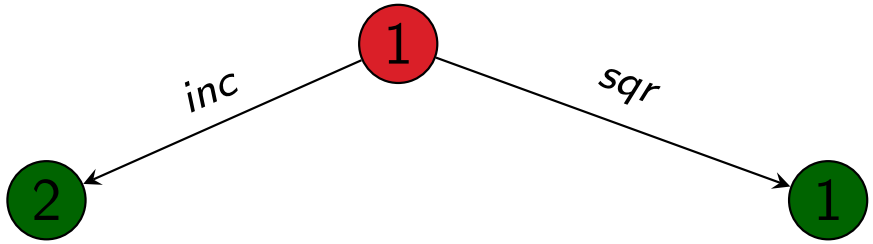
German: tiefenbeschränkte Suche

Depth-limited Search Example

Consider depth limit $\ell = 2$.

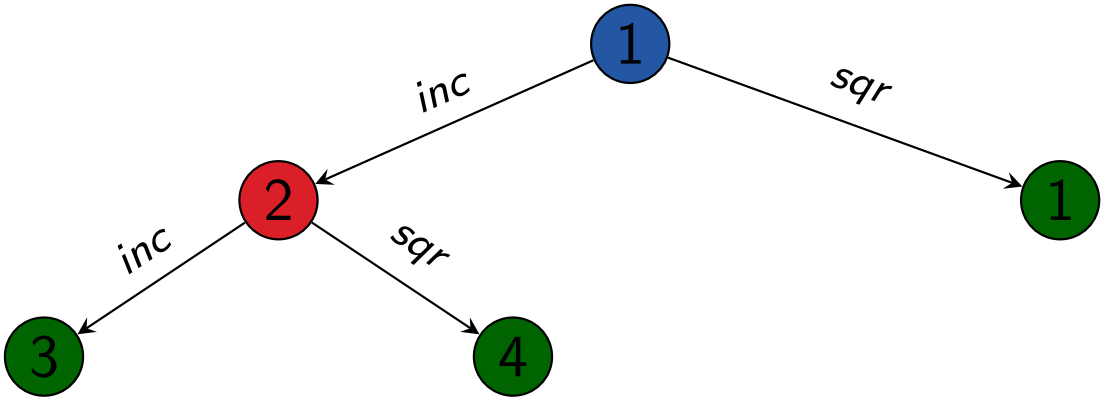
Depth-limited Search Example

Consider depth limit $\ell = 2$.



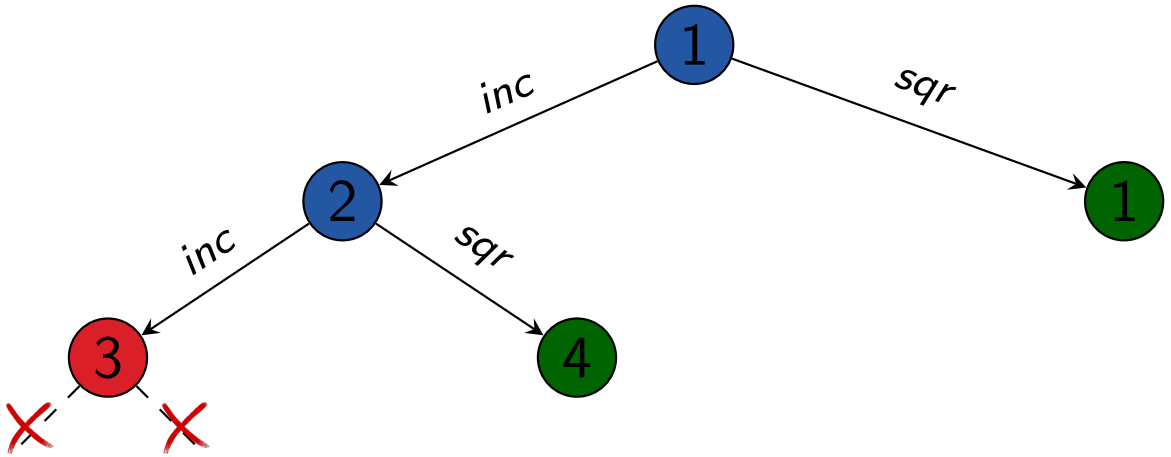
Depth-limited Search Example

Consider depth limit $\ell = 2$.



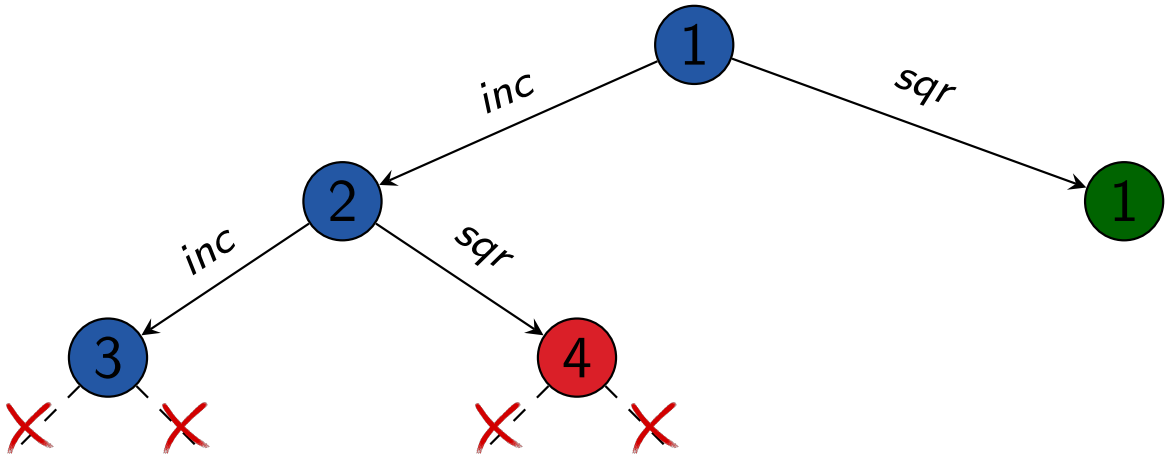
Depth-limited Search Example

Consider depth limit $\ell = 2$.



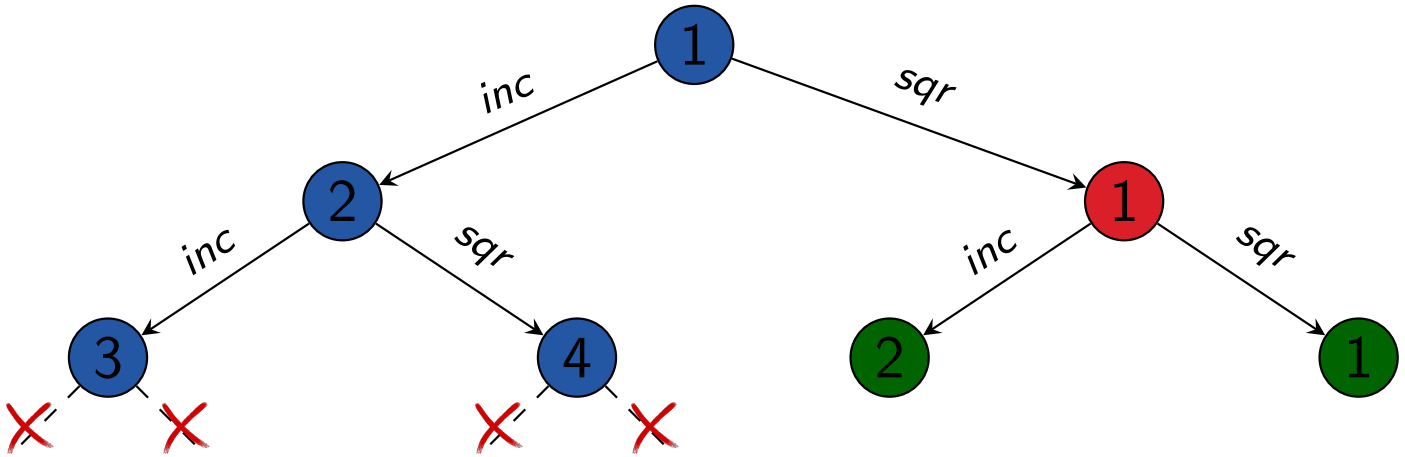
Depth-limited Search Example

Consider depth limit $\ell = 2$.



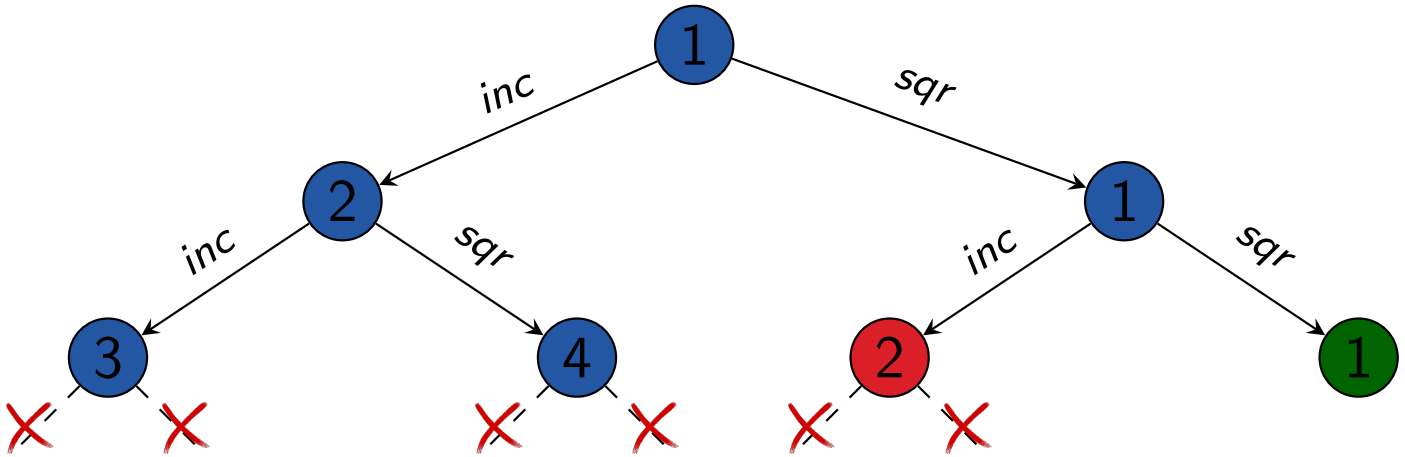
Depth-limited Search Example

Consider depth limit $\ell = 2$.



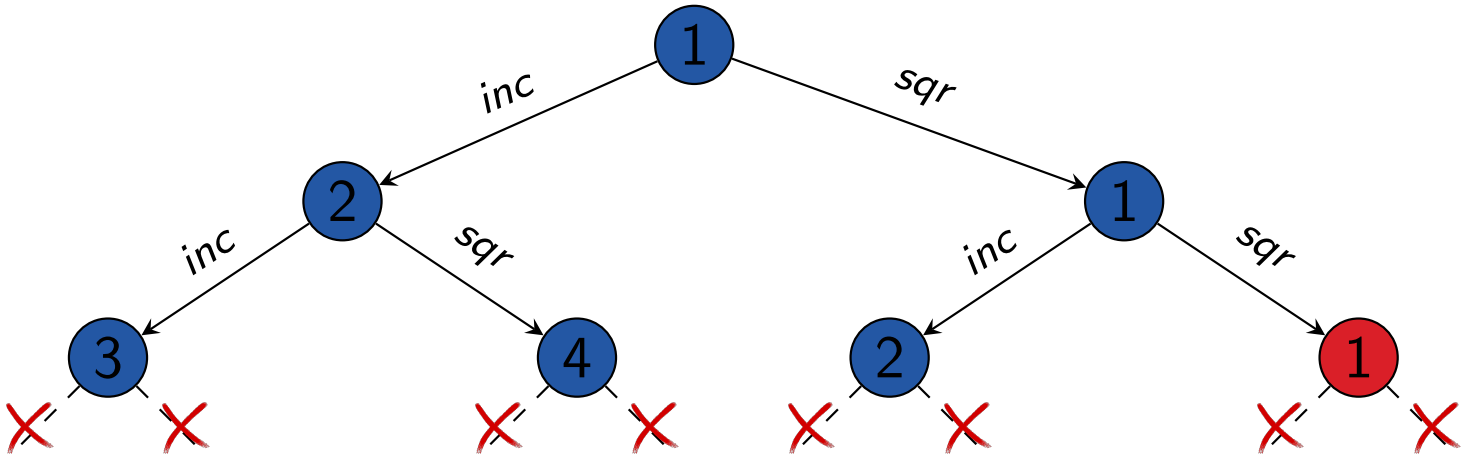
Depth-limited Search Example

Consider depth limit $\ell = 2$.



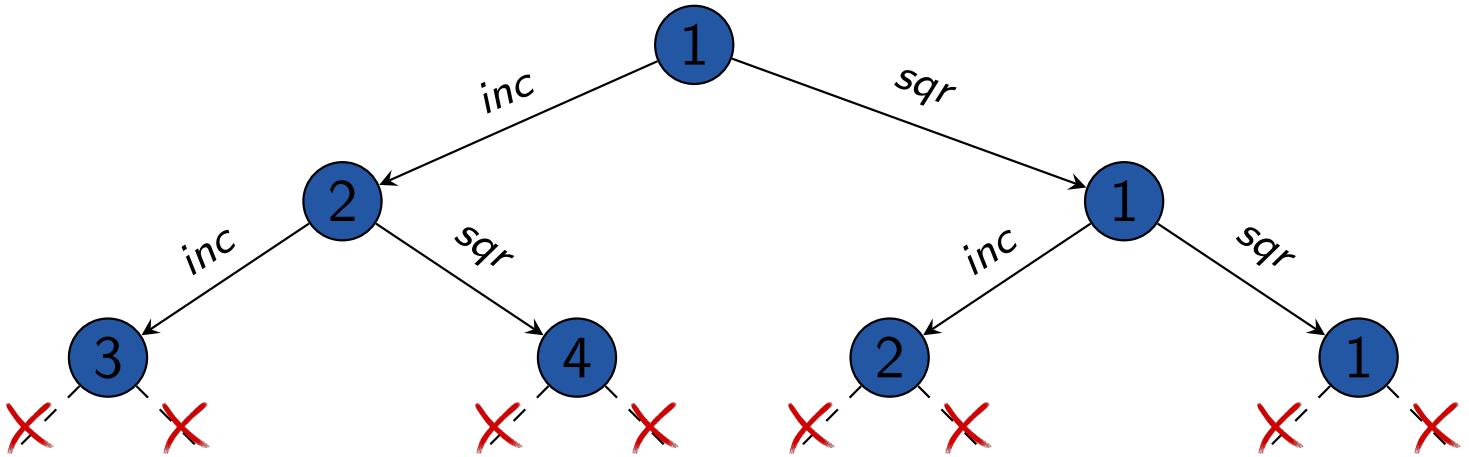
Depth-limited Search Example

Consider depth limit $\ell = 2$.



Depth-limited Search Example

Consider depth limit $\ell = 2$.



Depth-limited Search: Pseudo-Code

function depth_limited_search(s, *depth_limit*):

```
if is_goal(s):
    return ⟨⟩
if depth_limit > 0:
    for each ⟨a, s'⟩ ∈ succ(s):
        solution := depth_limited_search(s', depth_limit − 1)
        if solution ≠ none:
            solution.push_front(a)
            return solution
return none
```

Iterative Deepening Depth-first Search

iterative deepening depth-first search (iterative deepening DFS):

- idea: perform a sequence of depth-limited searches with increasing depth limit
- sounds wasteful (each iteration repeats all the useful work of all previous iterations)
- in fact overhead acceptable ($\leadsto$ analysis follows)

Iterative Deepening DFS

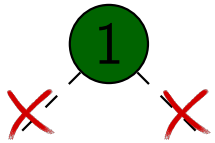
for *depth_limit* $\in \{0, 1, 2, \dots\}$:
    solution := depth_limited_search(init(), *depth_limit*)
    if *solution* $\neq$ **none**:
        return *solution*

German: iterative Tiefensuche

Example

depth limit: 0

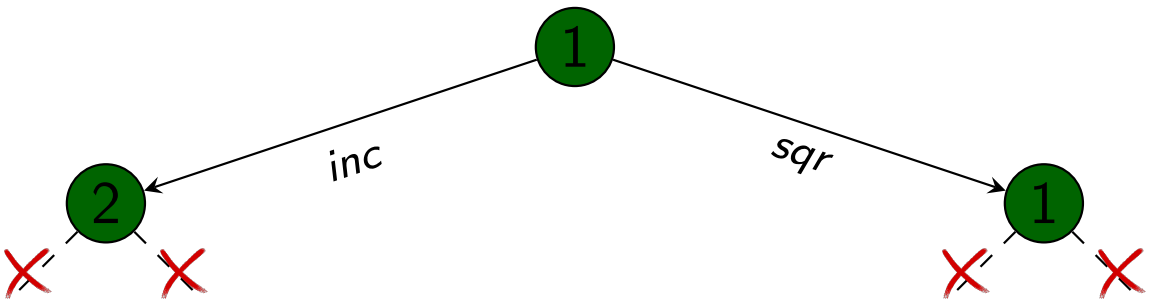
generated nodes: 1



Example

depth limit: 1

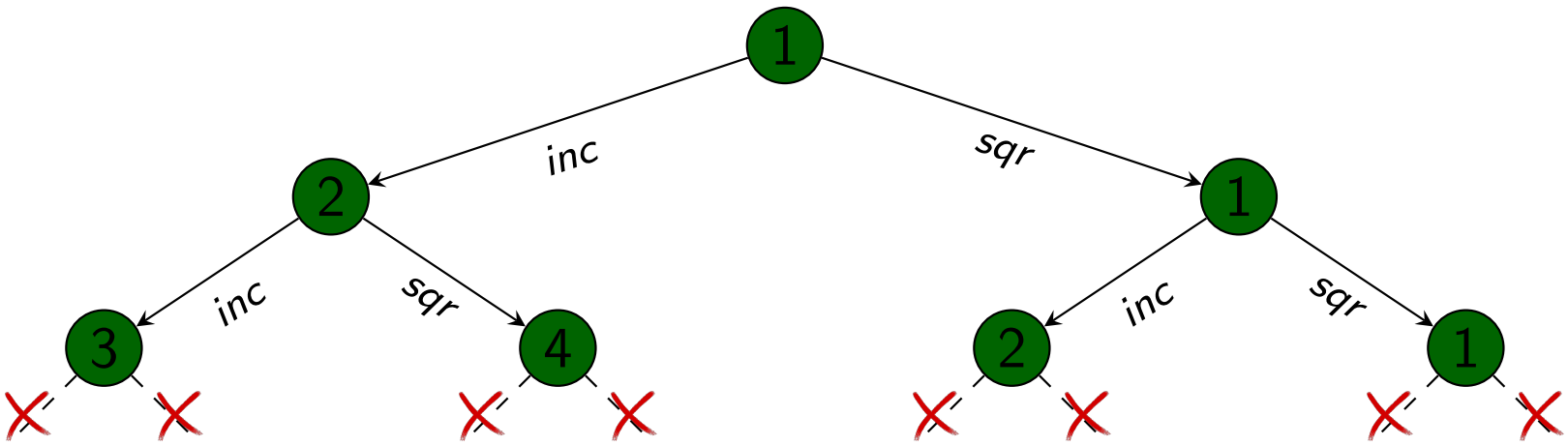
generated nodes: 1+3



Example

depth limit: 2

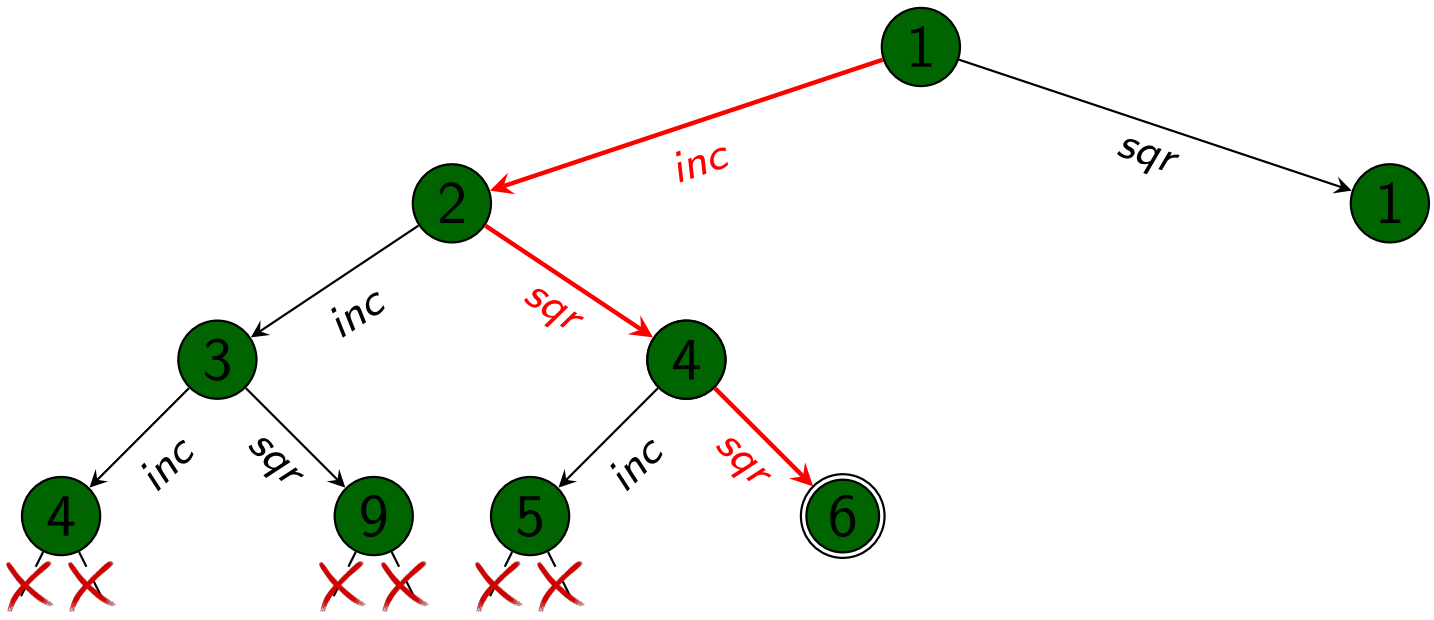
generated nodes: 1+3+7



Example

depth limit: 3

generated nodes: 1+3+7+9=20



Iterative Deepening DFS: Properties

combines advantages of breadth-first and depth-first search:

- (almost) like BFS: semi-complete (however, not complete)
- like BFS: optimal if all actions have same cost
- like DFS: only need to store nodes along one path
  $\rightsquigarrow$ space complexity $O(bd)$, where d minimal solution length
- time complexity only slightly higher than BFS
  ($\rightsquigarrow$ analysis soon)

Iterative Deepening DFS: Complexity Example

time complexity (generated nodes):

breadth-first search	$1 + b + b^2 + \cdots + b^{d-1} + b^d$
iterative deepening DFS	$(d+1) + db + (d-1)b^2 + \cdots + 2b^{d-1} + 1b^d$

example: $b = 10,\ d = 5$

breadth-first search	$1 + 10 + 100 + 1000 + 10000 + 100000 = 111111$
iterative deepening DFS	$6 + 50 + 400 + 3000 + 20000 + 100000 = 123456$

for $b = 10$, only 11% more nodes than breadth-first search

Iterative Deepening DFS: Time Complexity

Theorem (time complextive of iterative deepening DFS)

Let b be the branching factor and d be the minimal solution length of the given state space. Let $b \geq 2$.

Then the time complexity of iterative deepening DFS is

$$(d+1) + db + (d-1)b^2 + (d-2)b^3 + \cdots + 1b^d = O(b^d)$$

and the memory complexity is

$$O(bd).$$

Iterative Deepening DFS: Evaluation

Iterative Deepening DFS: Evaluation

Iterative Deepening DFS is often the method of choice if

- tree search is adequate (no duplicate elimination necessary),
- all action costs are identical, and
- the solution depth is unknown.

Summary

Summary

depth-first search: expand nodes in LIFO order

- usually as a tree search
- easy to implement recursively
- very memory-efficient
- can be combined with iterative deepening
  to combine many of the good aspects
  of breadth-first and depth-first search

Comparison of Blind Search Algorithms

completeness, optimality, time and space complexity

	search algorithm				
criterion	breadth-first	uniform cost	depth-first	depth-limited	iterative deepening
complete?	yes*	yes	no	no	semi
optimal?	yes**	yes	no	no	yes**
time	$O(b^d)$	$O(b^{\lfloor c^*/\varepsilon \rfloor + 1})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
space	$O(b^d)$	$O(b^{\lfloor c^*/\varepsilon \rfloor + 1})$	$O(bm)$	$O(b\ell)$	$O(bd)$

$b \geq 2$ branching factor
d minimal solution depth
m maximal search depth
ℓ depth limit
c^* optimal solution cost
$\varepsilon > 0$ minimal action cost

remarks:
* for BFS-Tree: semi-complete
** only with uniform action costs

Foundations of Artificial Intelligence

B9. State-Space Search: Heuristics

Malte Helmert

University of Basel

March 17, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Introduction

Informed Search Algorithms

search algorithms considered so far:

- uninformed ("blind"): use no information besides formal definition to solve a problem
- scale poorly: prohibitive time (and space) requirements for seemingly simple problems (time complexity usually $O(b^d)$)

Informed Search Algorithms

search algorithms considered so far:

- uninformed ("blind"): use no information besides formal definition to solve a problem
- scale poorly: prohibitive time (and space) requirements for seemingly simple problems (time complexity usually $O(b^d)$)

example: $b = 13$; 10^5 nodes/second

d	nodes	time
4	30 940	0.3 s
6	$5.2 \cdot 10^6$	52 s
8	$8.8 \cdot 10^8$	147 min
10	10^{11}	17 days
12	10^{13}	8 years
14	10^{15}	1 352 years
16	10^{17}	$2.2 \cdot 10^5$ years
18	10^{20}	$38 \cdot 10^6$ years

Informed Search Algorithms

Rubik's cube:

- branching factor: ≈ 13
- typical solution length: 18

example: $b = 13$; 10^5 nodes/second

d	nodes	time
4	30 940	0.3 s
6	$5.2 \cdot 10^6$	52 s
8	$8.8 \cdot 10^8$	147 min
10	10^{11}	17 days
12	10^{13}	8 years
14	10^{15}	1 352 years
16	10^{17}	$2.2 \cdot 10^5$ years
18	10^{20}	$38 \cdot 10^6$ years

Richard Korf, Finding Optimal Solutions to Rubik's Cube Using Pattern Databases (AAAI, 1997)

Informed Search Algorithms

Rubik's cube:

search algorithms considered now:

- idea: try to find (problem-specific) criteria to distinguish good and bad states
- heuristic ("informed") search algorithms prefer good states

- branching factor: ≈ 13
- typical solution length: 18

Richard Korf, Finding Optimal Solutions to Rubik's Cube Using Pattern Databases (AAAI, 1997)

Heuristics

Heuristics

Definition (heuristic)

Let $\mathcal{S}$ be a state space with states S.
A heuristic function or heuristic for $\mathcal{S}$ is a function

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\},$$

mapping each state to a nonnegative number (or ∞).

Heuristics: Intuition

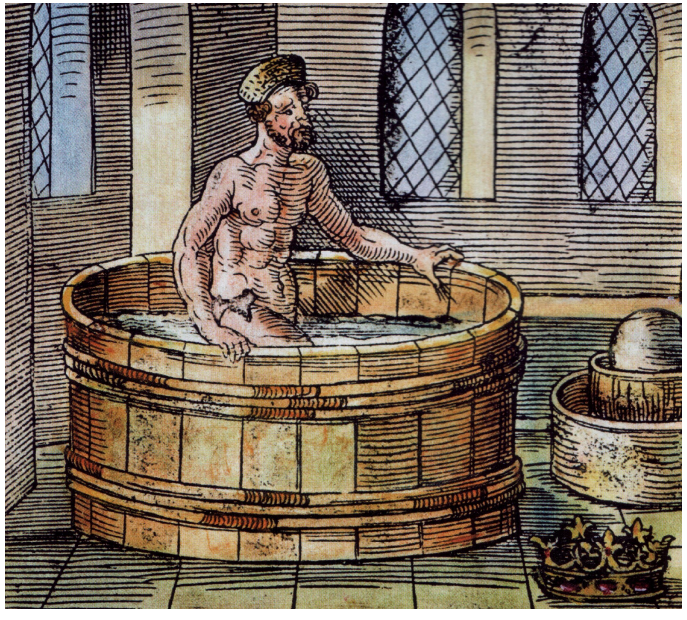
idea: $h(s)$ estimates distance (= cost of cheapest path) from s to closest goal state

- heuristics can be arbitrary functions
- intuition:
  1. the closer h is to true goal distance, the more efficient the search using h
  2. the better h separates states that are close to the goal from states that are far, the more efficient the search using h

Why "Heuristic"?

What does "heuristic" mean?

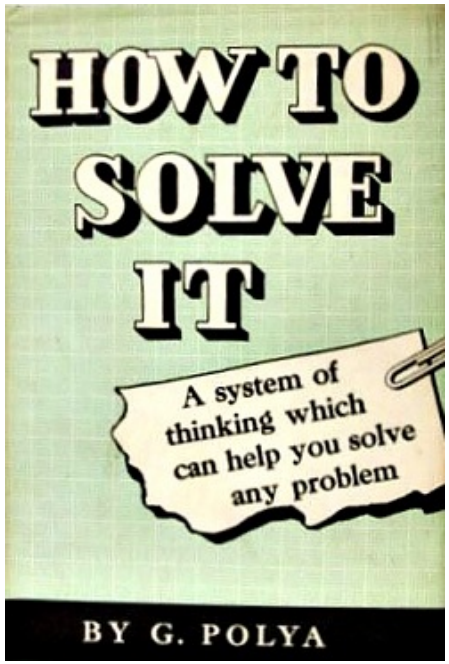
- from ancient Greek ἑυρισκω (= I find)
- same origin as ἑυρηκα!



Why "Heuristic"?

What does "heuristic" mean?

- from ancient Greek ἑυρισκω (= I find)
- same origin as ἑυρηκα!
- popularized by George Pólya: How to Solve It (1945)
- in computer science often used for: rule of thumb, inexact algorithm
- in state-space search technical term for goal distance estimator



Representation of Heuristics

In our black box model, heuristics are an additional element of the state space interface:

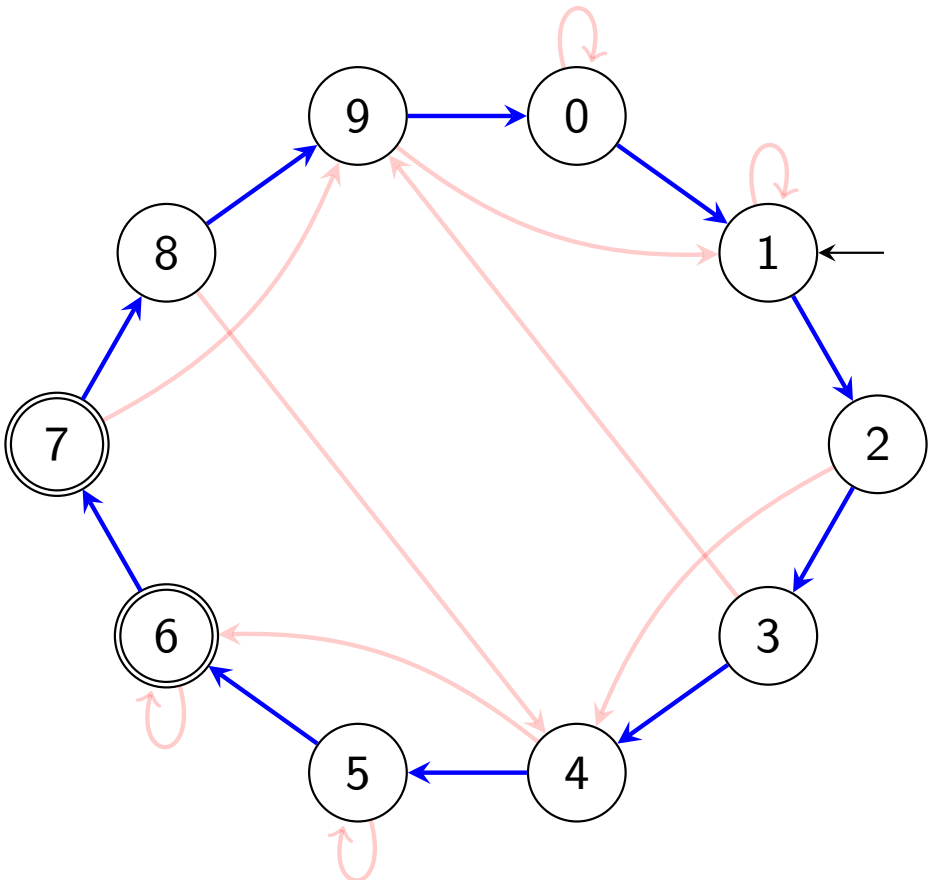
State Spaces as Black Boxes (Extended)

- init()
- is_goal(s)
- succ(s)
- cost(a)
- h(s): heuristic value for state s
  result: nonnegative integer or ∞

Examples

Bounded Inc-and-Square

bounded inc-and-square:



possible heuristics:

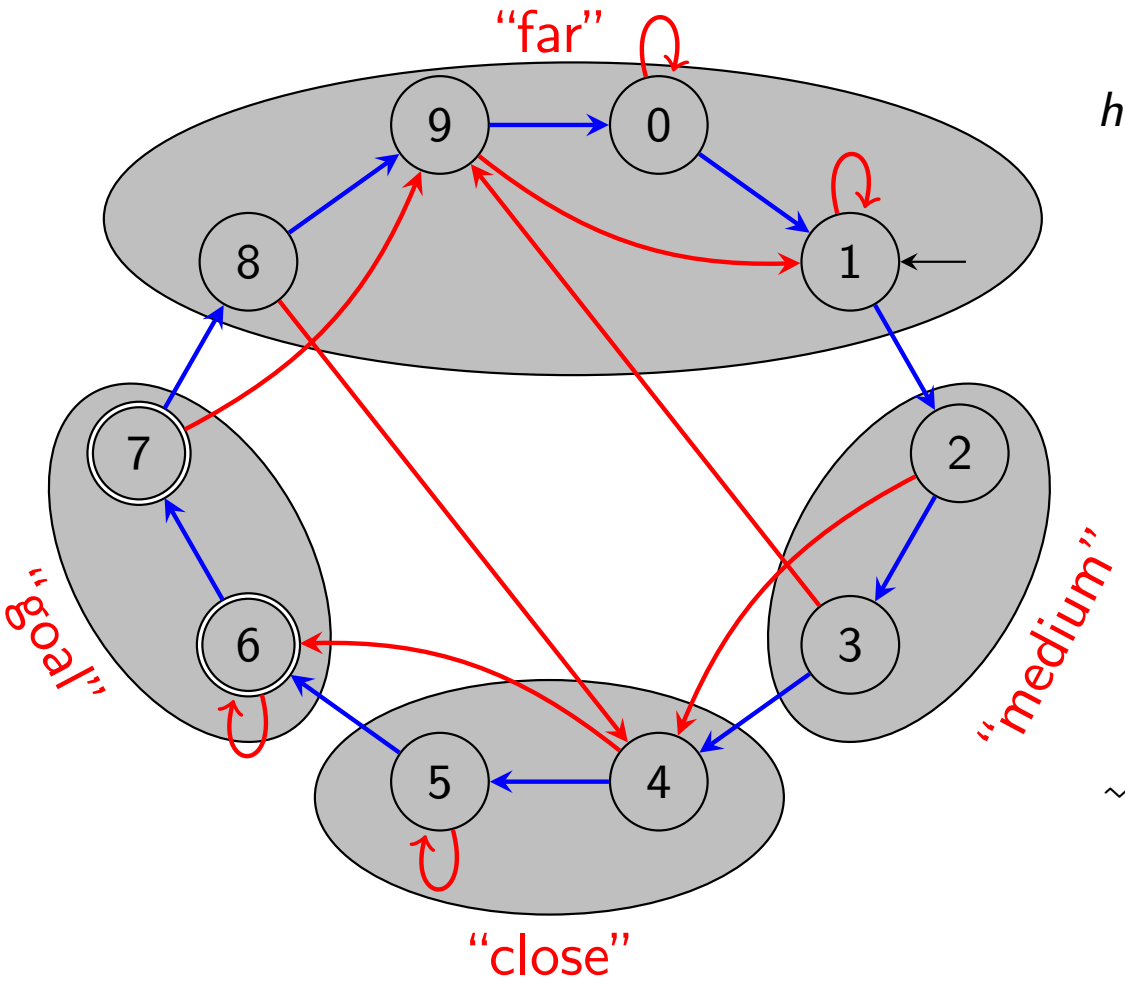
$$h_1(s) = \begin{cases} 0 & \text{if } s = 7 \\ (16 - s) \bmod 10 & \text{otherwise} \end{cases}$$

$\rightsquigarrow$ number of *inc* actions to goal

How accurate is this heuristic?

Bounded Inc-and-Square

bounded inc-and-square:



possible heuristics:

$$h_1(s) = \begin{cases} 0 & \text{if } s = 7 \\ (16 - s) \bmod 10 & \text{otherwise} \end{cases}$$

$\rightsquigarrow$ number of *inc* actions to goal

$$h_2(s) = \begin{cases} 0 & \text{if } s \text{ is a "goal"} \\ 1 & s \text{ is "close"} \\ 2 & s \text{ is "medium"} \\ 3 & s \text{ is "far"} \end{cases}$$

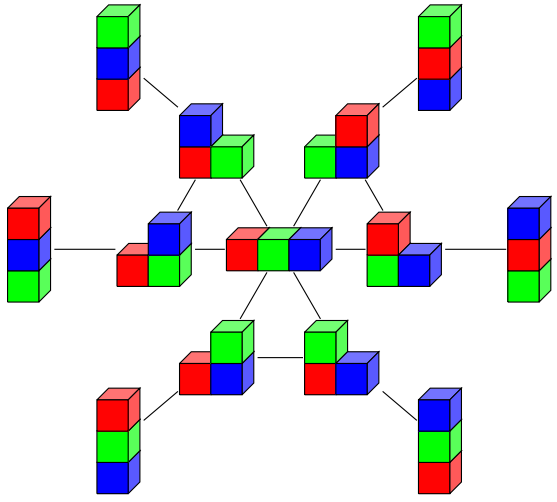
$\rightsquigarrow$ categorize states

How accurate is this heuristic?

Example: Blocks World

possible heuristic:

count blocks x that currently lie on y and must lie on $z \neq y$ in the goal (including case where y or z is the table)

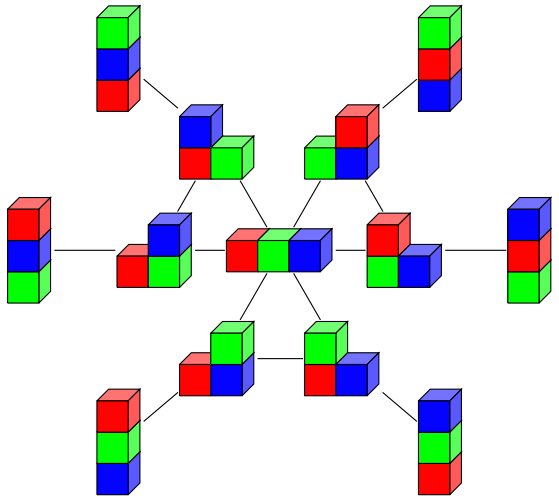


Example: Blocks World

possible heuristic:

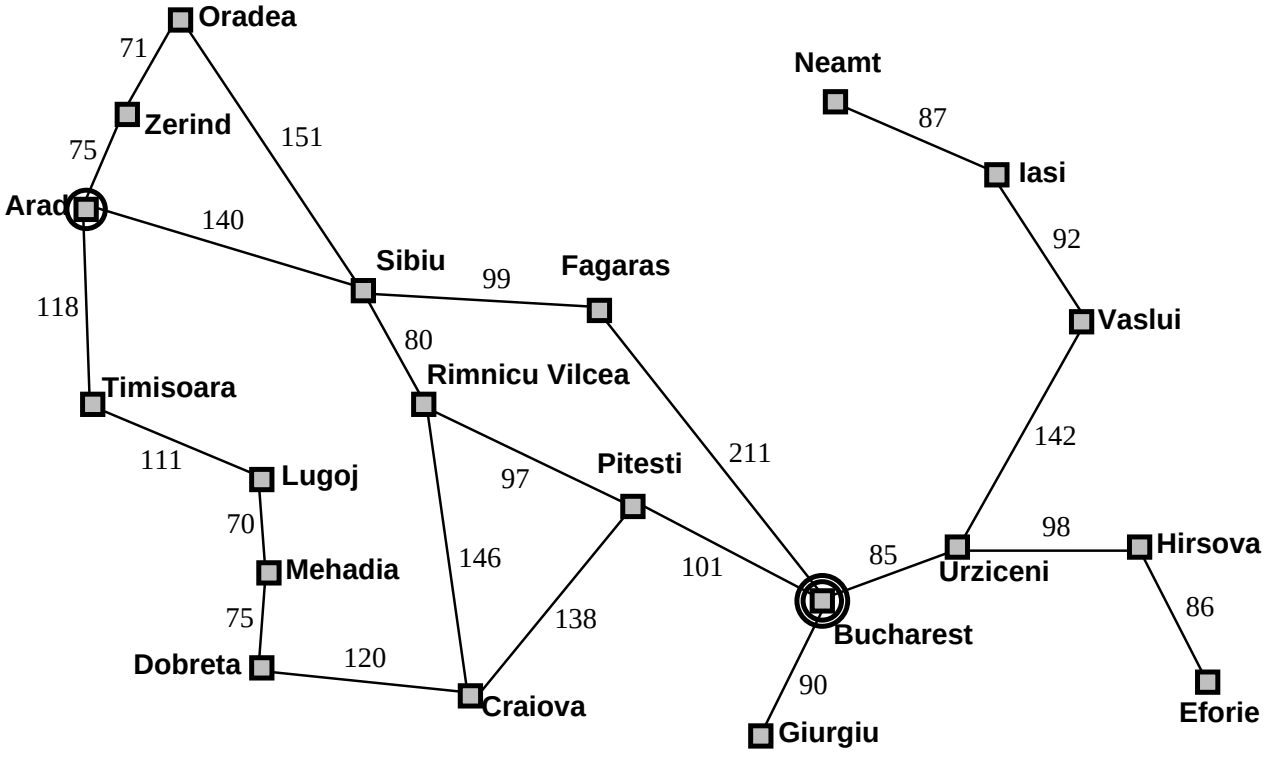
count blocks x that currently lie on y
and must lie on $z \neq y$ in the goal
(including case where y or z is the table)

How accurate is this heuristic?



Example: Route Planning in Romania

possible heuristic: straight-line distance to Bucharest



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Missionaries and Cannibals

Setting: Missionaries and Cannibals

- Six people must cross a river.
- Their rowing boat can carry one or two people across the river at a time (it is too small for three).
- Three people are missionaries, three are cannibals.
- Missionaries may never stay with a majority of cannibals.

possible heuristic: number of people on the wrong river bank

Example: Missionaries and Cannibals

Setting: Missionaries and Cannibals

- Six people must cross a river.
- Their rowing boat can carry one or two people across the river at a time (it is too small for three).
- Three people are missionaries, three are cannibals.
- Missionaries may never stay with a majority of cannibals.

possible heuristic: number of people on the wrong river bank

$\rightsquigarrow$ with our formulation of states as triples $\langle m, c, b \rangle$:
$h(\langle m, c, b \rangle) = m + c$

Summary

Summary

- heuristics estimate distance of a state to the goal
- can be used to focus search on promising states

$\rightsquigarrow$ soon: search algorithms that use heuristics

Foundations of Artificial Intelligence

B10. State-Space Search: Analysis of Heuristics

Malte Helmert

University of Basel

March 19, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Reminder: Heuristics

Definition (heuristic)

Let $\mathcal{S}$ be a state space with states S.
A heuristic function or heuristic for $\mathcal{S}$ is a function

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\},$$

mapping each state to a nonnegative number (or ∞).

Properties of Heuristics

Perfect Heuristic

Definition (perfect heuristic)

Let $\mathcal{S}$ be a state space with states S.

The perfect heuristic for $\mathcal{S}$, written h^*, maps each state $s \in S$

- to the cost of an optimal solution for s, or
- to ∞ if no solution for s exists.

German: perfekte Heuristik

Properties of Heuristics

Definition (safe, goal-aware, admissible, consistent)

Let $\mathcal{S}$ be a state space with states S.

A heuristic h for $\mathcal{S}$ is called

- safe if $h^*(s) = \infty$ for all $s \in S$ with $h(s) = \infty$
- goal-aware if $h(s) = 0$ for all goal states s
- admissible if $h(s) \leq h^*(s)$ for all states $s \in S$
- consistent if $h(s) \leq cost(a) + h(s')$ for all transitions $s \xrightarrow{a} s'$

German: sicher, zielerkennend, zulässig, konsistent

Properties of Heuristics

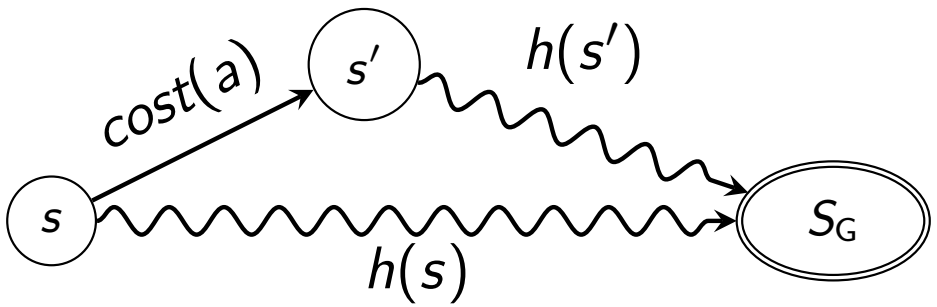
Definition (safe, goal-aware, admissible, consistent)

Let $\mathcal{S}$ be a state space with states S.

A heuristic h for $\mathcal{S}$ is called

- safe if $h^*(s) = \infty$ for all $s \in S$ with $h(s) = \infty$
- goal-aware if $h(s) = 0$ for all goal states s
- admissible if $h(s) \leq h^*(s)$ for all states $s \in S$
- consistent if $h(s) \leq cost(a) + h(s')$ for all transitions $s \xrightarrow{a} s'$

German: sicher, zielerkennend, zulässig, konsistent



Examples

Properties of Heuristics: Examples

Which of our three example heuristics have which properties?

Route Planning in Romania

straight-line distance:

- safe
- goal-aware
- admissible
- consistent

Why?

Properties of Heuristics: Examples

Which of our three example heuristics have which properties?

Blocks World

misplaced blocks:

- safe?
- goal-aware?
- admissible?
- consistent?

Properties of Heuristics: Examples

Which of our three example heuristics have which properties?

Missionaries and Cannibals

people on wrong river bank:

- safe?
- goal-aware?
- admissible?
- consistent?

Connections

Properties of Heuristics: Connections (1)

Theorem (admissible $\Longrightarrow$ safe + goal-aware)

Let h be an admissible heuristic.

Then h is safe and goal-aware.

Why?

Properties of Heuristics: Connections (2)

Theorem (goal-aware + consistent $\Longrightarrow$ admissible)

Let h be a goal-aware and consistent heuristic.

Then h is admissible.

Why?

Showing All Four Properties

How can one show most easily that a heuristic has all four properties?

Summary

Summary

- perfect heuristic h^*: true cost to the goal
- important properties: safe, goal-aware, admissible, consistent
- connections between these properties
  - admissible $\Longrightarrow$ safe and goal-aware
  - goal-aware and consistent $\Longrightarrow$ admissible

Foundations of Artificial Intelligence

B11. State-Space Search: Best-first Graph Search

Malte Helmert

University of Basel

March 19, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Introduction

Heuristic Search Algorithms

Heuristic Search Algorithms

Heuristic search algorithms use heuristic functions
to (partially or fully) determine the order of node expansion.

German: heuristische Suchalgorithmen

- this chapter: short introduction
- next chapters: more thorough analysis

Best-first Search

Best-first Search

Best-first search is a class of search algorithms that expand the "most promising" node in each iteration.

- decision which node is most promising uses heuristics...
- ...but not necessarily exclusively.

Best-first Search

Best-first search is a class of search algorithms that expand the "most promising" node in each iteration.

- decision which node is most promising uses heuristics...
- ...but not necessarily exclusively.

Best-first Search

A best-first search is a heuristic search algorithm that evaluates search nodes with an evaluation function f and always expands a node n with minimal $f(n)$ value.

German: Bestensuche, Bewertungsfunktion

- implementation essentially like uniform cost search
- different choices of $f \rightsquigarrow$ different search algorithms

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

- $f(n) = h(n.\text{state})$: greedy best-first search
  $\rightsquigarrow$ only the heuristic counts

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

- $f(n) = h(n.\text{state})$: greedy best-first search
  $\rightsquigarrow$ only the heuristic counts
- $f(n) = g(n) + h(n.\text{state})$: A*
  $\rightsquigarrow$ combination of path cost and heuristic

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

- $f(n) = h(n.\text{state})$: greedy best-first search
  $\rightsquigarrow$ only the heuristic counts
- $f(n) = g(n) + h(n.\text{state})$: A*
  $\rightsquigarrow$ combination of path cost and heuristic
- $f(n) = g(n) + w \cdot h(n.\text{state})$: weighted A*
  $w \in \mathbb{R}_0^+$ is a parameter
  $\rightsquigarrow$ interpolates between greedy best-first search and A*

German: gierige Bestensuche, A*, Weighted A*

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

- $f(n) = h(n.\text{state})$: greedy best-first search
  $\rightsquigarrow$ only the heuristic counts
- $f(n) = g(n) + h(n.\text{state})$: A*
  $\rightsquigarrow$ combination of path cost and heuristic
- $f(n) = g(n) + w \cdot h(n.\text{state})$: weighted A*
  $w \in \mathbb{R}_0^+$ is a parameter
  $\rightsquigarrow$ interpolates between greedy best-first search and A*

German: gierige Bestensuche, A*, Weighted A*

$\rightsquigarrow$ properties: next chapters

The Most Important Best-first Search Algorithms

the most important best-first search algorithms:

- $f(n) = h(n.\text{state})$: greedy best-first search
  $\rightsquigarrow$ only the heuristic counts
- $f(n) = g(n) + h(n.\text{state})$: A*
  $\rightsquigarrow$ combination of path cost and heuristic
- $f(n) = g(n) + w \cdot h(n.\text{state})$: weighted A*
  $w \in \mathbb{R}_0^+$ is a parameter
  $\rightsquigarrow$ interpolates between greedy best-first search and A*

German: gierige Bestensuche, A*, Weighted A*

$\rightsquigarrow$ properties: next chapters

What do we obtain with $f(n) := g(n)$?

Best-first Search: Graph Search or Tree Search?

Best-first search can be graph search or tree search.

- now: graph search (i.e., with duplicate elimination), which is the more common case
- Chapter B13: a tree search variant

Algorithm Details

Reminder: Uniform Cost Search

reminder from Chapter B7:

Uniform Cost Search

open := **new** MinHeap ordered by g
open.insert(make_root_node())
closed := **new** HashSet
while not *open*.is_empty():
    n := *open*.pop_min()
    if n.state $\notin$ *closed*:
        closed.insert(n.state)
        if is_goal(n.state):
            return extract_path(n)
        for each $\langle a, s' \rangle \in$ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable

Best-first Search without Reopening (1st Attempt)

Best-first Search without Reopening (1st Attempt)

```
open := new MinHeap ordered by f
open.insert(make_root_node())
closed := new HashSet
while not open.is_empty():
    n := open.pop_min()
    if n.state ∉ closed:
        closed.insert(n.state)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            n' := make_node(n, a, s')
            open.insert(n')
return unsolvable
```

Best-first Search w/o Reopening (1st Attempt): Discussion

Discussion:

This is already an acceptable implementation of best-first search.

Best-first Search w/o Reopening (1st Attempt): Discussion

Discussion:

This is already an acceptable implementation of best-first search.

two useful improvements:

- discard states considered unsolvable by the heuristic
  $\rightsquigarrow$ saves memory in *open*
- if multiple search nodes have identical f values,
  use h to break ties (preferring low h)
  - not always a good idea, but often
  - obviously unnecessary if $f = h$ (greedy best-first search)

Best-first Search without Reopening (Final Version)

Best-first Search without Reopening

open := **new** MinHeap ordered by $\langle f, h \rangle$
if $h(\text{init}()) < \infty$:
    open.insert(make_root_node())
closed := **new** HashSet
while not *open*.is_empty():
    n := *open*.pop_min()
    if n.state $\notin$ *closed*:
        closed.insert(n.state)
        if is_goal(n.state):
            return extract_path(n)
        for each $\langle a, s' \rangle \in$ succ(n.state):
            if $h(s') < \infty$:
                n' := make_node(n, a, s')
                open.insert(n')
return unsolvable

Best-first Search: Properties

properties:

- complete if h is safe (Why?)
- optimality depends on f $\rightsquigarrow$ next chapters

Reopening

Reopening

- reminder: uniform cost search expands nodes in order of increasing g values

$\rightsquigarrow$ guarantees that cheapest path to state of a node has been found when the node is expanded

- with arbitrary evaluation functions f in best-first search this does not hold in general

$\rightsquigarrow$ in order to find solutions of low cost, we may want to expand duplicate nodes when cheaper paths to their states are found (reopening)

German: Reopening

Best-first Search with Reopening

Best-first Search with Reopening

```
open := new MinHeap ordered by ⟨f, h⟩
if h(init()) < ∞:
    open.insert(make_root_node())
distances := new HashMap
while not open.is_empty():
    n := open.pop_min()
    if distances.lookup(n.state) = none or g(n) < distances[n.state]:
        distances[n.state] := g(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            if h(s') < ∞:
                n' := make_node(n, a, s')
                open.insert(n')
return unsolvable
```

$\rightsquigarrow$ *distances* controls reopening and replaces *closed*

Summary

Summary

- best-first search: expand node with minimal value of evaluation function f
  - $f = h$: greedy best-first search
  - $f = g + h$: A*
  - $f = g + w \cdot h$ with parameter $w \in \mathbb{R}_0^+$: weighted A*
- here: best-first search as a graph search
- reopening: expand duplicates with lower path costs to find cheaper solutions

Foundations of Artificial Intelligence

B12. State-Space Search: Greedy BFS, A*, Weighted A*

Malte Helmert

University of Basel

March 26, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Introduction

What Is It About?

In this chapter we study last chapter's algorithms in more detail:

- greedy best-first search
- A*
- weighted A*

Greedy Best-first Search

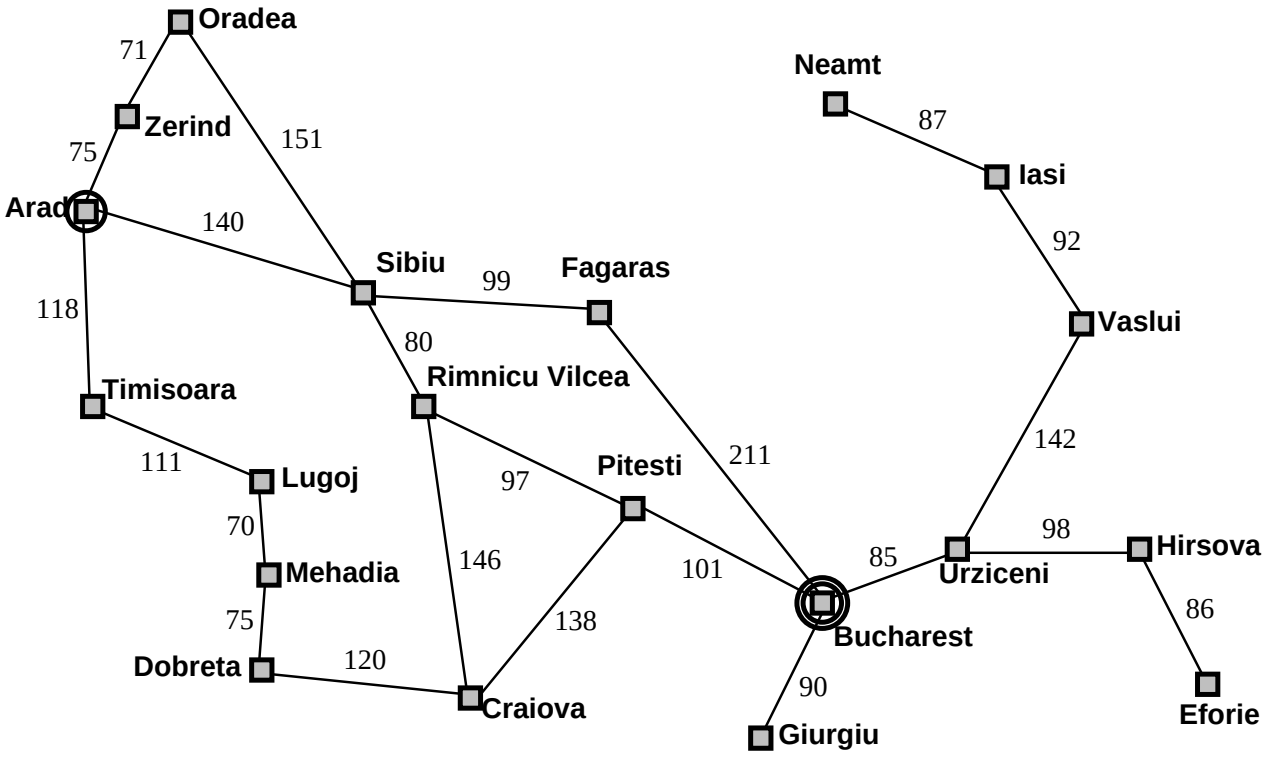
Greedy Best-first Search

Greedy Best-first Search

only consider the heuristic: $f(n) = h(n.\text{state})$

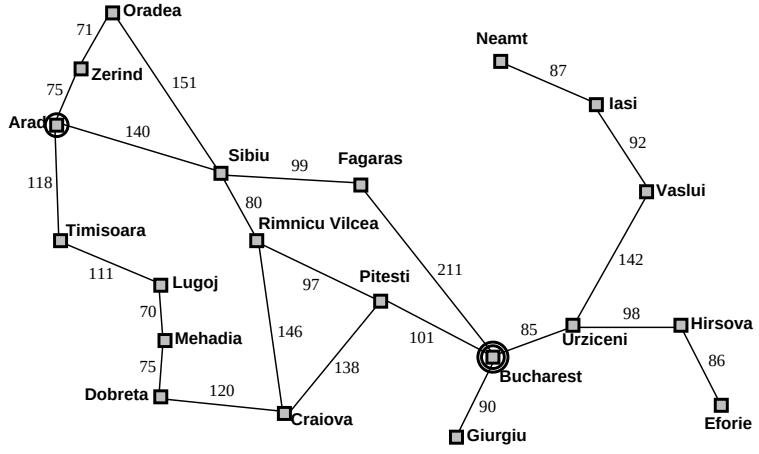
Note: usually without reopening (for reasons of efficiency)

Example: Greedy Best-first Search for Route Planning



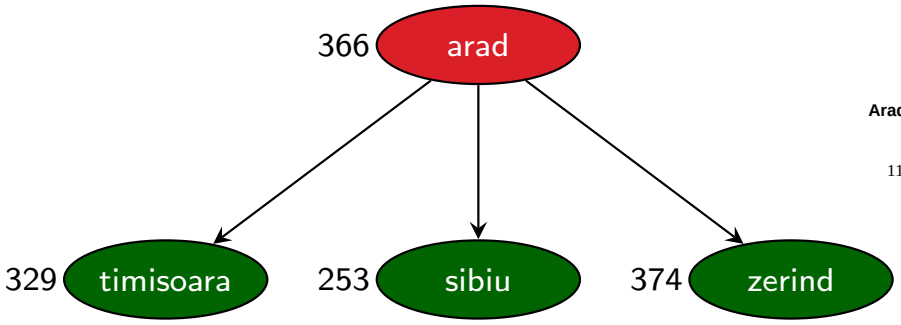
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

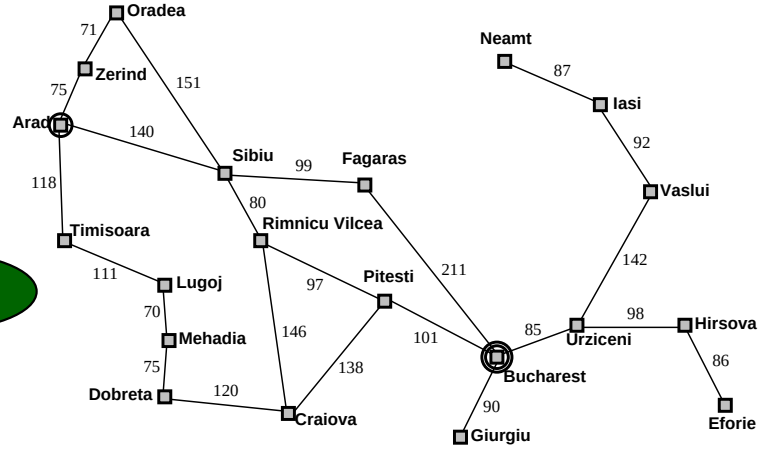
Example: Greedy Best-first Search for Route Planning



Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

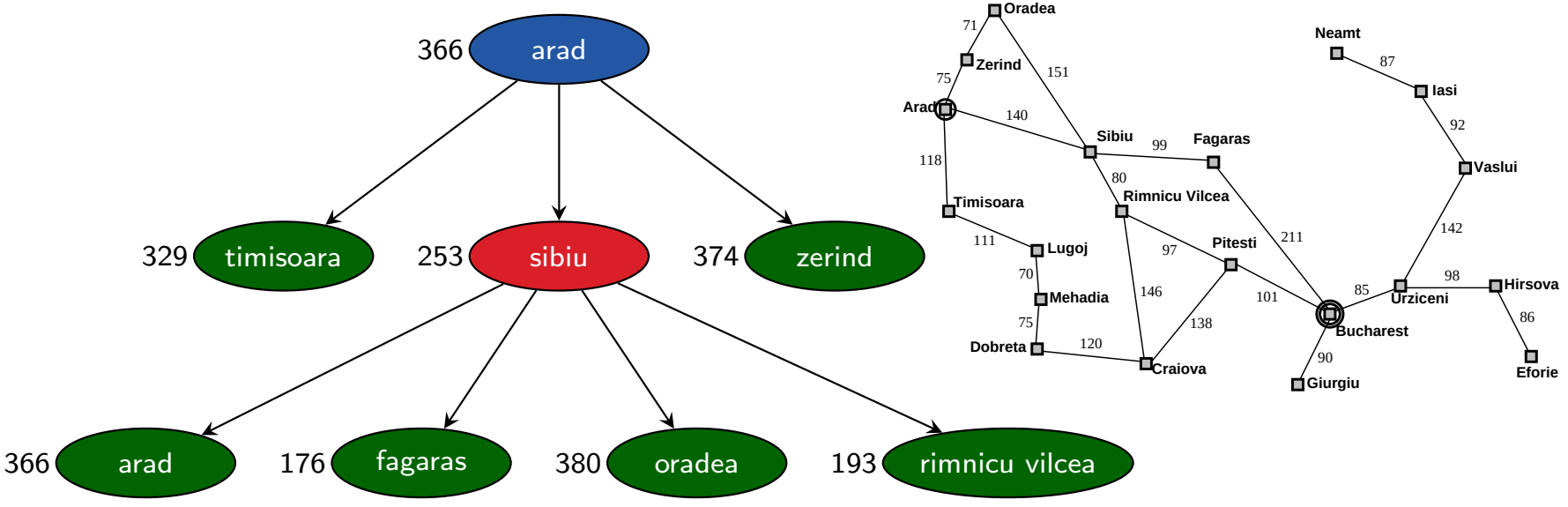
Example: Greedy Best-first Search for Route Planning





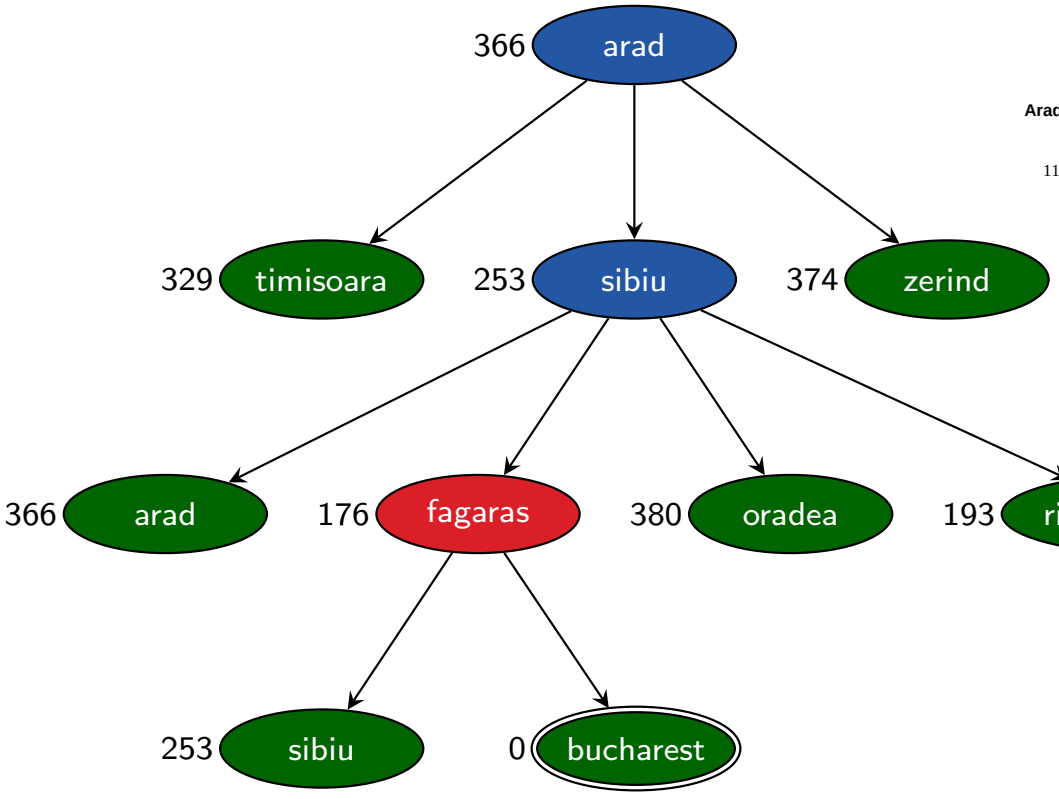
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

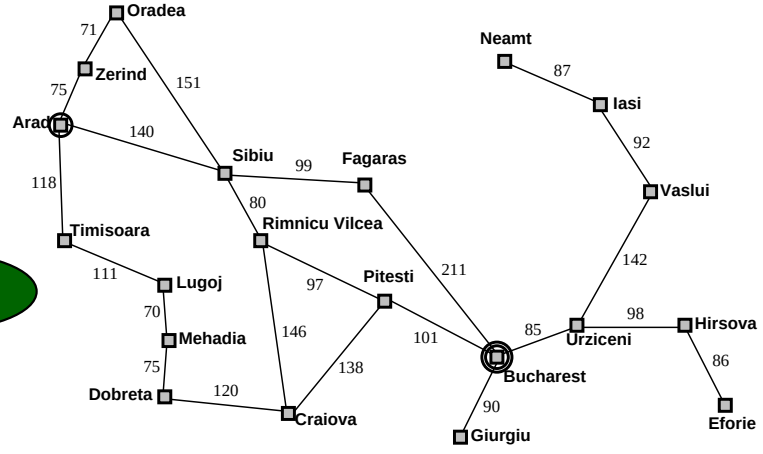
Example: Greedy Best-first Search for Route Planning



Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

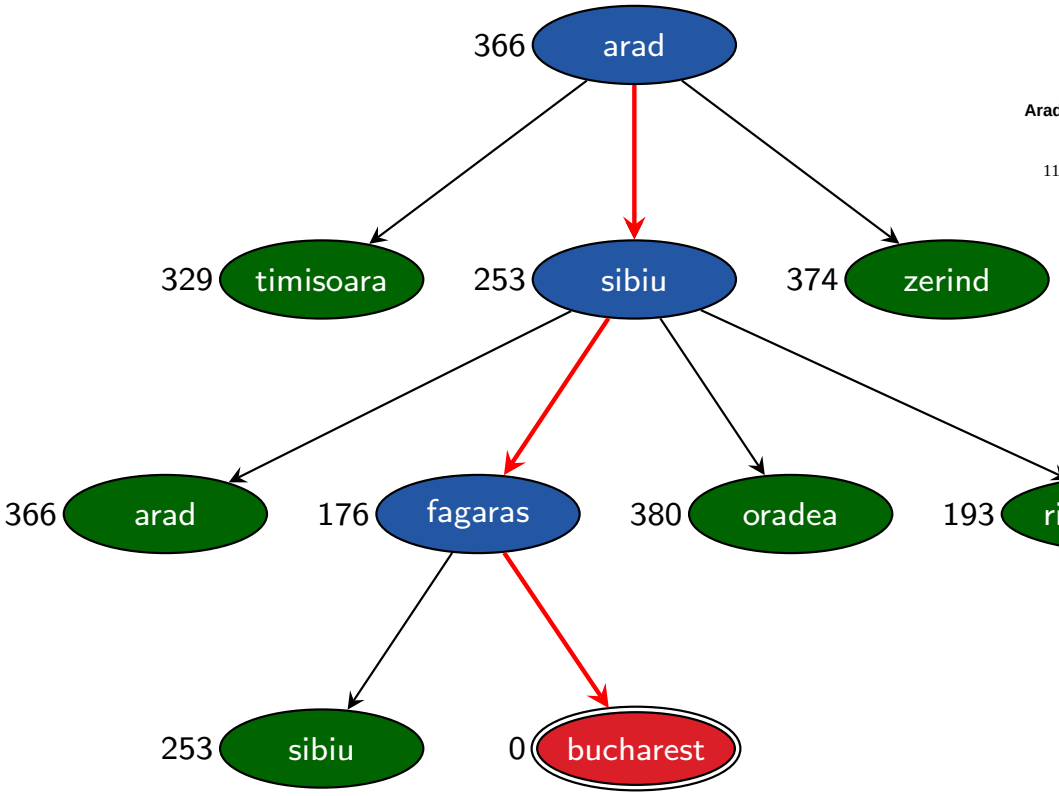
Example: Greedy Best-first Search for Route Planning

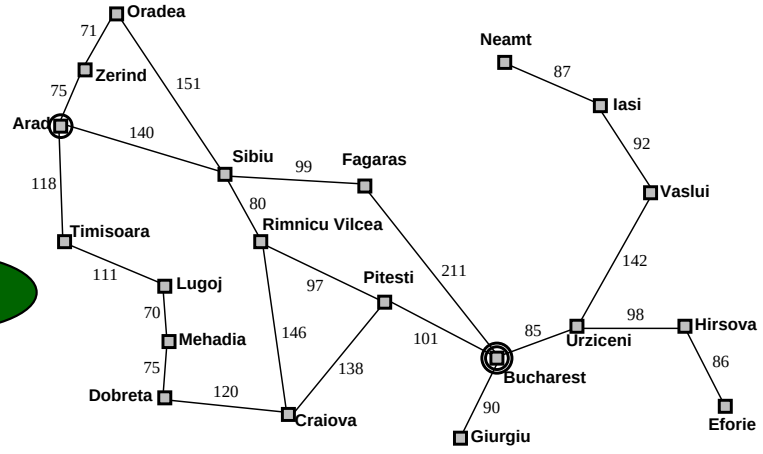




Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example: Greedy Best-first Search for Route Planning





Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Greedy Best-first Search: Properties

- complete with safe heuristics
  (like all variants of best-first graph search)
- suboptimal: solutions can be arbitrarily bad
- often very fast: one of the fastest search algorithms in practice
- monotonic transformations of h (e.g. scaling, additive constants) do not affect behaviour (Why is this interesting?)

A*

A*

A*

combine greedy best-first search with uniform cost search:
$f(n) = g(n) + h(n.\text{state})$

- trade-off between path cost and proximity to goal
- $f(n)$ estimates overall cost of cheapest solution
  from initial state via n to the goal

A*: Citations

hart nilsson raphael

About 16.300 results (**0,07** sec)

A formal basis for the heuristic determination of minimum cost paths

PE **Hart**, NJ **Nilsson**, B **Raphael** - IEEE transactions on Systems …, 1968 - ieeexplore.ieee.org

Although the problem of determining the minimum cost path through a graph arises naturally in a number of interesting applications, there has been no underlying theory to guide the …

☆ Save 99 Cite Cited by 17117 Related articles All 4 versions »

Correction to" a formal basis for the heuristic determination of minimum cost paths"

PE **Hart**, NJ **Nilsson**, B **Raphael** - ACM SIGART Bulletin, 1972 - dl.acm.org

Our paper on the use of heuristic information in graph searching defined a path-finding algorithm, A*, and proved that it had two important properties. In the notation of the paper, we …

☆ Save 99 Cite Cited by 592 Related articles All 11 versions

Research and applications: Artificial intelligence

B **Raphael**, RE Fikes, LJ Chaitin, PE **Hart**, RO Duda… - 1971 - ntrs.nasa.gov

A program of research in the field of artificial intelligence is presented. The research areas discussed include automatic theorem proving, representations of real-world environments, …

☆ Save 99 Cite Cited by 20 Related articles All 5 versions »

A*: Citations

hart nilsson raphael

About 16.300 results (**0,07** sec)

A formal basis for the heuristic determination of minimum cost paths

PE **Hart**, NJ **Nilsson**, B **Raphael** - IEEE transactions on Systems …, 1968 - ieeexplore.ieee.org

Although the problem of determining the minimum cost path through a graph arises naturally in a number of interesting applications, there has been no underlying theory to guide the …

☆ Save 99 Cite Cited by 17117 Related articles All 4 versions »

Correction to" a formal basis for the heuristic determination of minimum cost paths"

PE **Hart**, NJ **Nilsson**, B **Raphael** - ACM SIGART Bulletin, 1972 - dl.acm.org

Our paper on the use of heuristic information in graph searching defined a path-finding algorithm, A*, and proved that it had two important properties. In the notation of the paper, we …

☆ Save 99 Cite Cited by 592 Related articles All 11 versions

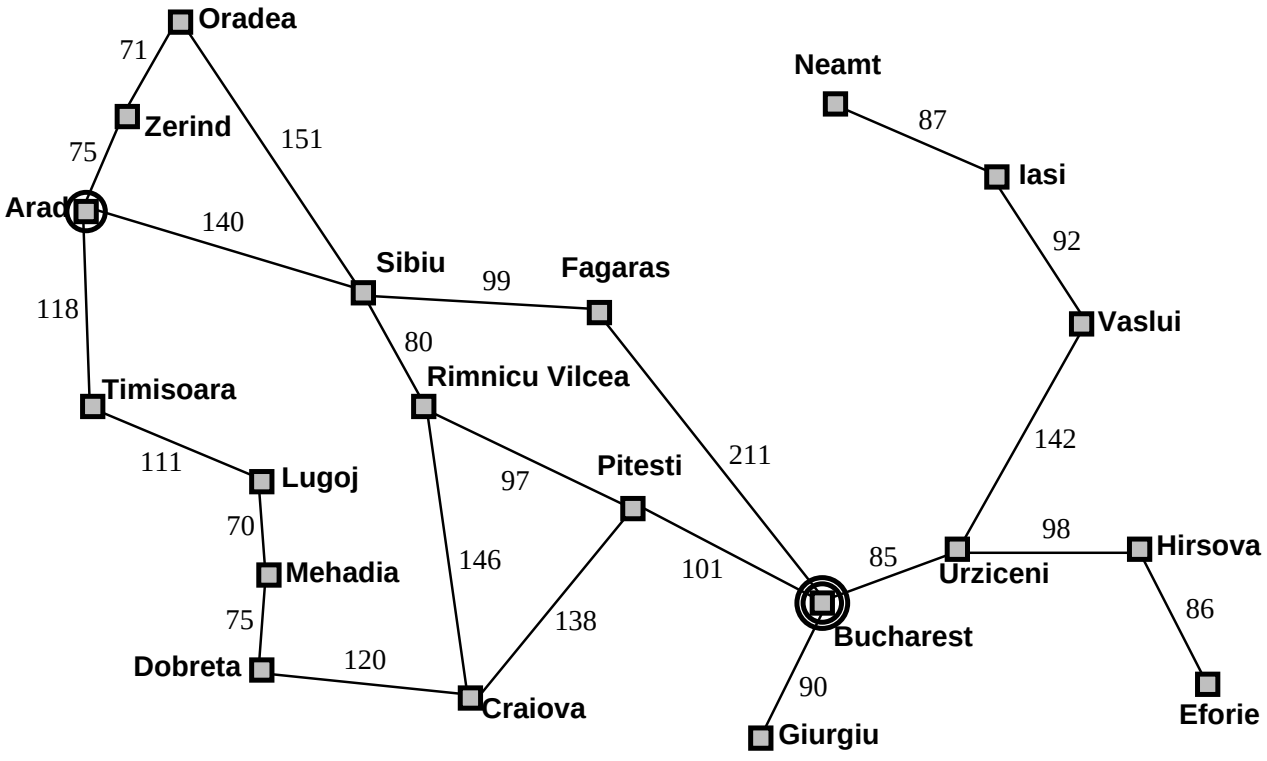
Research and applications: Artificial intelligence

B **Raphael**, RE Fikes, LJ Chaitin, PE **Hart**, RO Duda… - 1971 - ntrs.nasa.gov

A program of research in the field of artificial intelligence is presented. The research areas discussed include automatic theorem proving, representations of real-world environments, …

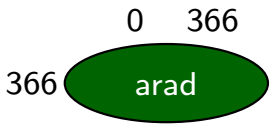
☆ Save 99 Cite Cited by 20 Related articles All 5 versions »

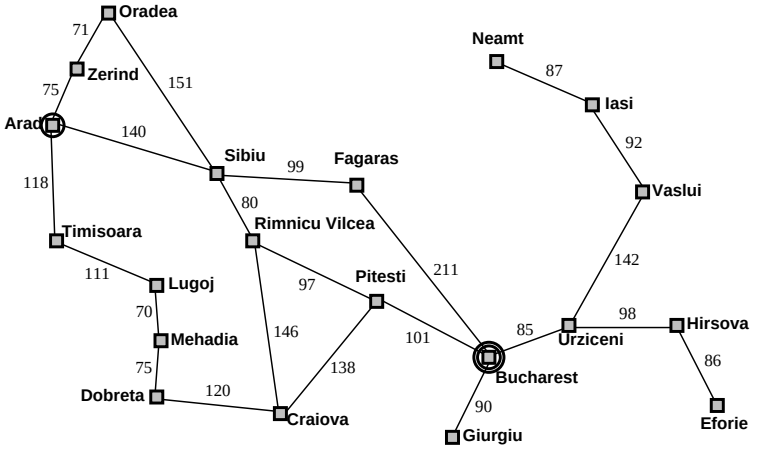
Example: A* for Route Planning



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

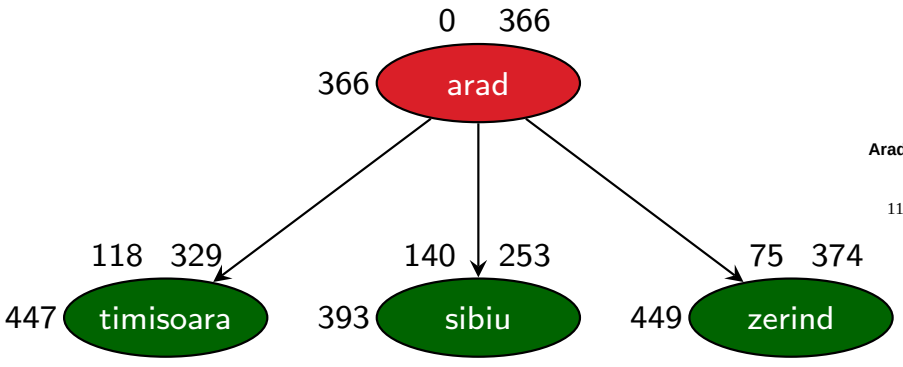
Example A* for Route Planning

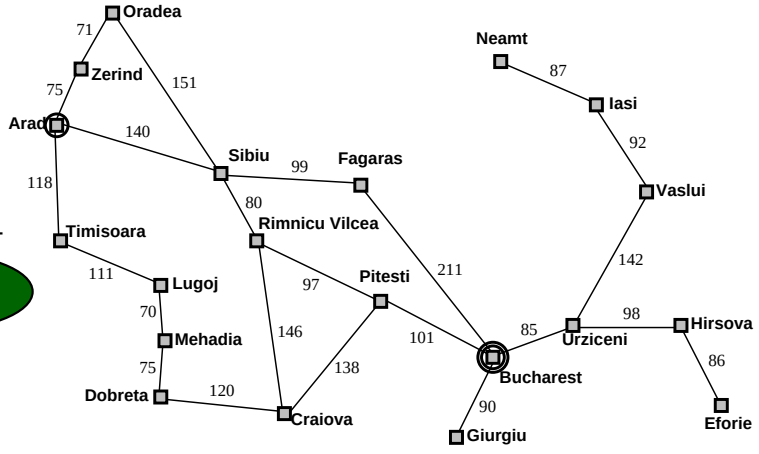




Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

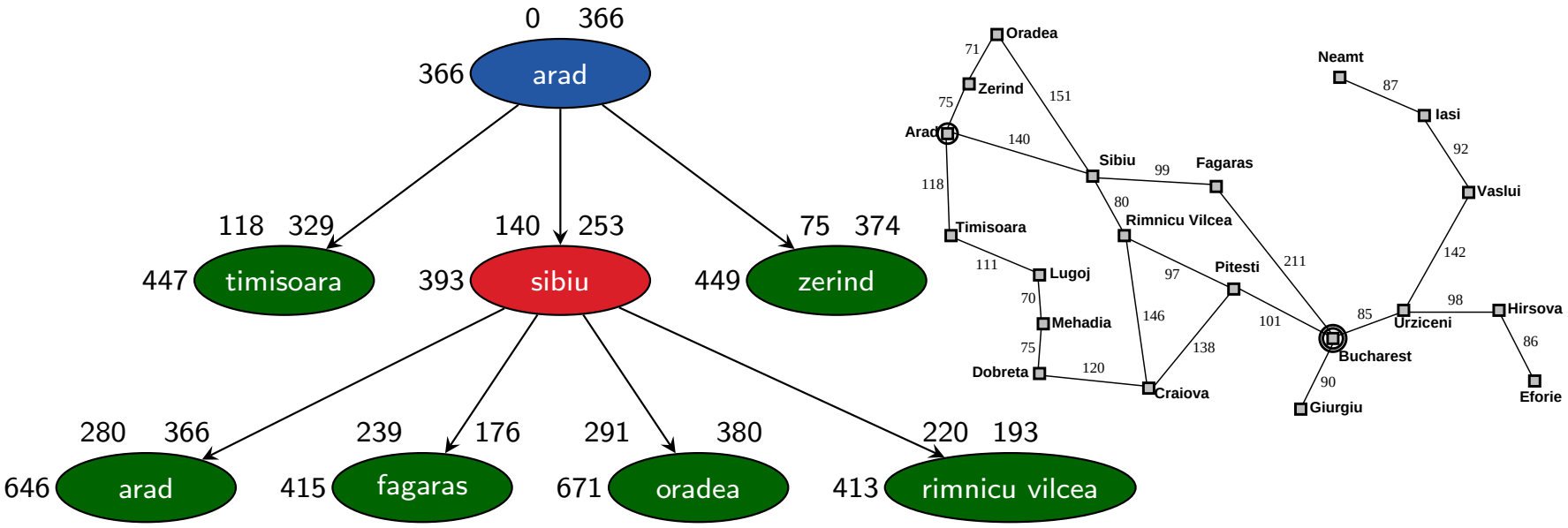
Example A* for Route Planning





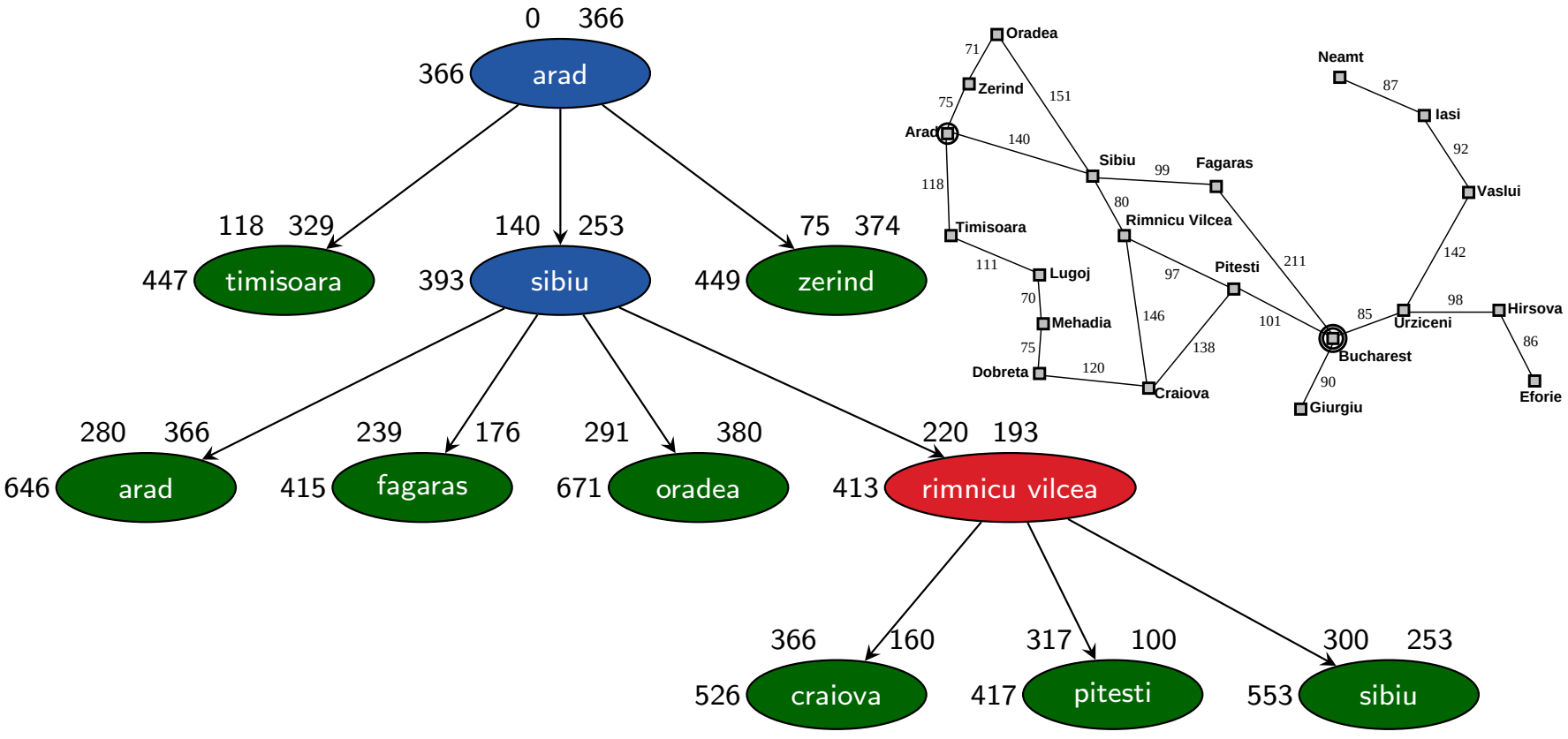
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



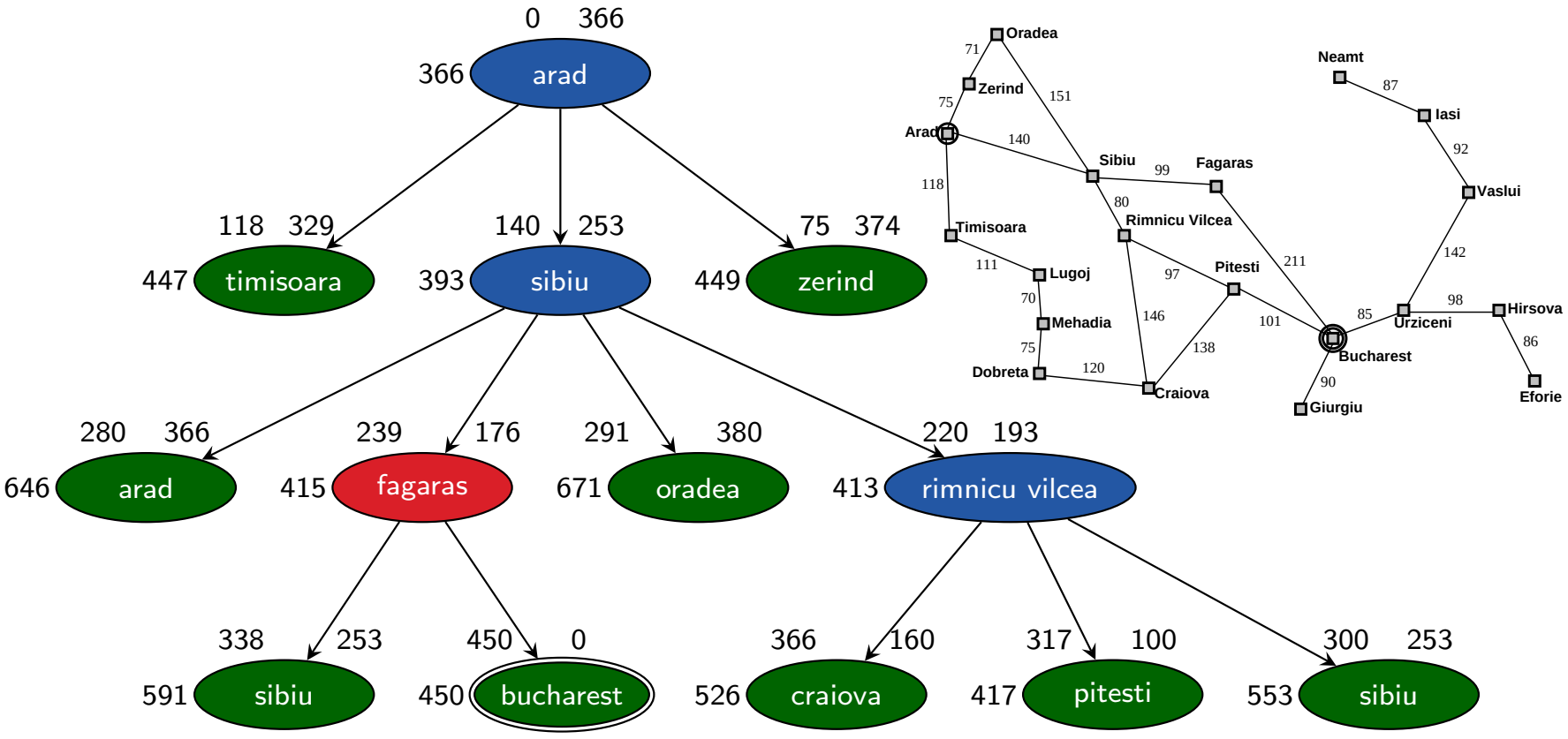
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



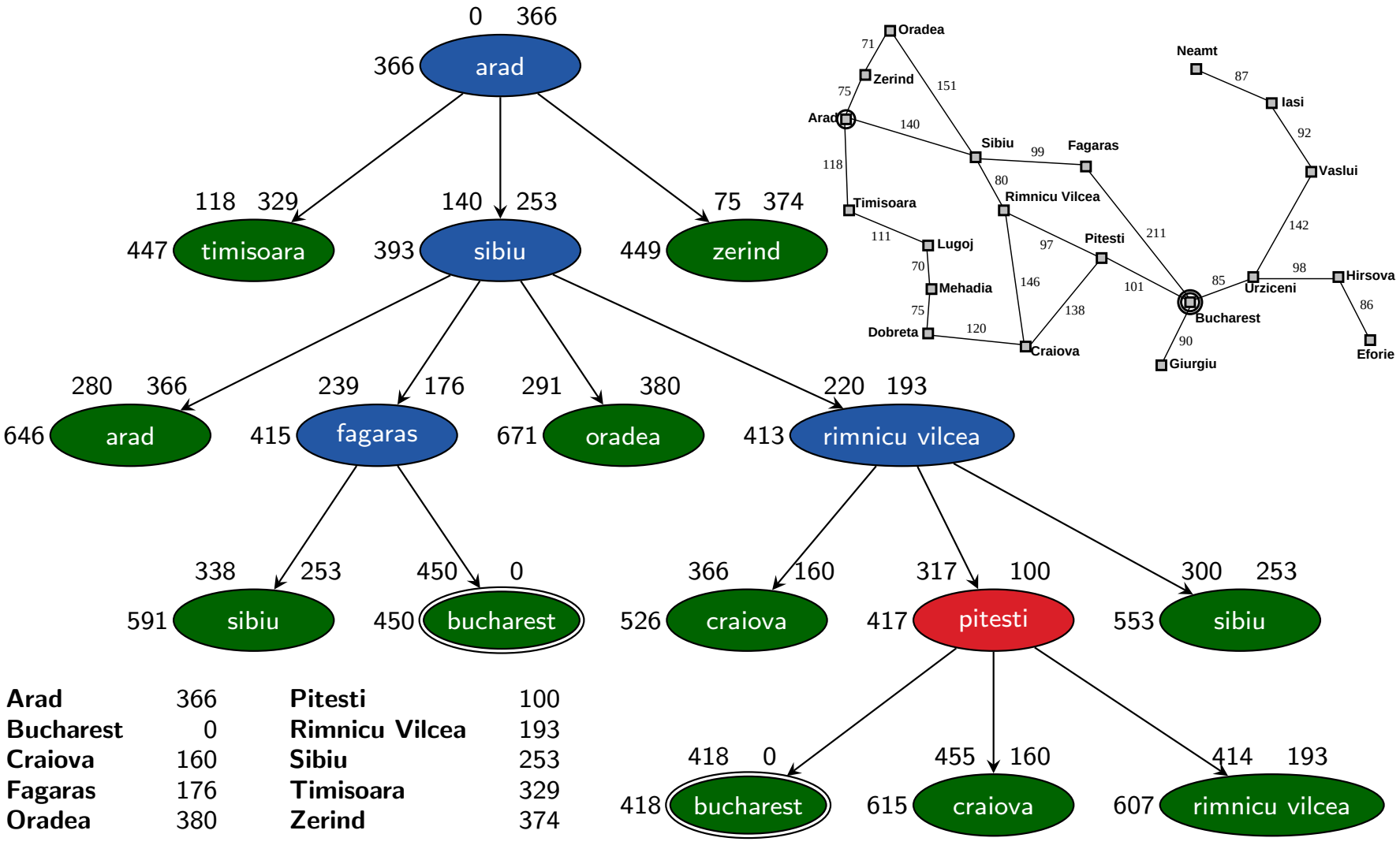
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



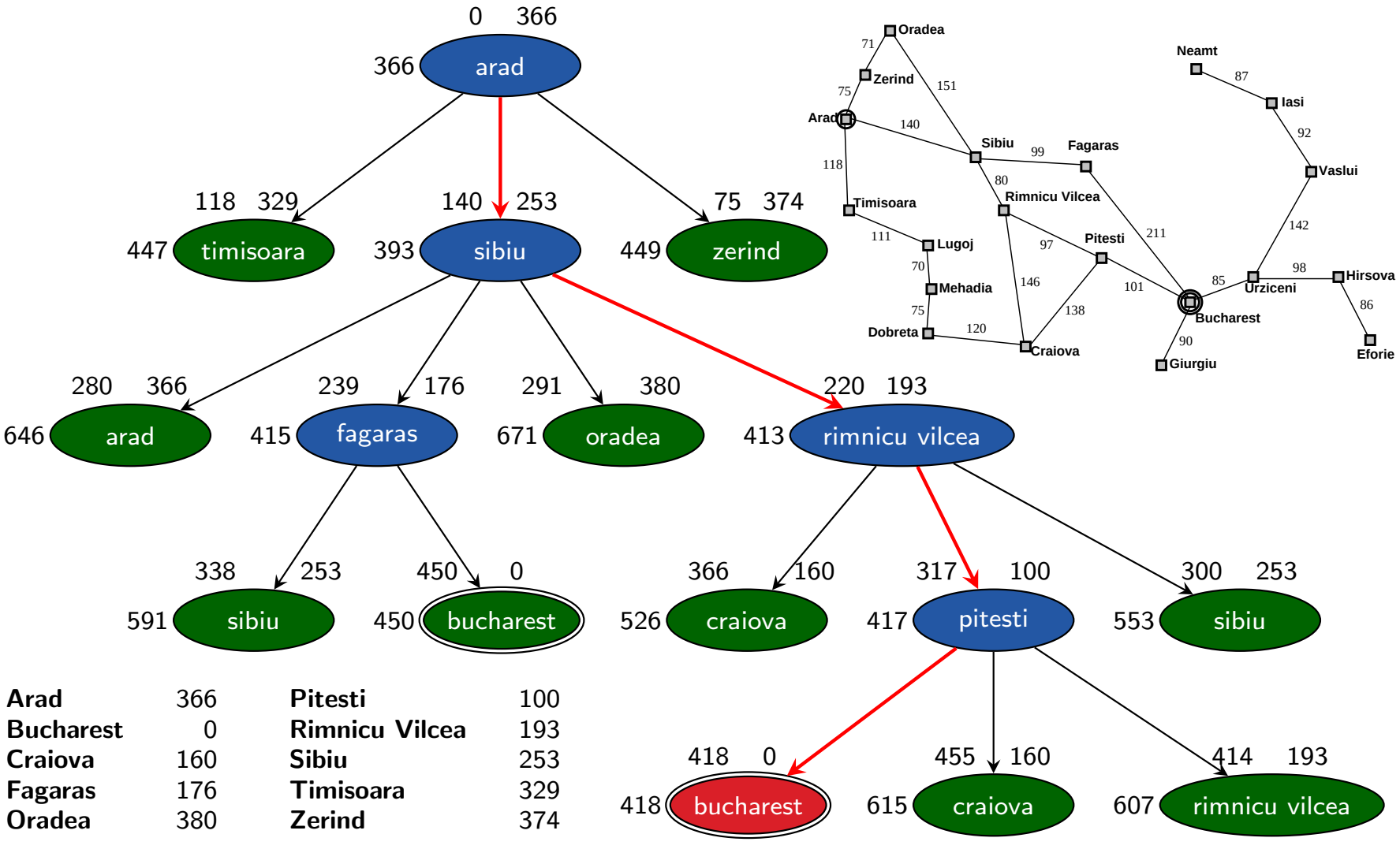
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

A*: Properties

- complete with safe heuristics
  (like all variants of best-first graph search)
- with reopening: optimal with admissible heuristics
- without reopening: optimal with heuristics
  that are admissible and consistent

$\rightsquigarrow$ proofs: Chapters B14 and B15

A*: Implementation Aspects

some practical remarks on implementing A*:

- common bug: reopening not implemented
  although heuristic is not consistent
- common bug: duplicate test "too early"
  (upon generation of search nodes)
- common bug: goal test "too early"
  (upon generation of search nodes)
- all these bugs lead to loss of optimality
  and can remain undetected for a long time

Weighted A*

Weighted A*

Weighted A*

A* with more heavily weighted heuristic:
$f(n) = g(n) + w \cdot h(n.\text{state})$,
where weight $w \in \mathbb{R}_0^+$ with $w \geq 1$ is a freely choosable parameter

Note: $w < 1$ is conceivable, but usually not a good idea
(Why not?)

Weighted A*: Properties

weight parameter controls "greediness" of search:

- $w = 0$: like uniform cost search
- $w = 1$: like A*
- $w \to \infty$: like greedy best-first search

with $w \geq 1$ properties analogous to A*:

- *h* admissible:
  found solution guaranteed to be at most w times as expensive as optimum when reopening is used
- *h* admissible and consistent:
  found solution guaranteed to be at most w times as expensive as optimum; no reopening needed

(without proof)

Summary

Summary

best-first graph search with evaluation function f:

- $f = h$: greedy best-first search
  suboptimal, often very fast
- $f = g + h$: A*
  optimal if h admissible and consistent
  or if h admissible and reopening is used
- $f = g + w \cdot h$: weighted A*
  for $w \geq 1$ suboptimality factor at most w
  under same conditions as for optimality of A*

Foundations of Artificial Intelligence

B13. State-Space Search: IDA*

Malte Helmert

University of Basel

March 26, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

IDA*: Idea

IDA*

The main drawback of the presented best-first graph search algorithms is their space complexity.

Idea: use the concepts of iterative-deepening DFS

IDA*

The main drawback of the presented best-first graph search algorithms is their space complexity.

Idea: use the concepts of iterative-deepening DFS

- depth-limited search with increasing limits
- instead of depth we limit f
  (in this chapter $f(n) := g(n) + h(n.\text{state})$ as in A*)

$\rightsquigarrow$ IDA* (iterative-deepening A*)

- tree search, unlike the previous best-first search algorithms

IDA*: Algorithm

Reminder: Iterative Deepening Depth-first Search

reminder from Chapter B8: iterative deepening depth-first search

Iterative Deepening DFS

for *depth_limit* $\in \{0, 1, 2, \dots\}$:
    solution := depth_limited_search(init(), *depth_limit*)
    if *solution* $\neq$ **none**:
        return *solution*

function depth_limited_search(s, *depth_limit*):

if is_goal(s):
    return $\langle\rangle$
if *depth_limit* > 0:
    for each $\langle a, s' \rangle \in$ succ(s):
        solution := depth_limited_search(s', *depth_limit* $- 1$)
        if *solution* $\neq$ **none**:
            solution.push_front(a)
            return *solution*
return none

First Attempt: IDA* Main Function

first attempt: iterative deepening A* (IDA*)

IDA* (First Attempt)

for *f_limit* $\in \{0, 1, 2, \dots\}$:
    solution := f_limited_search(init(), 0, *f_limit*)
    if *solution* $\neq$ **none**:
        return *solution*

First Attempt: f-Limited Search

function f_limited_search(s, g, f_limit):

if $g + h(s) > f\_limit$:
    return none

if is_goal(s):
    return $\langle\rangle$

for each $\langle a, s' \rangle \in$ succ(s):
    $solution :=$ f_limited_search($s', g +$ cost(a), f_limit)
    if $solution \neq$ **none**:
        $solution$.push_front(a)
        return $solution$

return none

IDA* First Attempt: Discussion

- The pseudo-code can be rewritten to be even more similar to our IDDFS pseudo-code. However, this would make our next modification more complicated.
- The algorithm follows the same principles as IDDFS, but takes path costs and heuristic information into account.
- For unit-cost state spaces and the trivial heuristic $h : s \mapsto 0$ for all states s, it behaves identically to IDDFS.
- For general state spaces, there is a problem with this first attempt, however.

Growing the f Limit

- In IDDFS, we grow the limit from the smallest limit that gives a non-empty search tree (0) by 1 at a time.
- This usually leads to exponential growth of the tree between rounds, so that re-exploration work can be amortized.
- In our first attempt at IDA*, there is no guarantee that increasing the f limit by 1 will lead to a larger search tree than in the previous round.
- This problem becomes worse if we also allow non-integer (fractional) costs, where increasing the limit by 1 would be very arbitrary.

Setting the Next f Limit

idea: let the f-limited search compute the next sensible f limit

- Start with $h(\text{init}())$, the smallest f limit that results in a non-empty search tree.
- In every round, increase the f limit to the smallest value that ensures that in the next round at least one additional path will be considered by the search.

$\rightsquigarrow$ f_limited_search now returns two values:

- the next f limit that would include at least one new node in the search tree (∞ if no such limit exists; **none** if a solution was found), and
- the solution that was found (or **none**).

Final Algorithm: IDA* Main Function

final algorithm: iterative deepening A* (IDA*)

IDA*

$f\_limit = h(\text{init}())$
while $f\_limit \neq \infty$:
    $\langle f\_limit, solution \rangle := \text{f\_limited\_search}(\text{init}(), 0, f\_limit)$
    if $solution \neq$ **none**:
        return $solution$
return unsolvable

Final Algorithm: f-Limited Search

function f_limited_search(s, g, f_limit):

if $g + h(s) > f\_limit$:
    return $\langle g + h(s), \textbf{none} \rangle$

if is_goal(s):
    return $\langle \textbf{none}, \langle\rangle \rangle$

$new\_limit := \infty$
for each $\langle a, s' \rangle \in$ succ(s):
    $\langle child\_limit, solution \rangle :=$ f_limited_search($s', g +$ cost(a), f_limit)
    if $solution \neq$ **none**:
        $solution$.push_front(a)
        return $\langle \textbf{none}, solution \rangle$
    $new\_limit := \min(new\_limit, child\_limit)$
return $\langle new\_limit, \textbf{none} \rangle$

Final Algorithm: f-Limited Search

function f_limited_search(s, g, f_limit):

if $g + h(s) > f\_limit$:
    return $\langle g + h(s), \textbf{none} \rangle$

if is_goal(s):
    return $\langle \textbf{none}, \langle\rangle \rangle$

$new\_limit := \infty$
for each $\langle a, s' \rangle \in$ succ(s):
    $\langle child\_limit, solution \rangle :=$ f_limited_search($s', g +$ cost(a), f_limit)
    if $solution \neq$ **none**:
        $solution$.push_front(a)
        return $\langle \textbf{none}, solution \rangle$
    $new\_limit :=$ min($new\_limit, child\_limit$)
return $\langle new\_limit, \textbf{none} \rangle$

IDA*: Properties

IDA*: Properties

Inherits important properties of A* and depth-first search:

- semi-complete if h safe and $cost(a) > 0$ for all actions a
- optimal if h admissible
- space complexity $O(\ell b)$, where
  - ℓ: length of longest generated path
    (for unit cost problems: bounded by optimal solution cost)
  - b: branching factor

We state these without proof.

IDA*: Discussion

- compared to A* potentially considerable overhead because no duplicates are detected
  - $\rightsquigarrow$ exponentially slower in many state spaces
  - $\rightsquigarrow$ often combined with partial duplicate elimination (cycle detection, transposition tables)
- overhead due to iterative increases of f limit often negligible, but not always
  - especially problematic if action costs vary a lot: then it can easily happen that each new f limit only considers a small number of new paths

Summary

Summary

- IDA* is a tree search variant of A*
  based on iterative deepening depth-first search
- main advantage: low space complexity
- disadvantage: repeated work can be significant
- most useful when there are few duplicates

Foundations of Artificial Intelligence

B14. State-Space Search: Properties of A*, Part I

Malte Helmert

University of Basel

March 31, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Introduction

Optimality of A*

- advantage of A* over greedy search:
  optimal for heuristics with suitable properties
- very important result!

$\rightsquigarrow$ next chapters: a closer look at A*

- A* with reopening $\rightsquigarrow$ this chapter
- A* without reopening $\rightsquigarrow$ next chapter

Optimality of A* with Reopening

In this chapter, we prove that A* with reopening is optimal when using admissible heuristics.

For this purpose, we

- give some basic definitions
- prove two lemmas regarding the behaviour of A*
- use these to prove the main result

Reminder: A* with Reopening

reminder from Chapter B11/B12: A* with reopening

A* with Reopening

```
open := new MinHeap ordered by ⟨f, h⟩
if h(init()) < ∞:
    open.insert(make_root_node())
distances := new HashMap
while not open.is_empty():
    n := open.pop_min()
    if distances.lookup(n.state) = none or g(n) < distances[n.state]:
        distances[n.state] := g(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            if h(s') < ∞:
                n' := make_node(n, a, s')
                open.insert(n')
return unsolvable
```

Solvable States

Definition (solvable)

A state s of a state space is called solvable if $h^*(s) < \infty$.

German: lösbar

Optimal Paths to States

Definition (g^*)

Let s be a state of a state space with initial state s_I.

We write $g^*(s)$ for the cost of an optimal (cheapest) path from s_I to s (∞ if s is unreachable).

Remarks:

- g is defined for nodes, g^* for states (Why?)
- $g^*(n.\mathrm{state}) \leq g(n)$ for all nodes n generated by a search algorithm (Why?)

Settled States in A*

Definition (settled)

A state s is called settled at a given point
during the execution of A* (with or without reopening)
if s is included in *distances* and $\mathit{distances}[s] = g^*(s)$.

German: erledigt

Optimal Continuation Lemma

Optimal Continuation Lemma

We now show the first important result for A* with reopening:

Lemma (optimal continuation lemma)

Consider A* with reopening using a safe heuristic at the beginning of any iteration of the **while** *loop.*

If

- *state s is settled,*
- *state s' is a solvable successor of s, and*
- *an optimal path from s_I to s' of the form $\langle s_\mathrm{I}, \ldots, s, s' \rangle$ exists,*

then

- *s' is settled or*
- *open contains a node n' with $n'.\mathrm{state} = s'$ and $g(n') = g^*(s')$.*

German: Optimale-Fortsetzungs-Lemma

Optimal Continuation Lemma: Intuition

(Proof follows on the next slides.)

Intuitively, the lemma states:

> *If no optimal path to a given state has been found yet, open must contain a "good" node that contributes to finding an optimal path to that state.*

(This potentially requires multiple applications of the lemma along an optimal path to the state.)

Optimal Continuation Lemma: Proof (1)

Proof.

Consider states s and s' with the given properties at the start of some iteration ("iteration A") of A*.

Optimal Continuation Lemma: Proof (1)

Proof.

Consider states s and s' with the given properties at the start of some iteration ("iteration A") of A*.

Because s is settled, an earlier iteration ("iteration B") set $\mathit{distances}[s] := g^*(s)$.

Optimal Continuation Lemma: Proof (1)

Proof.

Consider states s and s' with the given properties at the start of some iteration ("iteration A") of A*.

Because s is settled, an earlier iteration ("iteration B") set $\mathit{distances}[s] := g^*(s)$.

Thus iteration B removed a node n with $n.\text{state} = s$ and $g(n) = g^*(s)$ from *open*.

Optimal Continuation Lemma: Proof (1)

Proof.

Consider states s and s' with the given properties
at the start of some iteration ("iteration A") of A*.

Because s is settled, an earlier iteration ("iteration B")
set $\mathit{distances}[s] := g^*(s)$.

Thus iteration B removed a node n
with $n.\text{state} = s$ and $g(n) = g^*(s)$ from *open*.

A* did not terminate in iteration B.
(Otherwise iteration A would not exist.)
Hence n was expanded in iteration B. ...

Optimal Continuation Lemma: Proof (2)

Proof (continued).

This expansion considered the successor s' of s.
Because s' is solvable, we have $h^*(s') < \infty$.
Because h is safe, this implies $h(s') < \infty$.
Hence a successor node n' was generated for s'.

Optimal Continuation Lemma: Proof (2)

Proof (continued).

This expansion considered the successor s' of s.
Because s' is solvable, we have $h^*(s') < \infty$.
Because h is safe, this implies $h(s') < \infty$.
Hence a successor node n' was generated for s'.

This node n' satisfies the consequence of the lemma.
Hence the criteria of the lemma were satisfied for s and s' after iteration B.

Optimal Continuation Lemma: Proof (2)

Proof (continued).

This expansion considered the successor s' of s.
Because s' is solvable, we have $h^*(s') < \infty$.
Because h is safe, this implies $h(s') < \infty$.
Hence a successor node n' was generated for s'.

This node n' satisfies the consequence of the lemma.
Hence the criteria of the lemma were satisfied for s and s' after iteration B.

To complete the proof, we show: if the consequence of the lemma is satisfied at the beginning of an iteration, it is also satisfied at the beginning of the next iteration. ...

Optimal Continuation Lemma: Proof (3)

Proof (continued).

- If s' is settled at the beginning of an iteration, it remains settled until termination.

Optimal Continuation Lemma: Proof (3)

Proof (continued).

- If s' is settled at the beginning of an iteration, it remains settled until termination.
- If s' is not yet settled and *open* contains a node n' with $n'.\text{state} = s'$ and $g(n') = g^*(s')$ at the beginning of an iteration, then either the node remains in *open* during the iteration, or n' is removed during the iteration and s' becomes settled.

f-Bound Lemma

f-Bound Lemma

We need a second lemma:

Lemma (f-bound lemma)

Consider A* with reopening and an admissible heuristic applied to a solvable state space with optimal solution cost c^*.

Then open contains a node n with $f(n) \leq c^*$ at the beginning of each iteration of the **while** *loop.*

German: f-Schranken-Lemma

f-Bound Lemma: Proof (1)

Proof.

Consider the situation at the beginning of any iteration of the **while** loop.

Let $\langle s_0, \ldots, s_n \rangle$ with $s_0 := s_\mathrm{I}$ be an optimal solution.
(Here we use that the state space is solvable.)

f-Bound Lemma: Proof (1)

Proof.

Consider the situation at the beginning of any iteration of the **while** loop.

Let $\langle s_0, \ldots, s_n \rangle$ with $s_0 := s_\mathrm{I}$ be an optimal solution.
(Here we use that the state space is solvable.)

Let s_i be the first state in the sequence that is not settled.

(Not all states in the sequence can be settled:
s_n is a goal state, and when a goal state is inserted into *distances*, A* terminates.) ...

f-Bound Lemma: Proof (2)

Proof (continued).

Case 1: $i = 0$

Because $s_0 = s_\mathrm{I}$ is not settled yet, we are at the first iteration of the **while** loop.

f-Bound Lemma: Proof (2)

Proof (continued).

Case 1: $i = 0$

Because $s_0 = s_\mathrm{I}$ is not settled yet, we are at the first iteration of the **while** loop.

Because the state space is solvable and h is admissible, we have $h(s_0) < \infty$.

f-Bound Lemma: Proof (2)

Proof (continued).

Case 1: $i = 0$

Because $s_0 = s_\mathrm{I}$ is not settled yet, we are at the first iteration of the **while** loop.

Because the state space is solvable and h is admissible, we have $h(s_0) < \infty$.

Hence *open* contains the root n_0.

f-Bound Lemma: Proof (2)

Proof (continued).

Case 1: $i = 0$

Because $s_0 = s_\mathrm{I}$ is not settled yet, we are at the first iteration of the **while** loop.

Because the state space is solvable and h is admissible, we have $h(s_0) < \infty$.

Hence *open* contains the root n_0.

We obtain: $f(n_0) = g(n_0) + h(s_0) = 0 + h(s_0) \leq h^*(s_0) = c^*$, where "$\leq$" uses the admissibility of h.

This concludes the proof for this case. ...

f-Bound Lemma: Proof (3)

Proof (continued).

Case 2: $i > 0$

Then s_{i-1} is settled and s_i is not settled.
Moreover, s_i is a solvable successor of s_{i-1} and $\langle s_0, \ldots, s_{i-1}, s_i \rangle$ is an optimal path from s_0 to s_i.

f-Bound Lemma: Proof (3)

Proof (continued).

Case 2: $i > 0$

Then s_{i-1} is settled and s_i is not settled.
Moreover, s_i is a solvable successor of s_{i-1} and $\langle s_0, \ldots, s_{i-1}, s_i \rangle$ is an optimal path from s_0 to s_i.

We can hence apply the optimal continuation lemma (with $s = s_{i-1}$ and $s' = s_i$) and obtain:

(A) s_i is settled, or

(B) *open* contains n' with $n'.\text{state} = s_i$ and $g(n') = g^*(s_i)$.

f-Bound Lemma: Proof (3)

Proof (continued).

Case 2: $i > 0$

Then s_{i-1} is settled and s_i is not settled.
Moreover, s_i is a solvable successor of s_{i-1} and $\langle s_0, \ldots, s_{i-1}, s_i \rangle$ is an optimal path from s_0 to s_i.

We can hence apply the optimal continuation lemma (with $s = s_{i-1}$ and $s' = s_i$) and obtain:

(A) s_i is settled, or

(B) *open* contains n' with $n'.\text{state} = s_i$ and $g(n') = g^*(s_i)$.

Because (A) is false, (B) must be true.

f-Bound Lemma: Proof (3)

Proof (continued).

Case 2: $i > 0$

Then s_{i-1} is settled and s_i is not settled.
Moreover, s_i is a solvable successor of s_{i-1} and $\langle s_0, \ldots, s_{i-1}, s_i \rangle$ is an optimal path from s_0 to s_i.

We can hence apply the optimal continuation lemma (with $s = s_{i-1}$ and $s' = s_i$) and obtain:

(A) s_i is settled, or

(B) *open* contains n' with $n'.\text{state} = s_i$ and $g(n') = g^*(s_i)$.

Because (A) is false, (B) must be true.

We conclude: open contains n' with
$f(n') = g(n') + h(s_i) = g^*(s_i) + h(s_i) \leq g^*(s_i) + h^*(s_i) = c^*$,
where "$\leq$" uses the admissibility of h. □

Optimality of A* with Reopening

Optimality of A* with Reopening

We can now show the main result of this chapter:

Theorem (optimality of A* with reopening)

A with reopening is optimal when using an admissible heuristic.*

Optimality of A* with Reopening: Proof

Proof.

By contradiction: assume that the theorem is wrong.

Hence there is a state space with optimal solution cost c^* where A* with reopening and an admissible heuristic returns a solution with cost $c > c^*$.

Optimality of A* with Reopening: Proof

Proof.

By contradiction: assume that the theorem is wrong.

Hence there is a state space with optimal solution cost c^* where A* with reopening and an admissible heuristic returns a solution with cost $c > c^*$.

This means that in the last iteration, the algorithm removes a node n with $g(n) = c > c^*$ from *open*.

Optimality of A* with Reopening: Proof

Proof.

By contradiction: assume that the theorem is wrong.

Hence there is a state space with optimal solution cost c^* where A* with reopening and an admissible heuristic returns a solution with cost $c > c^*$.

This means that in the last iteration, the algorithm removes a node n with $g(n) = c > c^*$ from *open*.

With $h(n.\text{state}) = 0$ (because h is admissible and hence goal-aware), this implies:

Optimality of A* with Reopening: Proof

Proof.

By contradiction: assume that the theorem is wrong.

Hence there is a state space with optimal solution cost c^* where A* with reopening and an admissible heuristic returns a solution with cost $c > c^*$.

This means that in the last iteration, the algorithm removes a node n with $g(n) = c > c^*$ from *open*.

With $h(n.\text{state}) = 0$ (because h is admissible and hence goal-aware), this implies:

$$f(n) = g(n) + h(n.\text{state}) = g(n) + 0 = g(n) = c > c^*.$$

Optimality of A* with Reopening: Proof

Proof.

By contradiction: assume that the theorem is wrong.

Hence there is a state space with optimal solution cost c^* where A* with reopening and an admissible heuristic returns a solution with cost $c > c^*$.

This means that in the last iteration, the algorithm removes a node n with $g(n) = c > c^*$ from *open*.

With $h(n.\text{state}) = 0$ (because h is admissible and hence goal-aware), this implies:

$$f(n) = g(n) + h(n.\text{state}) = g(n) + 0 = g(n) = c > c^*.$$

A* always removes a node n with minimal f value from *open*. With $f(n) > c^*$, we get a contradiction to the f-bound lemma, which completes the proof. □

Summary

Summary

- A* with reopening using an admissible heuristic is optimal.
- The proof is based on the following lemmas
  that hold for solvable state spaces and admissible heuristics:
  - optimal continuation lemma: The open list always contains nodes that make progress towards an optimal solution.
  - f-bound lemma: The minimum f value in the open list at the beginning of each A* iteration is a lower bound on the optimal solution cost.

Foundations of Artificial Intelligence

B15. State-Space Search: Properties of A*, Part II

Malte Helmert

University of Basel

March 31, 2025

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms
  - B9. Heuristics
  - B10. Analysis of Heuristics
  - B11. Best-first Graph Search
  - B12. Greedy Best-first Search, A*, Weighted A*
  - B13. IDA*
  - B14. Properties of A*, Part I
  - B15. Properties of A*, Part II

Introduction

Optimality of A* without Reopening

We now study A* without reopening.

- For A* without reopening, admissibility and consistency together guarantee optimality.
- We prove this on the following slides, again beginning with a basic lemma.
- Either of the two properties on its own would not be sufficient for optimality. (How would one prove this?)

Reminder: A* without Reopening

reminder from Chapter B11/B12: A* without reopening

A* without Reopening

```
open := new MinHeap ordered by ⟨f, h⟩
if h(init()) < ∞:
    open.insert(make_root_node())
closed := new HashSet
while not open.is_empty():
    n := open.pop_min()
    if n.state ∉ closed:
        closed.insert(n)
        if is_goal(n.state):
            return extract_path(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            if h(s') < ∞:
                n' := make_node(n, a, s')
                open.insert(n')
return unsolvable
```

Monotonicity Lemma

A*: Monotonicity Lemma (1)

Lemma (monotonicity of A* with consistent heuristics)

Consider A with a consistent heuristic.*

Then:

1. *If n' is a child node of n, then $f(n') \geq f(n)$.*
2. *On all paths generated by A*, f values are non-decreasing.*
3. *The sequence of f values of the nodes expanded by A* is non-decreasing.*

German: Monotonielemma

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

- by definition of f: $f(n) = g(n) + h(s)$, $f(n') = g(n') + h(s')$

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

- by definition of f: $f(n) = g(n) + h(s)$, $f(n') = g(n') + h(s')$
- by definition of g: $g(n') = g(n) + cost(a)$

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

- by definition of f: $f(n) = g(n) + h(s)$, $f(n') = g(n') + h(s')$
- by definition of g: $g(n') = g(n) + cost(a)$
- by consistency of h: $h(s) \leq cost(a) + h(s')$

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

- by definition of f: $f(n) = g(n) + h(s)$, $f(n') = g(n') + h(s')$
- by definition of g: $g(n') = g(n) + cost(a)$
- by consistency of h: $h(s) \leq cost(a) + h(s')$

$\rightsquigarrow$ $f(n) = g(n) + h(s) \leq g(n) + cost(a) + h(s')$
$= g(n') + h(s') = f(n')$

A*: Monotonicity Lemma (2)

Proof.

on 1.:

Let n' be a child node of n via action a.

Let $s = n.\text{state}$, $s' = n'.\text{state}$.

- by definition of f: $f(n) = g(n) + h(s)$, $f(n') = g(n') + h(s')$
- by definition of g: $g(n') = g(n) + cost(a)$
- by consistency of h: $h(s) \leq cost(a) + h(s')$

$\rightsquigarrow$ $f(n) = g(n) + h(s) \leq g(n) + cost(a) + h(s')$
$= g(n') + h(s') = f(n')$

on 2.: follows directly from 1. ...

A*: Monotonicity Lemma (3)

Proof (continued).

on 3:

- Let f_b be the minimal f value in *open* at the beginning of a **while** loop iteration in A*.
  Let n be the removed node with $f(n) = f_b$.

A*: Monotonicity Lemma (3)

Proof (continued).

on 3:

- Let f_b be the minimal f value in *open* at the beginning of a **while** loop iteration in A*.
  Let n be the removed node with $f(n) = f_b$.
- to show: at the end of the iteration the minimal f value in *open* is at least f_b.

A*: Monotonicity Lemma (3)

Proof (continued).

on 3:

- Let f_b be the minimal f value in *open* at the beginning of a **while** loop iteration in A*. Let n be the removed node with $f(n) = f_b$.
- to show: at the end of the iteration the minimal f value in *open* is at least f_b.
- We must consider the operations modifying *open*: *open*.pop_min and *open*.insert.

A*: Monotonicity Lemma (3)

Proof (continued).

on 3:

- Let f_b be the minimal f value in *open* at the beginning of a **while** loop iteration in A*. Let n be the removed node with $f(n) = f_b$.
- to show: at the end of the iteration the minimal f value in *open* is at least f_b.
- We must consider the operations modifying *open*: *open*.pop_min and *open*.insert.
- *open*.pop_min can never decrease the minimal f value in *open* (only potentially increase it).

A*: Monotonicity Lemma (3)

Proof (continued).

on 3:

- Let f_b be the minimal f value in *open* at the beginning of a **while** loop iteration in A*. Let n be the removed node with $f(n) = f_b$.
- to show: at the end of the iteration the minimal f value in *open* is at least f_b.
- We must consider the operations modifying *open*: *open*.pop_min and *open*.insert.
- *open*.pop_min can never decrease the minimal f value in *open* (only potentially increase it).
- The nodes n' added with *open*.insert are children of n and hence satisfy $f(n') \geq f(n) = f_b$ according to part 1.

□

Optimality of A* without Reopening

Optimality of A* without Reopening

Theorem (optimality of A* without reopening)

A without reopening is optimal when using an admissible and consistent heuristic.*

Proof.

From the monotonicity lemma, the sequence of f values of nodes removed from the open list is non-decreasing.

- ⇝ If multiple nodes with the same state s are removed from the open list, then their g values are non-decreasing.
- ⇝ If we allowed reopening, it would never happen.
- ⇝ With consistent heuristics, A* without reopening behaves the same way as A* with reopening.

The result follows because A* with reopening and admissible heuristics is optimal. □

Time Complexity of A*

Time Complexity of A* (1)

What is the time complexity of A*?

- depends strongly on the quality of the heuristic
- an extreme case: $h = 0$ for all states
  - $\rightsquigarrow$ A* identical to uniform cost search
- another extreme case: $h = h^*$ and $cost(a) > 0$ for all actions a
  - $\rightsquigarrow$ A* only expands nodes along an optimal solution
  - $\rightsquigarrow$ $O(\ell^*)$ expanded nodes, $O(\ell^* b)$ generated nodes, where
    - ℓ^*: length of the found optimal solution
    - b: branching factor

Time Complexity of A* (2)

more precise analysis:

- dependency of the runtime of A* on heuristic error

example:

- unit cost problems with
- constant branching factor and
- constant absolute error: $|h^*(s) - h(s)| \leq c$ for all $s \in S$

time complexity:

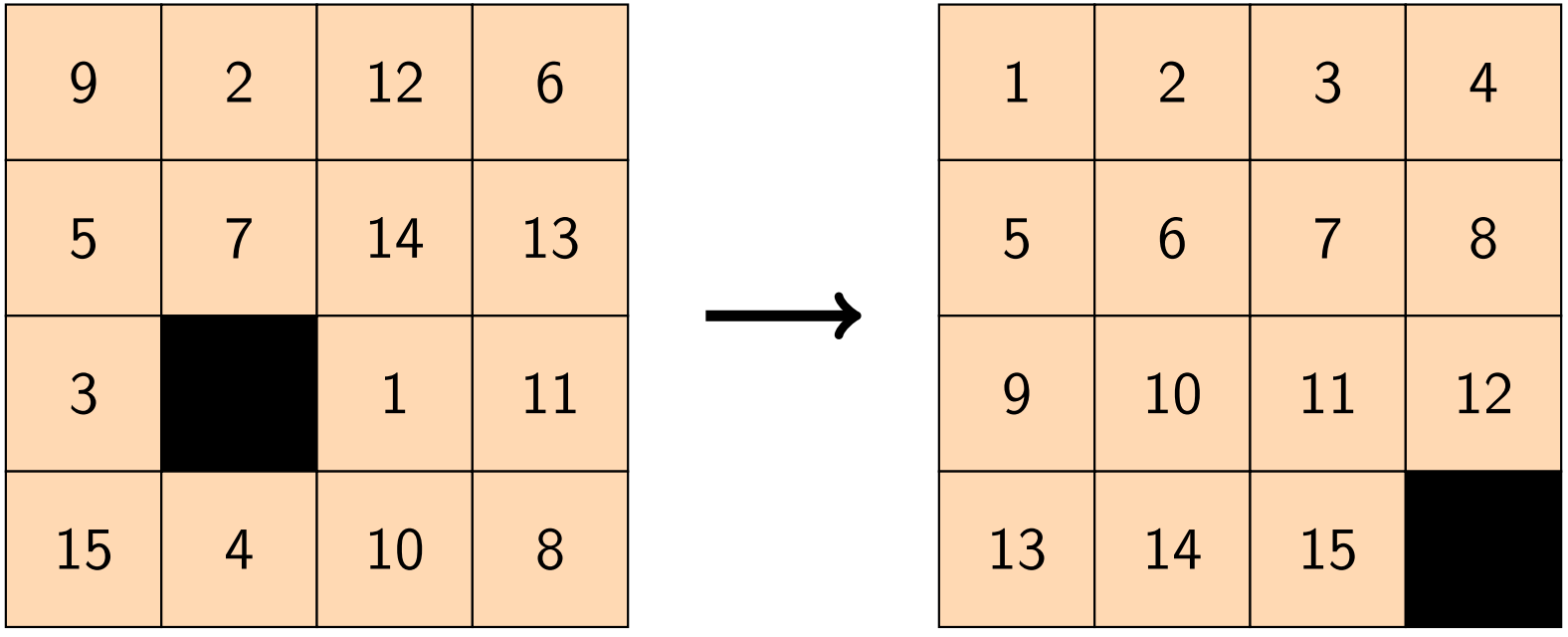
- if state space is a tree: time complexity of A* grows linearly in solution length (Pohl 1969; Gaschnig 1977)
- general search spaces: runtime of A* grows exponentially in solution length (Helmert & Röger 2008)

Overhead of Reopening

How does reopening affect runtime?

- For most practical state spaces and inconsistent admissible heuristics, the number of reopened nodes is negligible.
- exceptions exist:
  Martelli (1977) constructed state spaces with n states where exponentially many (in n) node reopenings occur in A*.
  ($\rightsquigarrow$ exponentially worse than uniform cost search)

Practical Evaluation of A* (1)



h_1: number of tiles in wrong cell (misplaced tiles)

h_2: sum of distances of tiles to their goal cell (Manhattan distance)

Practical Evaluation of A* (2)

- experiments with random initial states, generated by random walk from goal state
- entries show median of number of generated nodes for 101 random walks of the same length N

	generated nodes		
N	BFS-Graph	A* with h_1	A* with h_2
10	63	15	15
20	1,052	28	27
30	7,546	77	42
40	72,768	227	64
50	359,298	422	83
60	$>$ 1,000,000	7,100	307
70	$>$ 1,000,000	12,769	377
80	$>$ 1,000,000	62,583	849
90	$>$ 1,000,000	162,035	1,522
100	$>$ 1,000,000	690,497	4,964

Summary

Summary

- A* without reopening using an admissible and consistent heuristic is optimal
- key property monotonicity lemma (with consistent heuristics):
  - f values never decrease along paths considered by A*
  - sequence of f values of expanded nodes is non-decreasing
- time complexity depends on heuristic and shape of state space
  - precise details complex and depend on many aspects
  - reopening increases runtime exponentially in degenerate cases, but usually negligible overhead
  - small improvements in heuristic values often lead to exponential improvements in runtime

Foundations of Artificial Intelligence

C1. Combinatorial Optimization: Introduction and Hill-Climbing

Malte Helmert

University of Basel

April 2, 2025

Combinatorial Optimization: Overview

Chapter overview: combinatorial optimization

- C1. Introduction and Hill-Climbing
- C2. Advanced Techniques

Combinatorial Optimization

Introduction

previous chapters: classical state-space search

- find action sequence (path) from initial to goal state
- difficulty: large number of states ("state explosion")

next chapters: combinatorial optimization

$\rightsquigarrow$ similar scenario, but:

- no actions or transitions
- don't search for path, but for configuration ("state") with low cost/high quality

German: Zustandsraumexplosion, kombinatorische Optimierung, Konfiguration

Combinatorial Optimization: Example

Example: Nurse Scheduling Problem

- find a schedule for a hospital
- satisfy hard constraints
  - labor laws, hospital policies, ...
  - nurses working night shifts should not work early next day
  - have enough nurses with required skills present at all times
- maximize satisfaction of soft constraints
  - individual preferences, reduce overtime, fair distribution, ...

We are interested in a (high-quality) schedule, not a path to a goal.

Combinatorial Optimization Problems

Definition (combinatorial optimization problem)

A combinatorial optimization problem (COP) is given by a tuple $\langle C, S, opt, v \rangle$ consisting of:

- a finite set of (solution) candidates C
- a finite set of solutions $S \subseteq C$
- an objective sense $opt \in \{\min, \max\}$
- an objective function $v : S \to \mathbb{R}$

German: kombinatorisches Optimierungsproblem, Kandidaten, Lösungen, Optimierungsrichtung, Zielfunktion

Remarks:

- "problem" here in another sense (= "instance") than commonly used in computer science
- practically interesting COPs usually have too many candidates to enumerate explicitly

Optimal Solutions

Definition (optimal)

Let $\mathcal{O} = \langle C, S, opt, v \rangle$ be a COP.

The optimal solution quality v^* of $\mathcal{O}$ is defined as

$$v^* = \begin{cases} \min_{c \in S} v(c) & \text{if } opt = \min \\ \max_{c \in S} v(c) & \text{if } opt = \max \end{cases}$$

(v^* is undefined if $S = \emptyset$.)

A solution s of $\mathcal{O}$ is called optimal if $v(s) = v^*$.

German: optimale Lösungsqualität, optimal

Combinatorial Optimization

The basic algorithmic problem we want to solve:

Combinatorial Optimization

Find a solution of good (ideally, optimal) quality
for a combinatorial optimization problem $\mathcal{O}$
or prove that no solution exists.

Good here means close to v^* (the closer, the better).

Relevance and Hardness

- There is a huge number of practically important combinatorial optimization problems.
- Solving these is a central focus of operations research.
- Many important combinatorial optimization problems are NP-complete.
- Most "classical" NP-complete problems can be formulated as combinatorial optimization problems.

$\rightsquigarrow$ Examples: TSP, VertexCover, Clique, BinPacking, Partition

German: Unternehmensforschung, NP-vollständig

Search vs. Optimization

Combinatorial optimization problems have

- a search aspect (among all candidates C, find a solution from the set S) and
- an optimization aspect (among all solutions in S, find one of high quality).

Pure Search/Optimization Problems

Important special cases arise when one of the two aspects is trivial:

- pure search problems:
  - all solutions are of equal quality
  - difficulty is in finding a solution at all
  - formally: v is a constant function (e.g., constant 0); *opt* can be chosen arbitrarily (does not matter)
- pure optimization problems:
  - all candidates are solutions
  - difficulty is in finding solutions of high quality
  - formally: $S = C$

Example

Example: 8 Queens Problem

8 Queens Problem

How can we

- place 8 queens on a chess board
- such that no two queens threaten each other?

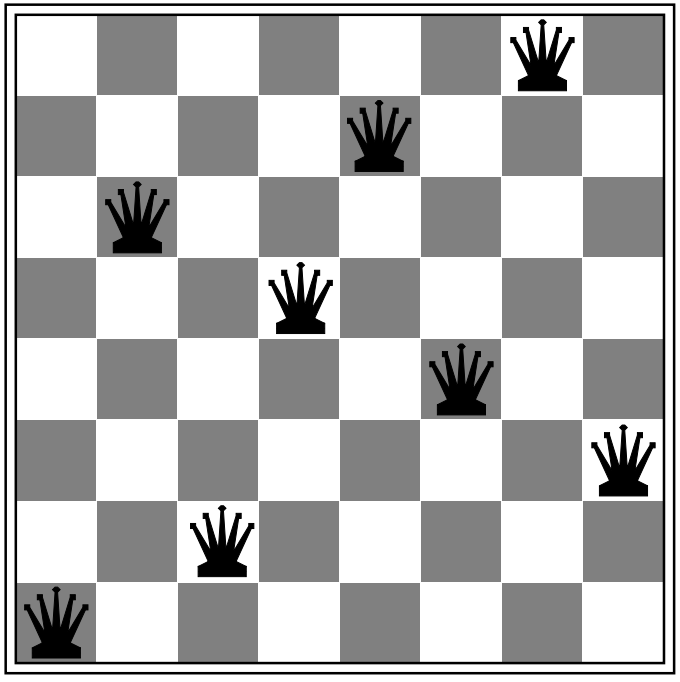
German: 8-Damen-Problem

- originally proposed in 1848
- variants: board size; other pieces; higher dimension

There are 92 solutions, or 12 solutions if we do not count symmetric solutions (under rotation or reflection) as distinct.

Example: 8 Queens Problem

Problem: Place 8 queens on a chess board
such that no two queens threaten each other.



Is this candidate a solution?

Formally: 8 Queens Problem

How can we formalize the problem?

idea:

- obviously there must be exactly one queen in each file ("column")
- describe candidates as 8-tuples, where the i-th entry denotes the rank ("row") of the queen in the i-th file

formally: $\mathcal{O} = \langle C, S, opt, v \rangle$ with

- $C = \{1, \ldots, 8\}^8$
- $S = \{\langle r_1, \ldots, r_8 \rangle \mid \forall 1 \leq i < j \leq 8 : r_i \neq r_j \wedge |r_i - r_j| \neq |i - j|\}$
- v constant, opt irrelevant (pure search problem)

Local Search: Hill Climbing

Algorithms for Combinatorial Optimization Problems

How can we algorithmically solve COPs?

- formulation as classical state-space search
- formulation as constraint network
- formulation as logical satisfiability problem
- formulation as mathematical optimization problem (LP/IP)
- local search

Algorithms for Combinatorial Optimization Problems

How can we algorithmically solve COPs?

- formulation as classical state-space search
  $\rightsquigarrow$ Part B
- formulation as constraint network $\rightsquigarrow$ Part D
- formulation as logical satisfiability problem $\rightsquigarrow$ Part E
- formulation as mathematical optimization problem (LP/IP)
  $\rightsquigarrow$ not in this course
- local search $\rightsquigarrow$ today (Part C)

Search Methods for Combinatorial Optimization

- main ideas of heuristic search applicable for COPs
  $\rightsquigarrow$ states $\approx$ candidates
- main difference: no "actions" in problem definition
  - instead, we (as algorithm designers) can choose which candidates to consider neighbors
  - definition of neighborhood critical aspect of designing good algorithms for a given COP
- "path to goal" irrelevant to the user
  - no path costs, parents or generating actions

  $\rightsquigarrow$ no search nodes needed

Local Search: Idea

main ideas of local search algorithms for COPs:

- heuristic h estimates quality of candidates
  - for pure optimization: often objective function v itself
  - for pure search: often distance estimate to closest solution (as in state-space search)
- do not remember paths, only candidates
- often only one current candidate $\rightsquigarrow$ very memory-efficient (however, not complete or optimal)
- often initialization with random candidate
- iterative improvement by hill climbing

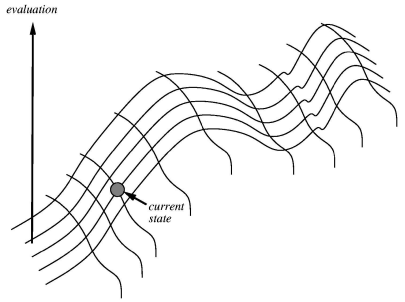
Hill Climbing

Hill Climbing (for Maximization Problems)

current := a random candidate
repeat:
    next := a neighbor of *current* with maximum h value
    if $h(next) \leq h(current)$:
        return *current*
    current := *next*

Remarks:

- search as walk "uphill" in a landscape
  defined by the neighborhood relation
- heuristic values define "height" of terrain
- analogous algorithm for minimization problems
  also traditionally called "hill climbing"
  even though the metaphor does not fully fit



Properties of Hill Climbing

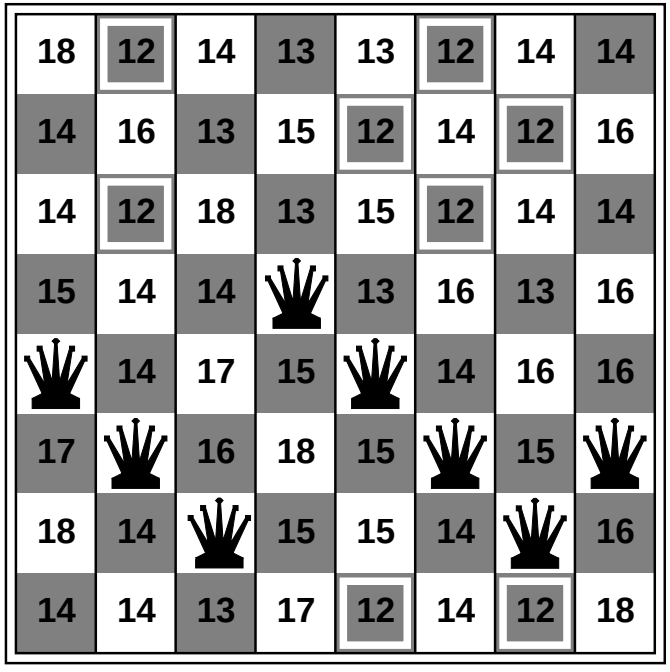
- always terminates (Why?)
- no guarantee that result is a solution
- if result is a solution, it is locally optimal w.r.t. h, but no global quality guarantees

Example: 8 Queens Problem

Problem: Place 8 queens on a chess board
such that no two queens threaten each other.

possible heuristic: no. of pairs of queens threatening each other
(formalization as minimization problem)

possible neighborhood: move one queen within its file



Performance of Hill Climbing for 8 Queens Problem

- problem has $8^8 \approx 17$ million candidates
  (reminder: 92 solutions among these)
- after random initialization, hill climbing finds a solution in around 14% of the cases
- only around 3–4 steps on average!

Summary

Summary

combinatorial optimization problems:

- find solution of good quality (objective value) among many candidates
- special cases:
  - pure search problems
  - pure optimization problems
- differences to state-space search:
  no actions, paths etc.; only "state" matters

often solved via local search:

- consider one candidate (or a few) at a time;
  try to improve it iteratively

Foundations of Artificial Intelligence

C2. Combinatorial Optimization: Advanced Techniques

Malte Helmert

University of Basel

April 2, 2025

Combinatorial Optimization: Overview

Chapter overview: combinatorial optimization

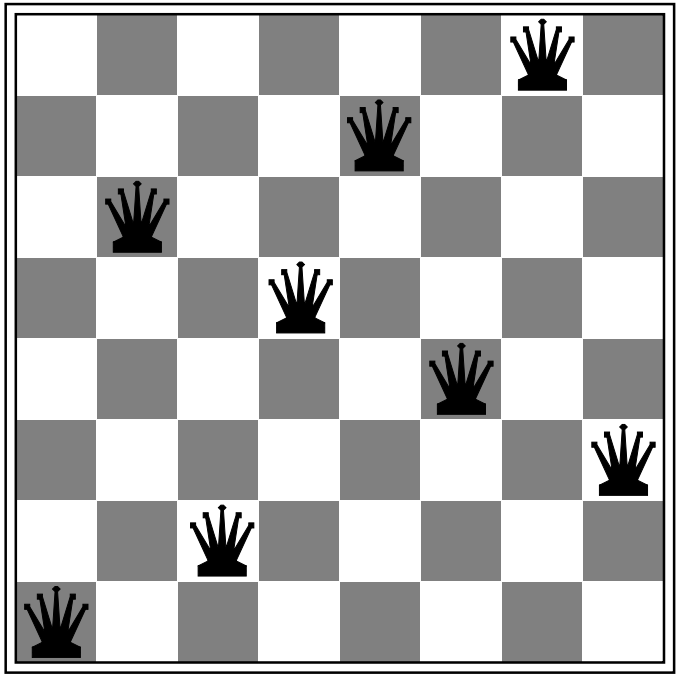
- C1. Introduction and Hill-Climbing
- C2. Advanced Techniques

Dealing with Local Optima

Example: Local Minimum in the 8 Queens Problem

local minimum:

- candidate has 1 conflict
- all neighbors have at least 2



Weaknesses of Local Search Algorithms

difficult situations for hill climbing:

- local optima: all neighbors worse than current candidate
- plateaus: many neighbors equally good as current candidate; none better

German: lokale Optima, Plateaus

consequence:

- algorithm gets stuck at current candidate

Combating Local Optima

possible remedies to combat local optima:

- allow stagnation (steps without improvement)
- include random aspects in the search neighborhood
- (sometimes) make random steps
- breadth-first search to better candidate
- restarts (with new random initial candidate)

Allowing Stagnation

allowing stagnation:

- do not terminate when no neighbor is an improvement
- limit number of steps to guarantee termination
- at end, return best visited candidate
  - pure search problems: terminate as soon as solution found

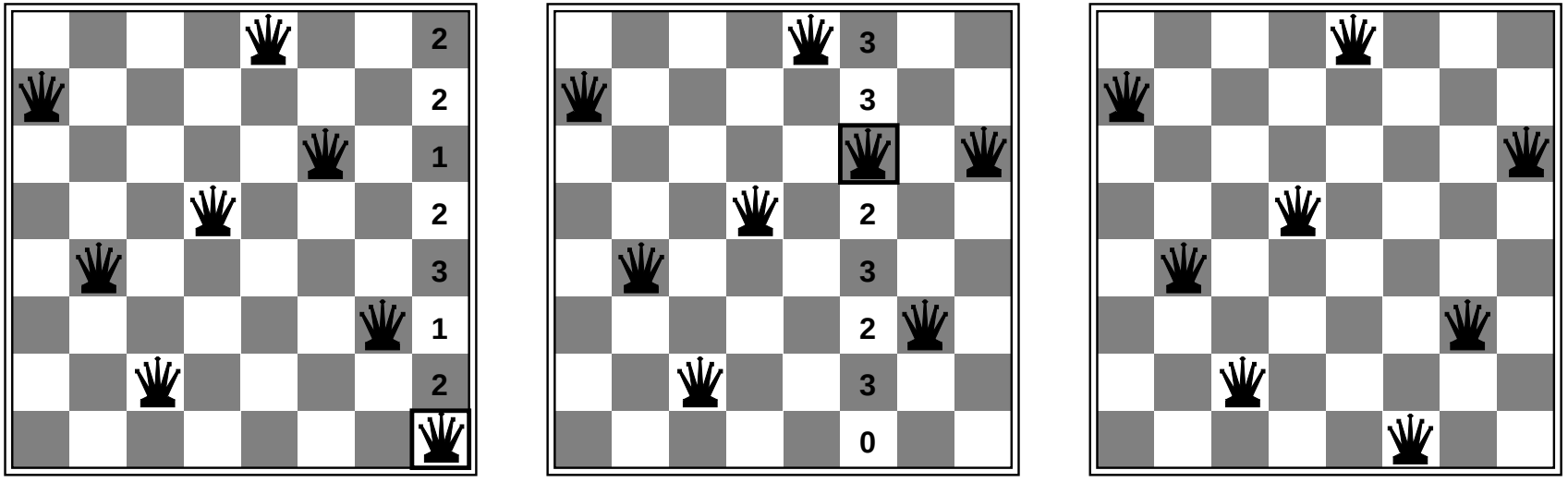
Example 8 queens problem:

- with a bound of 100 steps solution found in 96% of the cases
- on average 22 steps until solution found

$\rightsquigarrow$ works very well for this problem;
for more difficult problems often not good enough

Random Aspects in the Search Neighborhood

a possible variation of hill climbing for 8 queens:
Randomly select a file; move queen in this file
to square with minimal number of conflicts (null move possible).



$\rightsquigarrow$ Good local search approaches often combine
randomness (exploration) with heuristic guidance (exploitation).

German: Exploration, Exploitation

Outlook: Simulated Annealing

Simulated Annealing

Simulated annealing is a local search algorithm that systematically injects noise, beginning with high noise, then lowering it over time.

- walk with fixed number of steps N (variations possible)
- initially it is "hot", and the walk is mostly random
- over time temperature drops (controlled by a schedule)
- as it gets colder, moves to worse neighbors become less likely

very successful in some applications, e.g., VLSI layout

German: simulierte Abkühlung, Rauschen

Simulated Annealing: Pseudo-Code

Simulated Annealing (for Maximization Problems)

$curr :=$ a random candidate
$best :=$ **none**
for each $t \in \{1, \ldots, N\}$:
    if is_solution($curr$) **and** ($best$ **is none or** $v(curr) > v(best)$):
        $best := curr$
    $T :=$ schedule(t)
    $next :=$ a random neighbor of $curr$
    $\Delta E := h(next) - h(curr)$
    if $\Delta E \geq 0$ **or** with probability $e^{\frac{\Delta E}{T}}$:
        $curr := next$
return $best$

Outlook: Genetic Algorithms

Genetic Algorithms

Evolution often finds good solutions.

idea: simulate evolution by selection, crossover and mutation of individuals

ingredients:

- encode each candidate as a string of symbols (genome)
- fitness function: evaluates strength of candidates (= heuristic)
- population of k (e.g. 10–1000) individuals (candidates)

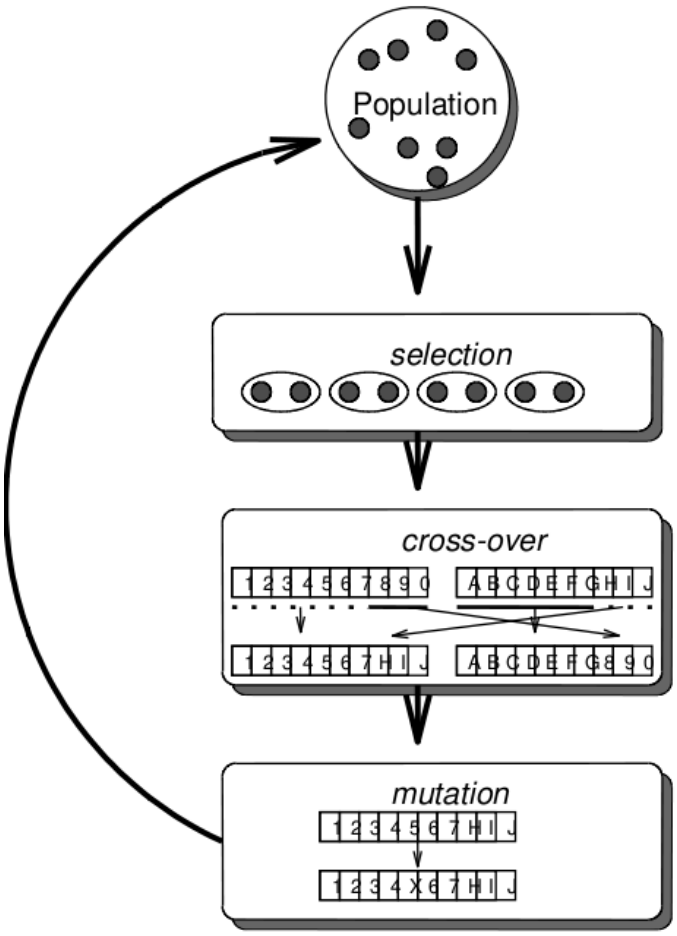
German: Evolution, Selektion, Kreuzung, Mutation, Genom, Fitnessfunktion, Population, Individuen

Genetic Algorithm: Example

example 8 queens problem:

- genome: encode candidate as string of 8 numbers
- fitness: number of non-attacking queen pairs
- use population of 100 candidates

Selection, Mutation and Crossover



many variants:

How to select?

How to perform crossover?

How to mutate?

select according to fitness function, followed by pairing

determine crossover points, then recombine

mutation: randomly modify each string position with a certain probability

Summary

Summary

- weakness of local search: local optima and plateaus
- remedy: balance exploration against exploitation
  (e.g., with randomness and restarts)
- simulated annealing and genetic algorithms
  are more complex search algorithms
  using the typical ideas of local search
  (randomization, keeping promising candidates)

Foundations of Artificial Intelligence

D1. Constraint Satisfaction Problems: Introduction and Examples

Malte Helmert

University of Basel

April 7, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
  - D1. Introduction and Examples
  - D2. Constraint Networks
- D3–D5. Basic Algorithms
- D6–D7. Problem Structure

Classification

classification:

Constraint Satisfaction Problems

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

Special case of a pure search combinatorial optimization problem

Introduction

Constraints

What is a Constraint?

a condition that every solution to a problem must satisfy

German: Einschränkung, Nebenbedingung (math.)

Examples: where do constraints occur?

- mathematics: requirements on solutions of optimization problems (e.g., equations, inequalities)
- software testing: specification of invariants to check data consistency (e.g., assertions)
- databases: integrity constraints

Constraint Satisfaction Problems: Informally

Given:

- set of variables with corresponding domains
- set of constraints that the variables must satisfy
  - most commonly binary, i.e., every constraint refers to two variables

Solution:

- assignment to the variables that satisfies all constraints

German: Variablen, Constraints, binär, Belegung

Examples

Examples

Examples

- 8 queens problem
- Latin squares
- Sudoku
- graph coloring
- satisfiability in propositional logic

German: 8-Damen-Problem, lateinische Quadrate, Sudoku, Graphfärbung, Erfüllbarkeitsproblem der Aussagenlogik

more complex examples:

- systems of equations and inequalities
- database queries

Example: 8 Queens Problem (Reminder)

(reminder from previous two chapters)

8 Queens Problem

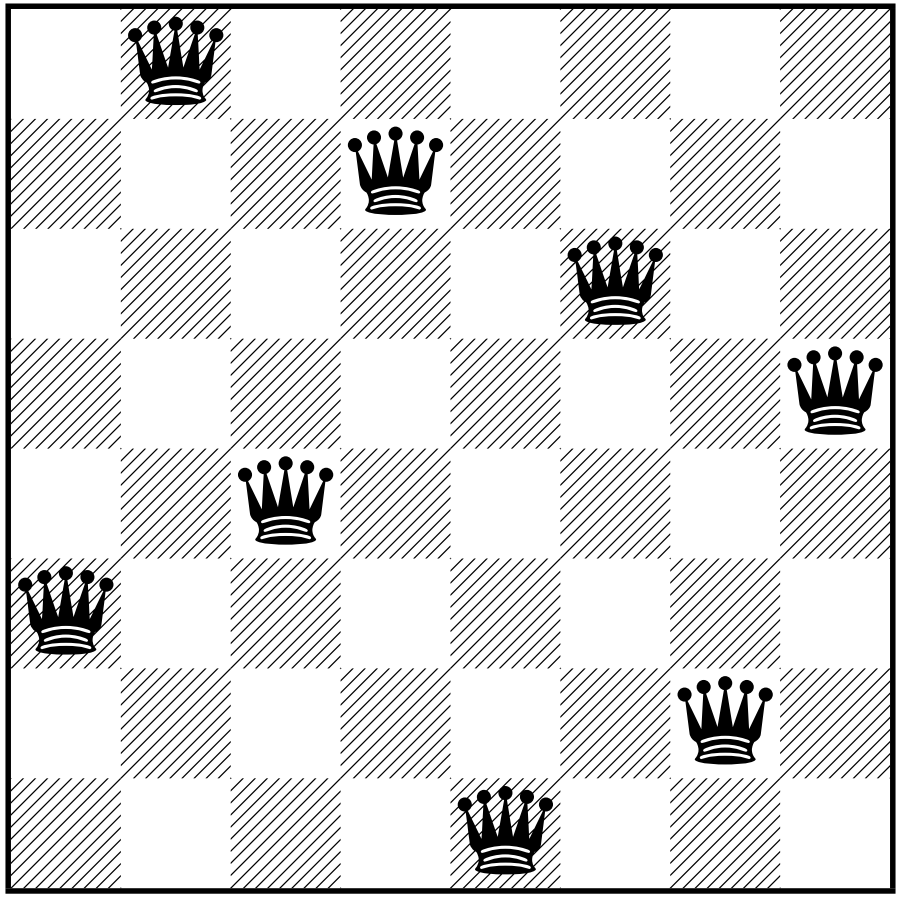
How can we

- place 8 queens on a chess board
- such that no two queens threaten each other?

- originally proposed in 1848
- variants: board size; other pieces; higher dimension

There are 92 solutions, or 12 solutions if we do not count symmetric solutions (under rotation or reflection) as distinct.

8 Queens Problem: Example Solution



example solution for the 8 queens problem

Example: Latin Squares

Latin Squares

How can we

- build an $n \times n$ matrix
  with n symbols
- such that every symbol occurs exactly once
  in every row and every column?

$$[1] \quad \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

There exist 12 different Latin squares of size 3,
576 of size 4, 161 280 of size 5, . . . ,
5 524 751 496 156 892 842 531 225 600 of size 9.

Example: Sudoku

Sudoku

How can we

- completely fill an already partially filled 9×9 matrix with numbers between 1–9
- such that each row, each column, and each of the nine 3×3 blocks contains every number exactly once?

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Example: Sudoku

Sudoku

How can we

- completely fill an already partially filled 9×9 matrix with numbers between 1–9
- such that each row, each column, and each of the nine 3×3 blocks contains every number exactly once?

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Example: Sudoku

Sudoku

How can we

- completely fill an already partially filled 9×9 matrix with numbers between 1–9
- such that each row, each column, and each of the nine 3×3 blocks contains every number exactly once?

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

relationship to Latin squares?

Sudoku: Trivia

- well-formed Sudokus have exactly one solution
- to achieve well-formedness, ≥ 17 cells must be filled already (McGuire et al., 2012)
- 6 670 903 752 021 072 936 960 solutions
- only 5 472 730 538 “non-symmetrical” solutions

Example: Graph Coloring

Graph Coloring

How can we

- color the vertices of a given graph using k colors
- such that two neighboring vertices never have the same color?

(The graph and k are problem parameters.)

Example: Graph Coloring

Graph Coloring

How can we

- color the vertices of a given graph using k colors
- such that two neighboring vertices never have the same color?

(The graph and k are problem parameters.)

NP-complete problem

- even for the special case of planar graphs and $k = 3$
- easy for $k = 2$ (also for general graphs)

Example: Graph Coloring

Graph Coloring

How can we

- color the vertices of a given graph using k colors
- such that two neighboring vertices never have the same color?

(The graph and k are problem parameters.)

NP-complete problem

- even for the special case of planar graphs and $k = 3$
- easy for $k = 2$ (also for general graphs)

Relationship to Sudoku?

Four Color Problem

famous problem in mathematics: Four Color Problem

- Is it always possible to color a planar graph with 4 colors?
- conjectured by Francis Guthrie (1852)
- 1890 first proof that 5 colors suffice
- several wrong proofs surviving for over 10 years

Four Color Problem

famous problem in mathematics: Four Color Problem

- Is it always possible to color a planar graph with 4 colors?
- conjectured by Francis Guthrie (1852)
- 1890 first proof that 5 colors suffice
- several wrong proofs surviving for over 10 years
- solved by Appel and Haken in 1976: 4 colors suffice
- Appel and Haken reduced the problem to 1936 cases, which were then checked by computers
- first famous mathematical problem solved (partially) by computers
  $\rightsquigarrow$ led to controversy: is this a mathematical proof?

Numberphile video:

`https://www.youtube.com/watch?v=NgbK43jB4rQ`

Satisfiability in Propositional Logic

Satisfiability in Propositional Logic

How can we

- assign truth values (true/false) to a set of propositional variables
- such that a given set of clauses (formulas of the form $X \vee \neg Y \vee Z$) is satisfied (true)?

Satisfiability in Propositional Logic

Satisfiability in Propositional Logic

How can we

- assign truth values (true/false) to a set of propositional variables
- such that a given set of clauses (formulas of the form $X \vee \neg Y \vee Z$) is satisfied (true)?

remarks:

- NP-complete (Cook 1971; Levin 1973)
- requiring clause form (instead of arbitrary propositional formulas) is no restriction
- clause length bounded by 3 would not be a restriction

Satisfiability in Propositional Logic

Satisfiability in Propositional Logic

How can we

- assign truth values (true/false) to a set of propositional variables
- such that a given set of clauses (formulas of the form $X \vee \neg Y \vee Z$) is satisfied (true)?

remarks:

- NP-complete (Cook 1971; Levin 1973)
- requiring clause form (instead of arbitrary propositional formulas) is no restriction
- clause length bounded by 3 would not be a restriction

relationship to previous problems (e.g., Sudoku)?

Practical Applications

- There are thousands of practical applications of constraint satisfaction problems.
- This statement is true already for the satisfiability problem of propositional logic.

some examples:

- verification of hardware and software
- timetabling (e.g., generating time schedules, room assignments for university courses)
- assignment of frequency spectra (e.g., broadcasting, mobile phones)

Running Example

Small Math Puzzle (informal description)

- assign a value from $\{1, 2, 3, 4\}$ to the variables w and y
- and from $\{1, 2, 3\}$ to x and z
- such that
  - $w = 2x$,
  - $w < z$ and
  - $y > z$.

We will use this example to explain definitions and algorithms in the next chapters.

Summary

Summary

- constraint satisfaction:
  - find assignment for a set of variables
  - with given variable domains
  - that satisfies a given set of constraints.
- examples:
  - 8 queens problem
  - Latin squares
  - Sudoku
  - graph coloring
  - satisfiability in propositional logic
  - many practical applications

Foundations of Artificial Intelligence

D2. Constraint Satisfaction Problems: Constraint Networks

Malte Helmert

University of Basel

April 7, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
  - D1. Introduction and Examples
  - D2. Constraint Networks
- D3–D5. Basic Algorithms
- D6–D7. Problem Structure

Constraint Networks

Constraint Networks: Informally

Constraint Networks: Informal Definition

A constraint network is defined by

- a finite set of variables
- a finite domain for each variable
- a set of constraints (here: binary relations)

The objective is to find a solution for the constraint network, i.e., an assignment of the variables that complies with all constraints.

Informally, people often just speak of constraint satisfaction problems (CSP) instead of constraint networks.

More formally, a "CSP" is the algorithmic problem of finding a solution for a constraint network.

Constraint Networks: Formally

Definition (binary constraint network)

A (binary) constraint network
is a 3-tuple $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ such that:

- V is a non-empty and finite set of variables,
- dom is a function that assigns a non-empty and finite domain to each variable $v \in V$, and
- $(R_{uv})_{u,v \in V, u \neq v}$ is a family of binary relations (constraints) over V where for all $u \neq v$: $R_{uv} \subseteq \mathrm{dom}(u) \times \mathrm{dom}(v)$

German: (binäres) Constraint-Netz, Variablen, Wertebereich, Constraints

possible generalizations:

- infinite domains (e.g., $\mathrm{dom}(v) = \mathbb{Z}$)
- constraints of higher arity
  (e.g., satisfiability in propositional logic)

Variables and Domains

Running Example (informally)

- assign a value from $\{1, 2, 3, 4\}$ to the variables w and y
- and from $\{1, 2, 3\}$ to x and z
- such that ...

Running Example (formally)

$\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ with

- $V = \{w, x, y, z\}$
- $\mathrm{dom}(w) = \mathrm{dom}(y) = \{1, 2, 3, 4\}$
- $\mathrm{dom}(x) = \mathrm{dom}(z) = \{1, 2, 3\}$
- ...

Binary Constraints (1)

binary constraints:

- For variables u, v, the constraint R_{uv} expresses which joint assignments to u and v are allowed in a solution.

Running Example (informally)

- ... such that
  - $\ldots, w < z, \ldots$

Running Example (formally)

$\ldots, R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}, \ldots$

Binary Constraints (2)

binary constraints:

- If $R_{uv} = \mathrm{dom}(u) \times \mathrm{dom}(v)$, the constraint is trivial: there is no restriction, and the constraint is typically not given explicitly in the constraint network description (although it formally always exists!).

Running Example

$$\ldots, R_{xz} = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle\}, \ldots$$

Binary Constraints (3)

binary constraints:

- Constraints R_{uv} and R_{vu} refer to the same variables.
  Hence, usually only one of them is given in the description.

Running Example (informally)

- ... such that
  - $\ldots, w < z, \ldots$

Running Example (formally)

$\ldots, R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}, \ldots$
$\ldots, R_{zw} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle\}, \ldots$

Unary Constraints

unary constraints:

- It is often useful to have additional restrictions on single variables as constraints.
- Such constraints are called unary constraints.
- A unary constraint R_v for $v \in V$ corresponds to a restriction of $\mathrm{dom}(v)$ to the values allowed by R_v.
- Formally, unary constraints are not necessary, but they often allow us to describe constraint networks more clearly.

German: unäre Constraints

Running Example

$\mathrm{dom}(z) = \{1, 2, 3\}$ could be described as
$\mathrm{dom}(z) = \{1, 2, 3, 4\}, R_z = \{1, 2, 3\}$

Example

Full Formal Model of Running Example

$\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ with

- variables:
  $V = \{w, x, y, z\}$
- domains:
  $\mathrm{dom}(w) = \mathrm{dom}(y) = \{1, 2, 3, 4\}$
  $\mathrm{dom}(x) = \mathrm{dom}(z) = \{1, 2, 3\}$
- constraints:
  $R_{wx} = \{\langle 2, 1\rangle, \langle 4, 2\rangle\}$
  $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$
  $R_{yz} = \{\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle,$
  $\langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle\}$

Compact Encodings and General Constraint Solvers

Constraint networks allow for compact encodings of large sets of assignments:

- Consider a network with n variables with domains of size k.

$\rightsquigarrow$ k^n assignments

Compact Encodings and General Constraint Solvers

Constraint networks allow for compact encodings of large sets of assignments:

- Consider a network with n variables with domains of size k.

$\rightsquigarrow$ k^n assignments

- For the description as constraint network, at most $\binom{n}{2}$, i.e., $O(n^2)$ constraints have to be provided. Every constraint in turn consists of at most $O(k^2)$ pairs.

$\rightsquigarrow$ encoding size $O(n^2 k^2)$

- We observe: The number of assignments is exponentially larger than the description of the constraint network.

Compact Encodings and General Constraint Solvers

Constraint networks allow for compact encodings of large sets of assignments:

- Consider a network with n variables with domains of size k.

$\rightsquigarrow$ k^n assignments

- For the description as constraint network, at most $\binom{n}{2}$, i.e., $O(n^2)$ constraints have to be provided.
  Every constraint in turn consists of at most $O(k^2)$ pairs.

$\rightsquigarrow$ encoding size $O(n^2 k^2)$

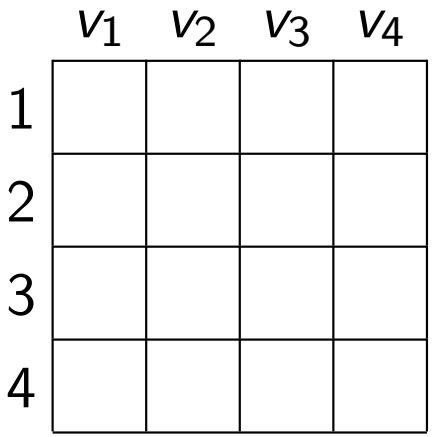
- We observe: The number of assignments is exponentially larger than the description of the constraint network.
- As a consequence, such descriptions can be used as inputs of general constraint solvers.

Examples

Example: 4 Queens Problem

4 Queens Problem as Constraint Network

- variables: $V = \{v_1, v_2, v_3, v_4\}$
  v_i encodes the rank of the queen in the i-th file
- domains:
  $\mathrm{dom}(v_1) = \mathrm{dom}(v_2) = \mathrm{dom}(v_3) = \mathrm{dom}(v_4) = \{1, 2, 3, 4\}$
- constraints: for all $1 \leq i < j \leq 4$, we set: $R_{v_i,v_j} = \{\langle k, l\rangle \in \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \mid k \neq l \wedge |k - l| \neq |i - j|\}$
  e.g. $R_{v_1,v_3} = \{\langle 1, 2\rangle, \langle 1, 4\rangle, \langle 2, 1\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle, \langle 3, 4\rangle, \langle 4, 1\rangle, \langle 4, 3\rangle\}$



Example: Sudoku

Sudoku as Constraint Network

- variables: $V = \{v_{ij} \mid 1 \leq i, j \leq 9\}$; v_{ij}: Value row i, column j
- domains: $\mathrm{dom}(v) = \{1, \ldots, 9\}$ for all $v \in V$
- unary constraints: $R_{v_{ij}} = \{k\}$,
  if $\langle i, j \rangle$ is a cell with predefined value k
- binary constraints: for all $v_{ij}, v_{i'j'} \in V$, we set
  $R_{v_{ij}v_{i'j'}} = \{\langle a, b \rangle \in \{1, \ldots, 9\} \times \{1, \ldots, 9\} \mid a \neq b\}$,
  if $i = i'$ (same row), or $j = j'$ (same column),
  or $\langle \lceil \frac{i}{3} \rceil, \lceil \frac{j}{3} \rceil \rangle = \langle \lceil \frac{i'}{3} \rceil, \lceil \frac{j'}{3} \rceil \rangle$ (same block)

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Assignments and Consistency

Assignments

Definition (assignment, partial assignment)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.
A partial assignment of $\mathcal{C}$ (or of V) is a function

$$\alpha : V' \to \bigcup_{v \in V} \mathrm{dom}(v)$$

with $V' \subseteq V$ and $\alpha(v) \in \mathrm{dom}(v)$ for all $v \in V'$.

If $V' = V$, then α is also called total assignment (or assignment).

German: partielle Belegung, (totale) Belegung

- $\rightsquigarrow$ partial assignments assign values to some or to all variables
- $\rightsquigarrow$ (total) assignments are defined on all variables

Example

Partial Assignments of Running Example

$\alpha_1 = \{w \mapsto 1, z \mapsto 2\}$
$\alpha_2 = \{w \mapsto 3, x \mapsto 1\}$

Total Assignments of Running Example

$\alpha_3 = \{w \mapsto 1, x \mapsto 1, y \mapsto 2, z \mapsto 2\}$
$\alpha_4 = \{w \mapsto 2, x \mapsto 1, y \mapsto 4, z \mapsto 3\}$

Consistency

Definition (inconsistent, consistent, violated)

A partial assignment α of a constraint network $\mathcal{C}$ is called inconsistent if there are variables u, v such that α is defined for both u and v, and $\langle \alpha(u), \alpha(v) \rangle \notin R_{uv}$.

In this case, we say α violates the constraint R_{uv}.

A partial assignment is called consistent if it is not inconsistent.

German: inkonsistent, verletzt, konsistent

trivial example: The empty assignment is always consistent.

Example

Consistent Partial Assignment

$\alpha_1 = \{w \mapsto 1, z \mapsto 2\}$

Inconsistent Partial Assignment

$\alpha_2 = \{w \mapsto 2, x \mapsto 2\}$
violates $R_{wx} = \{\langle 2, 1\rangle, \langle 4, 2\rangle\}$

Inconsistent Assignment

$\alpha_3 = \{w \mapsto 2, x \mapsto 1, y \mapsto 2, z \mapsto 2\}$
violates $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$ and
$R_{yz} = \{\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle\}$

Solution

Definition (solution, solvable)

Let $\mathcal{C}$ be a constraint network.

A consistent and total assignment of $\mathcal{C}$ is called a solution of $\mathcal{C}$.

If a solution of $\mathcal{C}$ exists, $\mathcal{C}$ is called solvable.

If no solution exists, $\mathcal{C}$ is called inconsistent.

German: Lösung, lösbar, inkonsistent

Solution of the Running Example

$\alpha = \{w \mapsto 2, x \mapsto 1, y \mapsto 4, z \mapsto 3\}$

Consistency vs. Solvability

Note: Consistent partial assignments α cannot necessarily be extended to a solution.

It only means that so far (i.e., on the variables where α is defined) no constraint is violated.

Example (4 queens problem): $\alpha = \{v_1 \mapsto 1, v_2 \mapsto 4, v_3 \mapsto 2\}$

	v_1	v_2	v_3	v_4
1	q			
2			q	
3				
4		q		

Complexity of Constraint Satisfaction Problems

Proposition (CSPs are NP-complete)

It is an NP-complete problem to decide whether a given constraint network is solvable.

Proof

Membership in NP:

Guess and check: guess a solution and check it for validity.
This can be done in polynomial time in the size of the input.

NP-hardness:

The graph coloring problem is a special case of CSPs and is already known to be NP-complete.

Tightness of Constraint Networks

Definition (tighter, strictly tighter)

Let $\mathcal{C} = \langle V, \mathrm{dom}, R_{uv} \rangle$ and $\mathcal{C}' = \langle V, \mathrm{dom}', R'_{uv} \rangle$ be constraint networks with equal variable sets V.

$\mathcal{C}$ is called tighter than $\mathcal{C}'$, in symbols $\mathcal{C} \sqsubseteq \mathcal{C}'$, if

- $\mathrm{dom}(v) \subseteq \mathrm{dom}'(v)$ for all $v \in V$, and
- $R_{uv} \subseteq R'_{uv}$ for all $u, v \in V$ (including trivial constraints).

If at least one of these subset equations is strict,
then $\mathcal{C}$ is called strictly tighter than $\mathcal{C}'$, in symbols $\mathcal{C} \sqsubset \mathcal{C}'$.

German: (echt) schärfer

Equivalence of Constraint Networks

Definition (equivalent)

Let $\mathcal{C}$ and $\mathcal{C}'$ be constraint networks with equal variable sets.

$\mathcal{C}$ and $\mathcal{C}'$ are called equivalent, in symbols $\mathcal{C} \equiv \mathcal{C}'$,
if they have the same solutions.

German: äquivalent

Outline and Summary

CSP Algorithms

In the following chapters, we will consider solution algorithms for constraint networks.

basic concepts:

- search: check partial assignments systematically
- backtracking: discard inconsistent partial assignments
- inference: derive equivalent, but tighter constraints to reduce the size of the search space

Summary

- formal definition of constraint networks: variables, domains, constraints
- compact encodings of exponentially many configurations
- unary and binary constraints
- assignments: partial and total
- consistency of assignments; solutions
- deciding solvability is NP-complete
- tightness of constraints
- equivalence of constraints

Foundations of Artificial Intelligence

D3. Constraint Satisfaction Problems: Backtracking

Malte Helmert

University of Basel

April 9, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
- D3–D5. Basic Algorithms
  - D3. Backtracking
  - D4. Arc Consistency
  - D5. Path Consistency
- D6–D7. Problem Structure

CSP Algorithms

CSP Algorithms

In the following chapters, we consider algorithms for solving constraint networks.

basic concepts:

- search: check partial assignments systematically
- backtracking: discard inconsistent partial assignments
- inference: derive equivalent, but tighter constraints to reduce the size of the search space

Naive Backtracking

Naive Backtracking (= Without Inference)

function NaiveBacktracking($\mathcal{C}, \alpha$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

select some variable v for which α is not defined

for each $d \in \text{dom}(v)$ in some order:
    $\alpha' := \alpha \cup \{v \mapsto d\}$
    $\alpha'' :=$ NaiveBacktracking($\mathcal{C}, \alpha'$)
    if $\alpha'' \neq$ **inconsistent**:
        return α''
return inconsistent

input: constraint network $\mathcal{C}$ and partial assignment α for $\mathcal{C}$
(first invocation: empty assignment $\alpha = \emptyset$)
result: solution of $\mathcal{C}$ or **inconsistent**

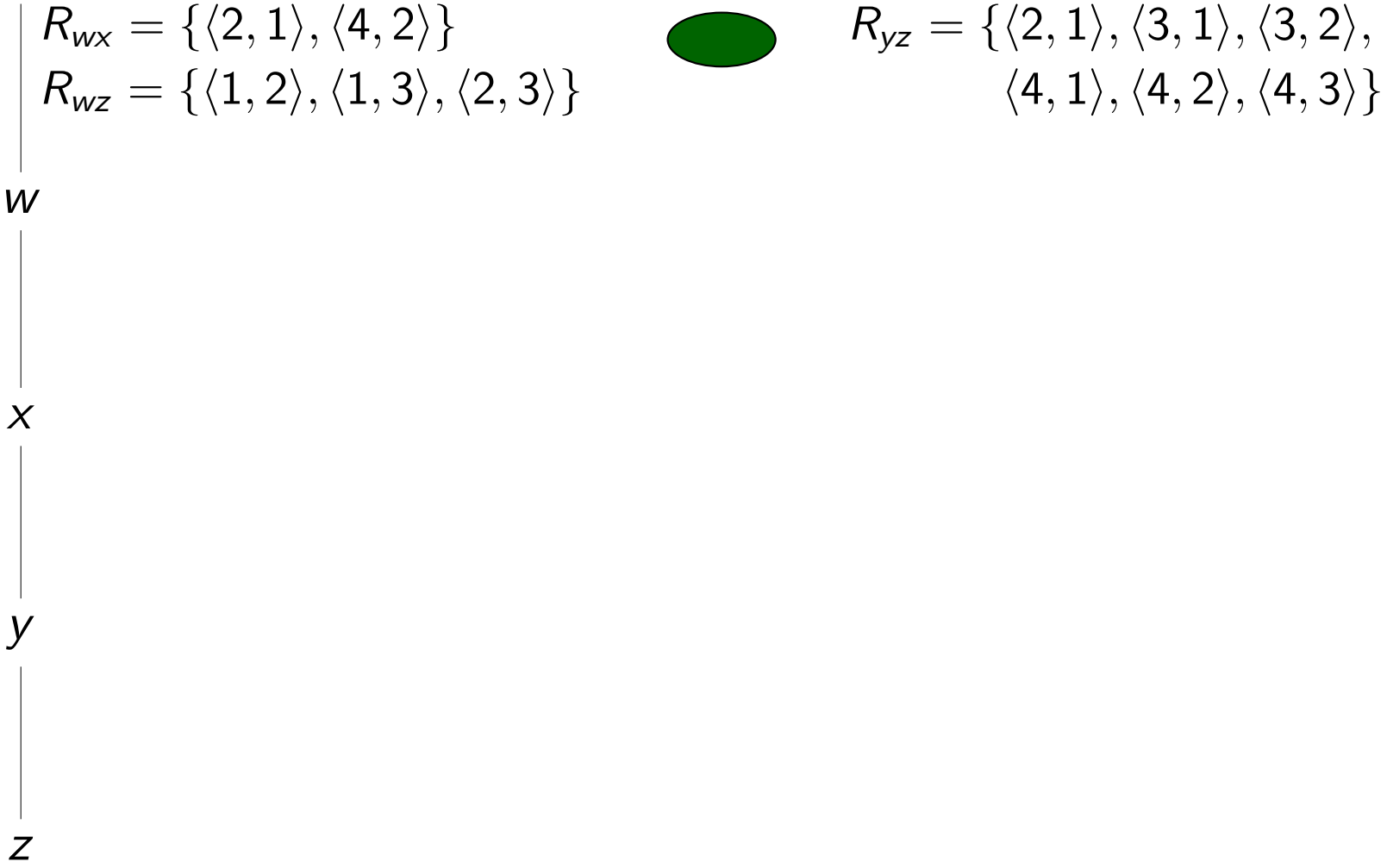
Running Example

Full Formal Model of Running Example

$\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ with

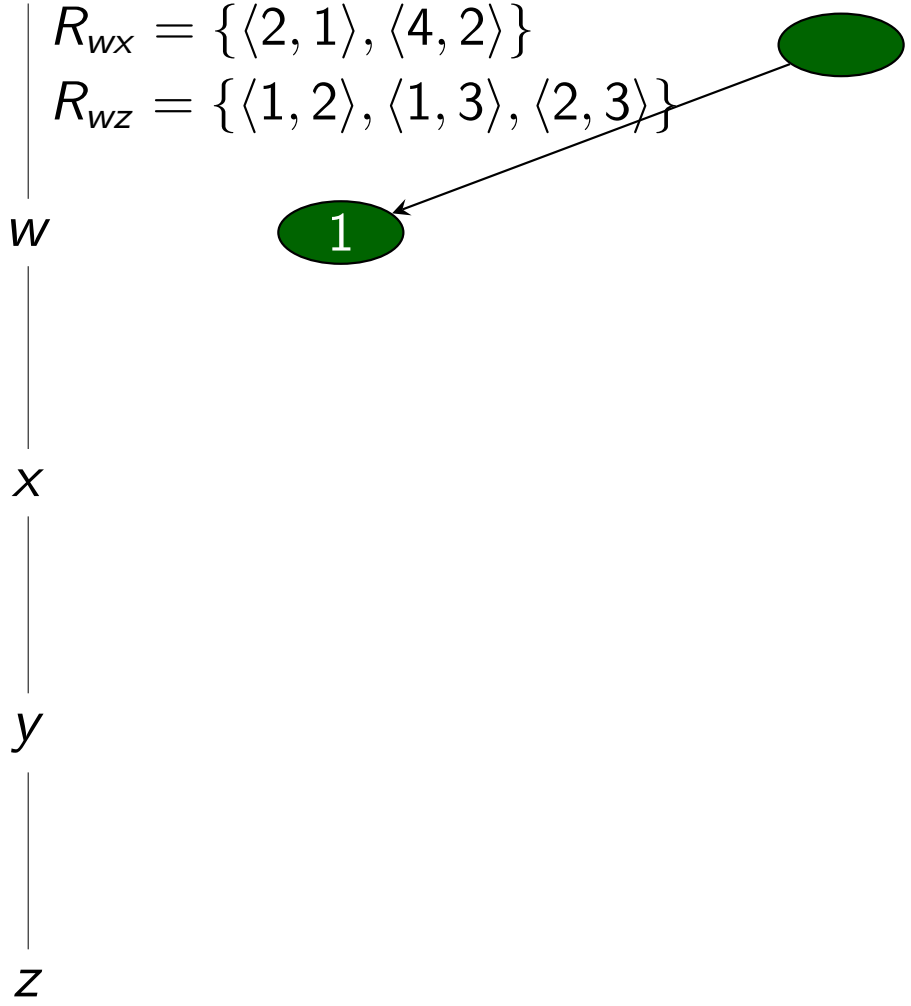
- variables:
  $V = \{w, x, y, z\}$
- domains:
  $\mathrm{dom}(w) = \mathrm{dom}(y) = \{1, 2, 3, 4\}$
  $\mathrm{dom}(x) = \mathrm{dom}(z) = \{1, 2, 3\}$
- constraints:
  $R_{wx} = \{\langle 2, 1\rangle, \langle 4, 2\rangle\}$
  $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$
  $R_{yz} = \{\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle,$
  $\langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle\}$

Running Example: Search Tree



Running Example: Search Tree

$R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

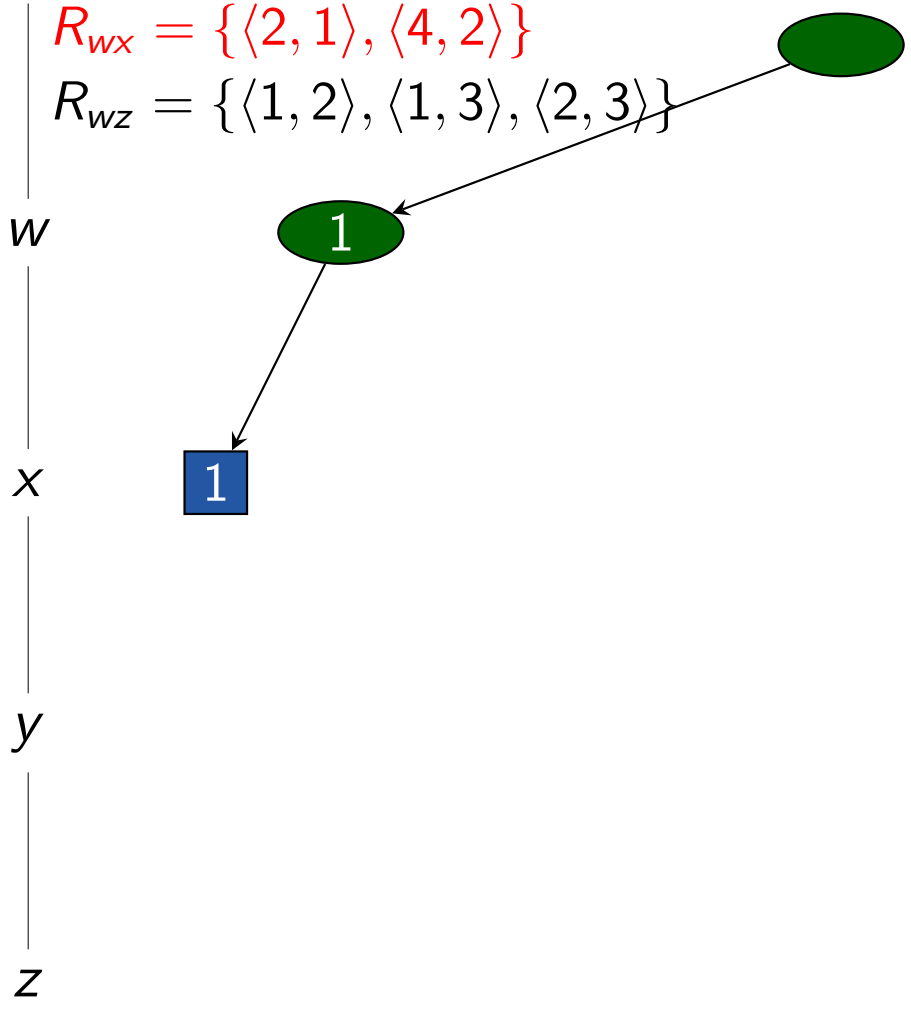


Running Example: Search Tree

$R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

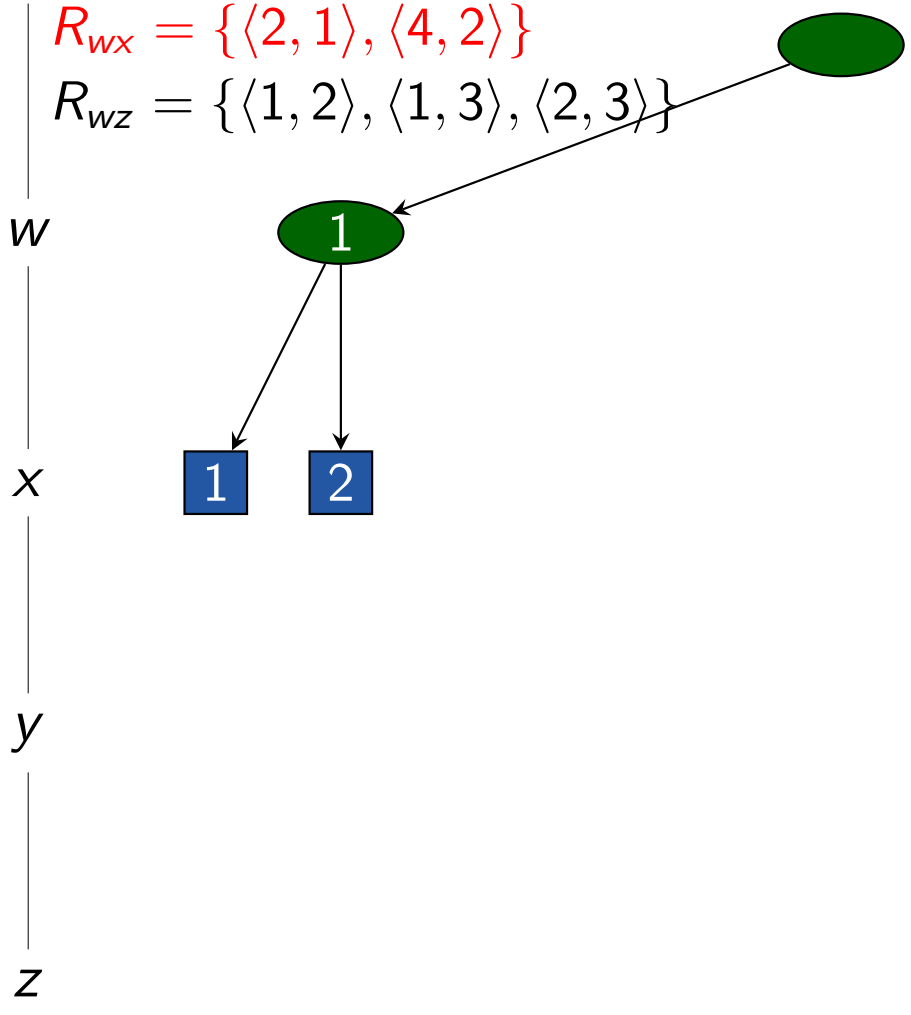


Running Example: Search Tree

$R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

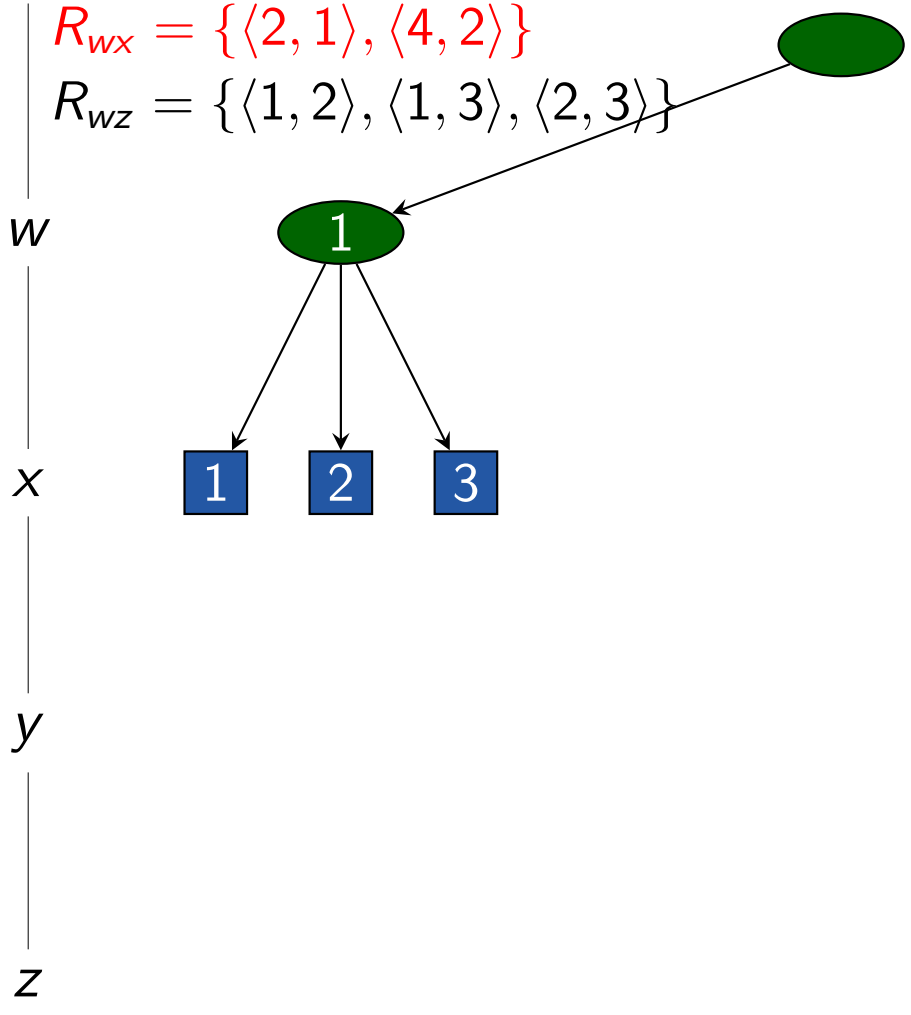


Running Example: Search Tree

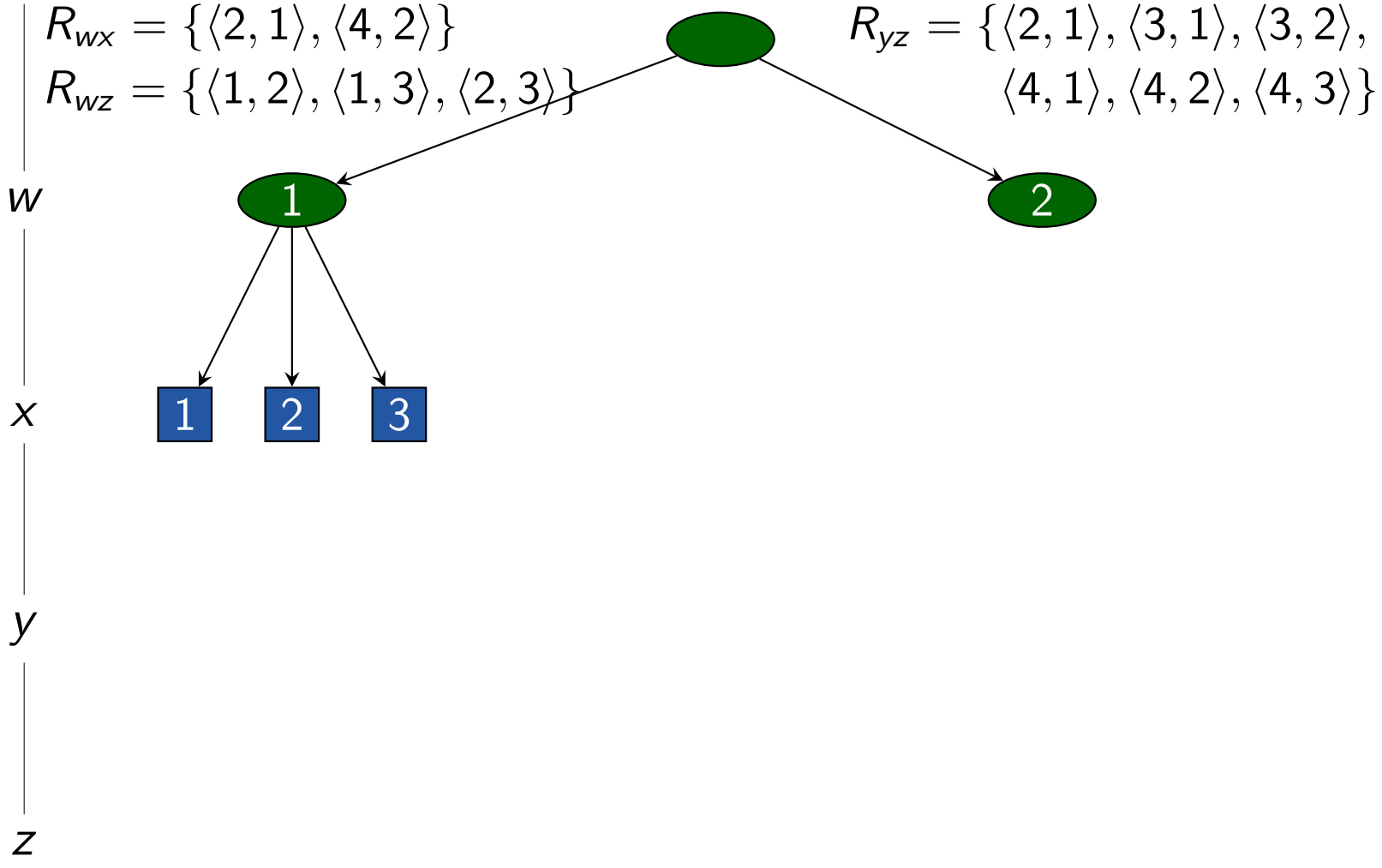
$R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

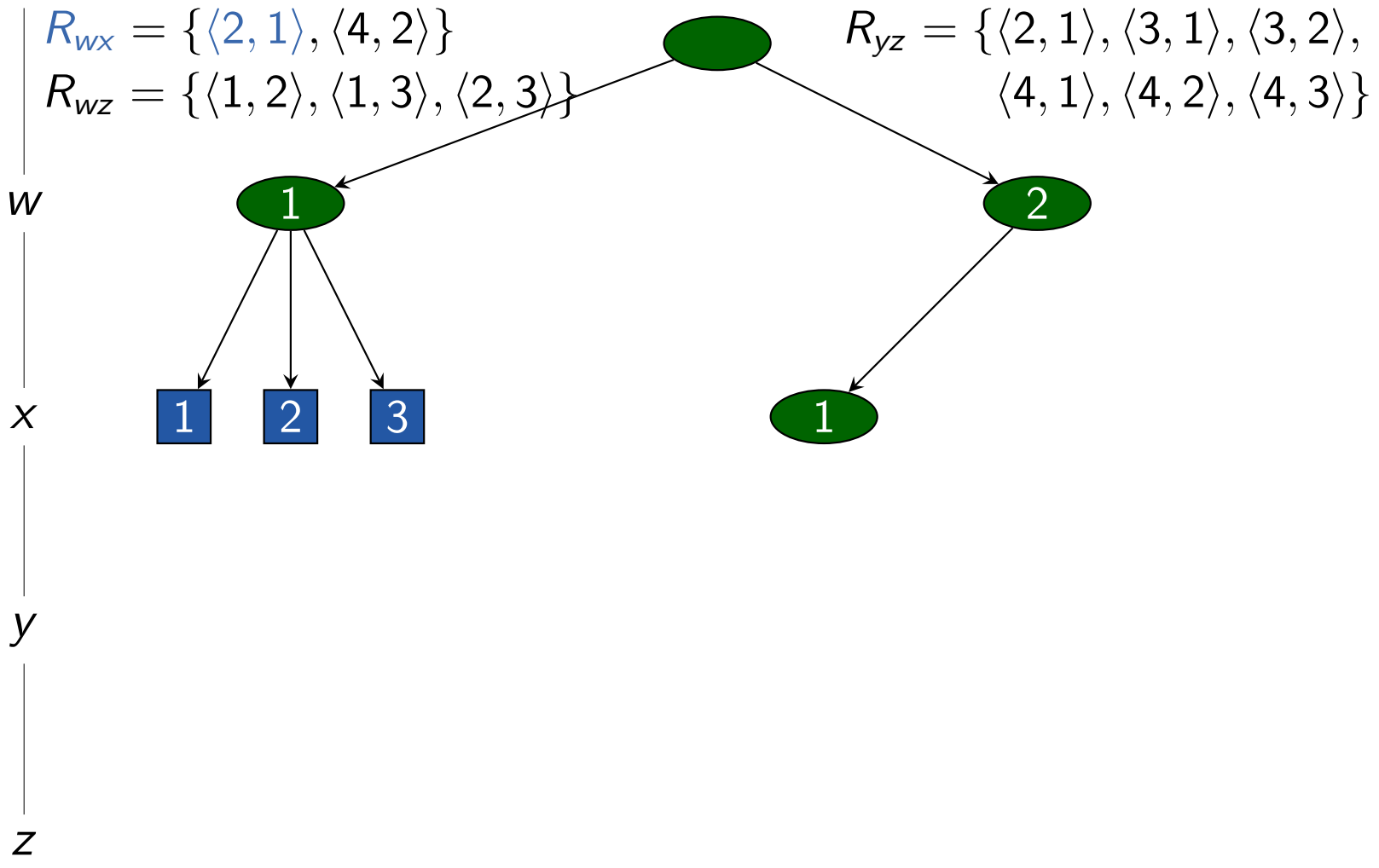
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$



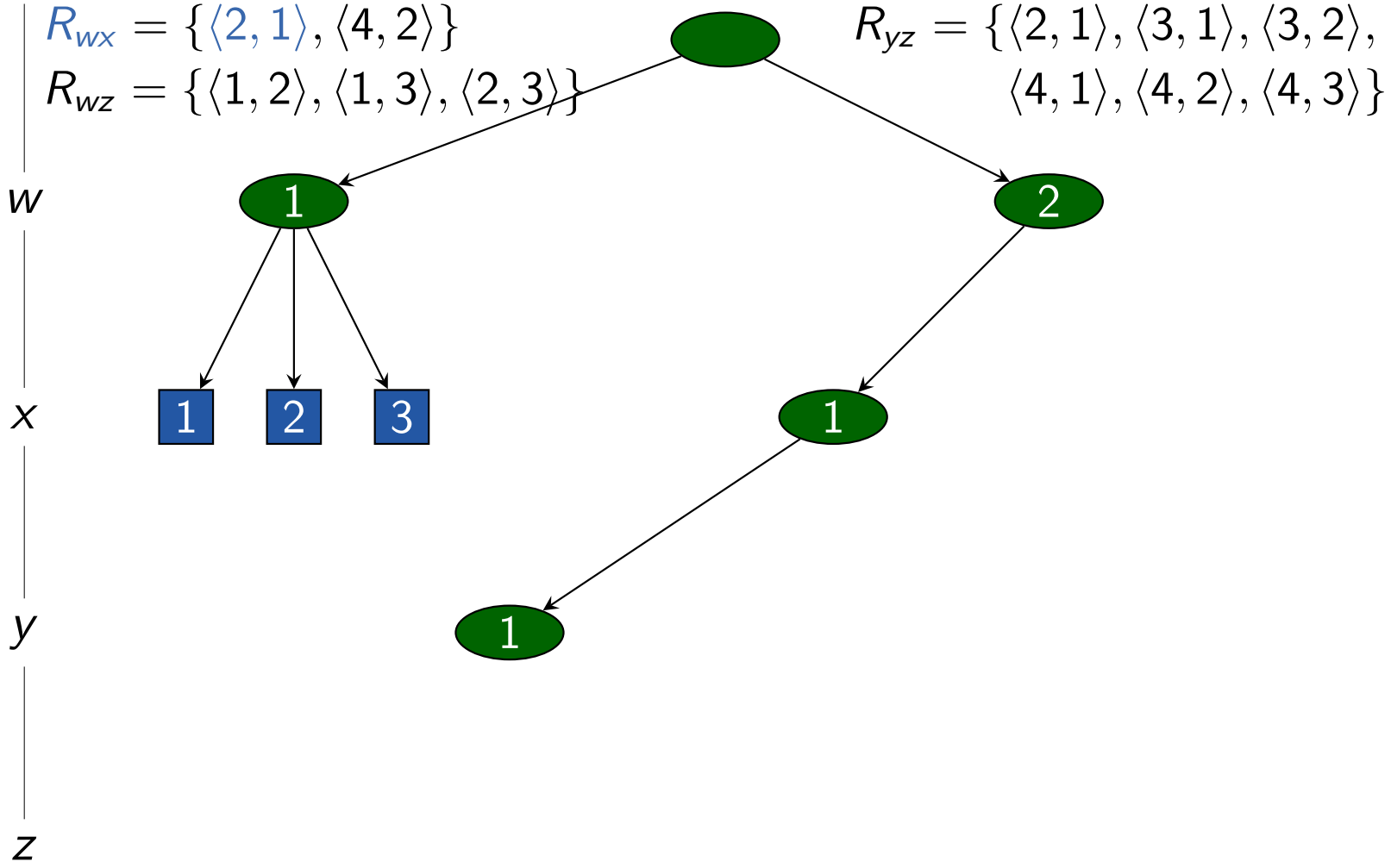
Running Example: Search Tree



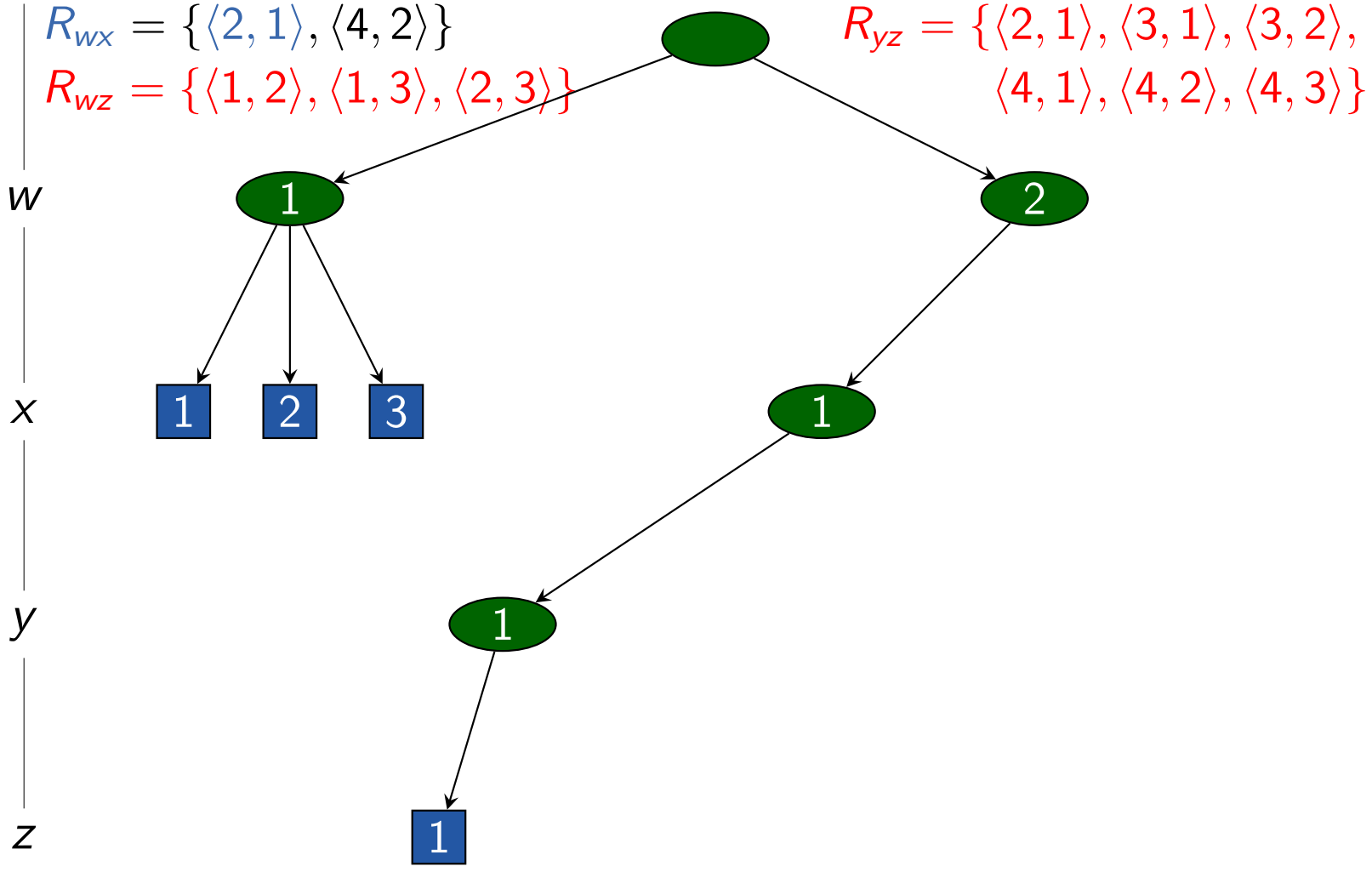
Running Example: Search Tree



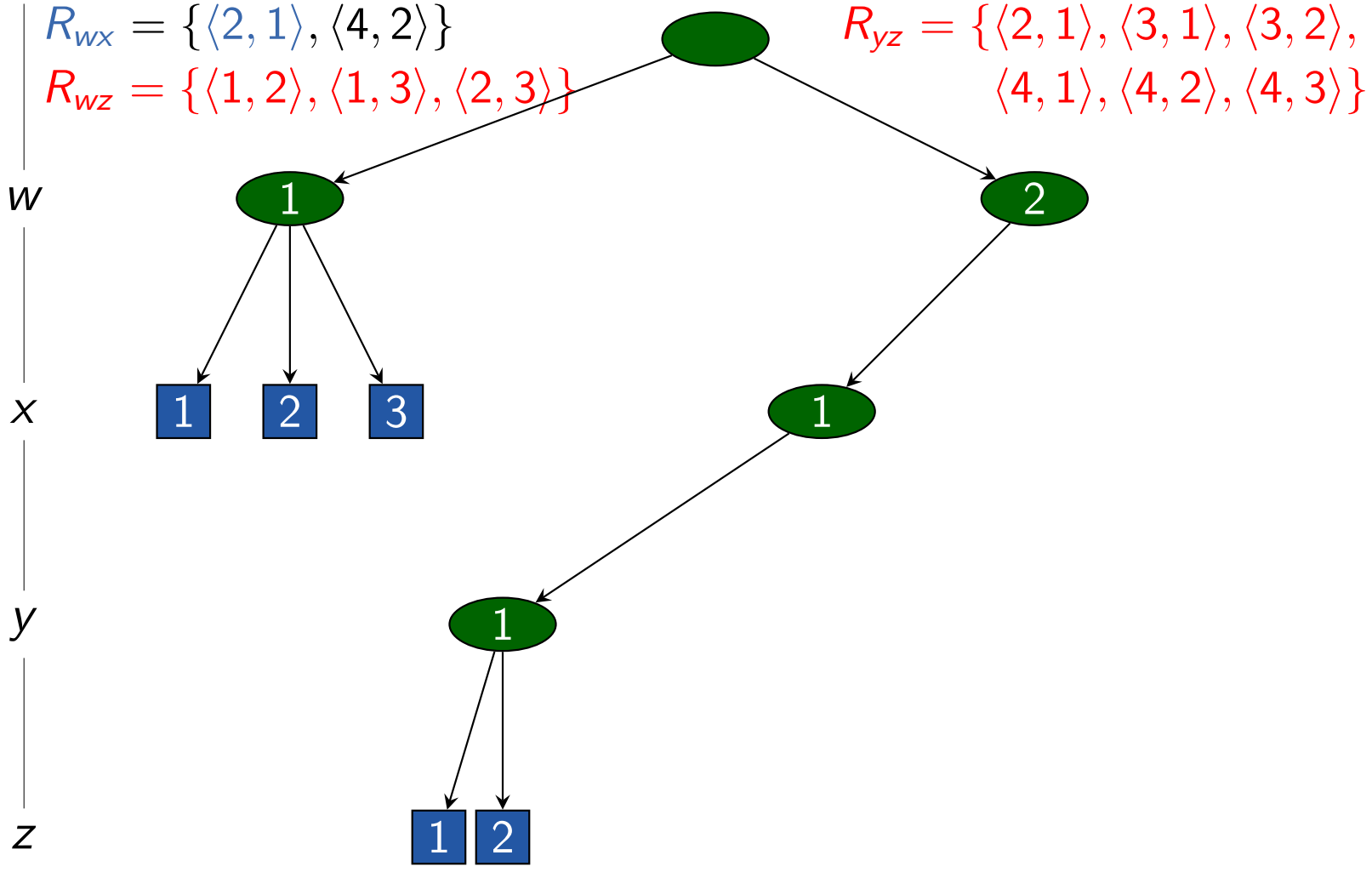
Running Example: Search Tree



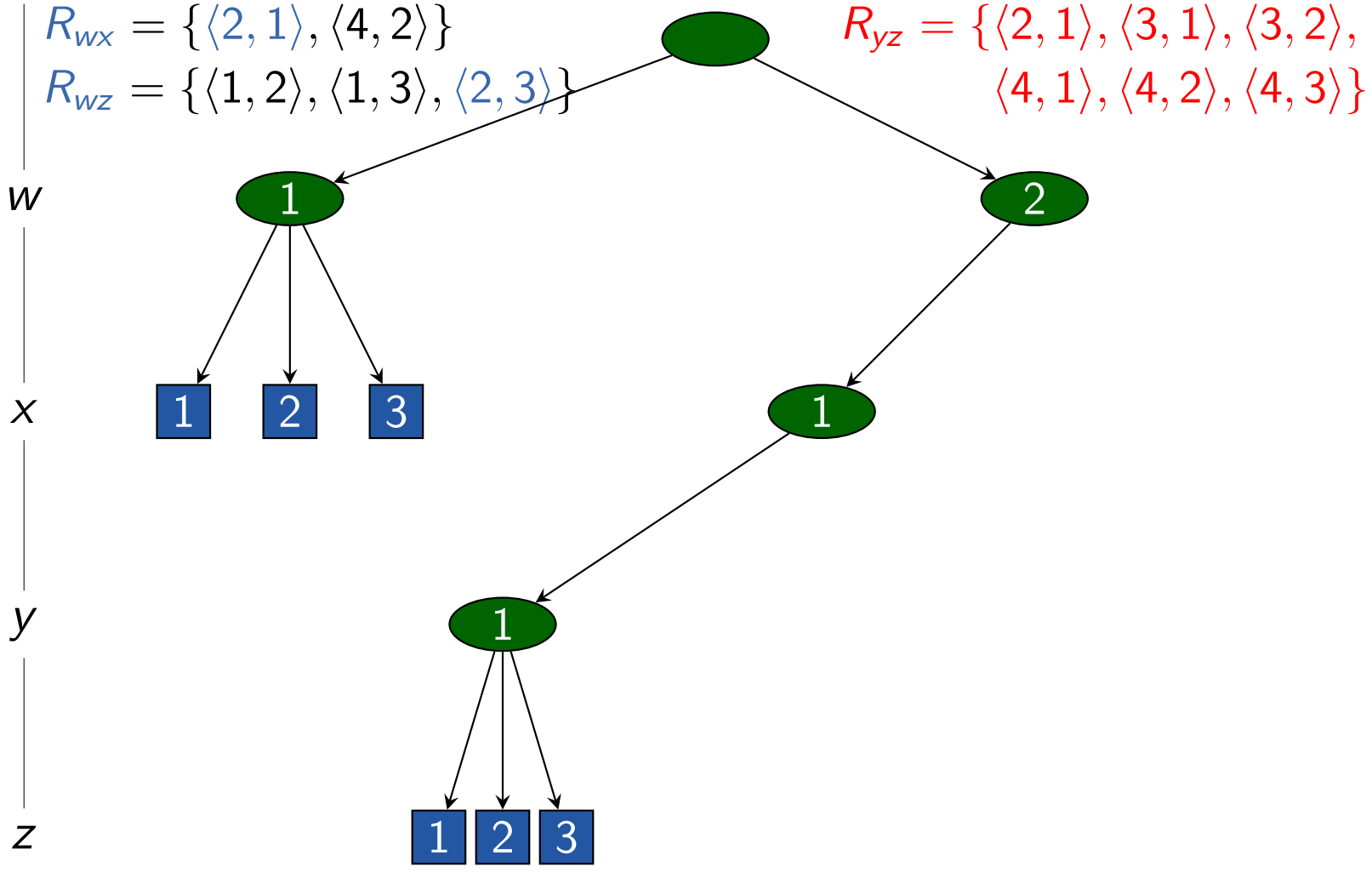
Running Example: Search Tree



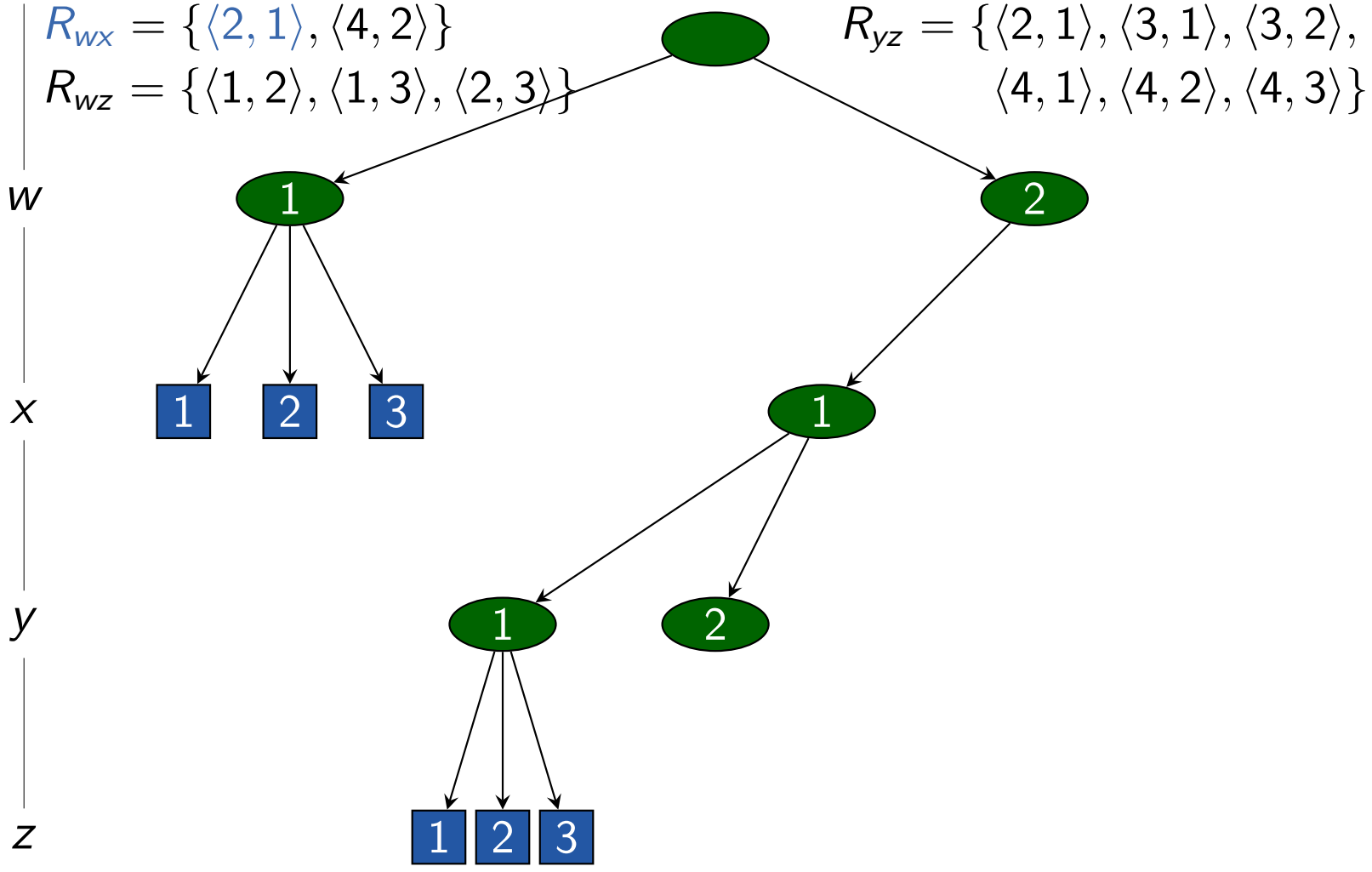
Running Example: Search Tree



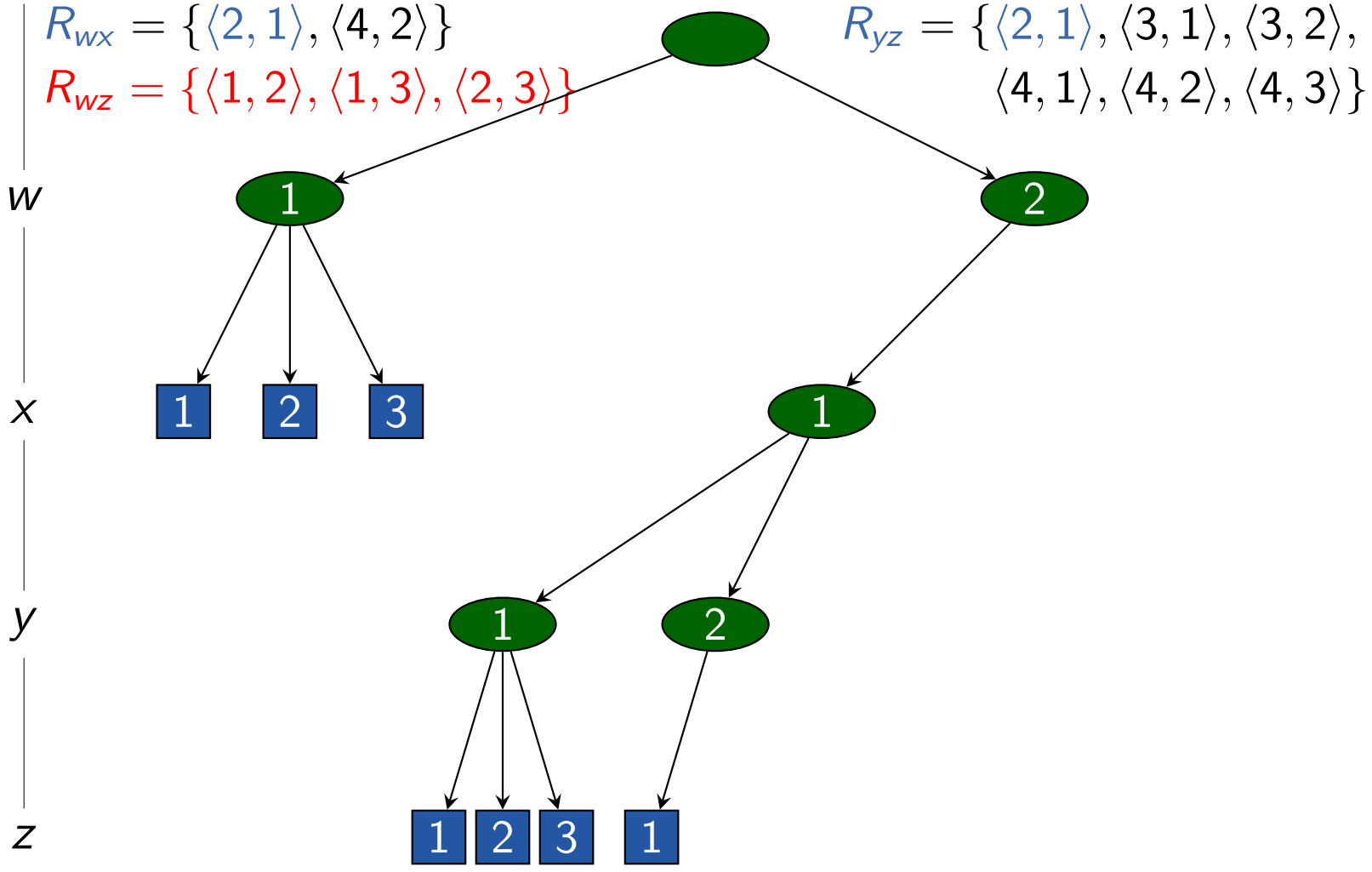
Running Example: Search Tree



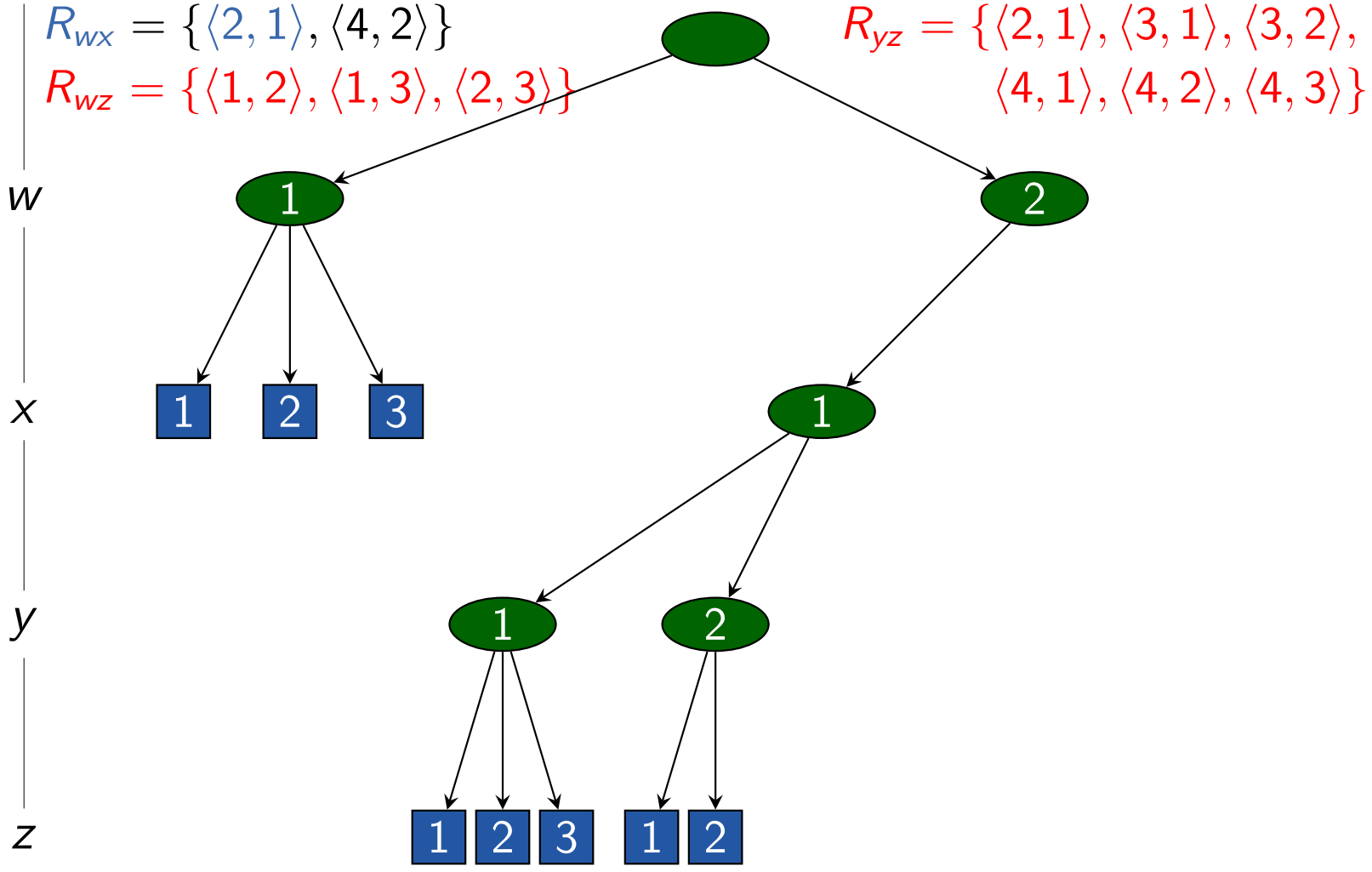
Running Example: Search Tree



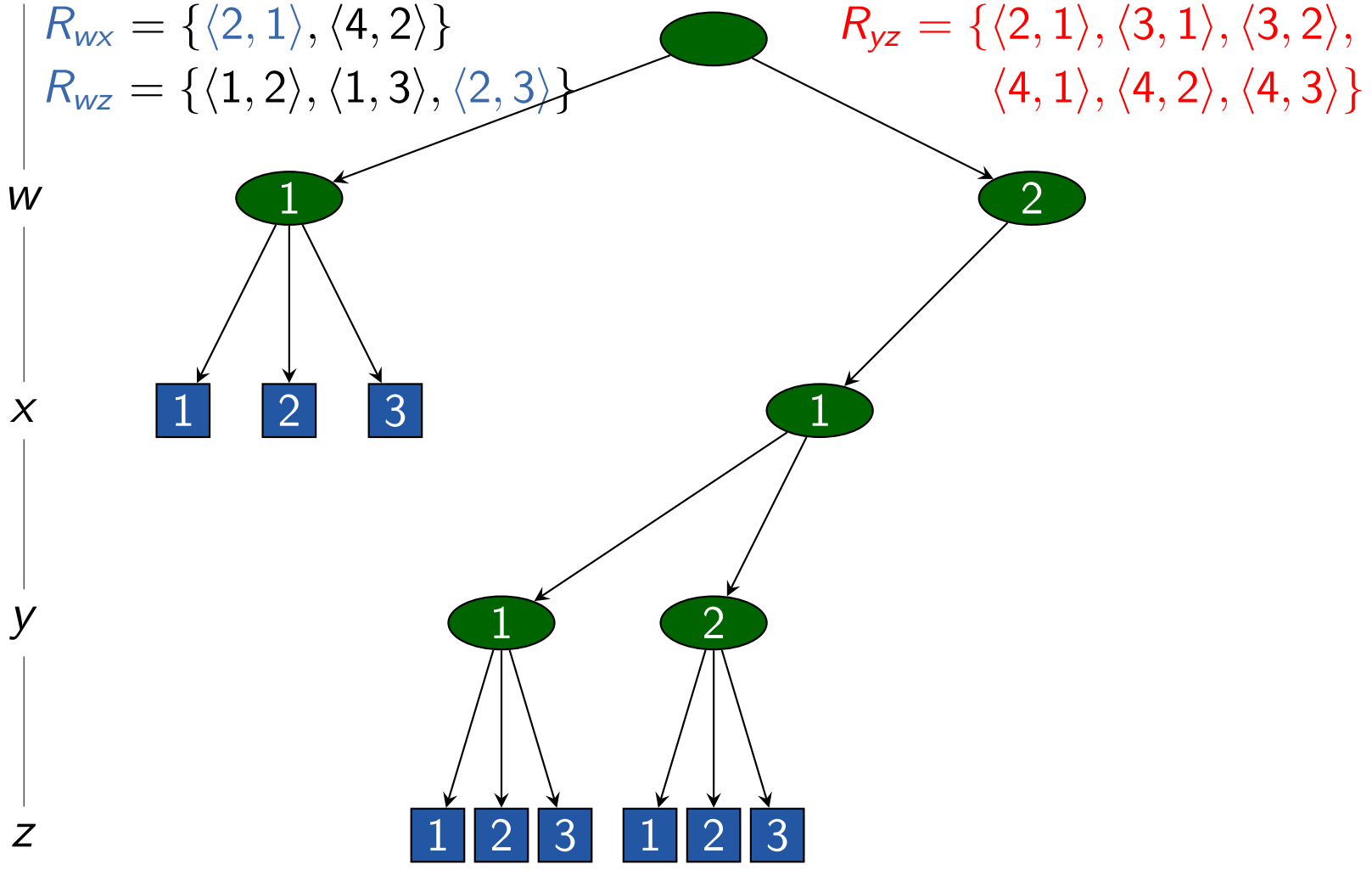
Running Example: Search Tree



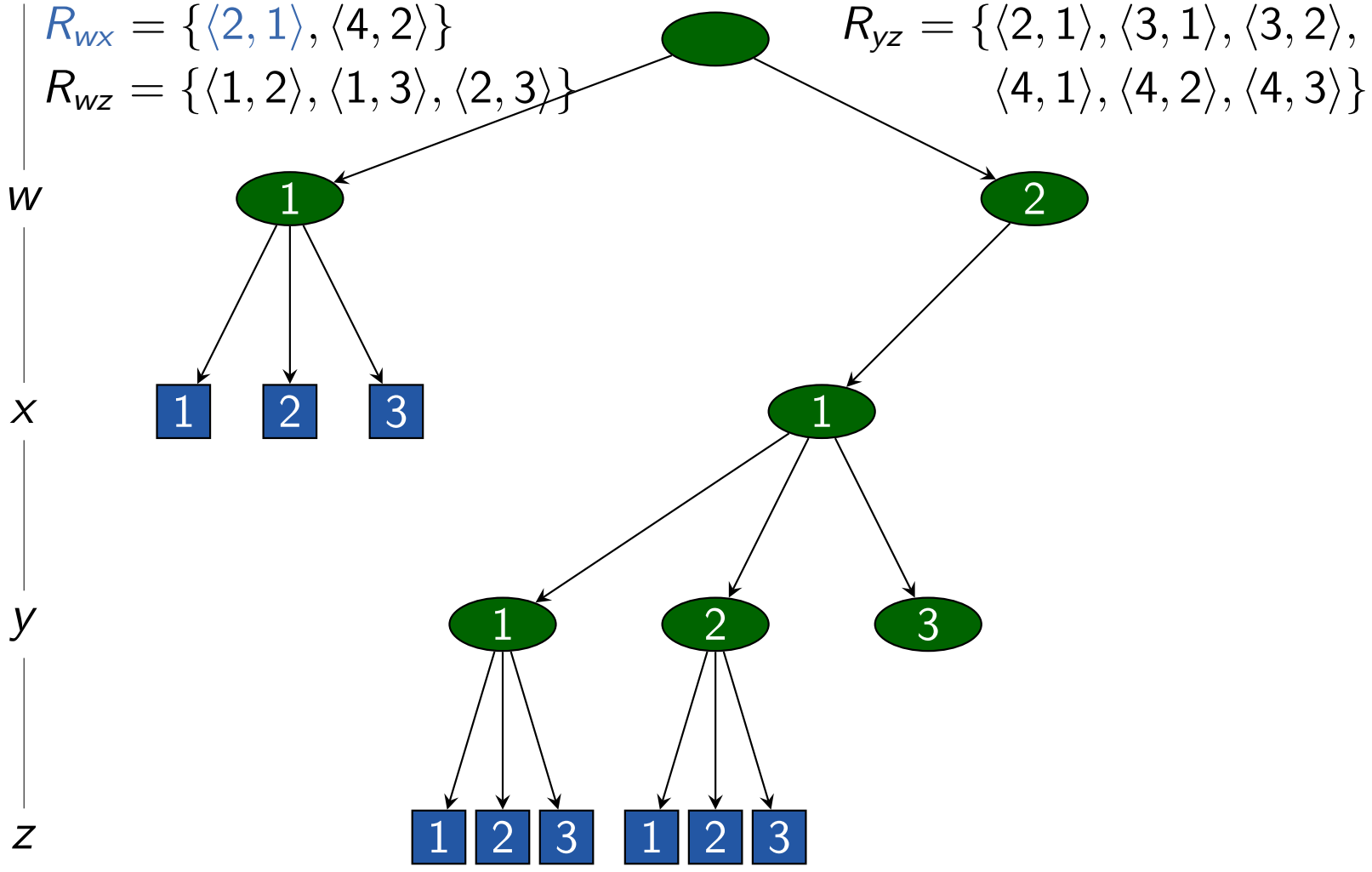
Running Example: Search Tree



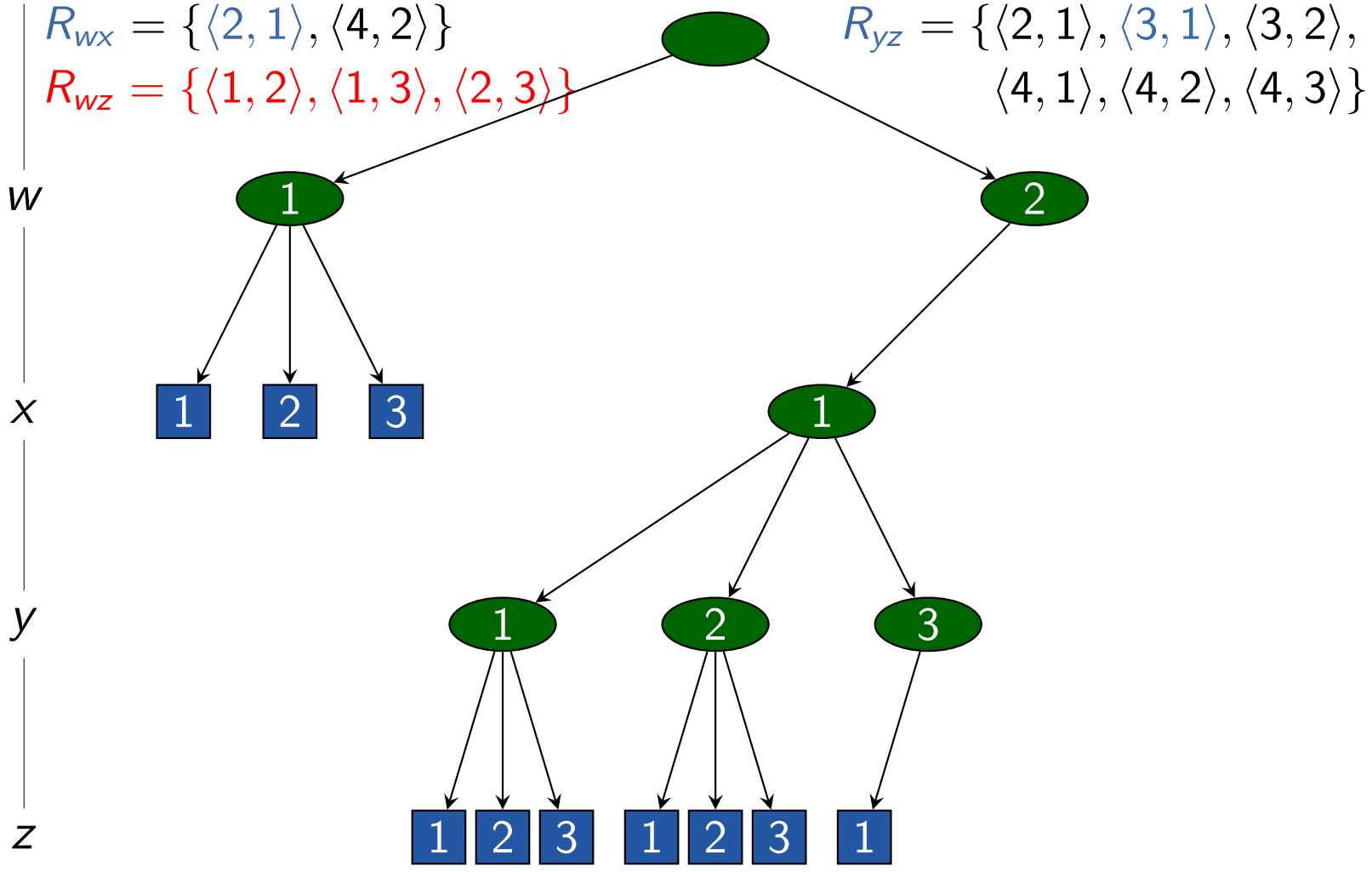
Running Example: Search Tree



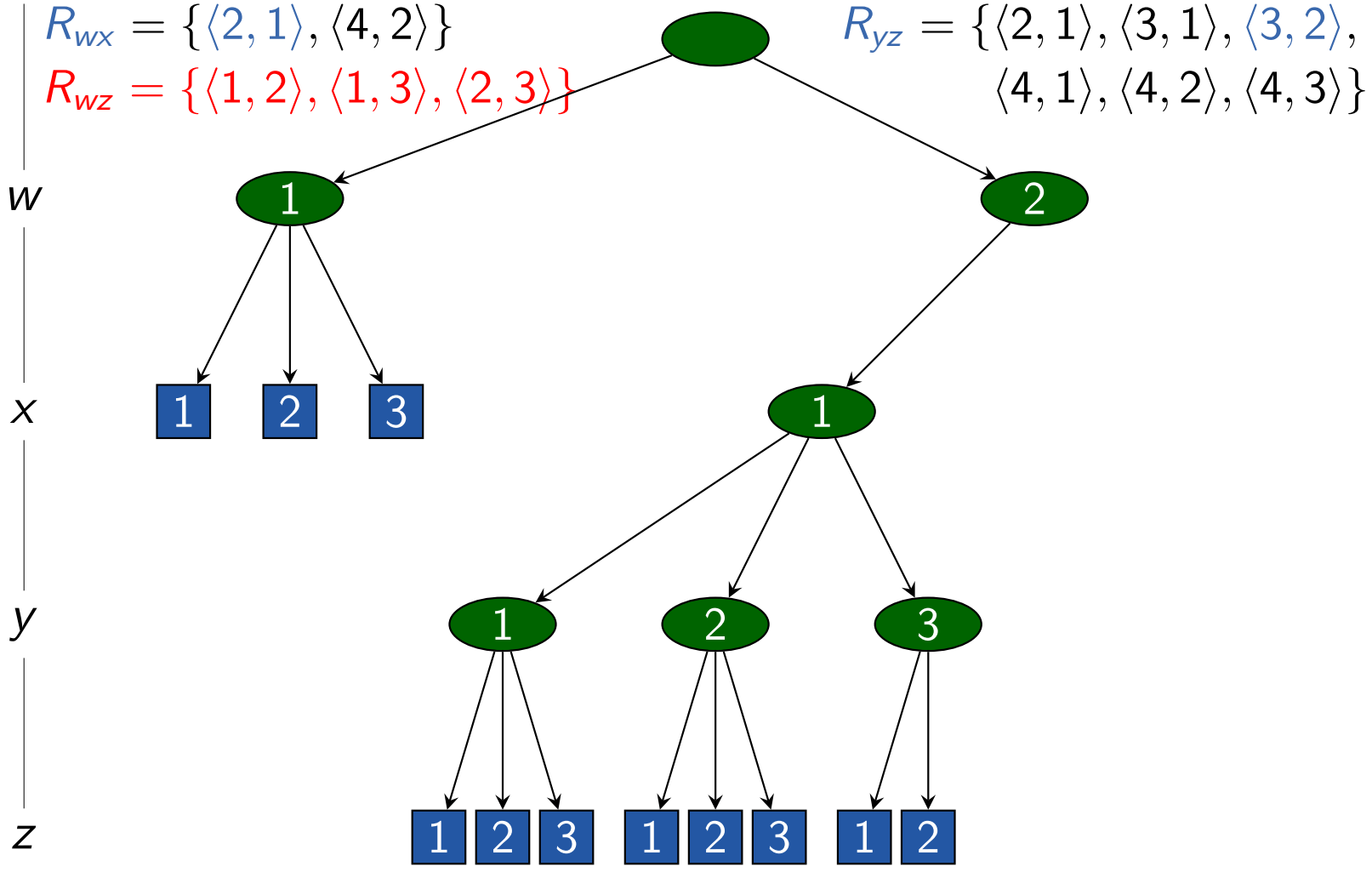
Running Example: Search Tree



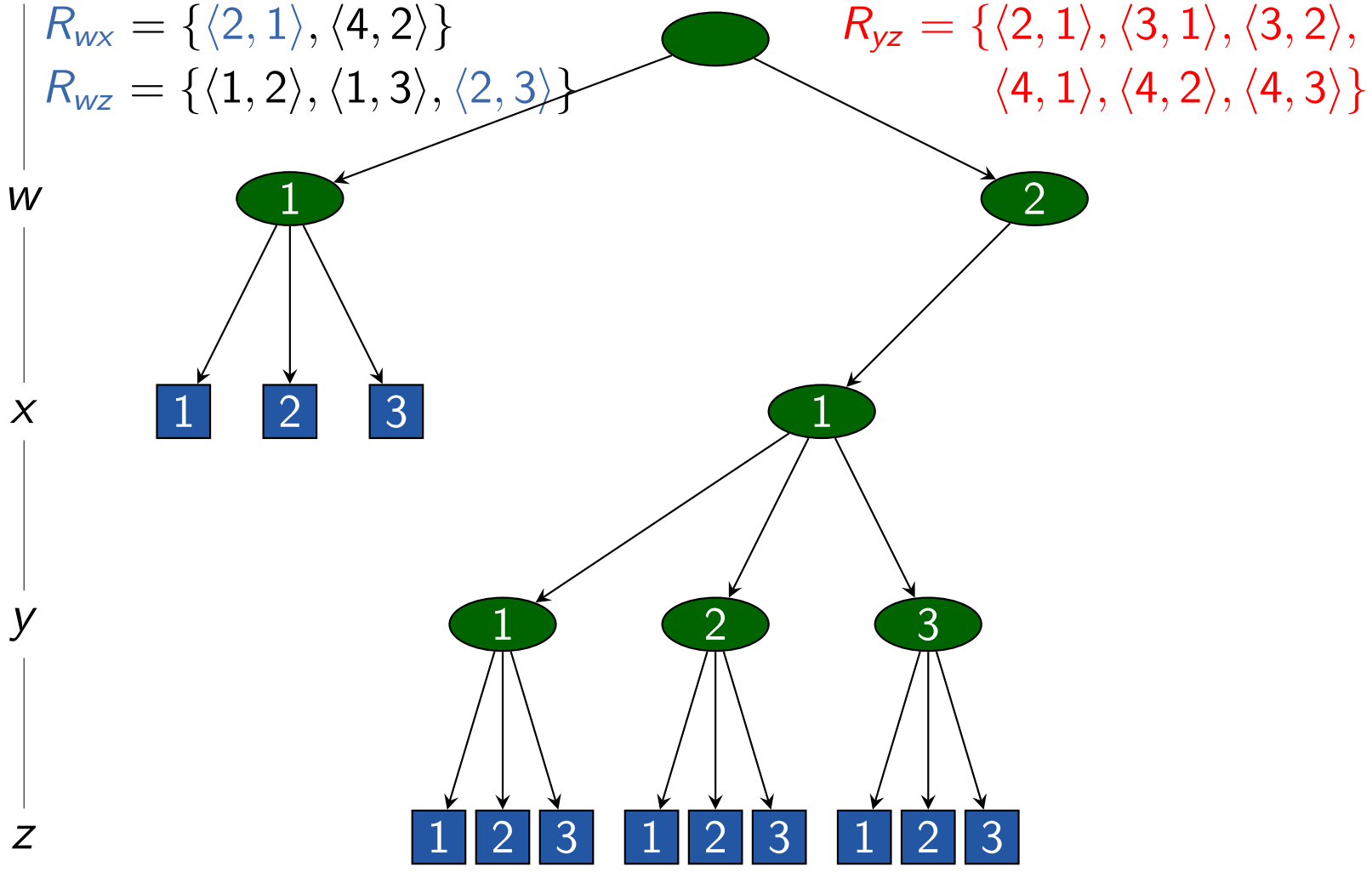
Running Example: Search Tree



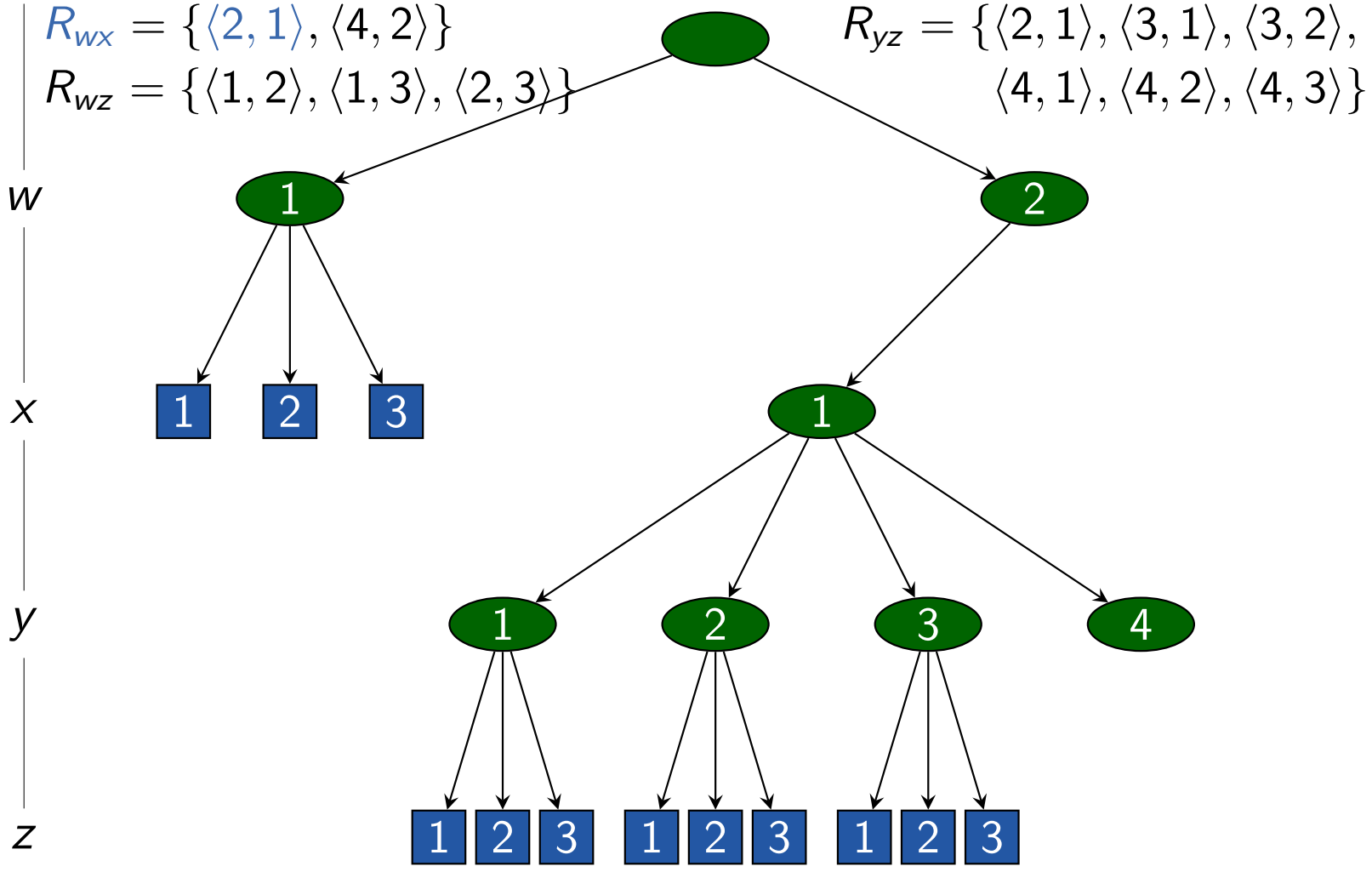
Running Example: Search Tree



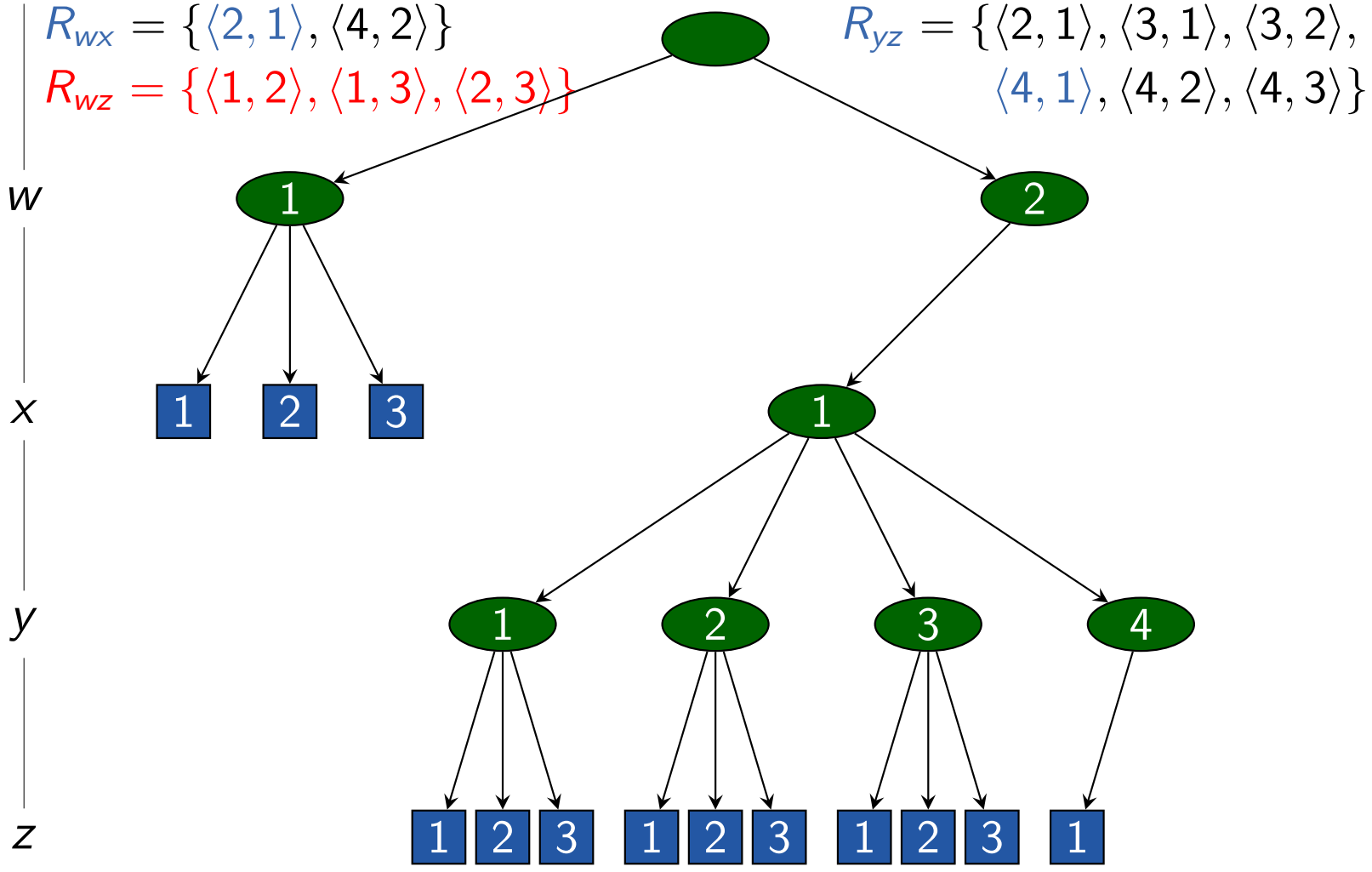
Running Example: Search Tree



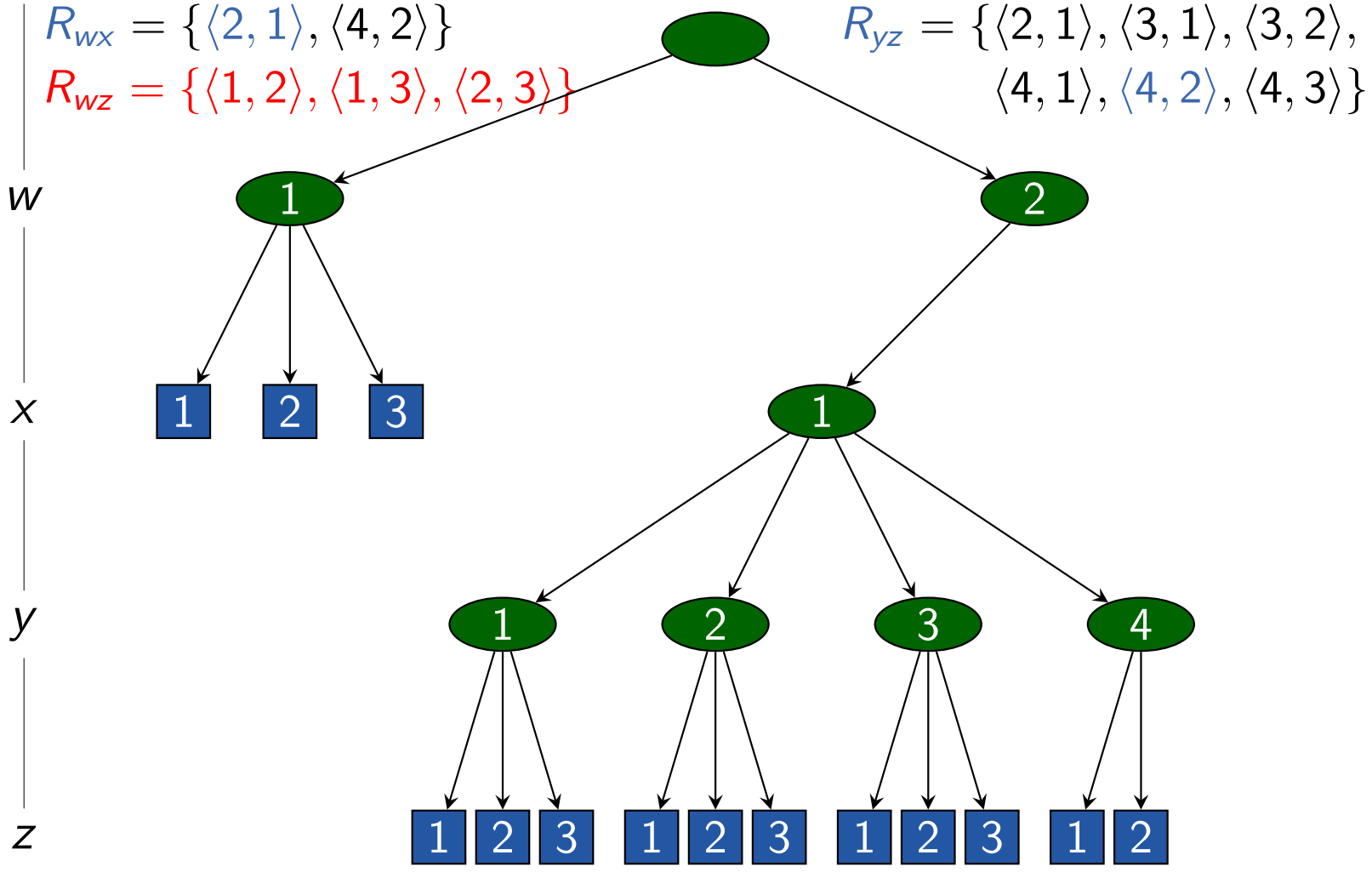
Running Example: Search Tree



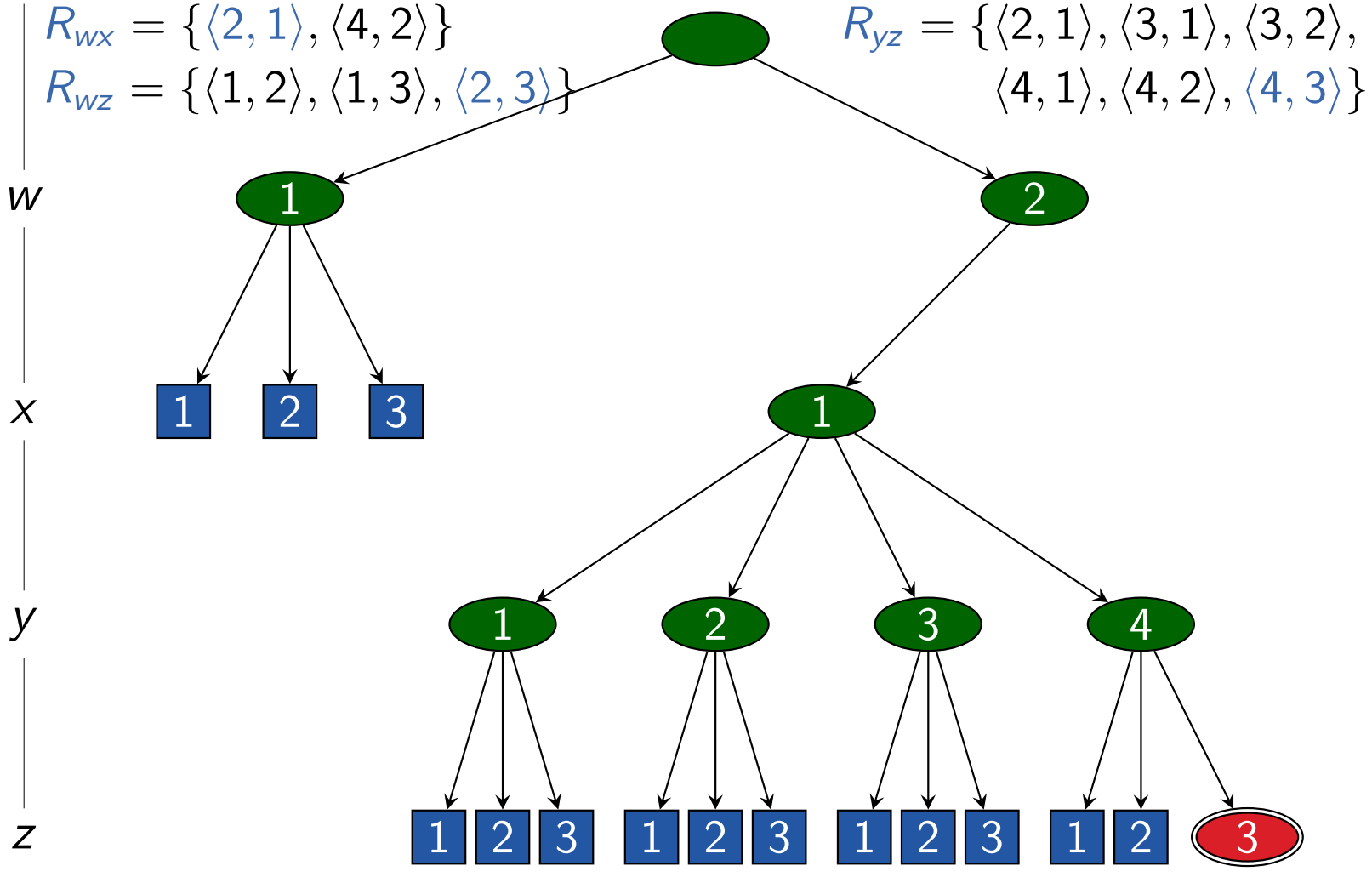
Running Example: Search Tree



Running Example: Search Tree



Running Example: Search Tree



Is This a New Algorithm?

We have already seen this algorithm:
Backtracking corresponds to depth-first search (Chapter B8)
with the following state space:

- states: partial assignments
- initial state: empty assignment $\emptyset$
- goal states: consistent total assignments
- actions: $assign_{v,d}$ assigns value $d \in \mathrm{dom}(v)$ to variable v
- action costs: all 0 (all solutions are of equal quality)
- transitions:
  - for each non-total consistent assignment α,
    choose variable $v = \mathrm{select}(\alpha)$ that is unassigned in α
  - transition $\alpha \xrightarrow{assign_{v,d}} \alpha \cup \{v \mapsto d\}$ for each $d \in \mathrm{dom}(v)$

Why Depth-First Search?

Depth-first search is particularly well-suited for CSPs:

- path length bounded (by the number of variables)
- solutions located at the same depth (lowest search layer)
- state space is directed tree, initial state is the root
  $\rightsquigarrow$ no duplicates (Why?)

Hence none of the problematic cases for depth-first search occurs.

Naive Backtracking: Discussion

- Naive backtracking often has to exhaustively explore similar search paths (i.e., partial assignments that are identical except for a few variables).
- "Critical" variables are not recognized and hence considered for assignment (too) late.
- Decisions that necessarily lead to constraint violations are only recognized when all variables involved in the constraint have been assigned.

$\rightsquigarrow$ more intelligence by focusing on critical decisions and by inference of consequences of previous decisions

Variable and Value Orders

Naive Backtracking

function NaiveBacktracking($\mathcal{C}, \alpha$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

select some variable v for which α is not defined

for each $d \in \text{dom}(v)$ in some order:
    $\alpha' := \alpha \cup \{v \mapsto d\}$
    $\alpha'' :=$ NaiveBacktracking($\mathcal{C}, \alpha'$)
    if $\alpha'' \neq$ **inconsistent**:
        return α''
return inconsistent

Variable Orders

- Backtracking does not specify in which order variables are considered for assignment.
- Such orders can strongly influence the search space size and hence the search performance.
  $\rightsquigarrow$ example: exercises
- Eventually we have to assign all variables
  $\rightsquigarrow$ prefer critical assignments (fail early)

German: Variablenordnung

Value Orders

- Backtracking does not specify in which order the values of the selected variable v are considered.
- This is not as important because it does not matter in subtrees without a solution. (Why not?)
- If there is a solution in the subtree, then ideally a value that leads to a solution should be chosen.
  $\rightsquigarrow$ prefer promising assignments

German: Werteordnung

Static vs. Dynamic Orders

we distinguish:

- static orders (fixed prior to search)
- dynamic orders (selected variable or value order depends on the search state)

comparison:

- dynamic orders obviously more powerful
- static orders $\rightsquigarrow$ no computational overhead during search

The following ordering criteria can be used statically, but are more effective combined with inference ($\rightsquigarrow$ later) and used dynamically.

Variable Orders

two common variable ordering criteria:

- minimum remaining values:
  prefer variables that have small domains
  - intuition: few subtrees $\rightsquigarrow$ smaller tree
  - extreme case: only one value $\rightsquigarrow$ forced assignment
- most constraining variable:
  prefer variables contained in many nontrivial constraints
  - intuition: constraints tested early
    $\rightsquigarrow$ inconsistencies recognized early $\rightsquigarrow$ smaller tree

combination: use minimum remaining values criterion,
then most constraining variable criterion to break ties

Value Orders

Definition (conflict)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.
For variables $v \neq v'$ and values $d \in \mathrm{dom}(v)$, $d' \in \mathrm{dom}(v')$,
the assignment $v \mapsto d$ is in conflict with $v' \mapsto d'$ if $\langle d, d' \rangle \notin R_{vv'}$.

value ordering criterion for partial assignment α
and selected variable v:

- minimum conflicts: prefer values $d \in \mathrm{dom}(v)$
  such that $v \mapsto d$ causes as few conflicts as possible
  with variables that are unassigned in α

Summary

Summary: Backtracking

basic search algorithm for constraint networks: backtracking

- extends the (initially empty) partial assignment step by step until an inconsistency or a solution is found
- is a form of depth-first search
- depth-first search particularly well-suited because state space is directed tree and all solutions at same (known) depth

Summary: Variable and Value Orders

- Variable orders influence the performance of backtracking significantly.
  - goal: critical decisions as early as possible
- Value orders influence the performance of backtracking on solvable constraint networks significantly.
  - goal: most promising assignments first

Foundations of Artificial Intelligence

D4. Constraint Satisfaction Problems: Arc Consistency

Malte Helmert

University of Basel

April 9, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
- D3–D5. Basic Algorithms
  - D3. Backtracking
  - D4. Arc Consistency
  - D5. Path Consistency
- D6–D7. Problem Structure

Inference

Inference

Inference

Derive additional constraints (here: unary or binary)
that are implied by the given constraints,
i.e., that are satisfied in all solutions.

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains:

- $\mathrm{dom}(w) = \{1,2,3,4\}$
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Can we use the constraint R_{wz} $(w < z)$ to come up with a unary constraint R_w?

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains (unary constraints):

- $\mathrm{dom}(w) = \{1,2\}$
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Can we use the constraint R_{wz} $(w < z)$ to come up with a unary constraint R_w?

$\rightsquigarrow$ tighten domain with unary constraint

(sometimes called node consistency)

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains (unary constraints):

- $\mathrm{dom}(w) = \{1,2\}$
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

How does this affect the binary constraint R_{wx}?

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2, 1\rangle\}$
- $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$
- $R_{yz} = \{\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle\}$

domains (unary constraints):

- $\mathrm{dom}(w) = \{1, 2\}$
- $\mathrm{dom}(x) = \{1, 2, 3\}$
- $\mathrm{dom}(y) = \{1, 2, 3, 4\}$
- $\mathrm{dom}(z) = \{1, 2, 3\}$

How does this affect the binary constraint R_{wx}?

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains (unary constraints):

- $\mathrm{dom}(w) = \{1,2\}$
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Can we generate a "new" binary constraint from $w < z$ and $z < y$? (i.e., tighten a trivial constraint)

Inference: Example

Running Example

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$
- $R_{wy} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$

domains (unary constraints):

- $\mathrm{dom}(w) = \{1,2\}$
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Can we generate a "new" binary constraint from $w < z$ and $z < y$? (i.e., tighten a trivial constraint)

Trade-Off Search vs. Inference

Inference formally

For a given constraint network $\mathcal{C}$, replace $\mathcal{C}$
with an equivalent, but tighter constraint network.

Trade-off:

- the more complex the inference, and
- the more often inference is applied,
- the smaller the resulting state space, but
- the higher the complexity per search node.

When to Apply Inference?

different possibilities to apply inference:

- once as preprocessing before search
- combined with search: before recursive calls during backtracking procedure
  - already assigned variable $v \mapsto d$ corresponds to $\mathrm{dom}(v) = \{d\}$
    $\rightsquigarrow$ more inferences possible
  - during backtracking, derived constraints have to be retracted because they were based on the given assignment

$\rightsquigarrow$ powerful, but possibly expensive

Backtracking with Inference

function BacktrackingWithInference($\mathcal{C}, \alpha$):

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

$\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle :=$ copy of $\mathcal{C}$
apply inference to $\mathcal{C}'$

if $\text{dom}'(v) \neq \emptyset$ for all variables v:

    select some variable v for which α is not defined

    for each $d \in$ copy of $\text{dom}'(v)$ in some order:
        $\alpha' := \alpha \cup \{v \mapsto d\}$
        $\text{dom}'(v) := \{d\}$
        $\alpha'' :=$ BacktrackingWithInference($\mathcal{C}', \alpha'$)
        if $\alpha'' \neq$ **inconsistent**:
            return α''

return inconsistent

Backtracking with Inference

function BacktrackingWithInference($\mathcal{C}, \alpha$):

if α is inconsistent with $\mathcal{C}$:
    return inconsistent

if α is a total assignment:
    return α

$\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle :=$ copy of $\mathcal{C}$
apply inference to $\mathcal{C}'$

if $\text{dom}'(v) \neq \emptyset$ for all variables v:

    select some variable v for which α is not defined

    for each $d \in$ copy of $\text{dom}'(v)$ in some order:
        $\alpha' := \alpha \cup \{v \mapsto d\}$
        $\text{dom}'(v) := \{d\}$
        $\alpha'' :=$ BacktrackingWithInference$(\mathcal{C}', \alpha')$
        if $\alpha'' \neq$ **inconsistent**:
            return α''

return inconsistent

Backtracking with Inference: Discussion

- Inference is a placeholder:
  different inference methods can be applied.
- Inference methods can recognize unsolvability (given α)
  and indicate this by clearing the domain of a variable.
- Efficient implementations of inference are often incremental:
  the last assigned variable/value pair $v \mapsto d$ is taken
  into account to speed up the inference computation.

Forward Checking

Forward Checking

We start with a simple inference method:

Forward Checking

Let α be a partial assignment.
Inference: For all unassigned variables v in α,
remove all values from the domain of v that are in conflict with already assigned variable/value pairs in α.

$\rightsquigarrow$ definition of conflict as in the previous chapter

Incremental computation:

- When adding $v \mapsto d$ to the assignment, delete all pairs that conflict with $v \mapsto d$.

Forward Checking: Example

Running Example

Removing values in conflict with $\alpha = \{w \mapsto 2\}$:

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains:

- w is already assigned
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Forward Checking: Example

Running Example

Removing values in conflict with $\alpha = \{w \mapsto 2\}$:

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains:

- w is already assigned
- $\mathrm{dom}(x) = \{1,2,3\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{1,2,3\}$

Forward Checking: Example

Running Example

Removing values in conflict with $\alpha = \{w \mapsto 2\}$:

binary constraints:

- $R_{wx} = \{\langle 2, 1\rangle, \langle 4, 2\rangle\}$
- $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$
- $R_{yz} = \{\langle 2, 1\rangle, \langle 3, 1\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle, \langle 4, 2\rangle, \langle 4, 3\rangle\}$

domains:

- w is already assigned
- $\mathrm{dom}(x) = \{1\}$
- $\mathrm{dom}(y) = \{1, 2, 3, 4\}$
- $\mathrm{dom}(z) = \{1, 2, 3\}$

Forward Checking: Example

Running Example

Removing values in conflict with $\alpha = \{w \mapsto 2\}$:

binary constraints:

- $R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
- $R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
- $R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

domains:

- w is already assigned
- $\mathrm{dom}(x) = \{1\}$
- $\mathrm{dom}(y) = \{1,2,3,4\}$
- $\mathrm{dom}(z) = \{3\}$

Forward Checking: Discussion

properties of forward checking:

- correct inference method (retains equivalence)
- affects domains (= unary constraints), but not binary constraints
- consistency check at the beginning of the backtracking procedure no longer needed (Why?)
- cheap, but often still useful inference method

$\rightsquigarrow$ apply at least forward checking in the backtracking procedure

In the following, we will consider more powerful inference methods.

Arc Consistency

Arc Consistency: Definition

Definition (Arc Consistent)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.

1. The variable $v \in V$ is arc consistent
   with respect to another variable $v' \in V$,
   if for every value $d \in \mathrm{dom}(v)$
   there exists a value $d' \in \mathrm{dom}(v')$ with $\langle d, d' \rangle \in R_{vv'}$.
2. The constraint network $\mathcal{C}$ is arc consistent,
   if every variable $v \in V$ is arc consistent
   with respect to every other variable $v' \in V$.

German: kantenkonsistent

remarks:

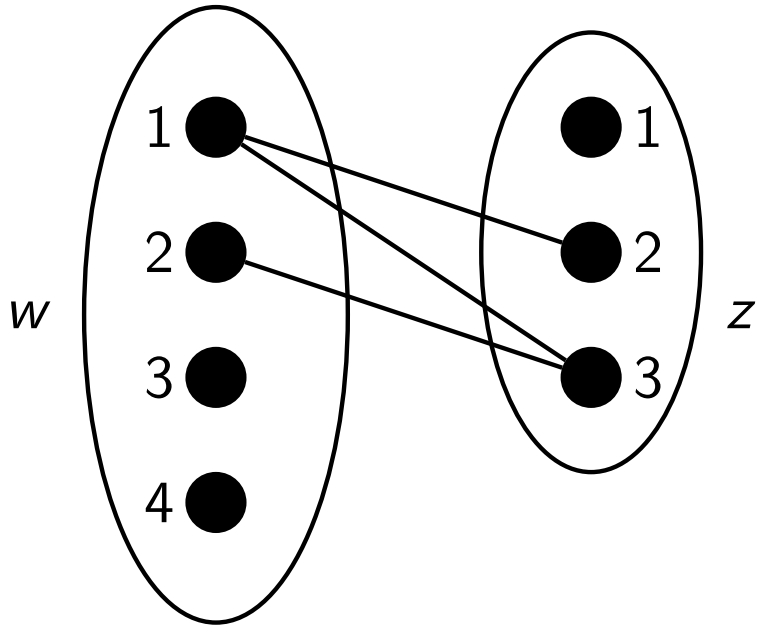
- definition for variable pair is not symmetrical
- v always arc consistent with respect to v'
  if the constraint between v and v' is trivial

Arc Consistency: Example

Running Example

Consider variables w and z from our running example:

- $\mathrm{dom}(w) = \{1, 2, 3, 4\}$
- $\mathrm{dom}(z) = \{1, 2, 3\}$
- $R_{wz} = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$



Arc consistency
of w with respect to z and
of z with respect to w
is violated.

Enforcing Arc Consistency

- Enforcing arc consistency, i.e., removing values from $\mathrm{dom}(v)$ that violate the arc consistency of v with respect to v', is a correct inference method. (Why?)
- more powerful than forward checking (Why?)

Enforcing Arc Consistency

- Enforcing arc consistency, i.e., removing values from $\mathrm{dom}(v)$ that violate the arc consistency of v with respect to v', is a correct inference method. (Why?)
- more powerful than forward checking (Why?)
  - $\rightsquigarrow$ Forward checking is a special case: enforcing arc consistency of all variables with respect to the just assigned variable corresponds to forward checking.

We will next consider algorithms that enforce arc consistency.

Processing Variable Pairs: revise

function revise($\mathcal{C}, v, v'$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

for each $d \in \mathrm{dom}(v)$:

  if there is no $d' \in \mathrm{dom}(v')$ with $\langle d, d' \rangle \in R_{vv'}$:

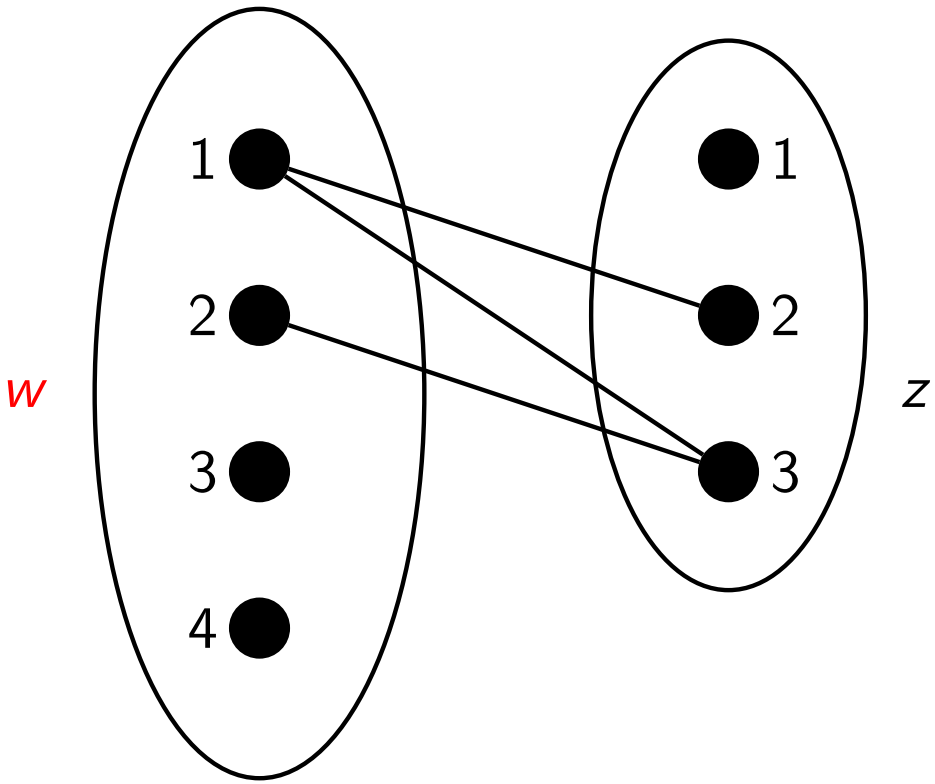
    remove d from $\mathrm{dom}(v)$

input: constraint network $\mathcal{C}$ and two variables v, v' of $\mathcal{C}$

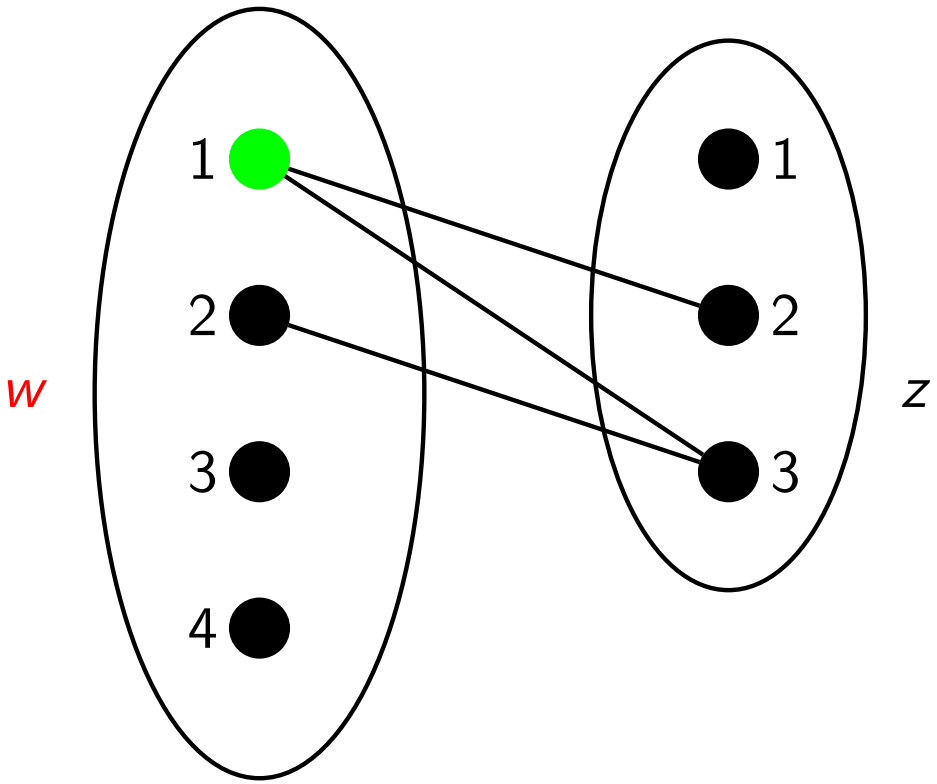
effect: v arc consistent with respect to v'.
All violating values in $\mathrm{dom}(v)$ are removed.

time complexity: $O(k^2)$, where k is maximal domain size

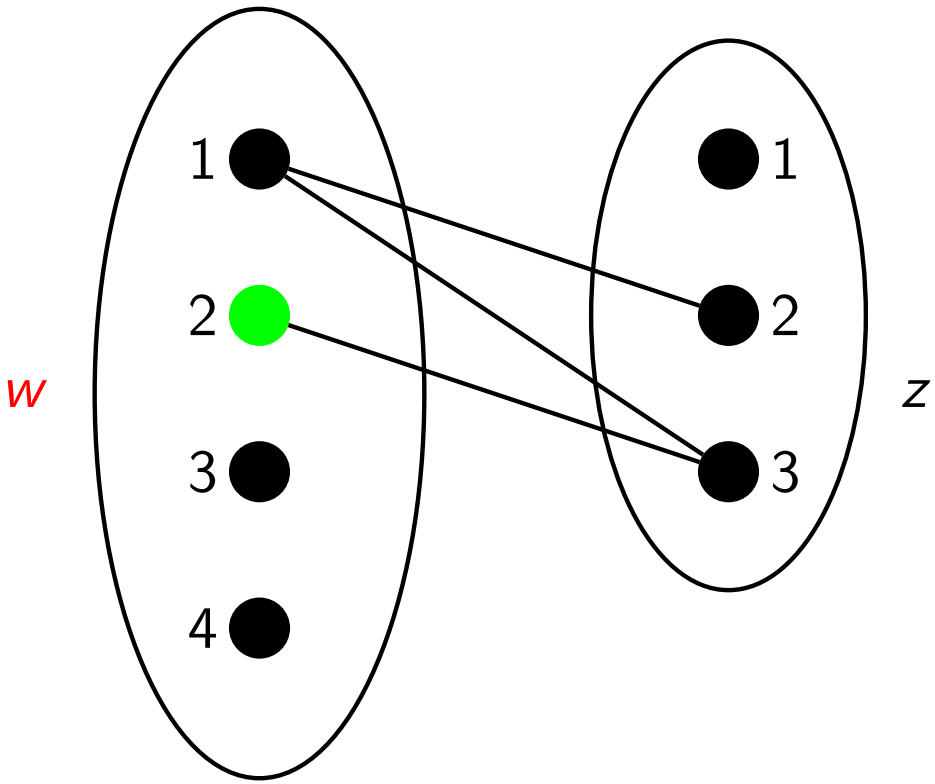
revise($\mathcal{C}, w, z$) in Running Example



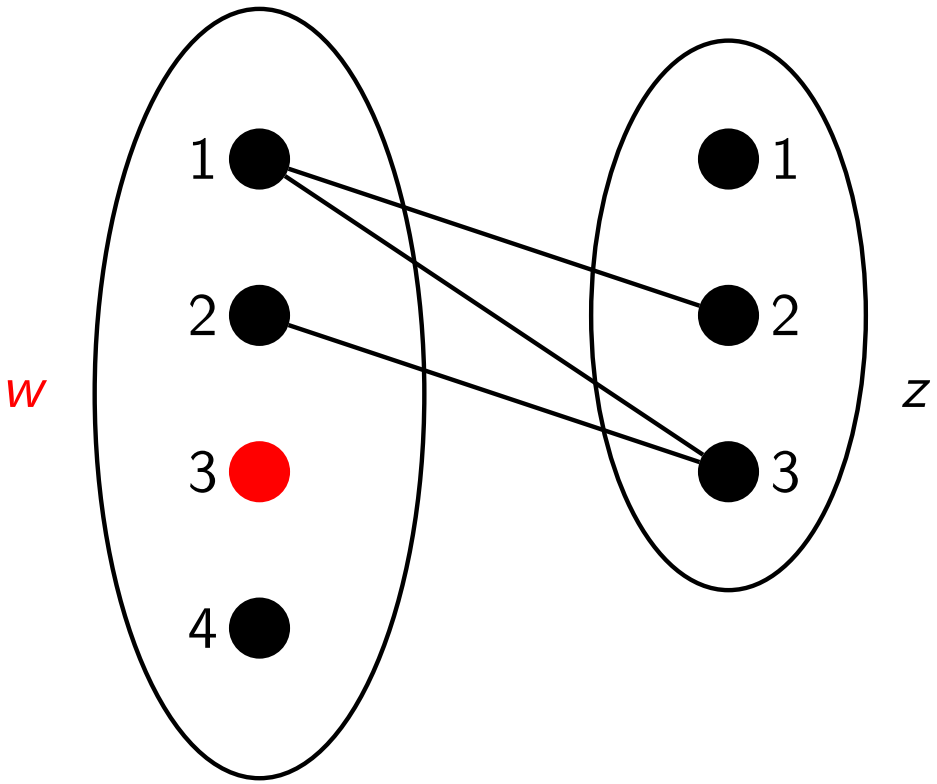
revise($\mathcal{C}, w, z$) in Running Example



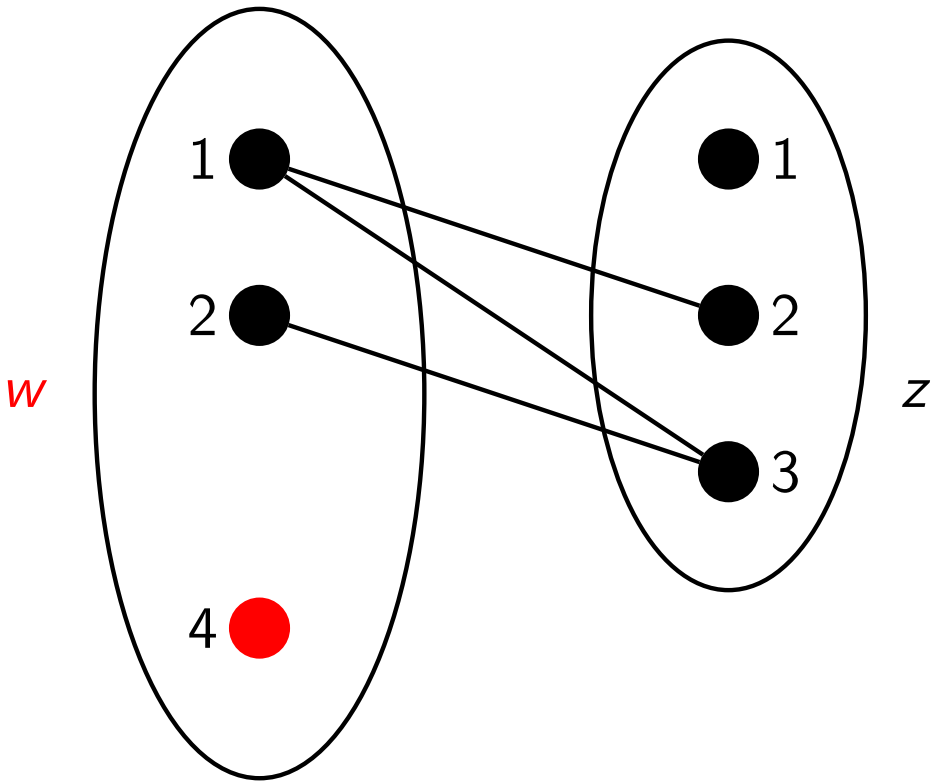
revise($\mathcal{C}, w, z$) in Running Example



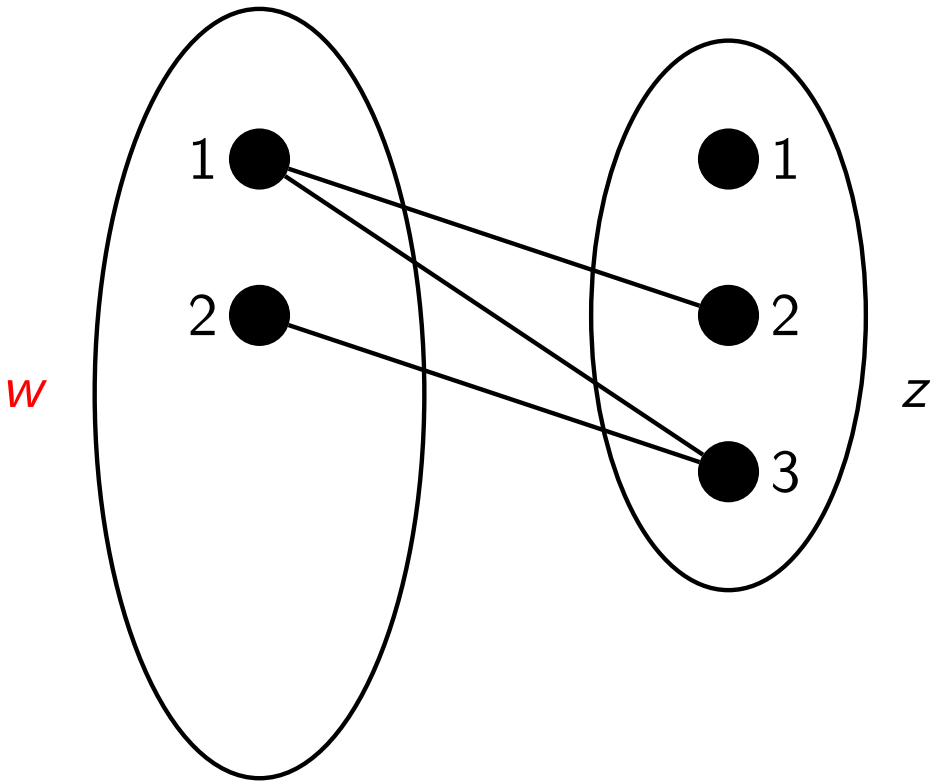
revise($\mathcal{C}, w, z$) in Running Example



revise($\mathcal{C}, w, z$) in Running Example



revise($\mathcal{C}, w, z$) in Running Example



Enforcing Arc Consistency: AC-1

function AC-1($\mathcal{C}$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

repeat

  for each nontrivial constraint R_{uv}:

    revise($\mathcal{C}, u, v$)

    revise($\mathcal{C}, v, u$)

until no domain has changed in this iteration

input: constraint network $\mathcal{C}$

effect: transforms $\mathcal{C}$ into equivalent arc consistent network

time complexity: ?

Enforcing Arc Consistency: AC-1

function AC-1($\mathcal{C}$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

repeat

  for each nontrivial constraint R_{uv}:

    revise($\mathcal{C}, u, v$)

    revise($\mathcal{C}, v, u$)

until no domain has changed in this iteration

input: constraint network $\mathcal{C}$

effect: transforms $\mathcal{C}$ into equivalent arc consistent network

time complexity: $O(n \cdot e \cdot k^3)$, with n variables,
e nontrivial constraints and maximal domain size k

AC-1: Discussion

- AC-1 does the job, but is rather inefficient.
- Drawback: Variable pairs are often checked again and again although their domains have remained unchanged.
- These (redundant) checks can be saved.

$\rightsquigarrow$ more efficient algorithm: AC-3

Enforcing Arc Consistency: AC-3

idea: store potentially inconsistent variable pairs in a queue

function AC-3($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

$queue := \emptyset$

for each nontrivial constraint R_{uv}:

    insert $\langle u, v \rangle$ into *queue*

    insert $\langle v, u \rangle$ into *queue*

while $queue \neq \emptyset$:

    remove an arbitrary element $\langle u, v \rangle$ from *queue*

    revise($\mathcal{C}, u, v$)

    if dom(u) changed in the call to revise:

        for each $w \in V \setminus \{u, v\}$ where R_{wu} is nontrivial:

            insert $\langle w, u \rangle$ into *queue*

AC-3: Discussion

- *queue* can be an arbitrary data structure
  that supports insert and remove operations
  (the order of removal does not affect the result)

$\rightsquigarrow$ use data structure with fast insertion and removal, e.g., stack

- AC-3 has the same effect as AC-1:
  it enforces arc consistency
- proof idea: invariant of the **while** loop:
  If $\langle u, v \rangle \notin \mathit{queue}$, then u is arc consistent with respect to v

AC-3: Time Complexity

Proposition (time complexity of AC-3)

Let $\mathcal{C}$ be a constraint network with e nontrivial constraints and maximal domain size k.

The time complexity of AC-3 *is $O(e \cdot k^3)$.*

AC-3: Time Complexity (Proof)

Proof.

Consider a pair $\langle u, v \rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

AC-3: Time Complexity (Proof)

Proof.

Consider a pair $\langle u, v \rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

Every time this pair is inserted to the queue (except for the first time) the domain of the second variable has just been reduced.

AC-3: Time Complexity (Proof)

Proof.

Consider a pair $\langle u, v \rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

Every time this pair is inserted to the queue (except for the first time) the domain of the second variable has just been reduced.

This can happen at most k times.

AC-3: Time Complexity (Proof)

Proof.

Consider a pair $\langle u, v \rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

Every time this pair is inserted to the queue (except for the first time) the domain of the second variable has just been reduced.

This can happen at most k times.

Hence every pair $\langle u, v \rangle$ is inserted into the queue
at most $k + 1$ times $\rightsquigarrow$ at most $O(ek)$ insert operations in total.

AC-3: Time Complexity (Proof)

Proof.

Consider a pair $\langle u, v \rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

Every time this pair is inserted to the queue (except for the first time) the domain of the second variable has just been reduced.

This can happen at most k times.

Hence every pair $\langle u, v \rangle$ is inserted into the queue at most $k + 1$ times $\rightsquigarrow$ at most $O(ek)$ insert operations in total.

This bounds the number of **while** iterations by $O(ek)$, giving an overall time complexity of $O(ek) \cdot O(k^2) = O(ek^3)$. □

Summary

Summary: Inference

- inference: derivation of additional constraints that are implied by the known constraints

$\rightsquigarrow$ tighter equivalent constraint network

- trade-off search vs. inference
- inference as preprocessing or integrated into backtracking

Summary: Forward Checking, Arc Consistency

- cheap and easy inference: forward checking
  - remove values that conflict with already assigned values
- more expensive and more powerful: arc consistency
  - iteratively remove values without a suitable "partner value" for another variable until fixed-point reached
  - efficient implementation of AC-3: $O(ek^3)$ with e: #nontrivial constraints, k: size of domain

Foundations of Artificial Intelligence

D5. Constraint Satisfaction Problems: Path Consistency

Malte Helmert

University of Basel

April 14, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
- D3–D5. Basic Algorithms
  - D3. Backtracking
  - D4. Arc Consistency
  - D5. Path Consistency
- D6–D7. Problem Structure

Beyond Arc Consistency

Beyond Arc Consistency: Path Consistency

idea of arc consistency:

- For every assignment to a variable u
  there must be a suitable assignment to every other variable v.
- If not: remove values of u for which
  no suitable "partner" assignment to v exists.

$\rightsquigarrow$ tighter unary constraint on u

This idea can be extended to three variables (path consistency):

- For every joint assignment to variables u, v
  there must be a suitable assignment to every third variable w.
- If not: remove pairs of values of u and v for which
  no suitable "partner" assignment to w exists.

$\rightsquigarrow$ tighter binary constraint on u and v

German: Pfadkonsistenz

Beyond Arc Consistency: i-Consistency

general concept of i-consistency for $i \geq 2$:

- For every joint assignment to variables $v_1, \ldots, v_{i-1}$ there must be a suitable assignment to every i-th variable v_i.
- If not: remove value tuples of $v_1, \ldots, v_{i-1}$ for which no suitable "partner" assignment for v_i exists.

$\rightsquigarrow$ tighter $(i-1)$-ary constraint on $v_1, \ldots, v_{i-1}$

- 2-consistency = arc consistency
- 3-consistency = path consistency (*)

We do not consider general i-consistency further as larger values than $i = 3$ are rarely used and we restrict ourselves to binary constraints in this course.

(*) usual definitions of 3-consistency vs. path consistency differ when ternary constraints are allowed

Path Consistency

Path Consistency: Definition

Definition (path consistent)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.

1. Two different variables $u, v \in V$ are path consistent with respect to a third variable $w \in V$ if for all values $d_u \in \mathrm{dom}(u), d_v \in \mathrm{dom}(v)$ with $\langle d_u, d_v \rangle \in R_{uv}$ there is a value $d_w \in \mathrm{dom}(w)$ with $\langle d_u, d_w \rangle \in R_{uw}$ and $\langle d_v, d_w \rangle \in R_{vw}$.
2. The constraint network $\mathcal{C}$ is path consistent if for all triples of different variables u, v, w, the variables u and v are path consistent with respect to w.

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

Are w and y path consistent with respect to z?

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

$R_{wy} = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle,$
$\langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle, \langle 4,4\rangle\}$

Are w and y path consistent with respect to z?

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$

$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

$R_{wy} = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle,$
$\langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle, \langle 4,4\rangle\}$

Are w and y path consistent with respect to z? No!

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$
$R_{wy} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$

Are w and y path consistent with respect to z?

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$
$R_{wy} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$

Are w and y path consistent with respect to z? Yes!

Path Consistency on Running Example

Running Example

$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$
$R_{wy} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$

Are w and y path consistent with respect to z? Yes!

Path Consistency on Running Example

Running Example

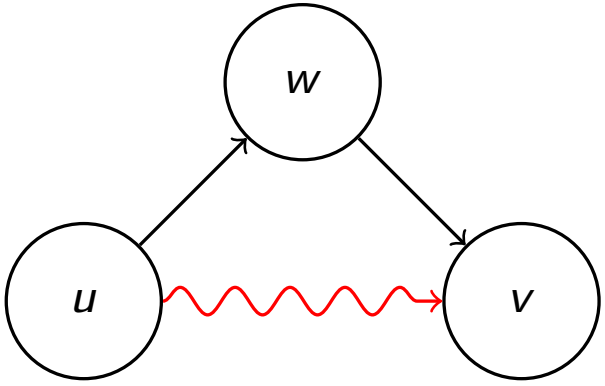
$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$
$R_{wy} = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$

Are w and y path consistent with respect to z? Yes!

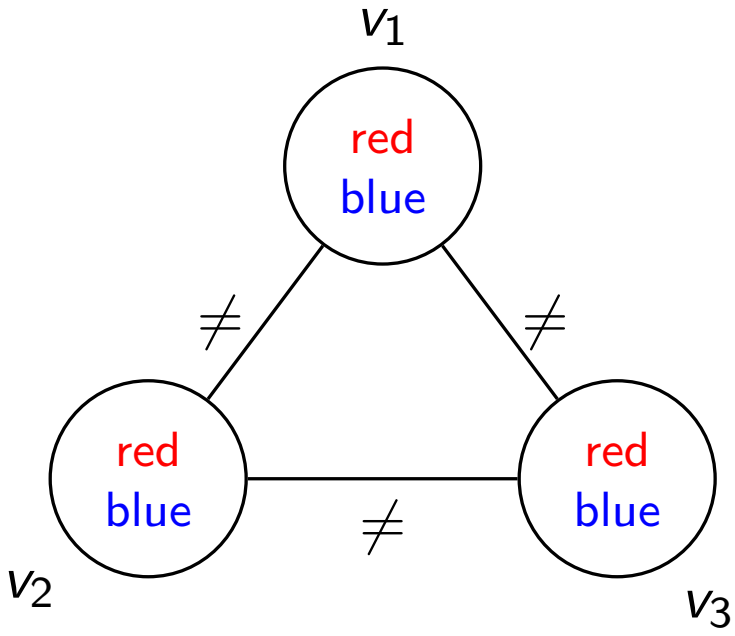
Path Consistency: Remarks

remarks:

- Even if the constraint R_{uv} is trivial, path consistency can infer nontrivial constraints between u and v.
- name "path consistency":
  path $u \to w \to v$ leads to new information on $u \to v$



Path Consistency: Example



arc consistent, but not path consistent

Processing Variable Triples: revise-3

analogous to revise for arc consistency:

function revise-3$(\mathcal{C}, u, v, w)$:

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

for each $\langle d_u, d_v \rangle \in R_{uv}$:

  if there is no $d_w \in \mathrm{dom}(w)$ with

    $\langle d_u, d_w \rangle \in R_{uw}$ **and** $\langle d_v, d_w \rangle \in R_{vw}$:

      remove $\langle d_u, d_v \rangle$ from R_{uv}

input: constraint network $\mathcal{C}$ and three variables u, v, w of $\mathcal{C}$

effect: u, v path consistent with respect to w.
All violating pairs are removed from R_{uv}.

time complexity: $O(k^3)$ where k is maximal domain size

Enforcing Path Consistency: PC-2

analogous to AC-3 for arc consistency:

function PC-2($\mathcal{C}$):

$\langle V, \mathrm{dom}, (R_{uv}) \rangle := \mathcal{C}$

$queue := \emptyset$

for each set of two variables $\{u, v\}$:

    for each $w \in V \setminus \{u, v\}$:

        insert $\langle u, v, w \rangle$ into $queue$

while $queue \neq \emptyset$:

    remove any element $\langle u, v, w \rangle$ from $queue$

    revise-3($\mathcal{C}, u, v, w$)

    if R_{uv} changed in the call to revise-3:

        for each $w' \in V \setminus \{u, v\}$:

            insert $\langle w', u, v \rangle$ into $queue$

            insert $\langle w', v, u \rangle$ into $queue$

PC-2: Discussion

The comments for AC-3 hold analogously.

- PC-2 enforces path consistency
- proof idea: invariant of the **while** loop:
  if $\langle u, v, w \rangle \notin \mathit{queue}$, then u, v path consistent with respect to w
- time complexity $O(n^3 k^5)$ for n variables and maximal domain size k (Why?)

Summary

Summary

- generalization of
  arc consistency (considers pairs of variables)
  to path consistency (considers triples of variables)
  and i-consistency (considers i-tuples of variables)
- arc consistency tightens unary constraints
- path consistency tightens binary constraints
- i-consistency tightens $(i-1)$-ary constraints
- higher levels of consistency more powerful
  but more expensive than arc consistency

Foundations of Artificial Intelligence

D6. Constraint Satisfaction Problems: Constraint Graphs

Malte Helmert

University of Basel

April 14, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
- D3–D5. Basic Algorithms
- D6–D7. Problem Structure
  - D6. Constraint Graphs
  - D7. Decomposition Methods

Constraint Graphs

Motivation

- To solve a constraint network consisting of n variables and k values, k^n assignments must be considered.
- Inference can alleviate this combinatorial explosion, but will not always avoid it.
- Many practically relevant constraint networks are efficiently solvable if their structure is taken into account.

Constraint Graphs

Definition (constraint graph)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.

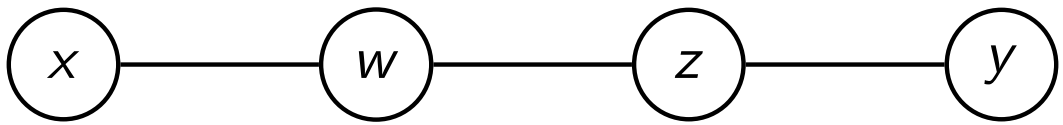
The constraint graph of $\mathcal{C}$ is the graph whose vertices are V and which contains an edge $\{u, v\}$ iff R_{uv} is a nontrivial constraint.

Constraint Graphs: Running Example

Nontrivial Constraints of Running Example

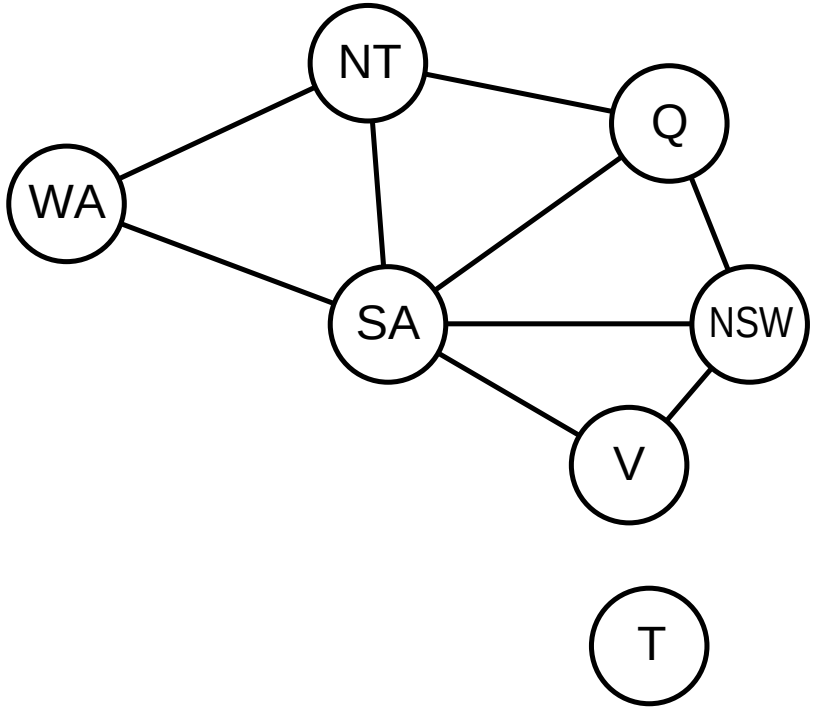
$R_{wx} = \{\langle 2,1\rangle, \langle 4,2\rangle\}$
$R_{wz} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$
$R_{yz} = \{\langle 2,1\rangle, \langle 3,1\rangle, \langle 3,2\rangle, \langle 4,1\rangle, \langle 4,2\rangle, \langle 4,3\rangle\}$

Resulting Constraint Graph:



Constraint Graphs: Better Example

Coloring of the Australian states (plus Northern Territory)



Unconnected Graphs

Unconnected Constraint Graphs

Proposition (unconnected constraint graphs)

If the constraint graph of $\mathcal{C}$ has multiple connected components, the subproblems induced by each component can be solved separately.

The union of the solutions of these subproblems is a solution for $\mathcal{C}$.

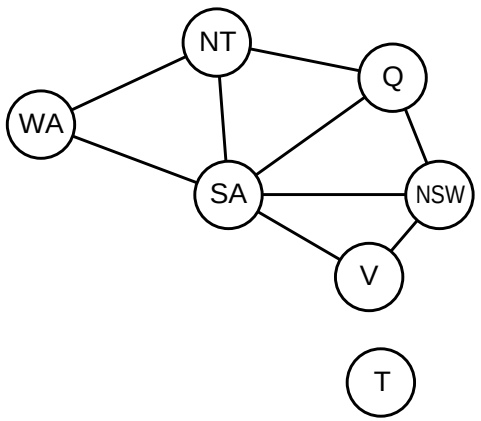
Proof.

A total assignment consisting of combined subsolutions satisfies all constraints that occur within the subproblems.

All constraints between two subproblems are trivial (follows from the definitions of constraint graphs and connected components). □

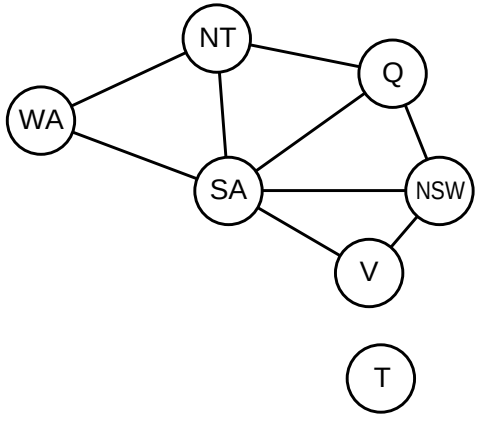
Unconnected Constraint Graphs: Example

example: Tasmania can be colored independently from the rest of Australia.



Unconnected Constraint Graphs: Example

example: Tasmania can be colored independently from the rest of Australia.

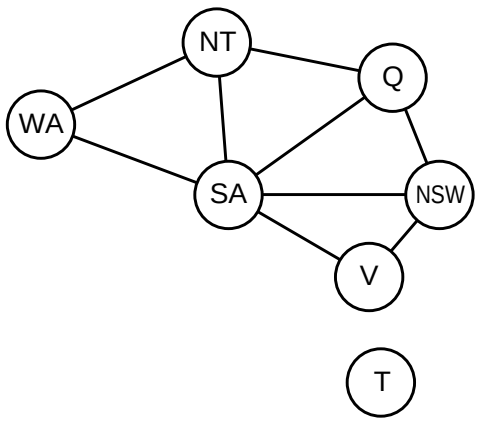


further example:
network with $k = 2$, $n = 30$ that decomposes into three components of equal size

savings?

Unconnected Constraint Graphs: Example

example: Tasmania can be colored independently from the rest of Australia.



further example:

network with $k = 2$, $n = 30$ that decomposes into three components of equal size

savings?

only $3 \cdot 2^{10} = 3072$ assignments instead of $2^{30} = 1073741824$

Trees

Trees as Constraint Graphs

Proposition (trees as constraint graphs)

Let $\mathcal{C}$ be a constraint network with n variables and maximal domain size k whose constraint graph is a tree or forest (i.e., does not contain cycles).

Then we can solve $\mathcal{C}$ or prove that no solution exists in time $O(nk^2)$.

example: $k = 5, n = 10$
$\rightsquigarrow k^n = 9765625,\ nk^2 = 250$

Trees as Constraint Graphs: Algorithm

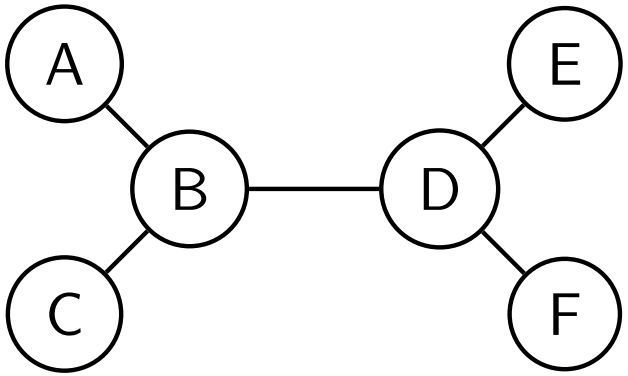
algorithm for trees:

- Build a directed tree for the constraint graph.
  Select an arbitrary variable as the root.
- Order variables $v_1, \ldots, v_n$ such that parents are ordered before their children.
- For $i \in \langle n, n-1, \ldots, 2 \rangle$: call revise$(v_{\text{parent}(i)}, v_i)$
  $\rightsquigarrow$ each variable is arc consistent with respect to its children
- If a domain becomes empty, the problem is unsolvable.
- Otherwise: solve with BacktrackingWithInference, variable order $v_1, \ldots, v_n$ and forward checking.
  $\rightsquigarrow$ solution is found without backtracking steps

proof: $\rightsquigarrow$ exercises

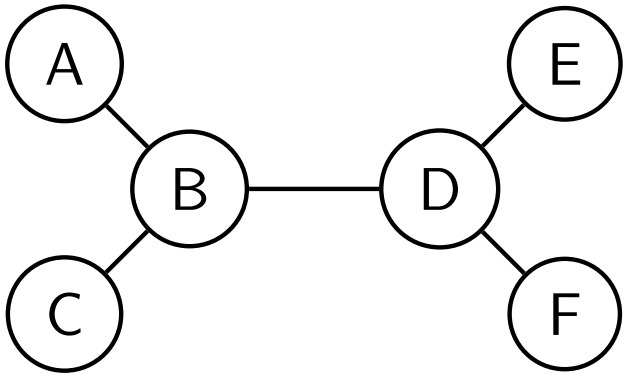
Trees as Constraint Graphs: Example

1. constraint graph:

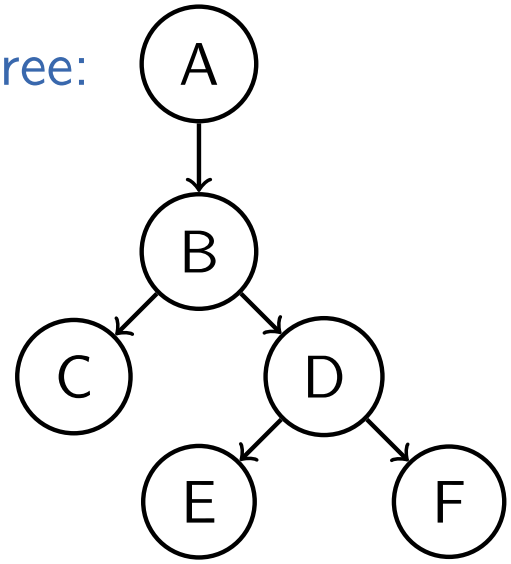


Trees as Constraint Graphs: Example

1. constraint graph:

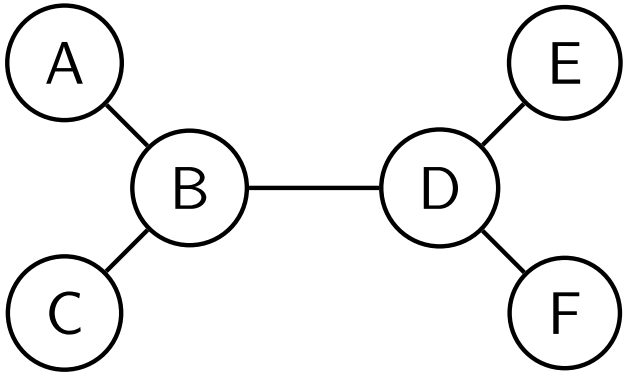


2. directed tree:

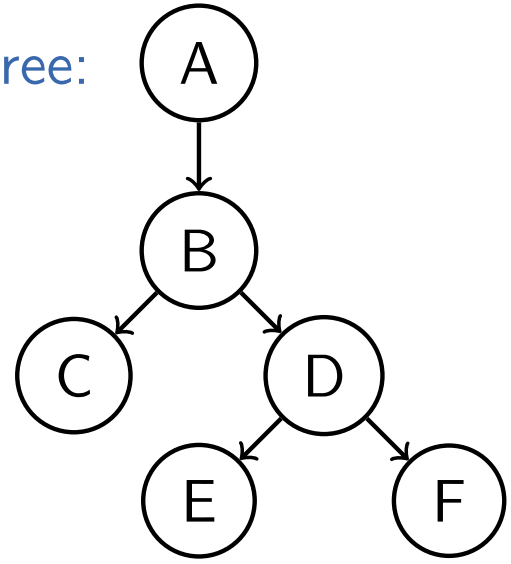


Trees as Constraint Graphs: Example

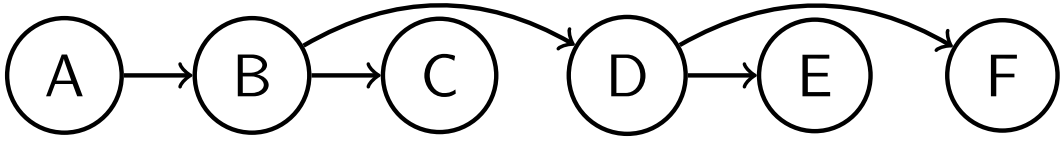
1. constraint graph:



2. directed tree:

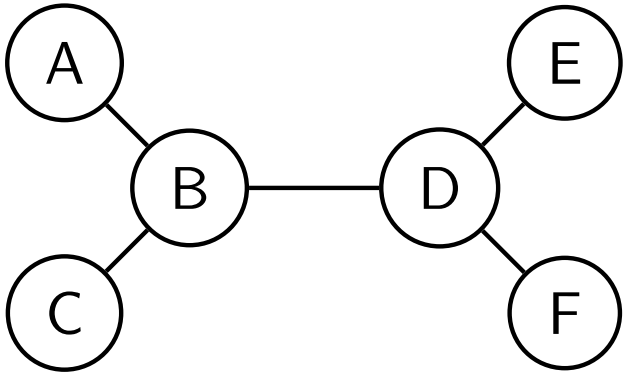


3. order:

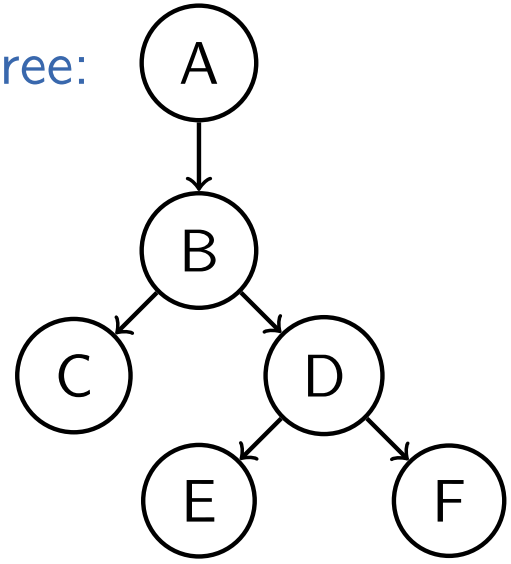


Trees as Constraint Graphs: Example

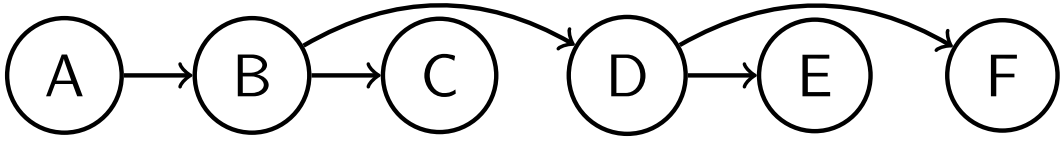
1. constraint graph:



2. directed tree:



3. order:

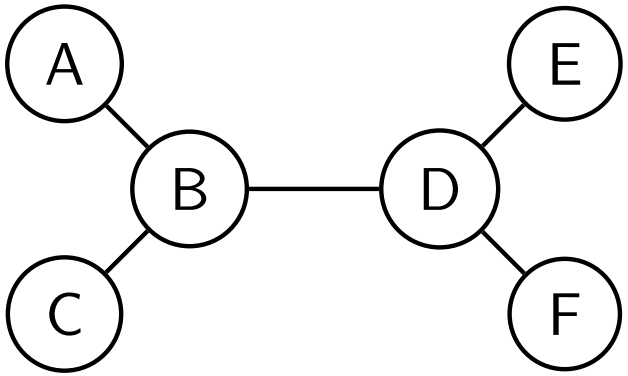


4. revise steps:

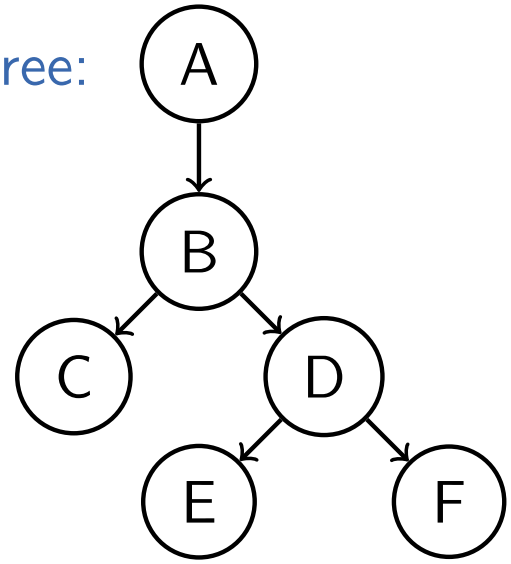
- revise(D, F)
- revise(D, E)
- revise(B, D)
- revise(B, C)
- revise(A, B)

Trees as Constraint Graphs: Example

1. constraint graph:



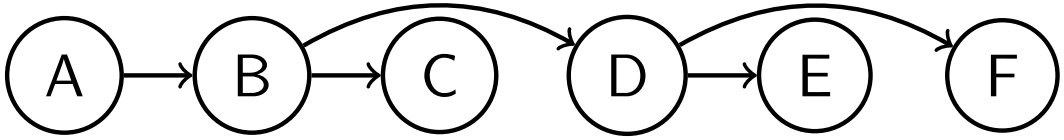
2. directed tree:



4. revise steps:

- revise(D, F)
- revise(D, E)
- revise(B, D)
- revise(B, C)
- revise(A, B)

3. order:



5. finding a solution:

backtracking with forward checking and order

$A \prec B \prec C \prec D \prec E \prec F$

Summary

Summary

- Constraint networks with simple structure are easy to solve.
- Constraint graphs formalize this structure:
  - several connected components:
    solve separately for each component
  - tree: algorithm linear in number of variables

Foundations of Artificial Intelligence

D7. Constraint Satisfaction Problems: Decomposition Methods

Malte Helmert

University of Basel

April 16, 2025

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems

- D1–D2. Introduction
- D3–D5. Basic Algorithms
- D6–D7. Problem Structure
  - D6. Constraint Graphs
  - D7. Decomposition Methods

Decomposition Methods

More Complex Graphs

What if the constraint graph is not a tree
and does not decompose into several components?

- idea 1: conditioning
- idea 2: tree decomposition

German: Konditionierung, Baumzerlegung

Conditioning

Conditioning

Conditioning

idea: Apply backtracking with forward checking until the constraint graph restricted to the remaining unassigned variables decomposes or is a tree.

remaining problem $\rightsquigarrow$ algorithms for simple constraint graphs

Conditioning

Conditioning

idea: Apply backtracking with forward checking until the constraint graph restricted to the remaining unassigned variables decomposes or is a tree.

remaining problem $\rightsquigarrow$ algorithms for simple constraint graphs

cutset conditioning:
Choose variable order such that early variables form a small cutset (i.e., set of variables such that removing these variables results in an acyclic constraint graph).

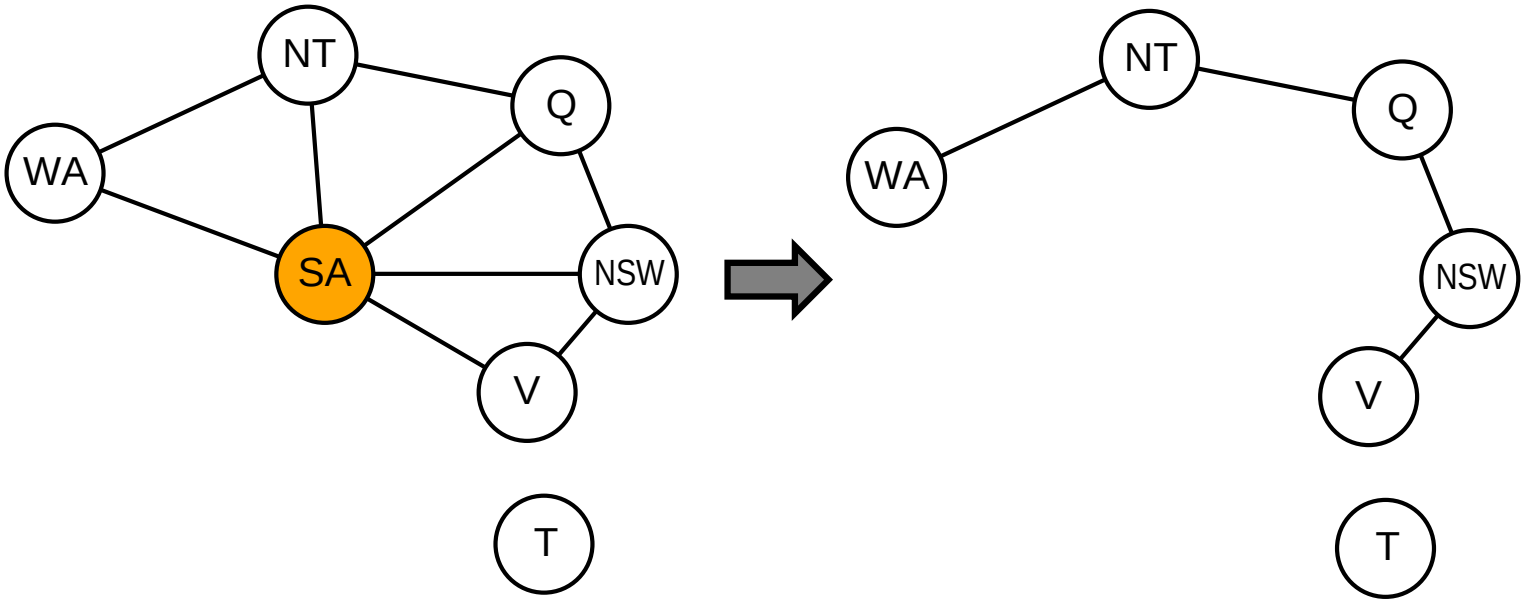
German: Cutset

time complexity: n variables, $m < n$ in cutset, maximal domain size k: $O(k^m \cdot (n-m)k^2)$

(Finding optimal cutsets is an NP-complete problem.)

Conditioning: Example

Australia example: Cutset of size 1 suffices:



Tree Decomposition

Tree Decomposition

basic idea of tree decomposition:

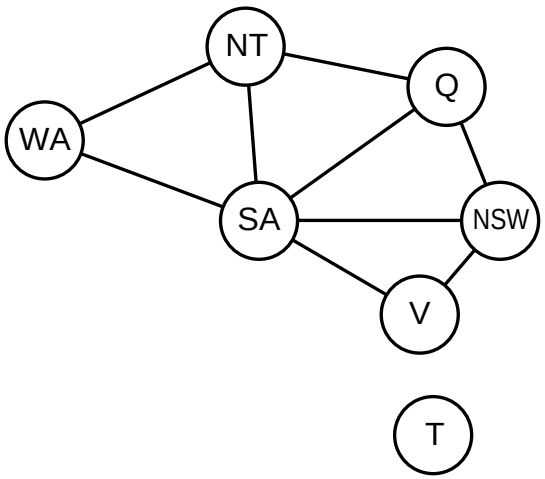
- Decompose constraint network into smaller subproblems (overlapping).
- Find solutions for the subproblems.
- Build overall solution based on the subsolutions.

more details:

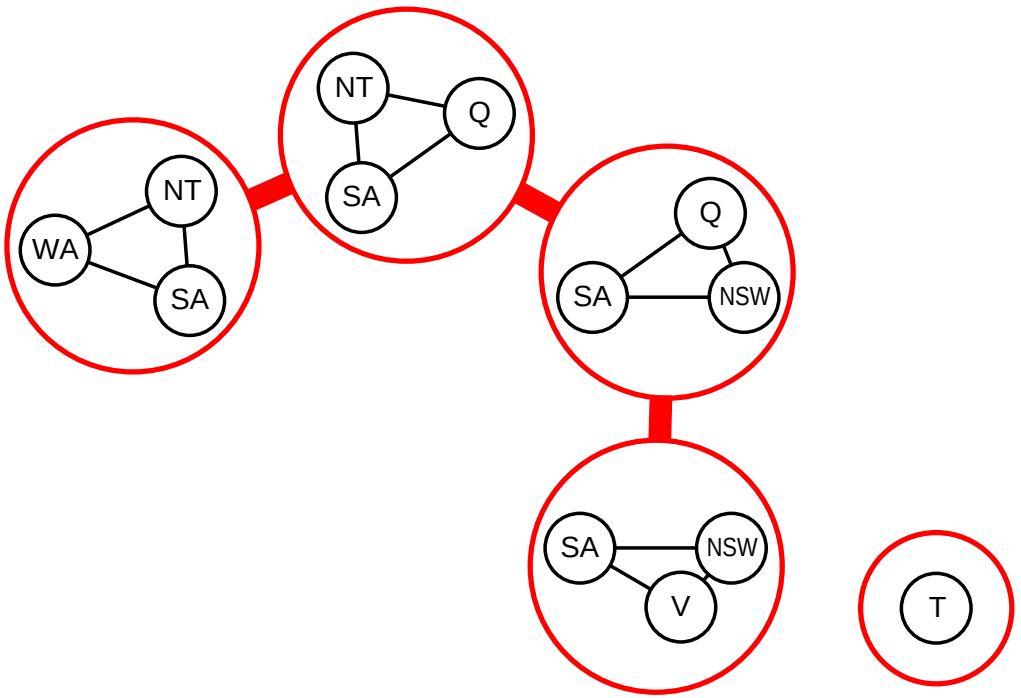
- "Overall solution building problem" based on subsolutions is a constraint network itself (meta constraint network).
- Choose subproblems in a way that the constraint graph of the meta constraint network is a tree/forest.
  $\rightsquigarrow$ build overall solution with efficient tree algorithm

Tree Decomposition: Example

constraint network:



tree decomposition:



Tree Decomposition: Definition

Definition (tree decomposition)

Consider a constraint network $\mathcal{C}$ with variables V.

A tree decomposition of $\mathcal{C}$
is a graph $\mathcal{T}$ with the following properties.

requirements on vertices:

- Every vertex of $\mathcal{T}$ corresponds to a subset of the variables V. Such a vertex (and corresponding variable set) is called a subproblem of $\mathcal{C}$.
- Every variable of V appears in at least one subproblem of $\mathcal{T}$.
- For every nontrivial constraint R_{uv} of $\mathcal{C}$, the variables u and v appear together in at least one subproblem in $\mathcal{T}$.

...

Tree Decomposition: Definition

Definition (tree decomposition)

Consider a constraint network $\mathcal{C}$ with variables V.

A tree decomposition of $\mathcal{C}$
is a graph $\mathcal{T}$ with the following properties.

...

requirements on edges:

- For each variable $v \in V$, let $\mathcal{T}_v$ be the set of vertices corresponding to the subproblems that contain v.
- For each variable v, the set $\mathcal{T}_v$ is connected, i.e., each vertex in $\mathcal{T}_v$ is reachable from every other vertex in $\mathcal{T}_v$ without visiting vertices not contained in $\mathcal{T}_v$.
- $\mathcal{T}$ is acyclic (a tree/forest)

Meta Constraint Network

meta constraint network $\mathcal{C}^{\mathcal{T}} = \langle V^{\mathcal{T}}, \mathrm{dom}^{\mathcal{T}}, (R^{\mathcal{T}}_{uv}) \rangle$
based on tree decomposition $\mathcal{T}$

- $V^{\mathcal{T}} :=$ vertices of $\mathcal{T}$ (i.e., subproblems of $\mathcal{C}$ occurring in $\mathcal{T}$)
- $\mathrm{dom}^{\mathcal{T}}(v) :=$ set of solutions of subproblem v
- $R^{\mathcal{T}}_{uv} := \{\langle s, t \rangle \mid s, t$ compatible solutions of subproblems $u, v\}$ if $\{u, v\}$ is an edge of $\mathcal{T}$. (All constraints between subproblems not connected by an edge of $\mathcal{T}$ are trivial.)

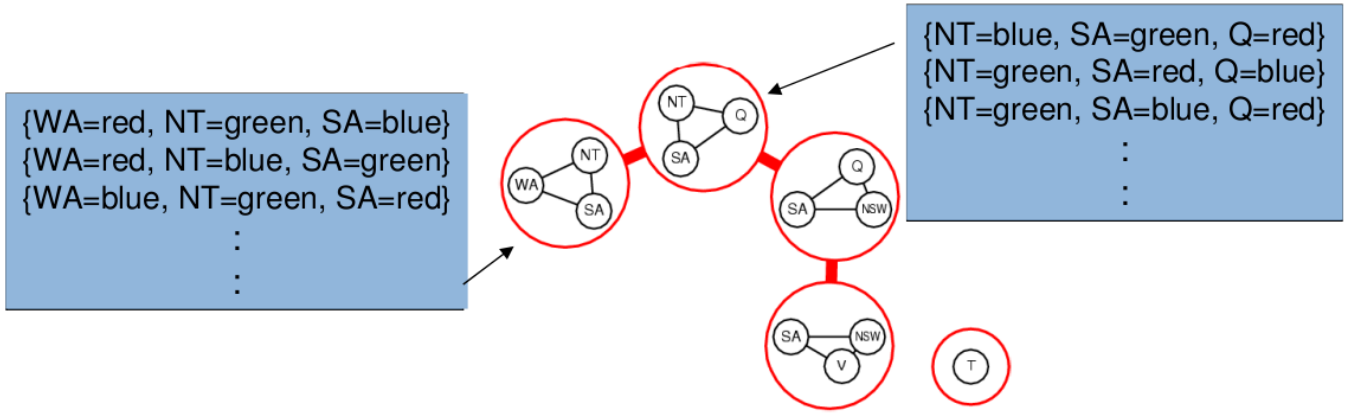
German: Meta-Constraintnetz

Solutions of two subproblems are called compatible
if all overlapping variables are assigned identically.

Solving with Tree Decompositions: Algorithm

algorithm:

- Find all solutions for all subproblems in the decomposition and build a tree-like meta constraint network.
- Constraints in meta constraint network: subsolutions must be compatible.
- Solve meta constraint network with an algorithm for tree-like networks.



Good Tree Decompositions

goal: each subproblem has as few variables as possible

- crucial: subproblem V' in $\mathcal{T}$ with highest number of variables
- number of variables in V' minus 1
  is called width of the decomposition
- best width over all decompositions: tree width
  of the constraint graph (computation is NP-complete)

time complexity of solving algorithm based on tree decompositions:
$O(nk^{w+1})$, where w is width of decomposition
(requires specialized version of revise; otherwise $O(nk^{2w+2})$.)

Summary

Summary: This Chapter

- Reduce complex constraint graphs to simple constraint graphs.
- cutset conditioning:
  - Choose as few variables as possible (cutset) such that an assignment to these variables yields a remaining problem which is structurally simple.
  - search over assignments of variables in cutset
- tree decomposition: build tree-like meta constraint network
  - meta variables: groups of original variables that jointly cover all variables and constraints
  - values correspond to consistent assignments to the groups
  - constraints between overlapping groups to ensure compatibility
  - overall algorithm exponential in width of decomposition (size of largest group)

Summary: CSPs

Constraint Satisfaction Problems (CSP)

General formalism for problems where

- values have to be assigned to variables
- such that the given constraints are satisfied.

- algorithms: backtracking search + inference
  (e.g., forward checking, arc consistency, path consistency)
- variable and value orders important
- more efficient: exploit structure of constraint graph
  (connected components; trees)

More Advanced Topics

more advanced topics (not considered in this course):

- backjumping: backtracking over several layers
- no-good learning: infer additional constraints based on information collected during backtracking
- local search methods in the space of total, but not necessarily consistent assignments
- tractable constraint classes: identification of constraint types that allow for polynomial algorithms
- solutions of different quality: constraint optimization problems (COP)

$\rightsquigarrow$ more than enough content for a one-semester course

Foundations of Artificial Intelligence

E1. Propositional Logic: Syntax and Semantics

Malte Helmert

University of Basel

April 16, 2025

Propositional Logic: Overview

Chapter overview: propositional logic

- E1. Syntax and Semantics
- E2. Equivalence and Normal Forms
- E3. Reasoning and Resolution
- E4. DPLL Algorithm
- E5. Local Search and Outlook

Classification

classification:

Propositional Logic

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

(applications also in more complex environments)

Motivation

Propositional Logic: Motivation

propositional logic

- modeling and representing problems and knowledge
- basis for general problem descriptions and solving strategies
  $\rightsquigarrow$ automated planning (Part F)
- allows for automated reasoning

German: Aussagenlogik, automatisches Schliessen

Relationship to CSPs

- previous part: constraint satisfaction problems
- satisfiability problem in propositional logic can be viewed as non-binary CSP over $\{\mathbf{F}, \mathbf{T}\}$
- formula encodes constraints
- solution: satisfying assignment of values to variables
- backtracking with inference $\approx$ DPLL (Chapter E4)

Propositional Logic: Description of State Spaces

propositional variables for missionaries and cannibals problem:

```
two-missionaries-are-on-left-shore
one-cannibal-is-on-left-shore
boat-is-on-left-shore
...
```

- problem description for general problem solvers
- states represented as truth values of atomic propositions

German: Aussagenvariablen

Propositional Logic: Intuition

propositions: atomic statements over the world
that cannot be divided further

Propositions with logical connectives like
"and", "or" and "not" form the propositional formulas.

German: logische Verknüpfungen

Syntax and Semantics

Today, we define syntax and semantics of propositional logic.
$\rightsquigarrow$ reminder from Discrete Mathematics in Computer Science

syntax:

- defines which symbols are allowed in formulas
  $(,), \aleph, \wedge, A, B, C, X, \heartsuit, \rightarrow, \nearrow, \ldots ?$
- . . . and which sequences of these symbols are correct formulas
  $(A \wedge B), \quad ((A) \wedge B), \quad \wedge)A(B, \ldots ?$

semantics:

- defines the meaning of formulas
- uses interpretations to describe a possible world
  $I = \{A \mapsto \mathbf{T}, B \mapsto \mathbf{F}\}$
- tells us which formulas are true in which worlds

Syntax

Alphabet of Propositions

- Logical formulas use an alphabet Σ of propositions, for example $\Sigma = \{P, Q, R, S\}$ or $\Sigma = \{X_1, X_2, X_3, \dots\}$.
- We do not mention the alphabet in the following.
- More formally, all definitions are parameterized by Σ.

German: Alphabet

Syntax

Definition (propositional formula)

- $\top$ and $\bot$ are formulas (constant true/constant false).
- Every proposition in Σ is a formula (atomic formula).
- If φ is a formula, then $\neg\varphi$ is a formula (negation).
- If φ and ψ are formulas, then so are
  - $(\varphi \wedge \psi)$ (conjunction)
  - $(\varphi \vee \psi)$ (disjunction)
  - $(\varphi \rightarrow \psi)$ (implication)

German: aussagenlogische Formel, konstant wahr/falsch, atomare Formel, Konjunktion, Disjunktion, Implikation

Note: minor differences to Discrete Mathematics course

Abbreviating Notations: Omitting Parenthesis

may omit redundant parentheses:

- outer parentheses of formula:
  - $(P \wedge Q) \vee R$ instead of $((P \wedge Q) \vee R)$
- multiple conjunctions/disjunctions:
  - $P \wedge Q \wedge \neg R \wedge S$ instead of $(((P \wedge Q) \wedge \neg R) \wedge S)$
- implicit binding strength: $(\neg) > (\wedge) > (\vee) > (\rightarrow)$:
  - $P \vee Q \wedge R$ instead of $P \vee (Q \wedge R)$
  - use responsibly

Abbreviating Notations: Prefix Notation

prefix notations used like $\sum$ for sums:

- $\bigvee_{i=1}^{4} X_i$ instead of $(X_1 \vee X_2 \vee X_3 \vee X_4)$
- $\bigwedge_{i=1}^{3} Y_i$ instead of $(Y_1 \wedge Y_2 \wedge Y_3)$

Semantics

Intuition for Semantics

A formula is true or false
depending on the interpretation of the propositions.

Semantics: Intuition

- A proposition P is either true or false.
  The truth value of P is determined by an interpretation.
- The truth value of a formula follows from
  the truth values of the propositions.

Example

example interpretations for $\varphi = (P \vee Q) \wedge R$:

- If P and Q are false and R is true, then φ is false.
- If P is false and Q and R are true, then φ is true.

Interpretations

Definition (interpretation)

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

Interpretations are sometimes called truth assignments.

German: Interpretation/Belegung/Wahrheitsbelegung

The Semantics of Formulas

When is a formula φ true under interpretation I?
symbolically: When does $I \models \varphi$ hold?

Definition (Models and the $\models$ Relation)

The relation "$\models$" is a relation between interpretations and formulas and is defined as follows:

- $I \models \top$ and $I \not\models \bot$
- $I \models P$ if $I(P) = \mathbf{T}$ for $P \in \Sigma$
- $I \models \neg\varphi$ if $I \not\models \varphi$
- $I \models (\varphi \wedge \psi)$ if $I \models \varphi$ and $I \models \psi$
- $I \models (\varphi \vee \psi)$ if $I \models \varphi$ or $I \models \psi$
- $I \models (\varphi \rightarrow \psi)$ if $I \not\models \varphi$ or $I \models \psi$

If $I \models \varphi$ ($I \not\models \varphi$), we say φ is true (false) under I.

Examples

Example (Interpretation I)

$I = \{P \mapsto \mathbf{T}, Q \mapsto \mathbf{T}, R \mapsto \mathbf{F}, S \mapsto \mathbf{F}\}$

Which formulas are true under I?

- $\varphi_1 = \neg(P \wedge Q) \wedge (R \wedge \neg S)$. Does $I \models \varphi_1$ hold?
- $\varphi_2 = (P \wedge Q) \wedge \neg(R \wedge \neg S)$. Does $I \models \varphi_2$ hold?
- $\varphi_3 = (R \rightarrow P)$. Does $I \models \varphi_3$ hold?

Properties of Formulas

Models of Formulas and Sets of Formulas

Definition (model)

An interpretation I is called a model of φ if $I \models \varphi$.

Definition ($I \models \Phi$)

Let Φ be a set of propositional formulas.
We write $I \models \Phi$ if $I \models \varphi$ for all $\varphi \in \Phi$.
Such an interpretation I is called a model of Φ.

If I is a model of formula φ, we also say "I satisfies φ" or "φ holds under I" (similarly for sets of formulas Φ).

German: Modell, erfüllt, gilt unter

Satisfiable, Unsatisfiable, Falsifiable, Valid

Definition (satisfiable etc.)

A formula φ is called

- satisfiable if there exists a model for φ
- unsatisfiable if there exists no model for φ
- valid (= a tautology) if all interpretations are models of φ
- falsifiable if not all interpretations are models of φ

German: erfüllbar, unerfüllbar, allgemeingültig (gültig, Tautologie), falsifizierbar

Truth Tables

Truth Tables

How to determine automatically if a given formula is (un)satisfiable, valid, falsifiable?

$\rightsquigarrow$ simple method: truth tables

example: Is $\varphi = ((P \vee H) \wedge \neg H) \rightarrow P$ valid?

P	H	$P \vee H$	$((P \vee H) \wedge \neg H)$	$((P \vee H) \wedge \neg H) \rightarrow P$
F	**F**	**F**	**F**	**T**
F	**T**	**T**	**F**	**T**
T	**F**	**T**	**T**	**T**
T	**T**	**T**	**F**	**T**

$I \models \varphi$ for all interpretations I, hence φ is valid.

- Is it satisfiable/unsatisfiable/falsifiable?

Terminology (Side Note)

What does "φ is true" mean?

- not formally defined
- the statement misses an interpretation
  - could be meant as "in the obvious interpretation" in some cases
  - or as "in all possible interpretations" (tautology)
- imprecise language $\rightsquigarrow$ avoid

Summary

Summary

- Propositional logic forms the basis for a general representation of problems and knowledge.
- Propositions (atomic formulas) are statements over the world that cannot be divided further.
- Propositional formulas combine constant and atomic formulas with $\neg$, $\wedge$, $\vee$, $\rightarrow$ to more complex statements.
- Interpretations determine which atomic formulas are true and which ones are false.
- Interpretations making a formula true are called models.
- important properties a formula may have: satisfiable, unsatisfiable, valid, falsifiable

Foundations of Artificial Intelligence

E2. Propositional Logic: Equivalence and Normal Forms

Malte Helmert

University of Basel

April 23, 2025

Propositional Logic: Overview

Chapter overview: propositional logic

- E1. Syntax and Semantics
- E2. Equivalence and Normal Forms
- E3. Reasoning and Resolution
- E4. DPLL Algorithm
- E5. Local Search and Outlook

Equivalence

Logical Equivalance

Definition (logically equivalent)

Formulas φ and ψ are called logically equivalent ($\varphi \equiv \psi$) if for all interpretations I: $I \models \varphi$ iff $I \models \psi$.

German: logisch äquivalent

Equivalences

Logical Equivalences

Let φ, ψ, and η be formulas.

- $(\varphi \wedge \psi) \equiv (\psi \wedge \varphi)$ and $(\varphi \vee \psi) \equiv (\psi \vee \varphi)$ (commutativity)
- $((\varphi \wedge \psi) \wedge \eta) \equiv (\varphi \wedge (\psi \wedge \eta))$ and
  $((\varphi \vee \psi) \vee \eta) \equiv (\varphi \vee (\psi \vee \eta))$ (associativity)
- $((\varphi \wedge \psi) \vee \eta) \equiv ((\varphi \vee \eta) \wedge (\psi \vee \eta))$ and
  $((\varphi \vee \psi) \wedge \eta) \equiv ((\varphi \wedge \eta) \vee (\psi \wedge \eta))$ (distributivity)
- $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ and
  $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$ (De Morgan)
- $\neg\neg\varphi \equiv \varphi$ (double negation)
- $(\varphi \rightarrow \psi) \equiv (\neg\varphi \vee \psi)$ ($(\rightarrow)$-elimination)

Commutativity and associativity are often used implicitly

$\rightsquigarrow$ We write $(X_1 \wedge X_2 \wedge X_3 \wedge X_4)$ instead of $(X_1 \wedge (X_2 \wedge (X_3 \wedge X_4)))$

Normal Forms

Normal Forms: Terminology

Definition (literal)

If $P \in \Sigma$, then the formulas P and $\neg P$ are called literals.

P is called positive literal, $\neg P$ is called negative literal.

The complementary literal to P is $\neg P$ and vice versa.
For a literal ℓ, the complementary literal to ℓ is denoted with $\bar{\ell}$.

German: Literal, positives/negatives/komplementäres Literal

Question: What is the difference between $\bar{\ell}$ and $\neg\ell$?

Normal Forms: Terminology

Definition (clause)

A disjunction of 0 or more literals is called a clause.
The empty clause (with 0 literals) is $\bot$.
Clauses consisting of exactly one literal are called unit clauses.

German: Klausel, leere Klausel, Einheitsklausel

Definition (monomial)

A conjunction of 0 or more literals is called a monomial.

German: Monom

Normal Forms

Definition (normal forms)

A formula φ is in conjunctive normal form (CNF, clause form) if φ is a conjunction of 0 or more clauses:

$$\varphi = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} \ell_{i,j} \right)$$

A formula φ is in disjunctive normal form (DNF) if φ is a disjunction of 0 or more monomials:

$$\varphi = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} \ell_{i,j} \right)$$

German: konjunktive Normalform, disjunktive Normalform

Normal Forms

For every propositional formula, there exists
a logically equivalent propositional formula in CNF and in DNF.

Conversion to CNF with equivalences

1. eliminate implications
   $(\varphi \to \psi) \equiv (\neg\varphi \vee \psi)$ ($(\to)$-elimination)
2. move negations inside
   $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ (De Morgan)
   $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$ (De Morgan)
   $\neg\neg\varphi \equiv \varphi$ (double negation)
3. distribute $\vee$ over $\wedge$
   $((\varphi \wedge \psi) \vee \eta) \equiv ((\varphi \vee \eta) \wedge (\psi \vee \eta))$ (distributivity)
4. simplify constant subformulas $(\top, \bot)$

There are formulas φ for which every logically equivalent formula in CNF and DNF is exponentially longer than φ.

Summary

Summary

- two formulas are logically equivalent
  if they have the same models
- different kinds of formulas:
  - atomic formulas and literals
  - clauses and monomials
  - conjunctive normal form (CNF) and
    disjunctive normal form (DNF)
- for every formula, there is a logically equivalent formula
  in CNF and a logically equivalent formula in DNF

Foundations of Artificial Intelligence

E3. Propositional Logic: Reasoning and Resolution

Malte Helmert

University of Basel

April 23, 2025

Propositional Logic: Overview

Chapter overview: propositional logic

- E1. Syntax and Semantics
- E2. Equivalence and Normal Forms
- E3. Reasoning and Resolution
- E4. DPLL Algorithm
- E5. Local Search and Outlook

Reasoning

Reasoning: Intuition

Reasoning: Intuition

- Generally, formulas only represent an incomplete description of the world.
- In many cases, we want to know if a formula logically follows from (a set of) other formulas.
- What does this mean?

Reasoning: Intuition

- example: $\varphi = (P \vee Q) \wedge (R \vee \neg P) \wedge S$
- S holds in every model of φ.
  What about P, Q and R?

$\rightsquigarrow$ consider all models of φ:

- $I_1 = \{P \mapsto \mathbf{F}, Q \mapsto \mathbf{T}, R \mapsto \mathbf{F}, S \mapsto \mathbf{T}\}$
- $I_2 = \{P \mapsto \mathbf{F}, Q \mapsto \mathbf{T}, R \mapsto \mathbf{T}, S \mapsto \mathbf{T}\}$
- $I_3 = \{P \mapsto \mathbf{T}, Q \mapsto \mathbf{F}, R \mapsto \mathbf{T}, S \mapsto \mathbf{T}\}$
- $I_4 = \{P \mapsto \mathbf{T}, Q \mapsto \mathbf{T}, R \mapsto \mathbf{T}, S \mapsto \mathbf{T}\}$

Observation

- In all models of φ, the formula $Q \vee R$ holds as well.
- We say: "$Q \vee R$ logically follows from φ."

Reasoning: Formally

Definition (logical consequence)

Let Φ be a set of formulas. A formula ψ logically follows from Φ (in symbols: $\Phi \models \psi$) if all models of Φ are also models of ψ.

German: logische Konsequenz, folgt logisch

In other words: for each interpretation $\mathcal{I}$,
if $\mathcal{I} \models \varphi$ for all $\varphi \in \Phi$, then also $\mathcal{I} \models \psi$.

Question

How can we automatically compute if $\Phi \models \psi$?

- One possibility: Build a truth table. (How?)
- Are there "better" possibilities that (potentially) avoid generating the whole truth table?

Reasoning: Deduction Theorem

Proposition (deduction theorem)

Let Φ be a finite set of formulas and let ψ be a formula. Then

$$\Phi \models \psi \quad \textit{iff} \quad \Big(\bigwedge_{\varphi \in \Phi} \varphi\Big) \to \psi \textit{ is a tautology.}$$

German: Deduktionssatz

Reasoning: Deduction Theorem

Proposition (deduction theorem)

Let Φ be a finite set of formulas and let ψ be a formula. Then

$$\Phi \models \psi \quad \textit{iff} \quad \Big(\bigwedge_{\varphi \in \Phi} \varphi\Big) \to \psi \textit{ is a tautology.}$$

German: Deduktionssatz

Proof.

$\Phi \models \psi$
iff for each interpretation I: if $I \models \varphi$ for all $\varphi \in \Phi$, then $I \models \psi$
iff for each interpretation I: if $I \models \bigwedge_{\varphi \in \Phi} \varphi$, then $I \models \psi$
iff for each interpretation I: $I \not\models \bigwedge_{\varphi \in \Phi} \varphi$ or $I \models \psi$
iff for each interpretation I: $I \models (\bigwedge_{\varphi \in \Phi} \varphi) \to \psi$
iff $(\bigwedge_{\varphi \in \Phi} \varphi) \to \psi$ is tautology □

Reasoning by Unsatisfiability Testing

Consequence of Deduction Theorem

Reasoning can be reduced to testing unsatisfiability.

Question: Does $\Phi \models \psi$ hold?

Idea:

- Let $\chi = (\bigwedge_{\varphi \in \Phi} \varphi) \rightarrow \psi$.
- We know that $\Phi \models \psi$ iff χ is a tautology.
- A formula is a tautology iff its negation is unsatisfiable.
- Hence, $\Phi \models \psi$ iff $\neg\chi$ is unsatisfiable.
- Use equivalences:
  $\neg\chi = \neg((\bigwedge_{\varphi \in \Phi} \varphi) \rightarrow \psi) \equiv \neg(\neg(\bigwedge_{\varphi \in \Phi} \varphi) \vee \psi)$
  $\equiv (\neg\neg(\bigwedge_{\varphi \in \Phi} \varphi) \wedge \neg\psi) \equiv \bigwedge_{\varphi \in \Phi} \varphi \wedge \neg\psi$
- We have that $\Phi \models \psi$ iff $\bigwedge_{\varphi \in \Phi} \varphi \wedge \neg\psi$ is unsatisfiable.

Algorithm for Reasoning

Question: Does $\Phi \models \psi$ hold?

Algorithm (given an algorithm for testing unsatisfiability):

1. Let $\eta = \bigwedge_{\varphi \in \Phi} \varphi \wedge \neg\psi$.
2. Test if η is unsatisfiable.
3. If yes, return "$\Phi \models \psi$".
4. Otherwise, return "$\Phi \not\models \psi$".

In the following: Can we test unsatisfiability in a more efficient way than by computing the whole truth table?

Resolution

Sets of Clauses

for the rest of this chapter:

- prerequisite: formulas in conjunctive normal form
- clause represented as a set C of literals
- formula represented as a set Δ of clauses

Example

Let $\varphi = (P \vee Q) \wedge \neg P$.

- φ in conjunctive normal form
- φ consists of clauses $(P \vee Q)$ and $\neg P$
- representation of φ as set of sets of literals: $\{\{P, Q\}, \{\neg P\}\}$

Sets of Clauses (Corner Cases)

Distinguish $\bot$ (empty clause = empty set of literals) vs. $\emptyset$ (empty set of clauses).

- $C = \bot\ (= \emptyset)$ represents a disjunction over zero literals:

$$\bigvee_{L \in \emptyset} L = \bot$$

- $\Delta_1 = \{\bot\}$ represents a conjunction over one clause:

$$\bigwedge_{\varphi \in \{\bot\}} \varphi = \bot$$

- $\Delta_2 = \emptyset$ represents a conjunction over zero clauses:

$$\bigwedge_{\varphi \in \emptyset} \varphi = \top$$

Resolution: Idea

Resolution

- method to test CNF formula φ for unsatisfiability
- idea: derive new clauses from φ that logically follow from φ
- if empty clause $\bot$ can be derived $\rightsquigarrow$ φ unsatisfiable

German: Resolution

The Resolution Rule

$$\frac{C_1 \cup \{\ell\}, C_2 \cup \{\bar{\ell}\}}{C_1 \cup C_2}$$

- "From $C_1 \cup \{\ell\}$ and $C_2 \cup \{\bar{\ell}\}$, we can conclude $C_1 \cup C_2$."
- $C_1 \cup C_2$ is resolvent of parent clauses $C_1 \cup \{\ell\}$ and $C_2 \cup \{\bar{\ell}\}$.
- The literals ℓ and $\bar{\ell}$ are called resolution literals, the corresponding proposition is called resolution variable.
- resolvent follows logically from parent clauses (Why?)

German: Resolutionsregel, Resolvent, Elternklauseln, Resolutionsliterale, Resolutionsvariable

Example

- resolvent of $\{A, B, \neg C\}$ and $\{A, D, C\}$?
- resolvents of $\{\neg A, B, \neg C\}$ and $\{A, D, C\}$?

Resolution: Derivations

Definition (derivation)

Notation: $R(\Delta) = \Delta \cup \{C \mid C \text{ is resolvent of two clauses in } \Delta\}$

A clause D can be derived from Δ (in symbols $\Delta \vdash D$) if there is a sequence of clauses $C_1, \ldots, C_n = D$ such that for all $i \in \{1, \ldots, n\}$ we have $C_i \in R(\Delta \cup \{C_1, \ldots, C_{i-1}\})$.

German: Ableitung, abgeleitet

Lemma (soundness of resolution)

If $\Delta \vdash D$, then $\Delta \models D$.

Does the converse direction hold as well (completeness)?

German: Korrektheit, Vollständigkeit

Resolution: Completeness?

The converse of the lemma does not hold in general.

example:

- $\{\{A, B\}, \{\neg B, C\}\} \models \{A, B, C\}$, but
- $\{\{A, B\}, \{\neg B, C\}\} \not\vdash \{A, B, C\}$

but: converse holds for special case of empty clause $\bot$ (no proof)

Theorem (refutation-completeness of resolution)

Δ is unsatisfiable iff $\Delta \vdash \bot$

German: Widerlegungsvollständigkeit

consequences:

- Resolution is a complete proof method for testing unsatisfiability of CNF formulas.
- Resolution can be used for general reasoning by reducing to a test for unsatisfiability of CNF formulas.

Example

Let $\Phi = \{P \vee Q, \neg P\}$. Does $\Phi \models Q$ hold?

Solution

- test if $((P \vee Q) \wedge \neg P) \rightarrow Q$ is tautology
- equivalently: test if $((P \vee Q) \wedge \neg P) \wedge \neg Q$ is unsatisfiable
- resulting set of clauses: $\Phi' = \{\{P, Q\}, \{\neg P\}, \{\neg Q\}\}$
- resolving $\{P, Q\}$ with $\{\neg P\}$ yields $\{Q\}$
- resolving $\{Q\}$ with $\{\neg Q\}$ yields $\bot$
- observation: empty clause can be derived, hence Φ' unsatisfiable
- consequently $\Phi \models Q$

Resolution: Discussion

- Resolution is a complete proof method
  to test formulas for unsatisfiability.
- In the worst case, resolution proofs can take exponential time.
- In practice, a strategy which determines
  the next resolution step is needed.
- In the following chapter, we discuss the DPLL algorithm,
  which is a combination of backtracking and resolution.

Summary

Summary

- Reasoning: the formula ψ follows from the set of formulas Φ if all models of Φ are also models of ψ.
- Reasoning can be reduced to testing validity (with the deduction theorem).
- Testing validity can be reduced to testing unsatisfiability.
- Resolution is a refutation-complete proof method applicable to formulas in conjunctive normal form.

$\rightsquigarrow$ can be used to test if a set of clauses is unsatisfiable

Foundations of Artificial Intelligence

E4. Propositional Logic: DPLL Algorithm

Malte Helmert

University of Basel

April 28, 2025

Propositional Logic: Overview

Chapter overview: propositional logic

- E1. Syntax and Semantics
- E2. Equivalence and Normal Forms
- E3. Reasoning and Resolution
- E4. DPLL Algorithm
- E5. Local Search and Outlook

Motivation

Propositional Logic: Motivation

- Propositional logic allows for the representation of knowledge and for deriving conclusions based on this knowledge.
- many practical applications can be directly encoded, e.g.
  - constraint satisfaction problems of all kinds
  - circuit design and verification
- many problems contain logic as ingredient, e.g.
  - automated planning
  - general game playing
  - description logic queries (semantic web)

Propositional Logic: Algorithmic Problems

main problems:

- reasoning ($\Phi \models \varphi$?):
  Does the formula φ logically follow from the formulas Φ?
- equivalence ($\varphi \equiv \psi$):
  Are the formulas φ and ψ logically equivalent?
- satisfiability (SAT):
  Is formula φ satisfiable? If yes, find a model.

German: Schlussfolgern, Äquivalenz, Erfüllbarkeit

The Satisfiability Problem

The Satisfiability Problem (SAT)

given:

propositional formula in conjunctive normal form (CNF)

usually represented as pair $\langle V, \Delta \rangle$:

- V set of propositional variables (propositions)
- Δ set of clauses over V
  (clause = set of literals v or $\neg v$ with $v \in V$)

find:

- satisfying interpretation (model)
- or proof that no model exists

SAT is a famous NP-complete problem (Cook 1971; Levin 1973).

Relevance of SAT

- The name "SAT" is often used for the satisfiability problem for general propositional formulas (instead of restriction to CNF).
- General SAT can be reduced to CNF case in linear time.
- All previously mentioned problems can be reduced to SAT or its complement UNSAT (is a given CNF formula unsatisfiable?) in linear time.

$\rightsquigarrow$ SAT algorithms important and intensively studied

this and next chapter: SAT algorithms

Systematic Search: DPLL

SAT vs. CSP

SAT can be considered a constraint satisfaction problem:

- CSP variables = propositions
- domains = $\{\mathbf{F}, \mathbf{T}\}$
- constraints = clauses

However, we often have constraints that affect > 2 variables.

Due to this relationship, all ideas for CSPs are applicable to SAT:

- search
- inference
- variable and value orders

The DPLL Algorithm

The DPLL algorithm (Davis/Putnam/Logemann/Loveland) corresponds to backtracking with inference for CSPs.

- recursive call DPLL(Δ, I)
  for clause set Δ and partial interpretation I
- result is a model of Δ that extends I;
  unsatisfiable if no such model exists
- first call DPLL($\Delta, \emptyset$)

inference in DPLL:

- simplify: after assigning value d to variable v,
  simplify all clauses that contain v
  $\rightsquigarrow$ forward checking (for constraints of arbitrary arity)
- unit propagation: variables that occur in clauses without other variables (unit clauses) are assigned immediately
  $\rightsquigarrow$ minimum remaining values variable order

The DPLL Algorithm: Pseudo-Code

function DPLL(Δ, I):

if $\bot \in \Delta$: [empty clause exists $\rightsquigarrow$ unsatisfiable]
    return unsatisfiable
else if $\Delta = \emptyset$: [no clauses left $\rightsquigarrow$ interpretation I satisfies formula]
    return I
else if there exists a unit clause $\{v\}$ or $\{\neg v\}$ in Δ: [unit propagation]
    Let v be such a variable, d the truth value that satisfies the clause.
    $\Delta' := \text{simplify}(\Delta, v, d)$
    return DPLL($\Delta', I \cup \{v \mapsto d\}$)
else: [splitting rule]
    Select some variable v which occurs in Δ.
    for each $d \in \{\mathbf{F}, \mathbf{T}\}$ in some order:
        $\Delta' := \text{simplify}(\Delta, v, d)$
        $I' :=$ DPLL($\Delta', I \cup \{v \mapsto d\}$)
        if $I' \neq$ **unsatisfiable**
            return I'
    return unsatisfiable

The DPLL Algorithm: simplify

function simplify(Δ, v, d)

Let ℓ be the literal for v that is satisfied by $v \mapsto d$.
$\Delta' := \{C \mid C \in \Delta \text{ such that } \ell \notin C\}$
$\Delta'' := \{C \setminus \{\bar{\ell}\} \mid C \in \Delta'\}$
return Δ''

- Remove clauses containing ℓ
  $\rightsquigarrow$ clause is satisfied by $v \mapsto d$
- Remove $\bar{\ell}$ from remaining clauses
  $\rightsquigarrow$ clause has to be satisfied with another variable

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

2a. $X \mapsto \mathbf{F}$
$\{\{Y\}, \{\neg Y\}\}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

2a. $X \mapsto \mathbf{F}$
$\{\{Y\}, \{\neg Y\}\}$

3a. unit propagation: $Y \mapsto \mathbf{T}$
$\{\bot\}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

2a. $X \mapsto \mathbf{F}$
$\{\{Y\}, \{\neg Y\}\}$

3a. unit propagation: $Y \mapsto \mathbf{T}$
$\{\bot\}$

2b. $X \mapsto \mathbf{T}$
$\{\{\neg Y\}\}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

2a. $X \mapsto \mathbf{F}$
$\{\{Y\}, \{\neg Y\}\}$

3a. unit propagation: $Y \mapsto \mathbf{T}$
$\{\bot\}$

2b. $X \mapsto \mathbf{T}$
$\{\{\neg Y\}\}$

3b. unit propagation: $Y \mapsto \mathbf{F}$
$\{\}$

Example (1)

$$\Delta = \{\{X, Y, \neg Z\}, \{\neg X, \neg Y\}, \{Z\}, \{X, \neg Y\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{X, Y\}, \{\neg X, \neg Y\}, \{X, \neg Y\}\}$
2. splitting rule:

2a. $X \mapsto \mathbf{F}$
$\{\{Y\}, \{\neg Y\}\}$

3a. unit propagation: $Y \mapsto \mathbf{T}$
$\{\bot\}$

2b. $X \mapsto \mathbf{T}$
$\{\{\neg Y\}\}$

3b. unit propagation: $Y \mapsto \mathbf{F}$
$\{\}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{W, \neg X, \neg Y\}, \{X\}, \{Y\}\}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{W, \neg X, \neg Y\}, \{X\}, \{Y\}\}$
2. unit propagation: $X \mapsto \mathbf{T}$
   $\{\{W, \neg Y\}, \{Y\}\}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{W, \neg X, \neg Y\}, \{X\}, \{Y\}\}$
2. unit propagation: $X \mapsto \mathbf{T}$
   $\{\{W, \neg Y\}, \{Y\}\}$
3. unit propagation: $Y \mapsto \mathbf{T}$
   $\{\{W\}\}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{W, \neg X, \neg Y\}, \{X\}, \{Y\}\}$
2. unit propagation: $X \mapsto \mathbf{T}$
   $\{\{W, \neg Y\}, \{Y\}\}$
3. unit propagation: $Y \mapsto \mathbf{T}$
   $\{\{W\}\}$
4. unit propagation: $W \mapsto \mathbf{T}$
   $\{\}$

Example (2)

$$\Delta = \{\{W, \neg X, \neg Y, \neg Z\}, \{X, \neg Z\}, \{Y, \neg Z\}, \{Z\}\}$$

1. unit propagation: $Z \mapsto \mathbf{T}$
   $\{\{W, \neg X, \neg Y\}, \{X\}, \{Y\}\}$
2. unit propagation: $X \mapsto \mathbf{T}$
   $\{\{W, \neg Y\}, \{Y\}\}$
3. unit propagation: $Y \mapsto \mathbf{T}$
   $\{\{W\}\}$
4. unit propagation: $W \mapsto \mathbf{T}$
   $\{\}$

Properties of DPLL

- DPLL is sound and complete.
- DPLL computes a model if a model exists.
  - Some variables possibly remain unassigned in the solution I; their values can be chosen arbitrarily.
- time complexity in general exponential

$\rightsquigarrow$ important in practice: good variable order and additional inference methods (in particular clause learning)

- Best known SAT algorithms are based on DPLL.

DPLL on Horn Formulas

Horn Formulas

important special case: Horn formulas

Definition (Horn formula)

A Horn clause is a clause with at most one positive literal,
i.e., of the form

$$\neg x_1 \vee \cdots \vee \neg x_n \vee y \text{ or } \neg x_1 \vee \cdots \vee \neg x_n$$

($n = 0$ is allowed.)

A Horn formula is a propositional formula
in conjunctive normal form that only consists of Horn clauses.

German: Hornformel

- foundation of logic programming (e.g., PROLOG)
- critical in many kinds of practical reasoning problems

DPLL on Horn Formulas

Proposition (DPLL on Horn formulas)

If the input formula φ is a Horn formula, then
the time complexity of DPLL is polynomial in the length of φ.

Proof.

properties:

1. If Δ is a Horn formula, then so is simplify(Δ, v, d). (Why?)

   $\rightsquigarrow$ all formulas encountered during DPLL search are Horn formulas if input is Horn formula

2. Every Horn formula without empty or unit clauses is satisfiable:
   - all such clauses consist of at least two literals
   - Horn property: at least one of them is negative
   - assigning **F** to all variables satisfies formula

...

DPLL on Horn Formulas (Continued)

Proof (continued).

3. From 2. we can conclude:
   - if splitting rule applied, then current formula satisfiable, and
   - if a wrong decision is taken, then this will be recognized without applying further splitting rules (i.e., only by applying unit propagation and by deriving the empty clause).
4. Hence the generated search tree for n variables can only contain at most n nodes where the splitting rule is applied (i.e., where the tree branches).
5. It follows that the search tree is of polynomial size, and hence the runtime is polynomial.

Summary

Summary

- satisfiability basic problem in propositional logic
  to which other problems can be reduced
- here: satisfiability for CNF formulas
- Davis-Putnam-Logemann-Loveland procedure (DPLL):
  systematic backtracking search with unit propagation
  as inference method
- DPLL successful in practice, in particular when combined
  with other ideas such as clause learning
- polynomial on Horn formulas
  (= at most one positive literal per clause)

Foundations of Artificial Intelligence

E5. Propositional Logic: Local Search and Outlook

Malte Helmert

University of Basel

April 28, 2025

Propositional Logic: Overview

Chapter overview: propositional logic

- E1. Syntax and Semantics
- E2. Equivalence and Normal Forms
- E3. Reasoning and Resolution
- E4. DPLL Algorithm
- E5. Local Search and Outlook

Local Search: GSAT

Local Search for SAT

- Apart from systematic search, there are also successful local search methods for SAT.
- These are usually not complete and in particular cannot prove unsatisfiability for a formula.
- They are often still interesting because they can find models for hard problems.
- However, all in all, DPLL-based methods have been more successful in recent years.

Local Search for SAT: Ideas

local search methods directly applicable to SAT:

- candidates: (complete) assignments
- solutions: satisfying assignments
- search neighborhood: change assignment of one variable
- heuristic: depends on algorithm; e.g., #unsatisfied clauses

GSAT (Greedy SAT): Pseudo-Code

auxiliary functions:

- violated(Δ, I): number of clauses in Δ not satisfied by I
- flip(I, v): assignment that results from I when changing the valuation of proposition v

function GSAT(Δ):

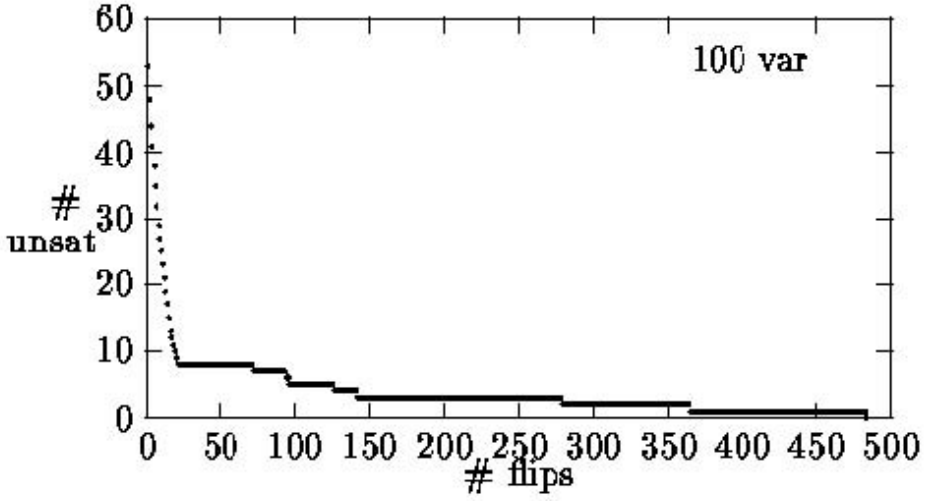
repeat *max-tries* **times**:
    $I :=$ a random assignment
    repeat *max-flips* **times**:
        if $I \models \Delta$:
            return I
        $V_{\text{greedy}} :=$ the set of variables v occurring in Δ
                for which violated$(\Delta, \text{flip}(I, v))$ is minimal
        randomly select $v \in V_{\text{greedy}}$
        $I := \text{flip}(I, v)$
return no solution found

GSAT: Discussion

GSAT has the usual ingredients of local search methods:

- hill climbing
- randomness (although relatively little!)
- restarts

empirically, much time is spent on plateaus:



Local Search: Walksat

Walksat: Pseudo-Code

$\text{lost}(\Delta, I, v)$: #clauses in Δ satisfied by I, but not by $\text{flip}(I, v)$

function Walksat(Δ):

repeat *max-tries* **times**:
    $I :=$ a random assignment
    repeat *max-flips* **times**:
        if $I \models \Delta$:
            return I
        $C :=$ randomly chosen unsatisfied clause in Δ
        if there is a variable v in C with $\text{lost}(\Delta, I, v) = 0$:
            $V_{\text{choices}} :=$ all such variables in C
        else with probability p_{noise}:
            $V_{\text{choices}} :=$ all variables occurring in C
        else:
            $V_{\text{choices}} :=$ variables v in C that minimize $\text{lost}(\Delta, I, v)$
        randomly select $v \in V_{\text{choices}}$
        $I := \text{flip}(I, v)$
return no solution found

Walksat vs. GSAT

Comparison GSAT vs. Walksat:

- much more randomness in Walksat
  because of random choice of considered clause
- "counter-intuitive" steps that temporarily increase
  the number of unsatisfied clauses are possible in Walksat

$\rightsquigarrow$ smaller risk of getting stuck in local minima

How Difficult Is SAT?

How Difficult is SAT in Practice?

- SAT is NP-complete.

$\rightsquigarrow$ known algorithms like DPLL need exponential time in the worst case

- What about the average case?
- depends on how the average is computed (no "obvious" way to define the average)

SAT: Polynomial Average Runtime

Good News (Goldberg 1979)

construct random CNF formulas
with n variables and k clauses as follows:

In every clause, every variable occurs

- positively with probability $\frac{1}{3}$,
- negatively with probability $\frac{1}{3}$,
- not at all with probability $\frac{1}{3}$.

Then the runtime of DPLL in the average case
is polynomial in n and k.

$\rightsquigarrow$ not a realistic model for practically relevant CNF formulas
(because almost all of the random formulas are satisfiable)

Phase Transitions

How to find interesting random problems?

conjecture of Cheeseman et al.:

Cheeseman et al., IJCAI 1991

Every NP-complete problem has at least one size parameter such that the difficult instances are close to a critical value of this parameter.

This so-called phase transition separates two problem regions, e.g., an over-constrained and an under-constrained region.

$\rightsquigarrow$ confirmed for, e.g., graph coloring, Hamiltonian paths and SAT

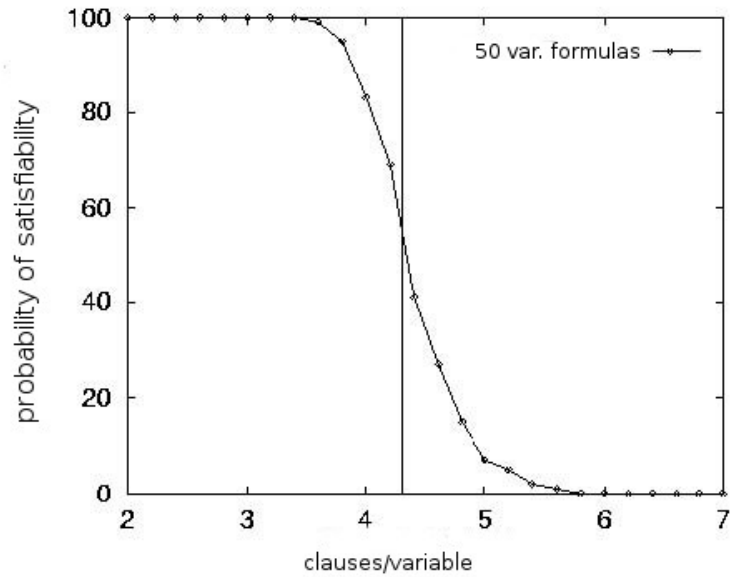
Phase Transitions for 3-SAT

Problem Model of Mitchell et al., AAAI 1992

- fixed clause size of 3
- in every clause, choose the variables randomly
- literals positive or negative with equal probability

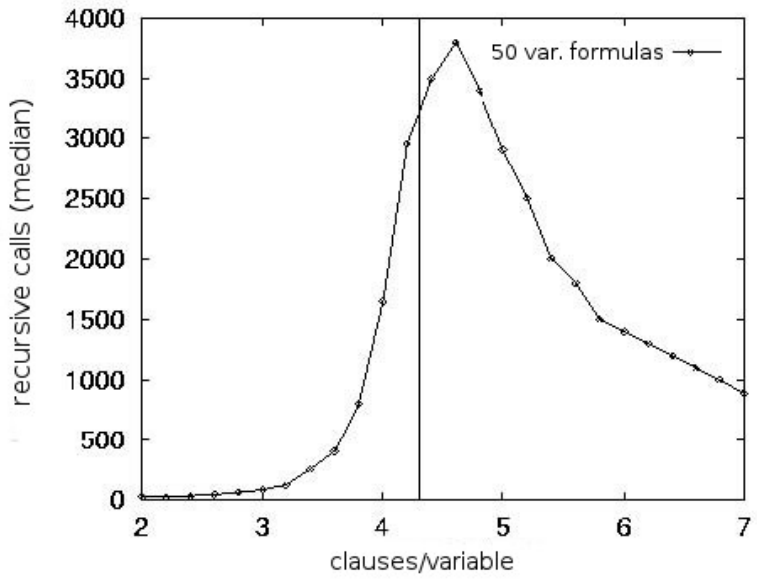
critical parameter: #clauses divided by #variables

phase transition at ratio ≈ 4.3



Phase Transition of DPLL

DPLL shows high runtime close to the phase transition region:



Phase Transition: Intuitive Explanation

- If there are many clauses and hence the instance is unsatisfiable with high probability, this can be shown efficiently with unit propagation.
- If there are few clauses, there are many satisfying assignments, and it is easy to find one of them.
- Close to the phase transition, there are many "almost-solutions" that have to be considered by the search algorithm.

Outlook

State of the Art

- research on SAT in general:
  $\rightsquigarrow$ `http://www.satlive.org/`
- conferences on SAT since 1996 (annually since 2000)
  $\rightsquigarrow$ `http://www.satisfiability.org/`
- competitions for SAT algorithms since 1992
  $\rightsquigarrow$ `http://www.satcompetition.org/`
  - largest instances have more than 1 000 000 literals
  - different tracks (e.g., SAT vs. SAT+UNSAT; industrial vs. random instances)

More Advanced Topics

DPLL-based SAT algorithms:

- efficient implementation techniques
- accurate variable orders
- clause learning

local search algorithms:

- efficient implementation techniques
- adaptive search methods ("difficult" clauses are recognized after some time and then prioritized)

SAT modulo theories:

- extension with background theories (e.g., real numbers, data structures, ...)

Summary

Summary (1)

- local search for SAT searches in the space of interpretations; neighbors: assignments that differ only in one variable
- has typical properties of local search methods: evaluation functions, randomization, restarts
- example: GSAT (Greedy SAT)
  - hill climbing with heuristic function: #unsatisfied clauses
  - randomization through tie-breaking and restarts
- example: Walksat
  - focuses on randomly selected unsatisfied clauses
  - does not follow the heuristic always, but also injects noise
  - consequence: more randomization as GSAT and lower risk of getting stuck in local minima

Summary (2)

- more detailed analysis of SAT shows: the problem is NP-complete, but not all instances are difficult
- randomly generated SAT instances are
  easy to satisfy if they contain few clauses, and
  easy to prove unsatisfiable if they contain many clauses
- in between: phase transition

Foundations of Artificial Intelligence

F1. Automated Planning: Introduction

Malte Helmert

University of Basel

April 30, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

Classification

classification:

Automated Planning

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent

problem solving method:

- problem-specific vs. general vs. learning

(applications also in more complex environments)

Introduction

Automated Planning

What is Automated Planning?

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

— P. Haslum

$\rightsquigarrow$ finding plans (sequences of actions)
that lead from an initial state to a goal state

our topic in this course: classical planning

- general approach to finding solutions
  for state-space search problems (Part B)
- classical = static, deterministic, fully observable
- variants: probabilistic planning, planning under partial observability, online planning, ...

Planning: Informally

given:

- state space description in terms of suitable problem description language (planning formalism)

required:

- a plan, i.e., a solution for the described state space (sequence of actions from initial state to goal)
- or a proof that no plan exists

distinguish between

- optimal planning: guarantee that returned plans are optimal, i.e., have minimal overall cost
- suboptimal planning (satisficing): suboptimal plans are allowed

What is New?

Many previously encountered problems are planning tasks:

- blocks world
- missionaries and cannibals
- 15-puzzle

New: we are now interested in general algorithms, i.e., the developer of the search algorithm does not know the tasks that the algorithm needs to solve.

- $\rightsquigarrow$ no problem-specific heuristics!
- $\rightsquigarrow$ input language to model the planning task

Repetition: State Spaces

Formal Models for State-Space Search

To cleanly study search problems we need a formal model.

Nothing New Here!

This section is a repetition of Section B1.2
of the chapter "State-Space Search: State Spaces".

State Spaces

Definition (state space)

A state space or transition system is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ with

- finite set of states S
- finite set of actions A
- action costs $cost : A \to \mathbb{R}_0^+$
- transition relation $T \subseteq S \times A \times S$ that is deterministic in $\langle s, a \rangle$ (see next slide)
- initial state $s_I \in S$
- set of goal states $S_G \subseteq S$

German: Zustandsraum, Transitionssystem, Zustände, Aktionen, Aktionskosten, Transitions-/Übergangsrelation, deterministisch, Anfangszustand, Zielzustände

State Spaces: Terminology & Notation

Definition (transition, deterministic)

Let $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_I, S_G \rangle$ be a state space.

The triples $\langle s, a, s' \rangle \in T$ are called (state) transitions.

We say $\mathcal{S}$ has the transition $\langle s, a, s' \rangle$ if $\langle s, a, s' \rangle \in T$.
We write this as $s \xrightarrow{a} s'$, or $s \to s'$ when a does not matter.

Transitions are deterministic in $\langle s, a \rangle$: it is forbidden to have both $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ with $s_1 \neq s_2$.

Graph Interpretation

state spaces are often depicted as directed, labeled graphs

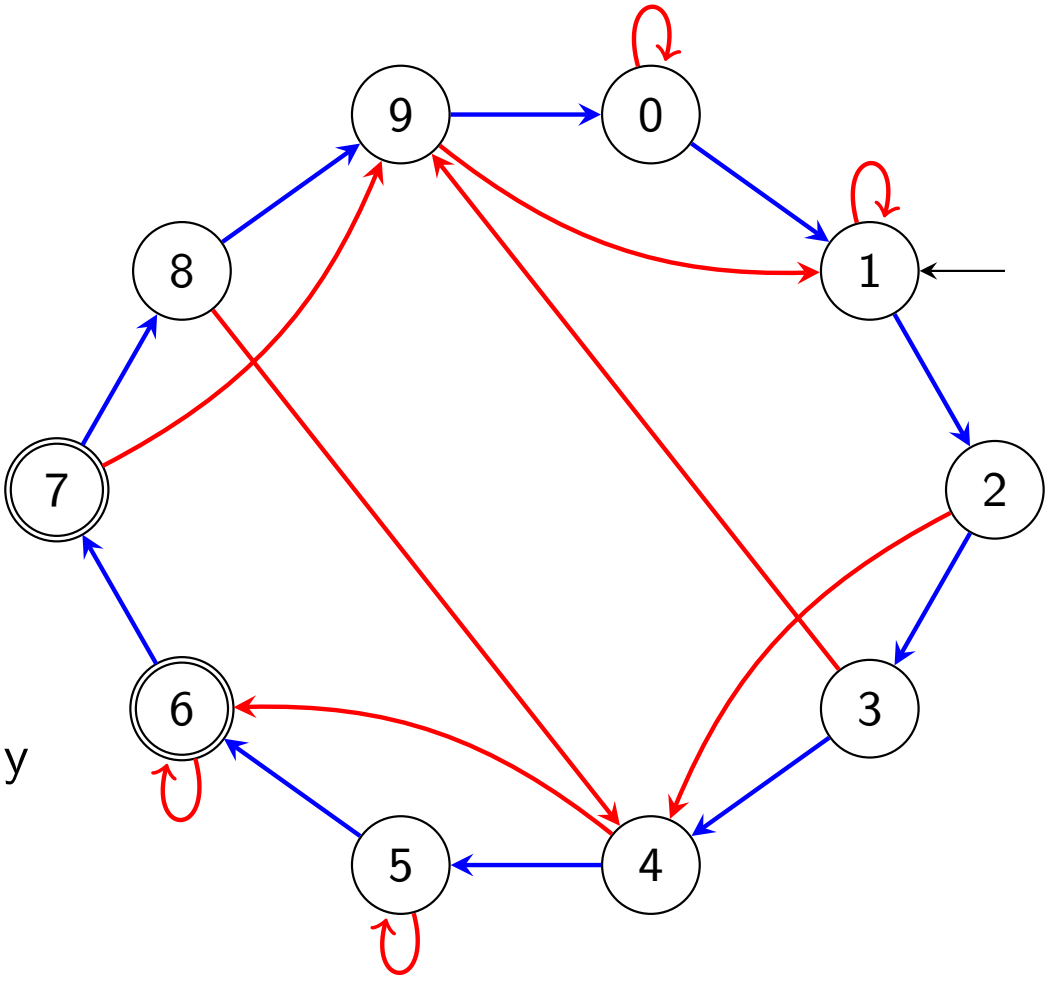
- states: graph vertices
- transitions: labeled arcs

- initial state: incoming arrow
- goal states: double circles
- actions: the arc labels
- action costs: described separately
  (or implicitly $= 1$)

Graph Interpretation

state spaces are often depicted as directed, labeled graphs

- states: graph vertices
- transitions: labeled arcs
  (here: colors instead of labels)
- initial state: incoming arrow
- goal states: double circles
- actions: the arc labels
- action costs: described separately
  (or implicitly $= 1$)



State Spaces: Terminology

terminology:

- predecessor, successor
- applicable action
- path, length, costs
- reachable
- solution, optimal solution

German: Vorgänger, Nachfolger, anwendbare Aktion, Pfad, Länge, Kosten, erreichbar, Lösung, optimale Lösung

Compact Descriptions

State Spaces with Declarative Representations

How do we represent state spaces in the computer?

previously: as black box

now: as declarative description

reminder: Chapter B2

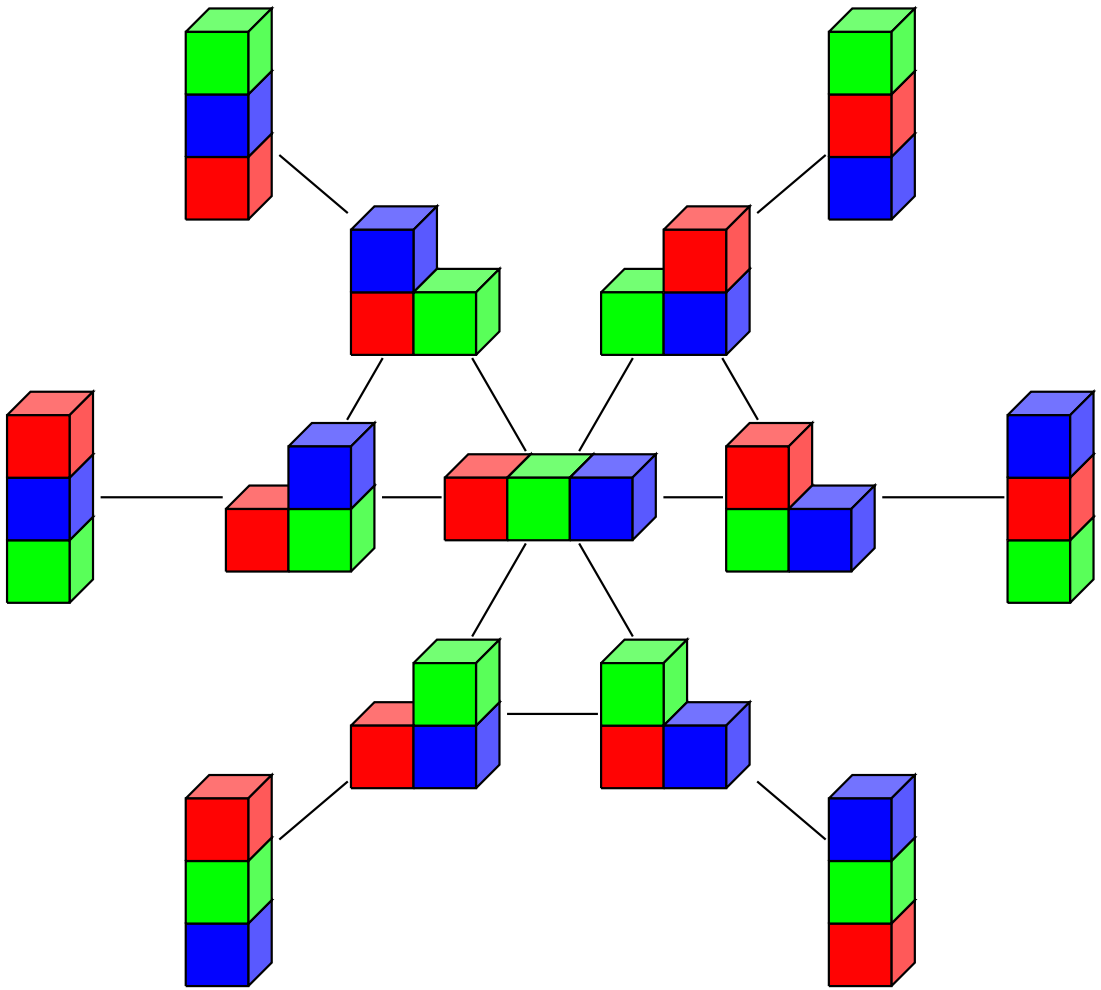
State Spaces with Declarative Representations

represent state spaces declaratively:

- compact description of state space as input to algorithms
  $\rightsquigarrow$ state spaces exponentially larger than the input
- algorithms directly operate on compact description

$\rightsquigarrow$ allows automatic reasoning about problem:
reformulation, simplification, abstraction, etc.

Reminder: Blocks World



problem: n blocks $\rightsquigarrow$ more than $n!$ states

Compact Description of State Spaces

How to describe state spaces compactly?

Compact Description of Several States

- introduce state variables
- states: assignments to state variables

$\rightsquigarrow$ e.g., n binary state variables can describe 2^n states

- transitions and goal states are compactly described with a logic-based formalism

different variants: different planning formalisms

Summary

Summary

- planning: search in general state spaces
- input: compact, declarative description of state space

Foundations of Artificial Intelligence

F2. Automated Planning: Planning Formalisms

Malte Helmert

University of Basel

April 30, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

Four Formalisms

Four Planning Formalisms

- A description language for state spaces (planning tasks) is called a planning formalism.
- We introduce four planning formalisms:
  1. STRIPS (Stanford Research Institute Problem Solver)
  2. ADL (Action Description Language)
  3. SAS$^+$ (Simplified Action Structures)
  4. PDDL (Planning Domain Definition Language)
- STRIPS and SAS$^+$ are the most simple formalisms; in the next chapters, we only consider these.

STRIPS

STRIPS: Basic Concepts

basic concepts of STRIPS:

- STRIPS is the most simple common planning formalism.
- state variables are binary (true or false)

STRIPS: Basic Concepts

basic concepts of STRIPS:

- STRIPS is the most simple common planning formalism.
- state variables are binary (true or false)
- states s (based on a given set of state variables V)
  can be represented in two equivalent ways:
  - as assignments $s : V \to \{\mathbf{F}, \mathbf{T}\}$
  - as sets $s \subseteq V$,
    where s encodes the set of state variables that are true in s

  We will use the set representation.

STRIPS: Basic Concepts

basic concepts of STRIPS:

- STRIPS is the most simple common planning formalism.
- state variables are binary (true or false)
- states s (based on a given set of state variables V)
  can be represented in two equivalent ways:
  - as assignments $s : V \to \{\mathbf{F}, \mathbf{T}\}$
  - as sets $s \subseteq V$,
    where s encodes the set of state variables that are true in s

  We will use the set representation.
- goals and preconditions of actions
  are given as sets of variables that must be true
  (values of other variables do not matter)
- effects of actions are given as sets of variables
  that are set to true and set to false, respectively

STRIPS Planning Task

Definition (STRIPS Planning Task)

A STRIPS planning task is a 4 tuple $\Pi = \langle V, I, G, A \rangle$ with

- V: finite set of state variables
- $I \subseteq V$: the initial state
- $G \subseteq V$: the set of goals
- A: finite set of actions,
  where for all actions $a \in A$, the following is defined:
  - $pre(a) \subseteq V$: the preconditions of a
  - $add(a) \subseteq V$: the add effects of a
  - $del(a) \subseteq V$: the delete effects of a
  - $cost(a) \in \mathbb{N}_0$: the costs of a

German: STRIPS-Planungsaufgabe, Zustandsvariablen, Anfangszustand, Ziele, Aktionen, Add-/Delete-Effekte, Kosten

remark: action costs are an extension of "traditional" STRIPS

State Space for STRIPS Planning Task

Definition (state space induced by STRIPS planning task)

Let $\Pi = \langle V, I, G, A \rangle$ be a STRIPS planning task.

Then Π induces the state space $\mathcal{S}(\Pi) = \langle S, A, \mathit{cost}, T, s_\mathsf{I}, S_\mathsf{G} \rangle$:

- set of states: $S = 2^V$ (= power set of V)
- actions: actions A as defined in Π
- action costs: cost as defined in Π
- transitions: $s \xrightarrow{a} s'$ for states $s, s' \in S$ and action $a \in A$ iff
  - $\mathit{pre}(a) \subseteq s$ (preconditions satisfied)
  - $s' = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$ (effects are applied)
- initial state: $s_\mathsf{I} = I$
- goal states: $s \in S_\mathsf{G}$ for state s iff $G \subseteq s$ (goals reached)

German: induziert den Zustandsraum

Example: Blocks World in STRIPS

Example (A Blocks World Planning Task in STRIPS)

$\Pi = \langle V, I, G, A \rangle$ with:

- $V = \{on_{R,B}, on_{R,G}, on_{B,R}, on_{B,G}, on_{G,R}, on_{G,B},$
  $\text{on-table}_R, \text{on-table}_B, \text{on-table}_G,$
  $\text{clear}_R, \text{clear}_B, \text{clear}_G\}$
- $I = \{on_{G,R}, \text{on-table}_R, \text{on-table}_B, \text{clear}_G, \text{clear}_B\}$
- $G = \{on_{R,B}, on_{B,G}\}$
- $A = \{\text{move}_{R,B,G}, \text{move}_{R,G,B}, \text{move}_{B,R,G},$
  $\text{move}_{B,G,R}, \text{move}_{G,R,B}, \text{move}_{G,B,R},$
  $\text{to-table}_{R,B}, \text{to-table}_{R,G}, \text{to-table}_{B,R},$
  $\text{to-table}_{B,G}, \text{to-table}_{G,R}, \text{to-table}_{G,B},$
  $\text{from-table}_{R,B}, \text{from-table}_{R,G}, \text{from-table}_{B,R},$
  $\text{from-table}_{B,G}, \text{from-table}_{G,R}, \text{from-table}_{G,B}\}$

...

Example: Blocks World in STRIPS

Example (A Blocks World Planning Task in STRIPS)

move actions encode moving a block
from one block to another

example:

- $pre(move_{R,B,G}) = \{on_{R,B}, clear_R, clear_G\}$
- $add(move_{R,B,G}) = \{on_{R,G}, clear_B\}$
- $del(move_{R,B,G}) = \{on_{R,B}, clear_G\}$
- $cost(move_{R,B,G}) = 1$

Example: Blocks World in STRIPS

Example (A Blocks World Planning Task in STRIPS)

to-table actions encode moving a block
from a block to the table

example:

- $pre(\textit{to-table}_{R,B}) = \{on_{R,B}, clear_R\}$
- $add(\textit{to-table}_{R,B}) = \{\textit{on-table}_R, clear_B\}$
- $del(\textit{to-table}_{R,B}) = \{on_{R,B}\}$
- $cost(\textit{to-table}_{R,B}) = 1$

Example: Blocks World in STRIPS

Example (A Blocks World Planning Task in STRIPS)

from-table actions encode moving a block
from the table to a block

example:

- $pre(\textit{from-table}_{R,B}) = \{\textit{on-table}_R, \textit{clear}_R, \textit{clear}_B\}$
- $add(\textit{from-table}_{R,B}) = \{\textit{on}_{R,B}\}$
- $del(\textit{from-table}_{R,B}) = \{\textit{on-table}_R, \textit{clear}_B\}$
- $cost(\textit{from-table}_{R,B}) = 1$

Why STRIPS?

- STRIPS is particularly simple.

$\rightsquigarrow$ simplifies the design and implementation of planning algorithms

- often cumbersome for the user to model tasks directly in STRIPS
- but: STRIPS is equally "powerful" to much more complex planning formalisms

$\rightsquigarrow$ automatic "compilers" exist that translate more complex formalisms (like ADL and SAS$^+$) to STRIPS

ADL, SAS$^+$ and PDDL

Basic Concepts of ADL

basic concepts of ADL:

- Like STRIPS, ADL uses propositional variables (true/false) as state variables.
- preconditions of actions and goal are arbitrary logic formulas (action applicable/goal reached in states that satisfy the formula)
- in addition to STRIPS effects, there are conditional effects: variable v is only set to true/false if a given logical formula is true in the current state

Basic Concepts of SAS$^+$

basic concepts of SAS$^+$:

- very similar to STRIPS: state variables not necessarily binary, but with given finite domain (cf. CSPs)
- states are assignments to these variables (cf. CSPs)
- preconditions and goals given as partial assignments

  example: $\{v_1 \mapsto a, v_3 \mapsto b\}$ as preconditions (or goals)
  - If $s(v_1) = a$ and $s(v_3) = b$,
    then the action is applicable in s (or goal is reached)
  - values of other variables do not matter
- effects are assignments to subset of variables

  example: effect $\{v_1 \mapsto b, v_2 \mapsto c\}$ means
  - In the successor state s', $s'(v_1) = b$ and $s'(v_2) = c$.
  - All other variables retain their values.

Basic Concept of PDDL

- PDDL is the standard language used in practice to describe planning tasks.
- descriptions in (restricted) predicate logic instead of propositional logic ($\rightsquigarrow$ even more compact)
- other features like numeric variables and derived variables (axioms) for defining complex logical conditions (formulas that are automatically evaluated in every state and can, e.g., be used in preconditions)
- There exist defined PDDL fragments for STRIPS and ADL; many planners only support the STRIPS fragment.

example: blocks world in PDDL

Summary

Summary

planning formalisms:

- STRIPS: particularly simple, easy to handle for algorithms
  - binary state variables
  - preconditions, add and delete effects, goals: sets of variables
- ADL: extension of STRIPS
  - logic formulas for complex preconditions and goals
  - conditional effects
- SAS$^+$: extension of STRIPS
  - state variables with arbitrary finite domains
- PDDL: input language used in practice
  - based on predicate logic (more compact than propositional logic)
  - only partly supported by most algorithms (e.g., STRIPS or ADL fragment)

Foundations of Artificial Intelligence

F3. Automated Planning: Delete Relaxation

Malte Helmert

University of Basel

May 5, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

How to Design Heuristics?

A Simple Planning Heuristic

The STRIPS planner (Fikes & Nilsson, 1971) uses the number of goals not yet satisfied in a STRIPS planning task as heuristic:

$$h(s) = |G \setminus s|.$$

intuition: fewer unsatisfied goals $\rightsquigarrow$ closer to goal state

$\rightsquigarrow$ STRIPS heuristic

Problems of STRIPS Heuristic

drawback of STRIPS heuristic?

- rather uninformed:
  For state s, if there is no applicable action a in s such that applying a in s satisfies strictly more (or fewer) goals, then all successor states have the same heuristic value as s.
- ignores almost the whole task structure:
  The heuristic values do not depend on the actions.

$\rightsquigarrow$ we need better methods to design heuristics

Planning Heuristics

We consider two basic ideas for general heuristics:

- delete relaxation $\rightsquigarrow$ this and next chapter
- abstraction $\rightsquigarrow$ Chapters F5–F6

Delete Relaxation: Basic Idea

Estimate solution costs by considering a simplified planning task, where all negative action effects are ignored.

Delete Relaxation

Relaxed Planning Tasks: Idea

In STRIPS planning tasks,
good and bad effects are easy to distinguish:

- Add effects are always useful.
- Delete effects are always harmful.

Why?

idea for designing heuristics: ignore all delete effects

Relaxed Planning Tasks

Definition (relaxation of actions)

The relaxation a^+ of STRIPS action a is the action with

- $pre(a^+) = pre(a)$,
- $add(a^+) = add(a)$,
- $cost(a^+) = cost(a)$, and
- $del(a^+) = \emptyset$.

German: Relaxierung von Aktionen

Definition (relaxation of planning tasks)

The relaxation Π^+ of a STRIPS planning task $\Pi = \langle V, I, G, A \rangle$ is the task $\Pi^+ = \langle V, I, G, \{a^+ \mid a \in A\} \rangle$.

German: Relaxierung von Planungsaufgaben

Relaxed Planning Tasks: Terminology

- STRIPS planning tasks without delete effects
  are called relaxed planning tasks
  or delete-free planning tasks.
- Plans for relaxed planning tasks are called relaxed plans.
- If Π is a STRIPS planning task and π^+ is a plan for Π^+,
  then π^+ is called relaxed plan for Π.

Optimal Relaxation Heuristic

Definition (optimal relaxation heuristic h^+)

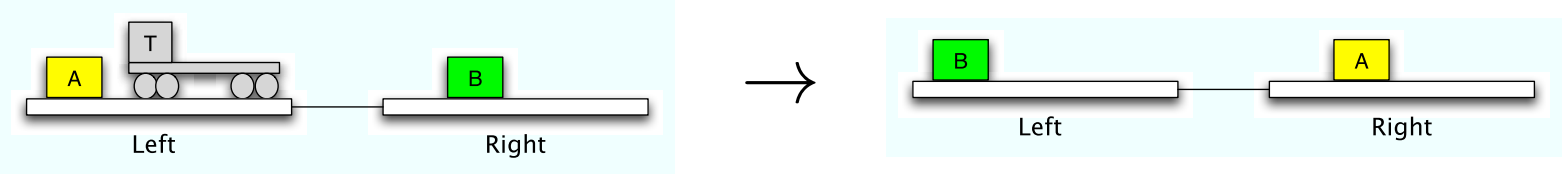
Let Π be a STRIPS planning task with the relaxation $\Pi^+ = \langle V, I, G, A^+ \rangle$.

The optimal relaxation heuristic h^+ for Π maps each state s to the cost of an optimal plan for the planning task $\langle V, s, G, A^+ \rangle$.

In other words, the heuristic value for s is the optimal solution cost in the relaxation of Π with s as the initial state.

Examples

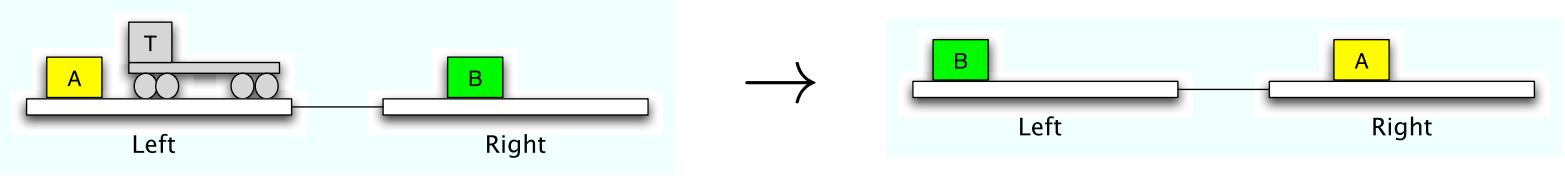
Example: Logistics



Example (Logistics Task)

- variables: $V = \{at_{AL}, at_{AR}, at_{BL}, at_{BR}, at_{TL}, at_{TR}, in_{AT}, in_{BT}\}$
- initial state: $I = \{at_{AL}, at_{BR}, at_{TL}\}$
- goals: $G = \{at_{AR}, at_{BL}\}$
- actions: $\{move_{LR}, move_{RL}, load_{AL}, load_{AR}, load_{BL}, load_{BR}, unload_{AL}, unload_{AR}, unload_{BL}, unload_{BR}\}$
- ...

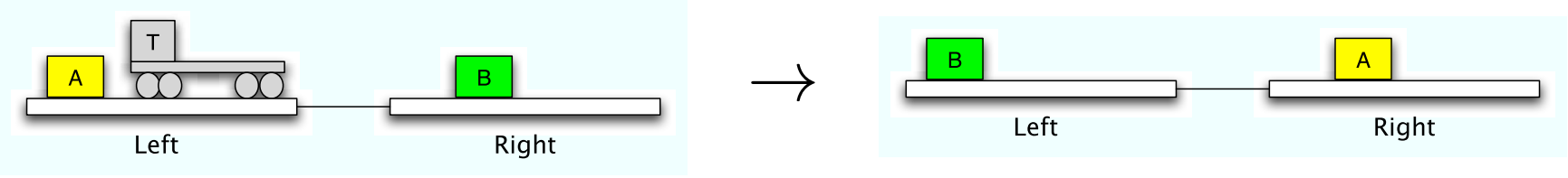
Example: Logistics



Example (Logistics Task)

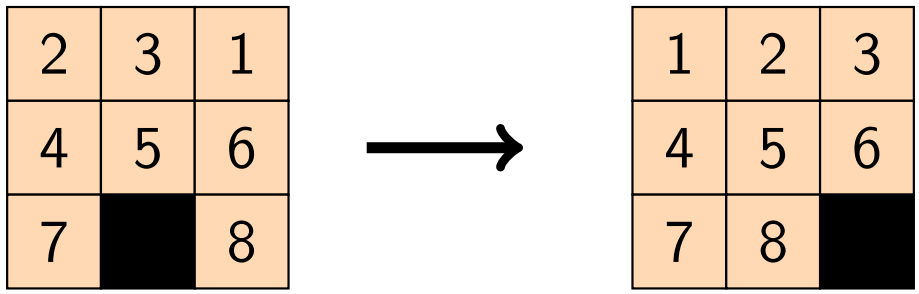
- $pre(move_{\mathrm{LR}}) = \{at_{\mathrm{TL}}\}$, $add(move_{\mathrm{LR}}) = \{at_{\mathrm{TR}}\}$, $del(move_{\mathrm{LR}}) = \{at_{\mathrm{TL}}\}$, $cost(move_{\mathrm{LR}}) = 1$
- $pre(load_{\mathrm{AL}}) = \{at_{\mathrm{TL}}, at_{\mathrm{AL}}\}$, $add(load_{\mathrm{AL}}) = \{in_{\mathrm{AT}}\}$, $del(load_{\mathrm{AL}}) = \{at_{\mathrm{AL}}\}$, $cost(load_{\mathrm{AL}}) = 1$
- $pre(unload_{\mathrm{AL}}) = \{at_{\mathrm{TL}}, in_{\mathrm{AT}}\}$, $add(unload_{\mathrm{AL}}) = \{at_{\mathrm{AL}}\}$, $del(unload_{\mathrm{AL}}) = \{in_{\mathrm{AT}}\}$, $cost(unload_{\mathrm{AL}}) = 1$
- ...

Example: Logistics



- optimal plan:
  1. $load_{AL}$
  2. $move_{LR}$
  3. $unload_{AR}$
  4. $load_{BR}$
  5. $move_{RL}$
  6. $unload_{BL}$
- optimal relaxed plan: ?
- $h^*(I) = 6$, $h^+(I) = ?$

Example: 8-Puzzle



- actual goal distance: $h^*(s) = 17$
- Manhattan distance: $h^{\mathrm{MD}}(s) = 5$
- optimal delete relaxation: $h^+(s) = 7$

relationship (no proof):
h^+ dominates the Manhattan distance in the sliding tile puzzle
(i.e., $h^{\mathrm{MD}}(s) \leq h^+(s) \leq h^*(s)$ for all states s)

Relaxed Solutions: Suboptimal or Optimal?

- For general STRIPS planning tasks, h^+ is an admissible and consistent heuristic (no proof).
- Can h^+ be computed efficiently?
  - It is easy to solve delete-free planning tasks suboptimally. (How?)
  - optimal solution (and hence the computation of h^+) is NP-hard (reduction from Set Cover)
- In practice, heuristics approximate h^+ from below or above.

Summary

Summary

delete relaxation:

- ignore negative effects (delete effects) of actions
- use solution costs of relaxed planning task
  as heuristic for solution costs of the original planning task
- computation of optimal relaxed solution costs h^+ is NP-hard,
  hence usually approximated from below or above

Foundations of Artificial Intelligence

F4. Automated Planning: Delete Relaxation Heuristics

Malte Helmert

University of Basel

May 5, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

Relaxed Planning Graphs

Relaxed Planning Graphs

- relaxed planning graphs: represent which variables in Π^+ can be reached and how
- graphs with variable layers V^i and action layers A^i
  - variable layer V^0 contains the variable vertex v^0 for all $v \in I$
  - action layer A^{i+1} contains the action vertex a^{i+1} for action a if V^i contains the vertex v^i for all $v \in pre(a)$
  - variable layer V^{i+1} contains the variable vertex v^{i+1} if previous variable layer contains v^i, or previous action layer contains a^{i+1} with $v \in add(a)$

German: relaxierter Planungsgraph, Variablenknoten, Aktionsknoten

Relaxed Planning Graphs (Continued)

- a goal vertex g if $v^n \in V^n$ for all $v \in G$, where n is last layer
- graph can be constructed for arbitrary many layers but stabilizes after a bounded number of layers $\rightsquigarrow V^{i+1} = V^i$ and $A^{i+1} = A^i$ (Why?)
- directed edges:
  - from v^i to a^{i+1} if $v \in pre(a)$ (precondition edges)
  - from a^i to v^i if $v \in add(a)$ (effect edges)
  - from v^i to v^{i+1} (no-op edges)
  - from v^n to g if $v \in G$ (goal edges)

German: Zielknoten, Vorbedingungskanten, Effektkanten, Zielkanten, No-Op-Kanten

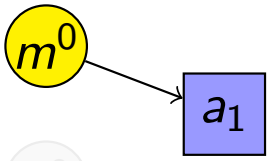
Illustrative Example

We write actions a with $pre(a) = \{p_1, \ldots, p_k\}$, $add(a) = \{q_1, \ldots, q_l\}$, $del(a) = \emptyset$ and $cost(a) = c$ as $p_1, \ldots, p_k \xrightarrow{c} q_1, \ldots, q_l$

$$V = \{m, n, o, p, q, r, s, t\}$$
$$I = \{m\}$$
$$G = \{o, p, q, r, s\}$$
$$A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$$
$$a_1 = m \xrightarrow{3} n, o$$
$$a_2 = m, o \xrightarrow{1} p$$
$$a_3 = n, o \xrightarrow{1} q$$
$$a_4 = n \xrightarrow{1} r$$
$$a_5 = p \xrightarrow{1} q, r$$
$$a_6 = p \xrightarrow{1} s$$

Illustrative Example: Relaxed Planning Graph

Illustrative Example: Relaxed Planning Graph



n^0

o^0

p^0

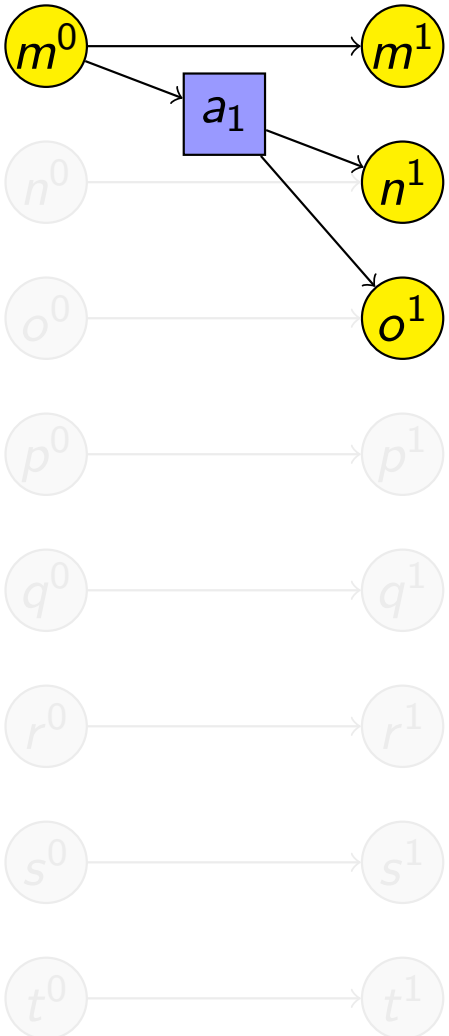
q^0

r^0

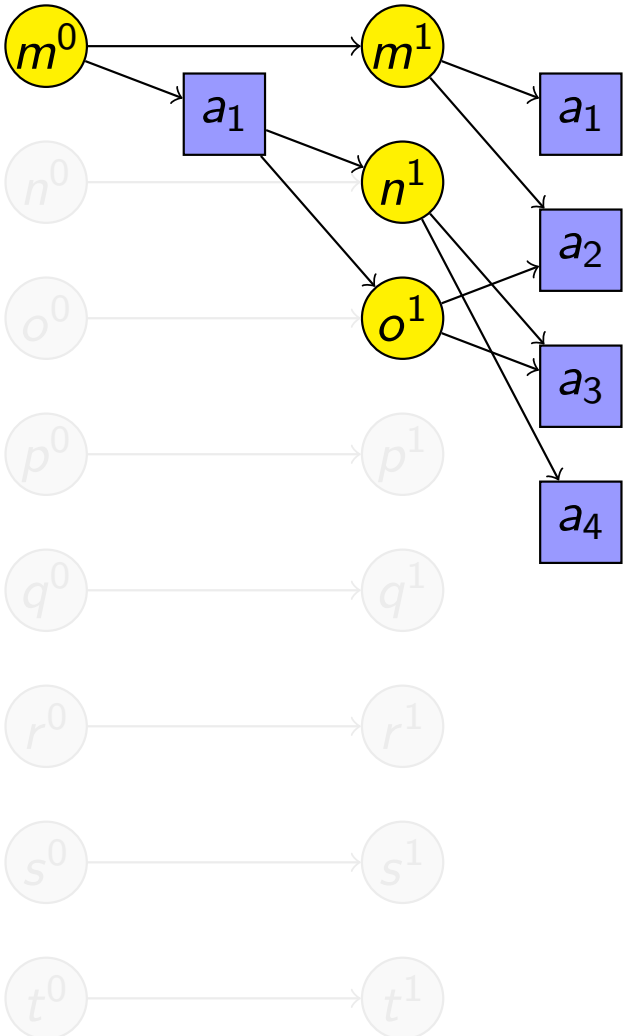
s^0

t^0

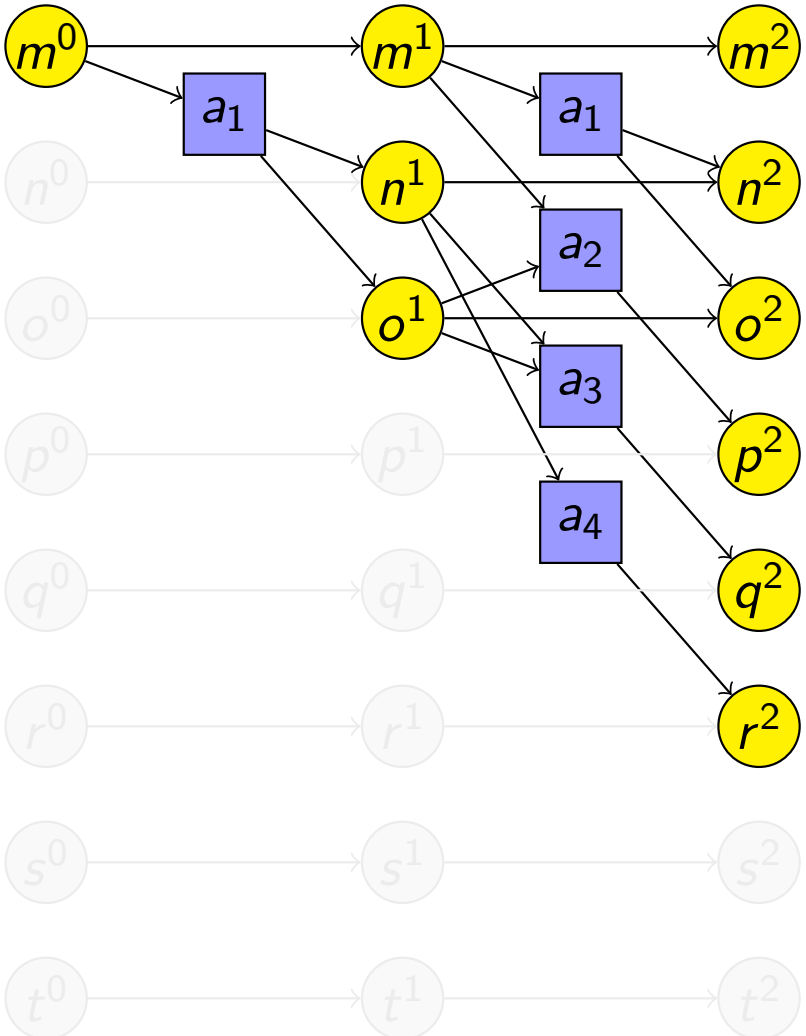
Illustrative Example: Relaxed Planning Graph



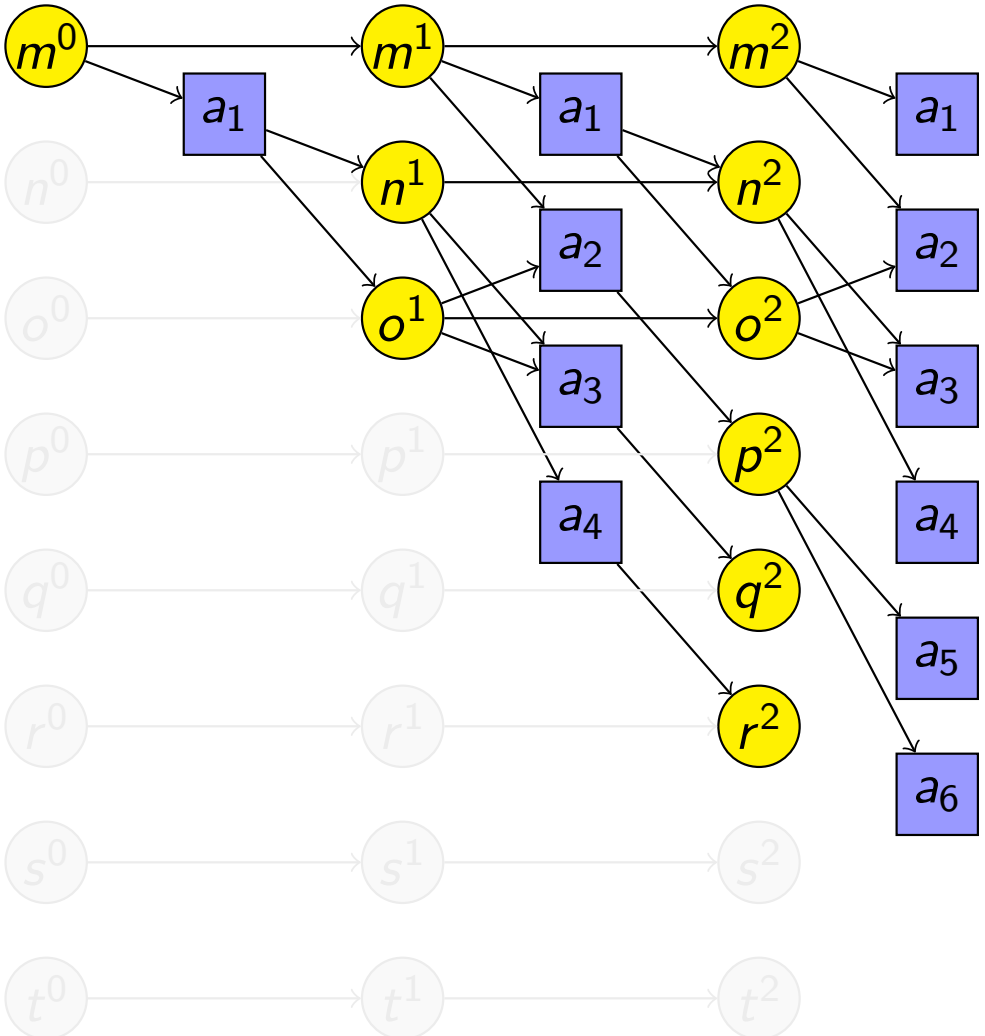
Illustrative Example: Relaxed Planning Graph



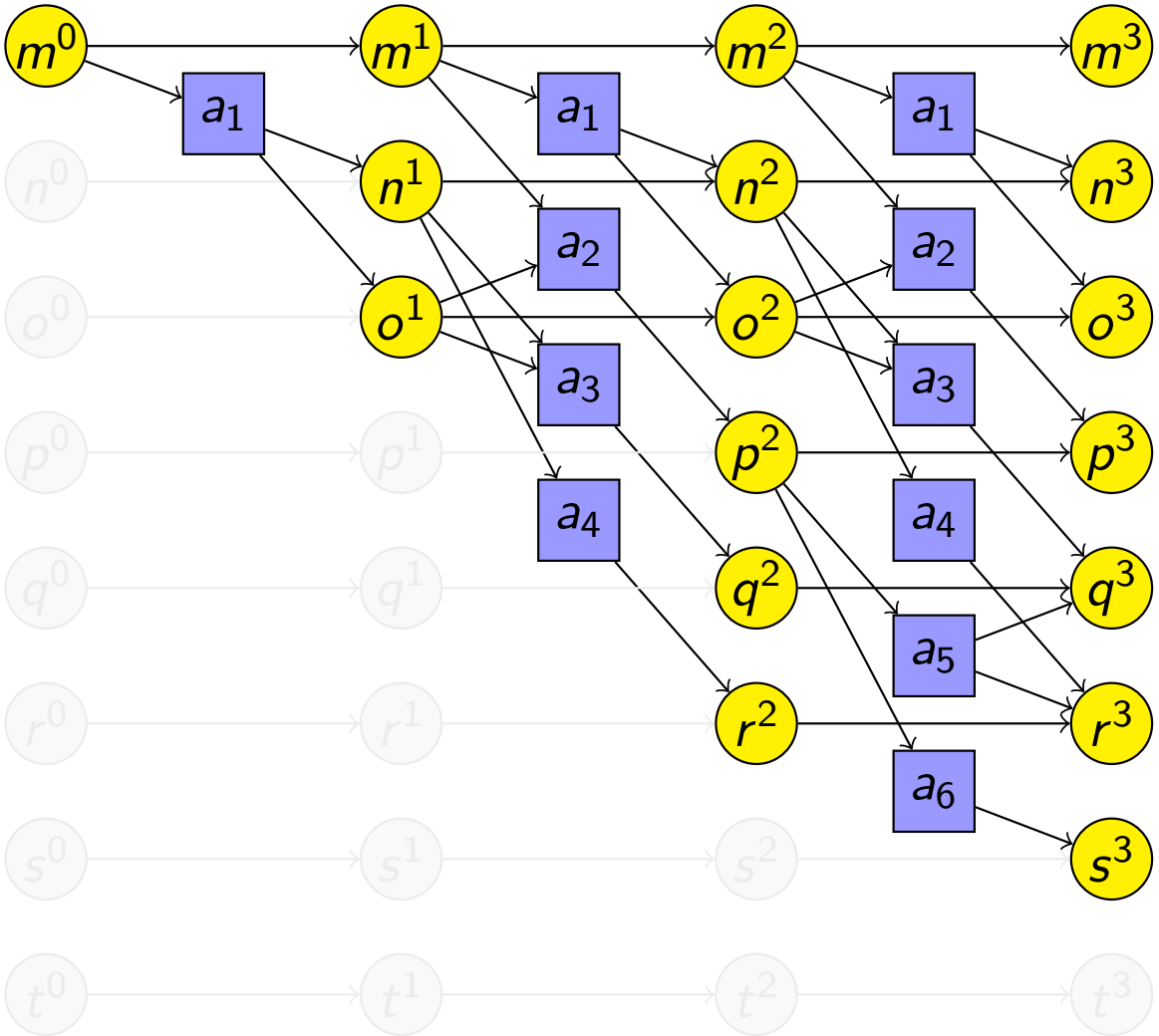
Illustrative Example: Relaxed Planning Graph



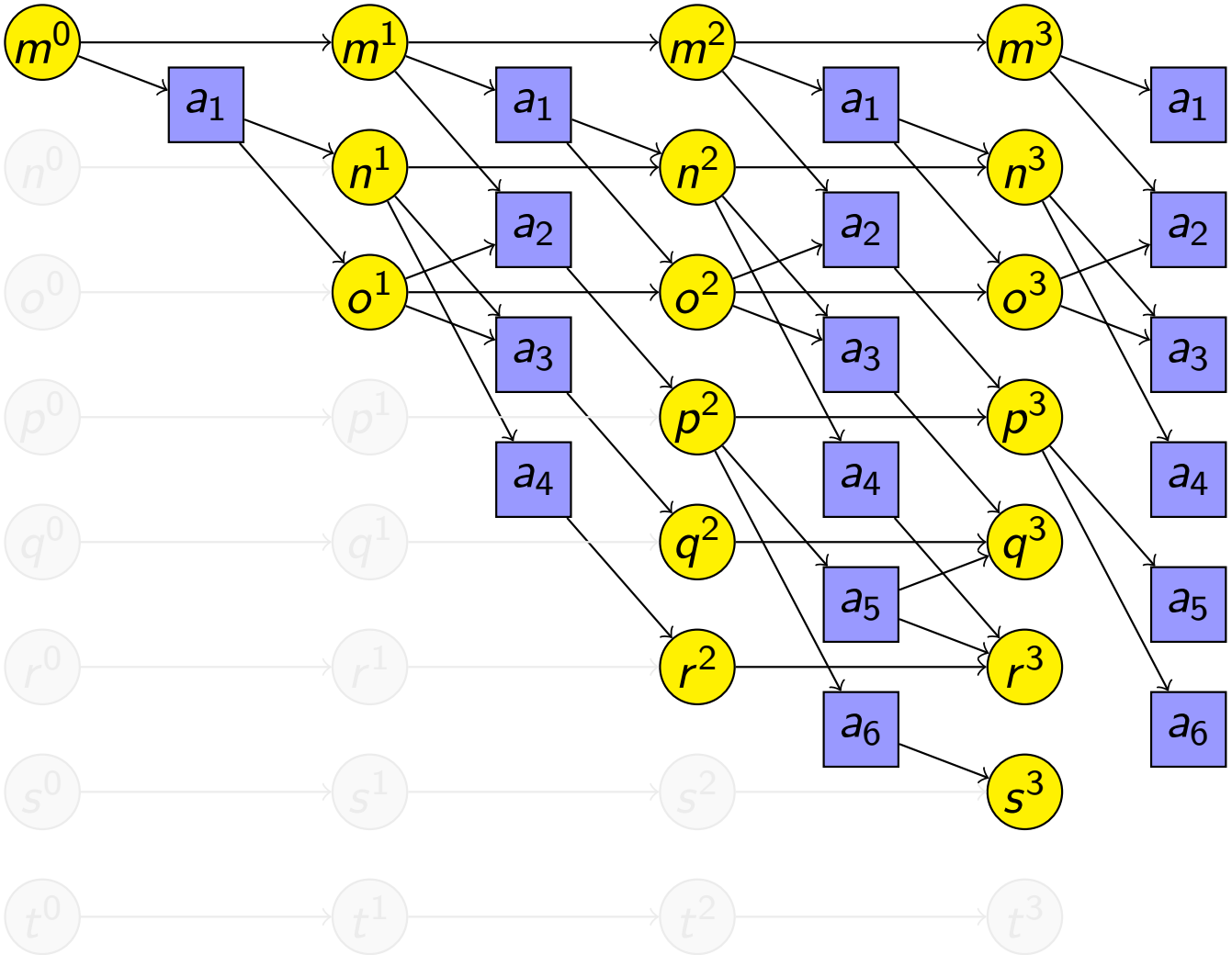
Illustrative Example: Relaxed Planning Graph



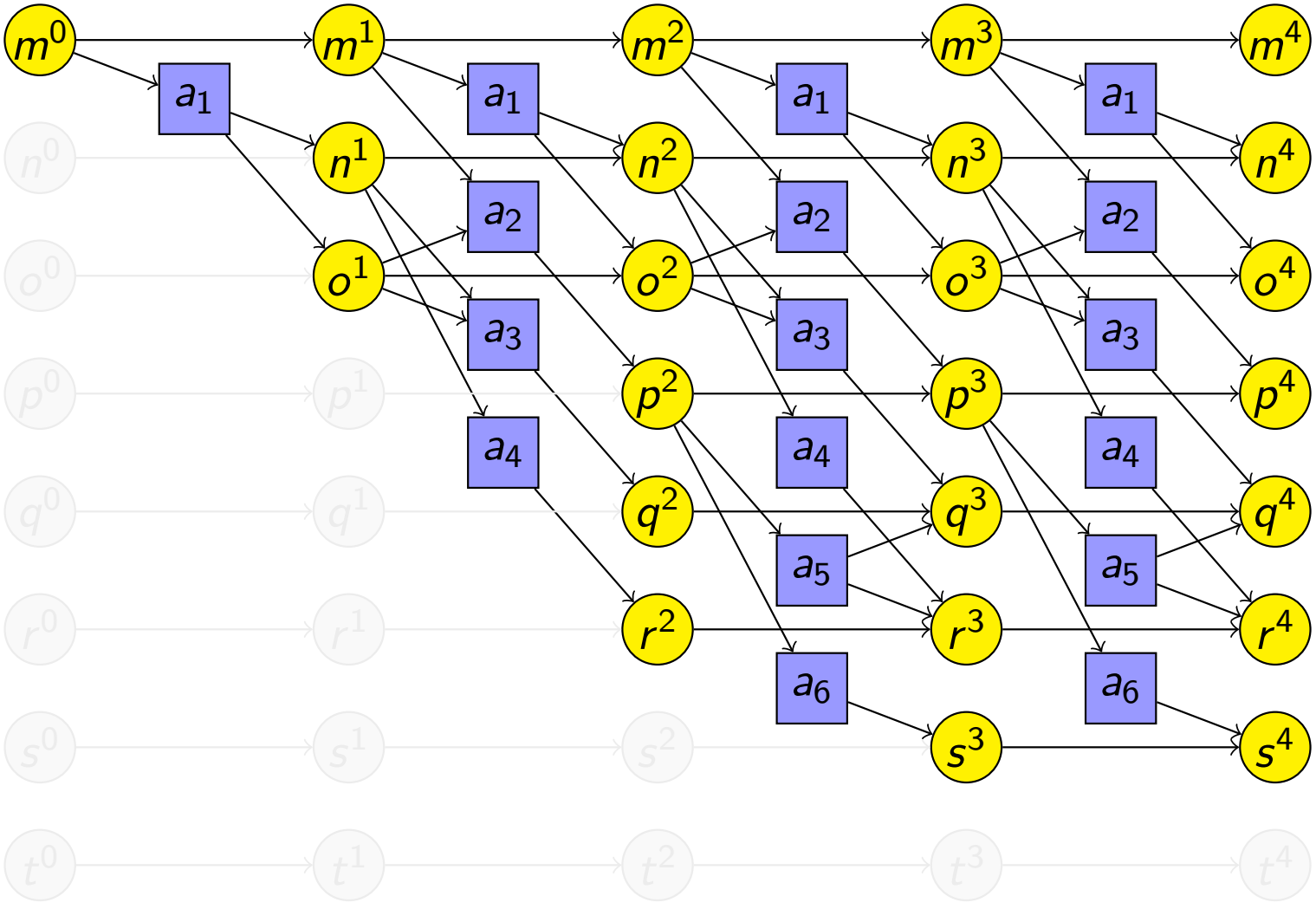
Illustrative Example: Relaxed Planning Graph



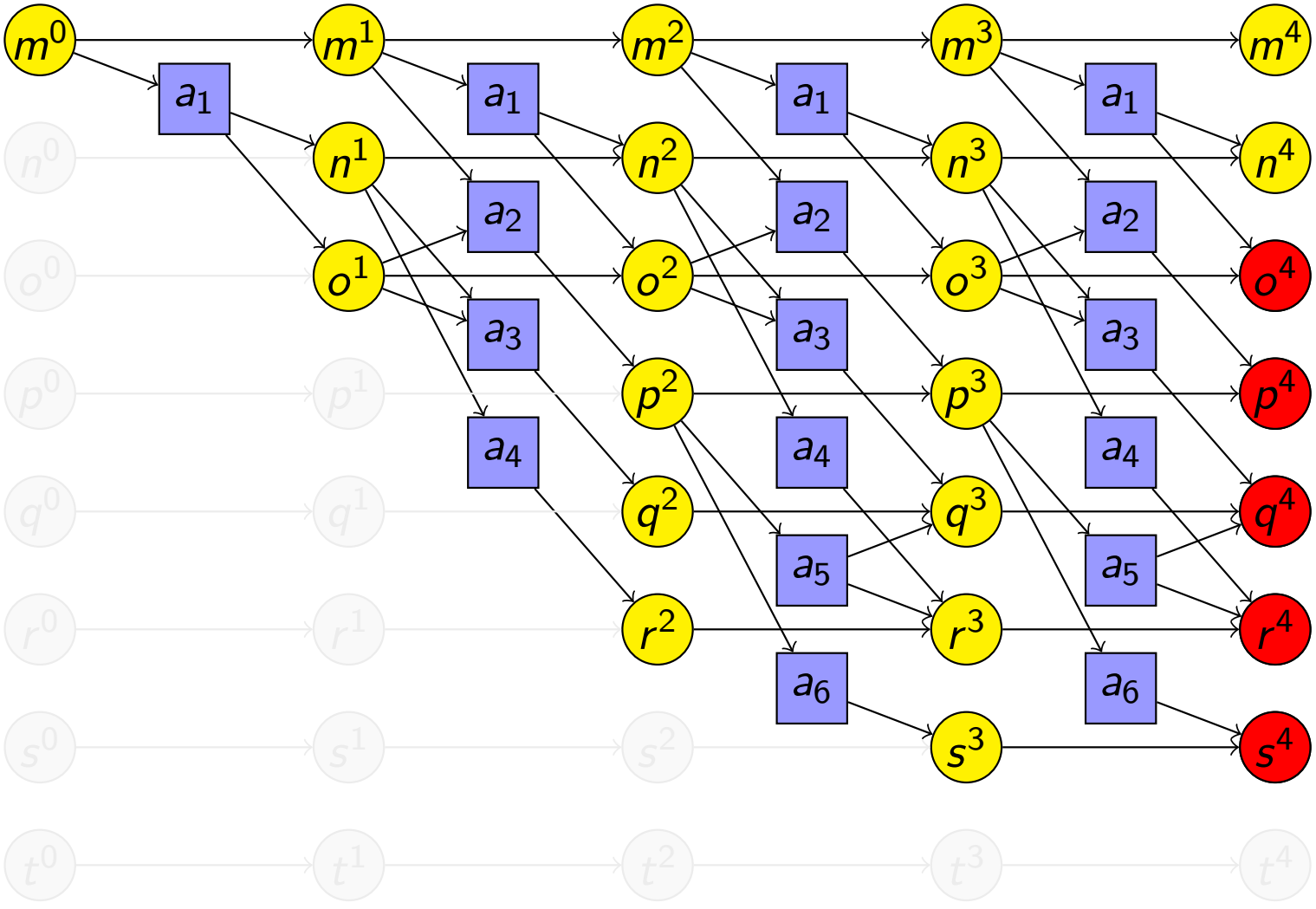
Illustrative Example: Relaxed Planning Graph



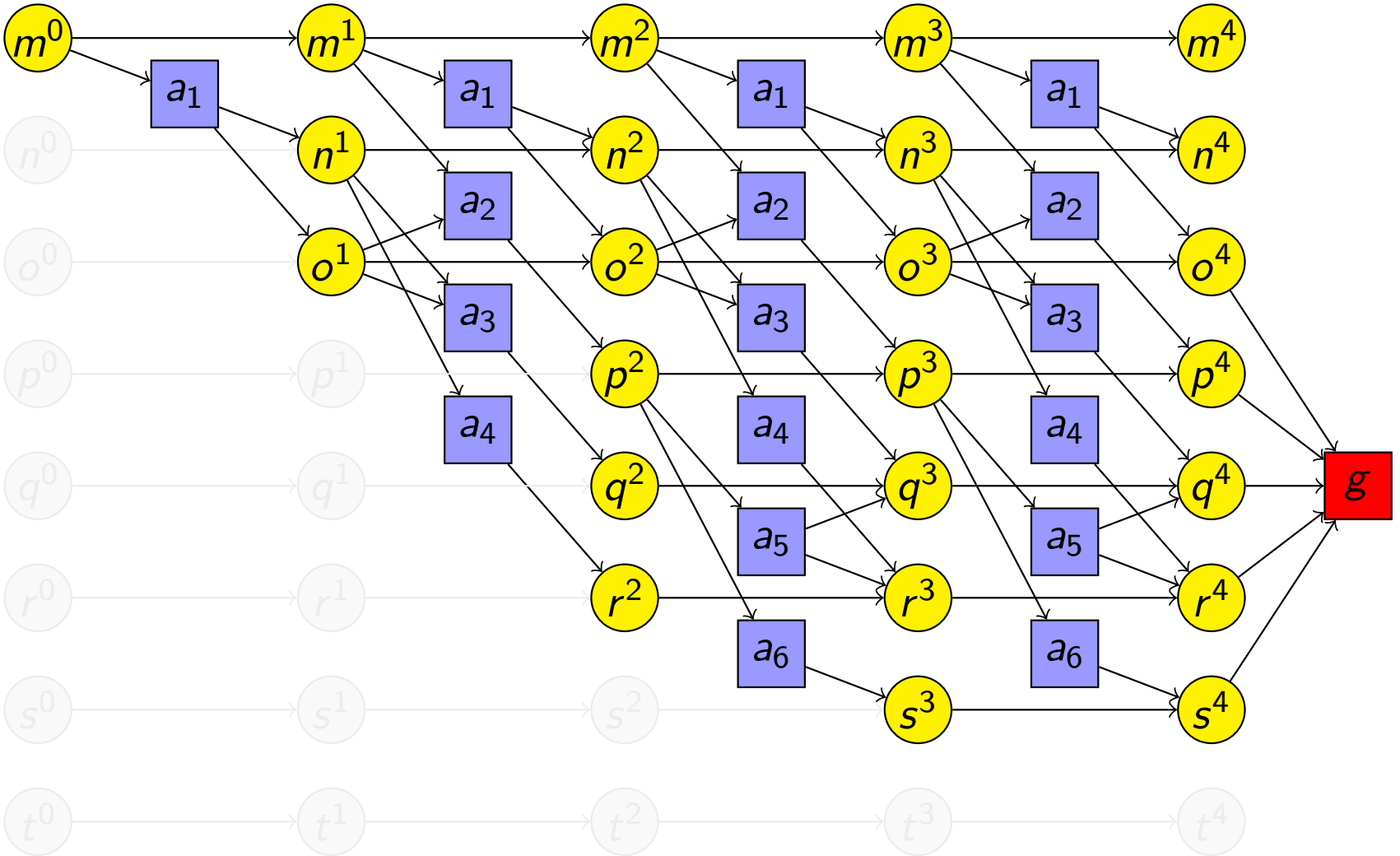
Illustrative Example: Relaxed Planning Graph



Illustrative Example: Relaxed Planning Graph



Illustrative Example: Relaxed Planning Graph



Generic Relaxed Planning Graph Heuristic

Heuristic Values from Relaxed Planning Graph

function *generic-rpg-heuristic*$(\langle V, I, G, A\rangle, s)$:
    $\Pi^+ := \langle V, s, G, A^+\rangle$
    for $k \in \{0, 1, 2, \dots\}$:
        $rpg := RPG_k(\Pi^+)$ [relaxed planning graph to layer k]
        if *rpg* contains a goal node:
            Annotate nodes of *rpg*.
            if termination criterion is true:
                return heuristic value from annotations
        else if graph has stabilized:
            return ∞

- $\rightsquigarrow$ general template for RPG heuristics
- $\rightsquigarrow$ to obtain concrete heuristic: instantiate highlighted elements

Concrete Examples for Generic RPG Heuristic

Many planning heuristics fit this general template.

In this course:

- maximum heuristic $h^{\max}$ (Bonet & Geffner, 1999)
- additive heuristic h^{add} (Bonet, Loerincs & Geffner, 1997)
- Keyder & Geffner's (2008) variant of the FF heuristic h^{FF} (Hoffmann & Nebel, 2001)

German: Maximum-Heuristik, additive Heuristik, FF-Heuristik

remark:

- The most efficient implementations of these heuristics do not use explicit planning graphs, but rather alternative (equivalent) definitions.

Maximum and Additive Heuristics

Maximum and Additive Heuristics

- $h^{\max}$ and h^{add} are the simplest RPG heuristics.
- Vertex annotations are numerical values.
- The vertex values estimate the costs
  - to make a given variable true
  - to reach and apply a given action
  - to reach the goal

Maximum and Additive Heuristics: Filled-in Template

$h^{\max}$ and h^{add}

computation of annotations:

- costs of variable vertices:
  0 in layer 0;
  otherwise minimum of the costs of predecessor vertices
- costs of action and goal vertices:
  maximum ($h^{\max}$) or sum (h^{add}) of predecessor vertex costs;
  for action vertices a^i, also add $cost(a)$

termination criterion:

- stability: terminate if $V^i = V^{i-1}$ and costs of all vertices in V^i equal corresponding vertex costs in V^{i-1}

heuristic value:

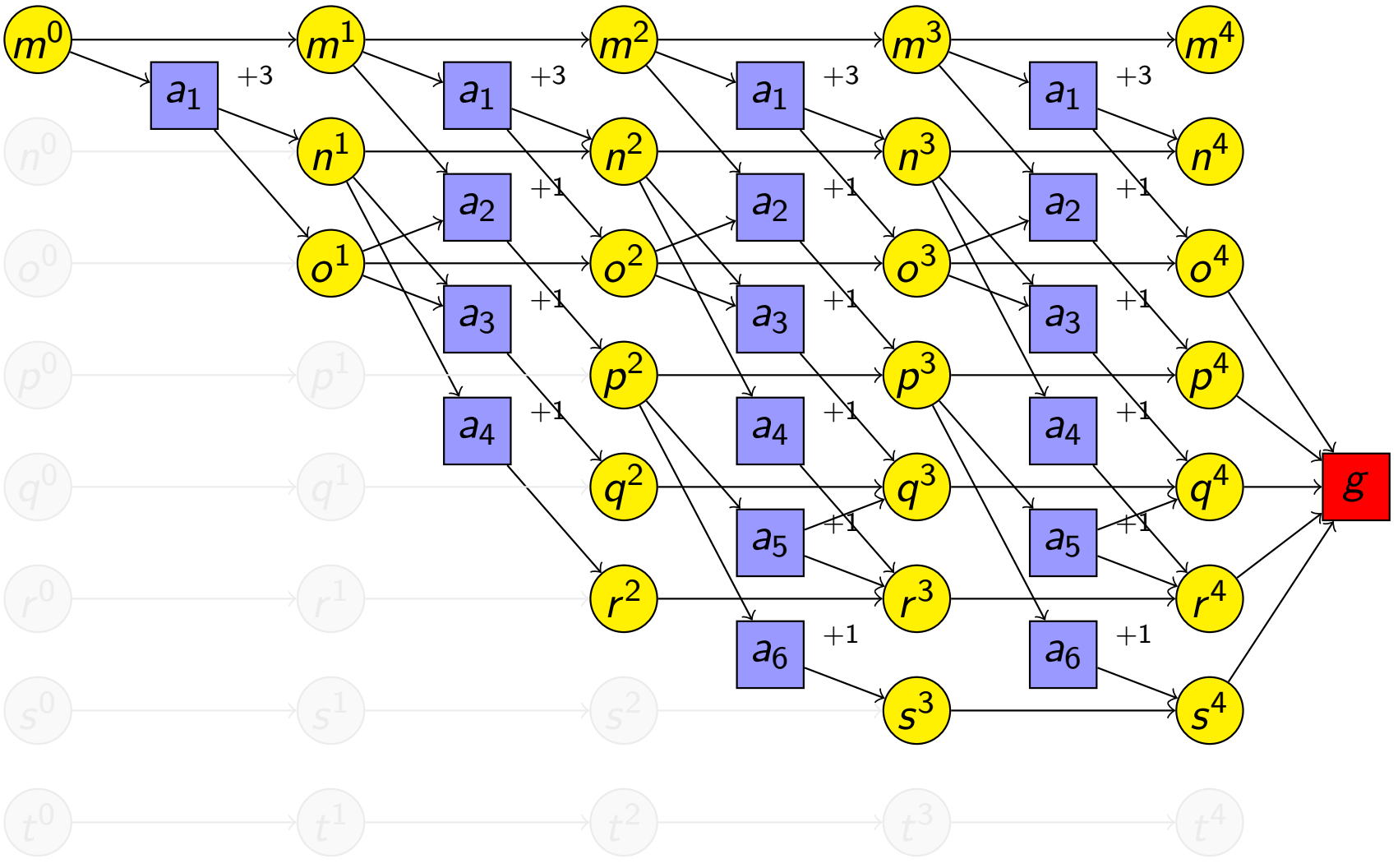
- value of goal vertex

Maximum and Additive Heuristics: Intuition

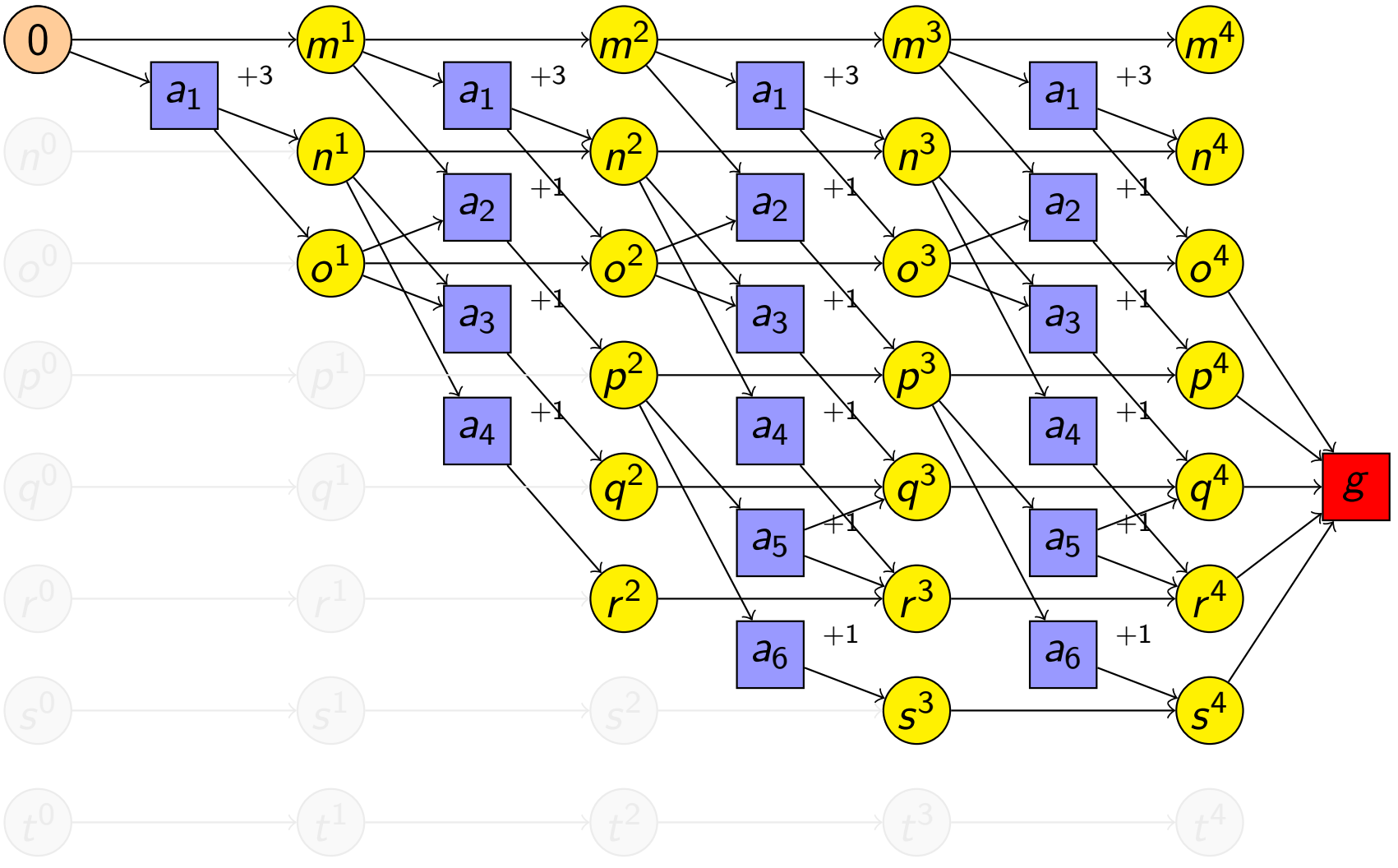
intuition:

- variable vertices:
  - choose cheapest way of reaching the variable
- action/goal vertices:
  - $h^{\max}$ is optimistic: assumption:
    when reaching the most expensive precondition variable,
    we can reach the other precondition variables in parallel
    (hence maximization of costs)
  - h^{add} is pessimistic: assumption:
    all precondition variables must be reached completely
    independently of each other (hence summation of costs)

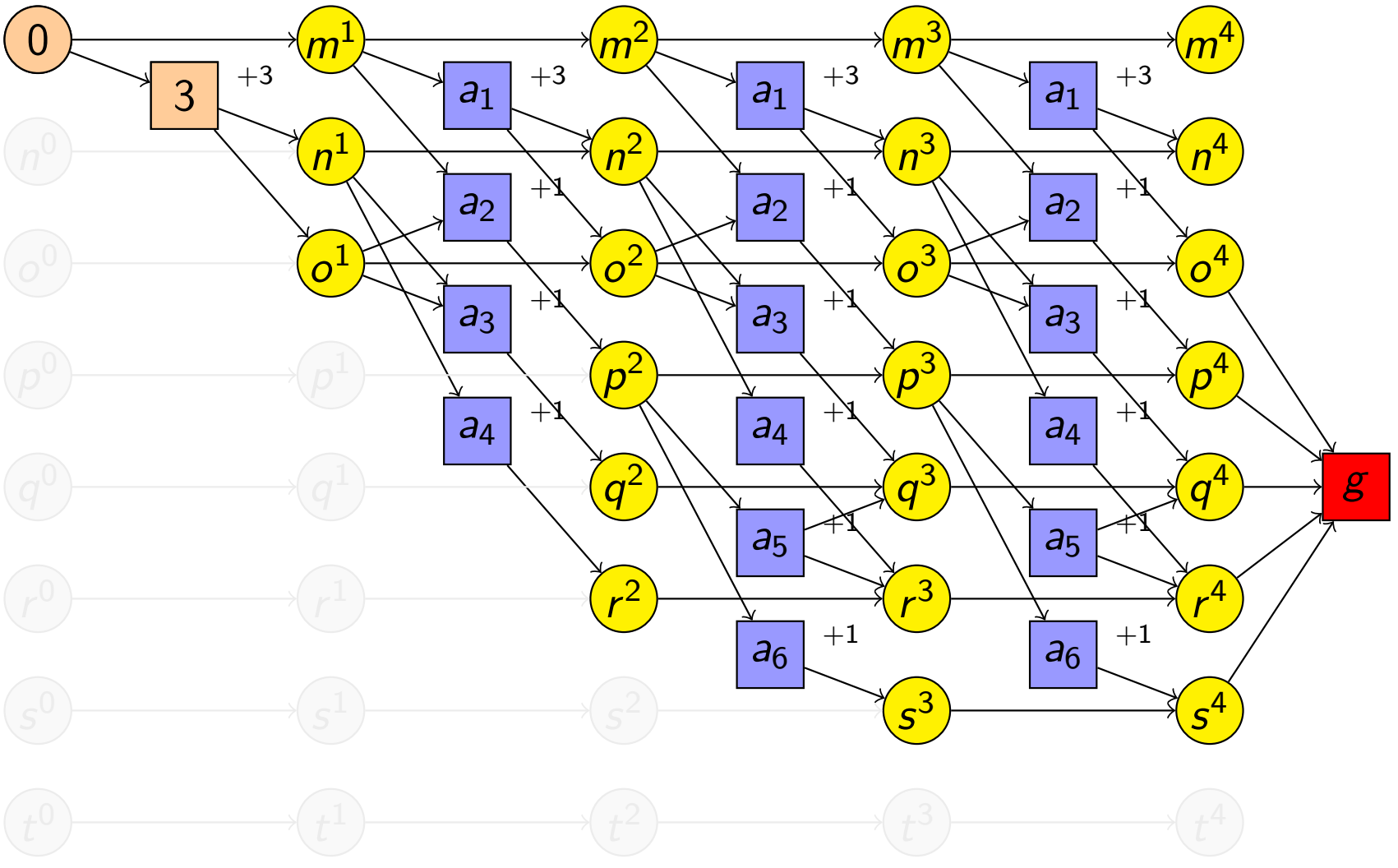
Illustrative Example: $h^{\max}$



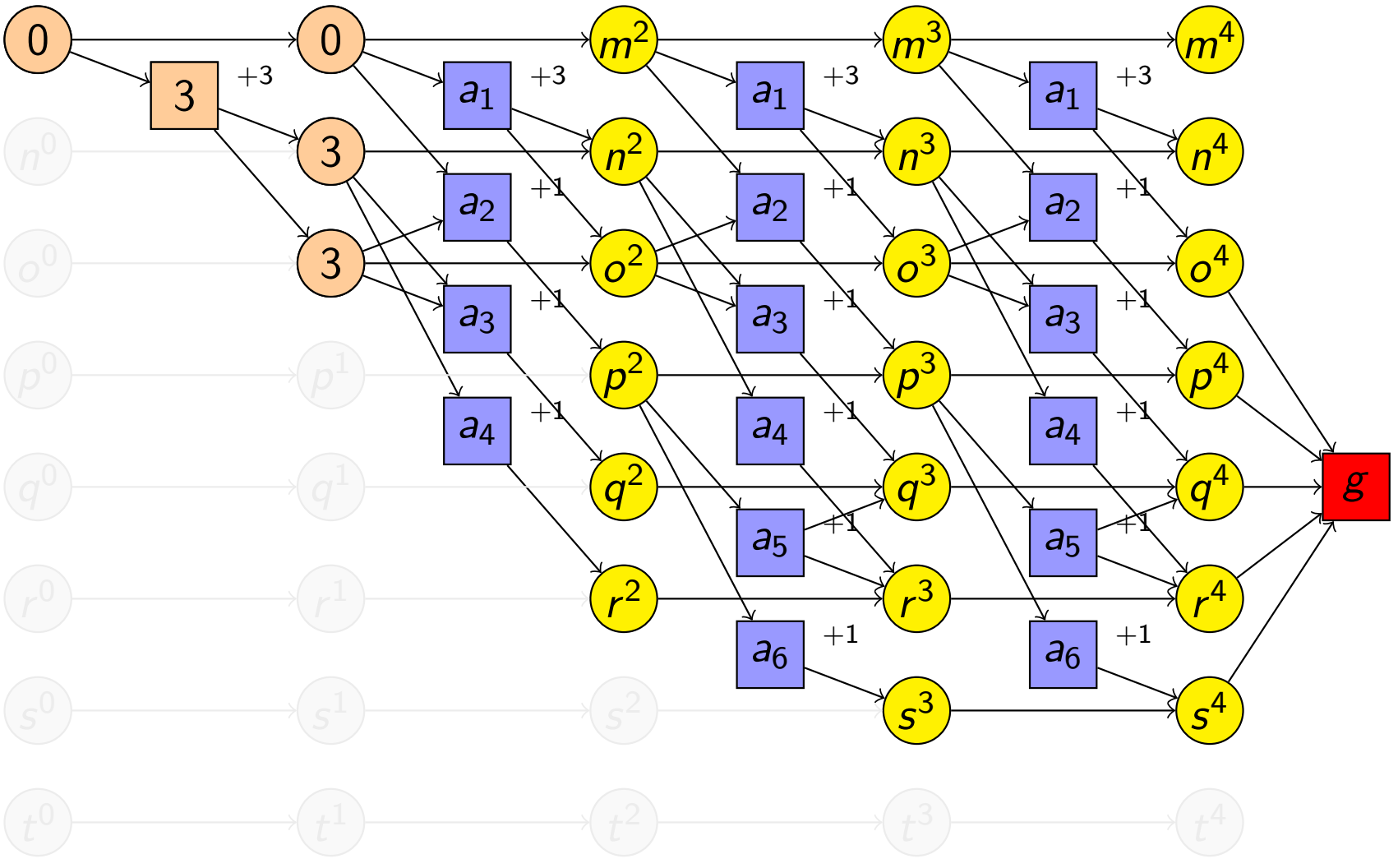
Illustrative Example: $h^{\max}$



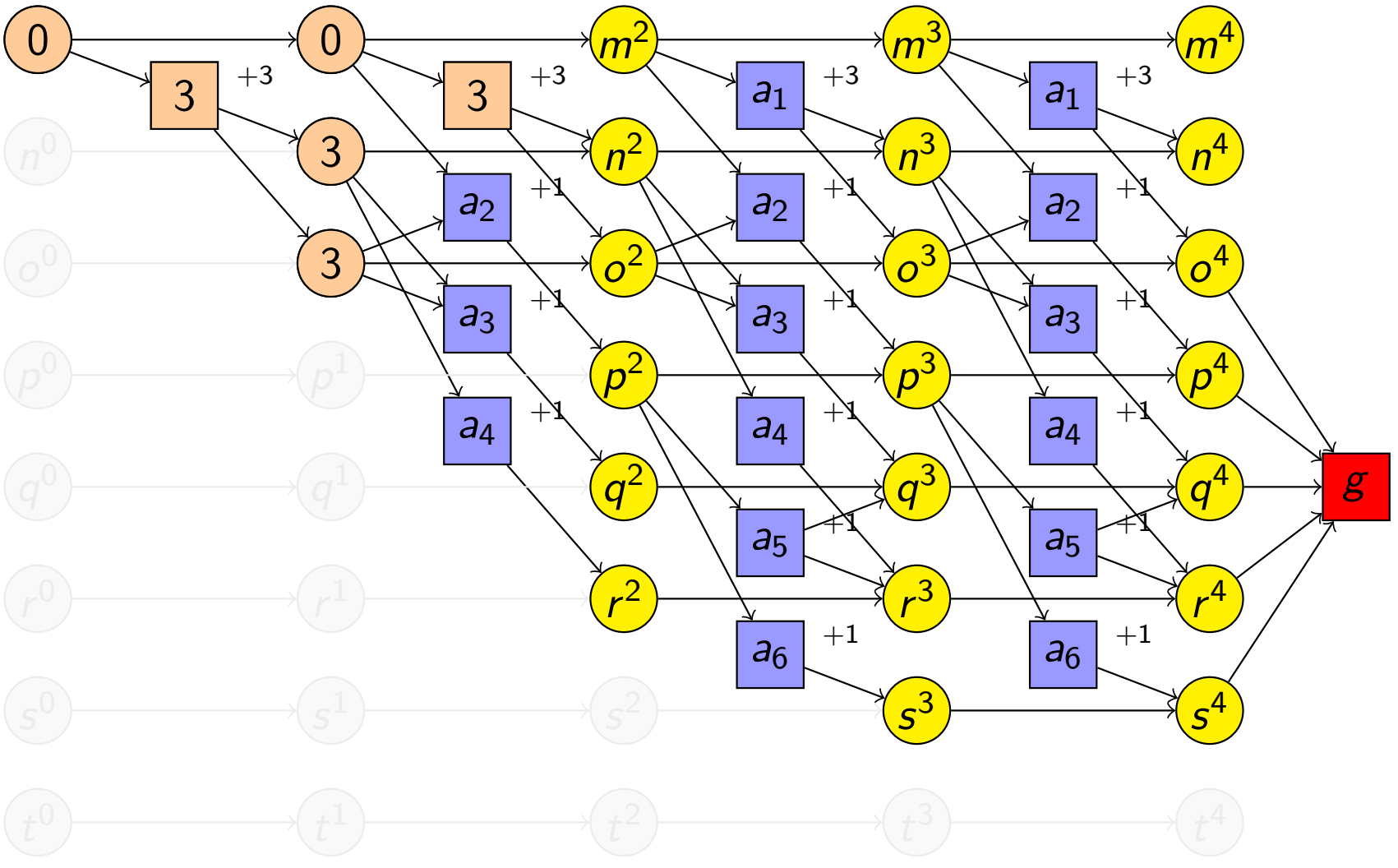
Illustrative Example: $h^{\max}$



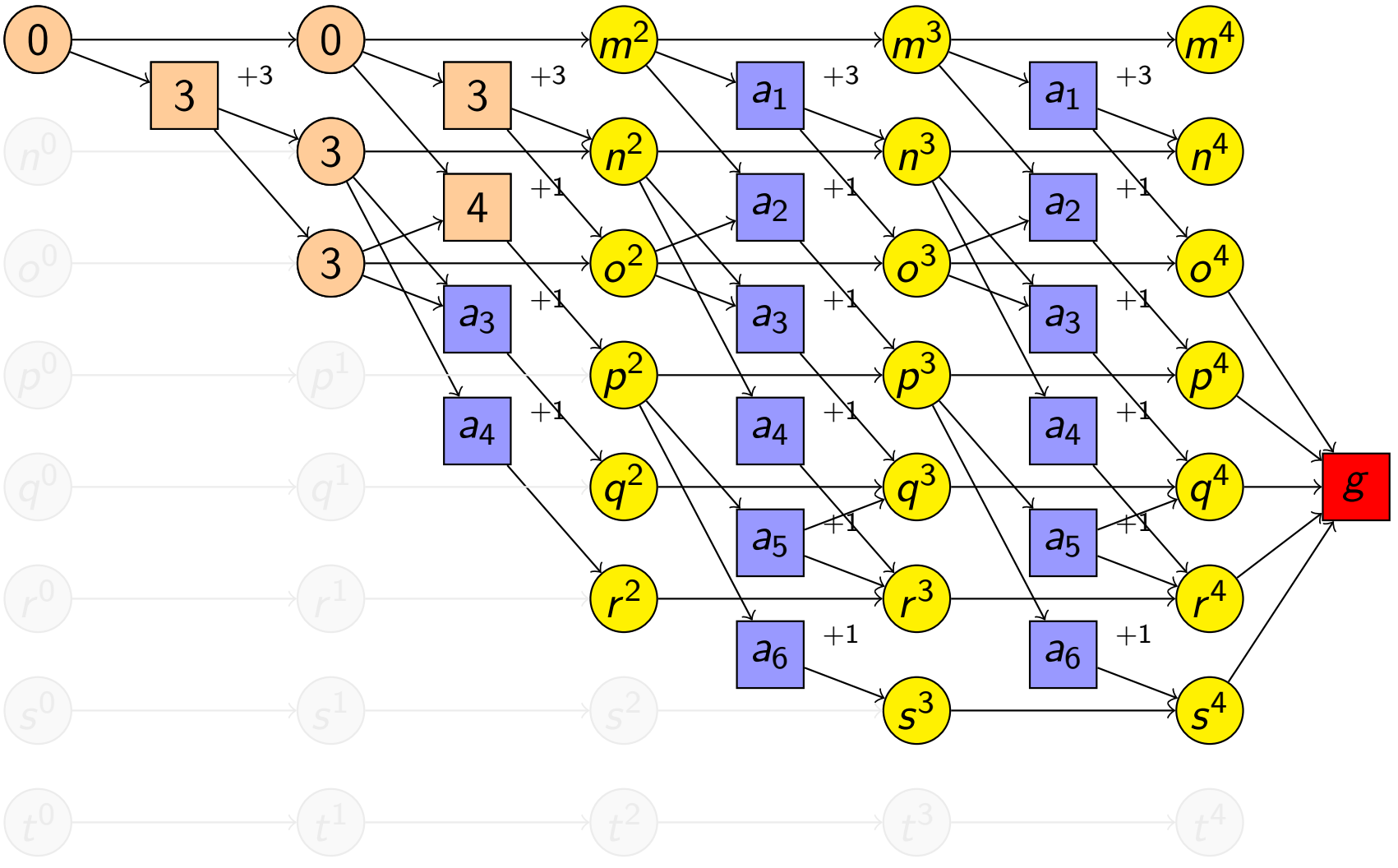
Illustrative Example: $h^{\max}$



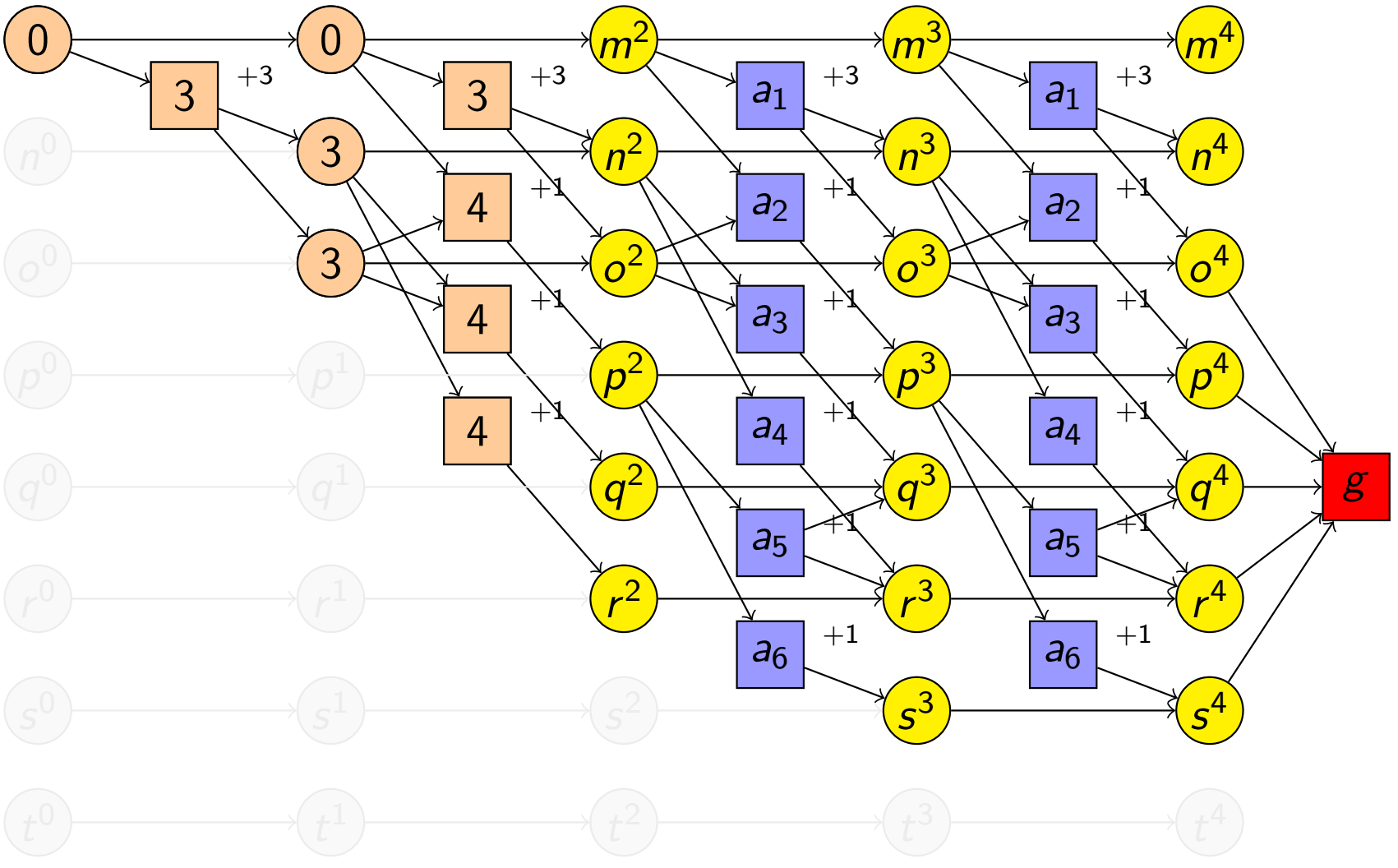
Illustrative Example: $h^{\max}$



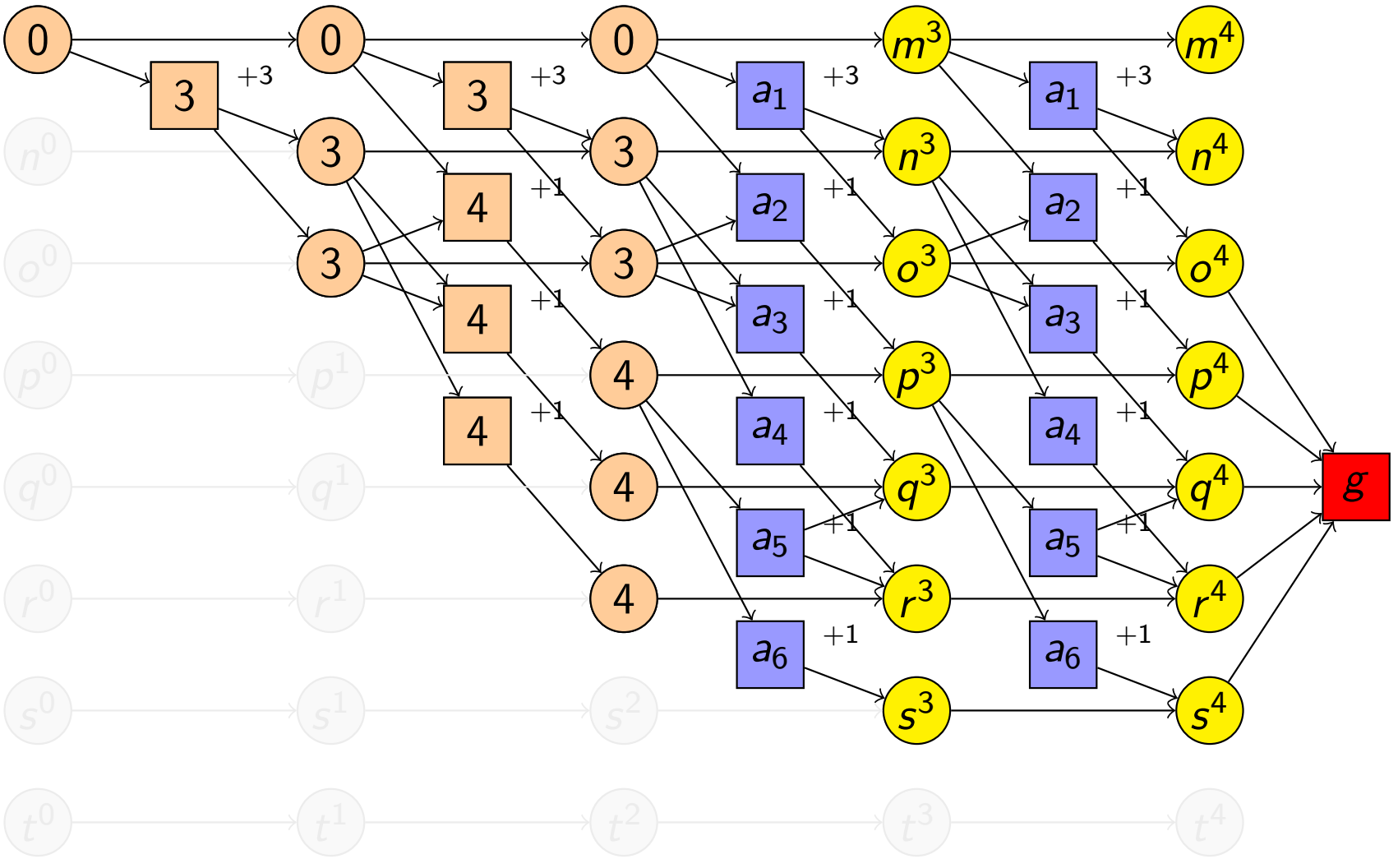
Illustrative Example: $h^{\max}$



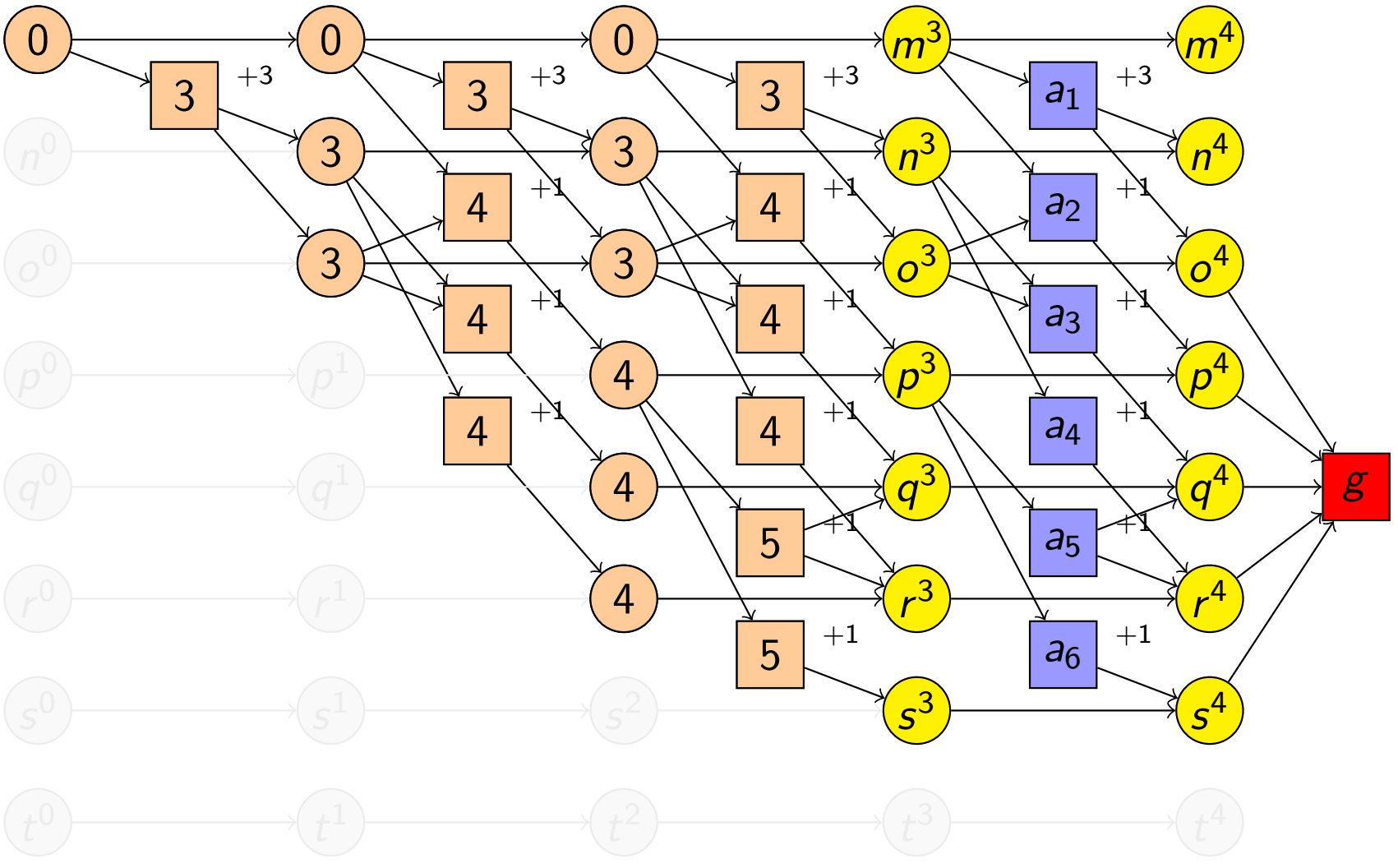
Illustrative Example: $h^{\max}$



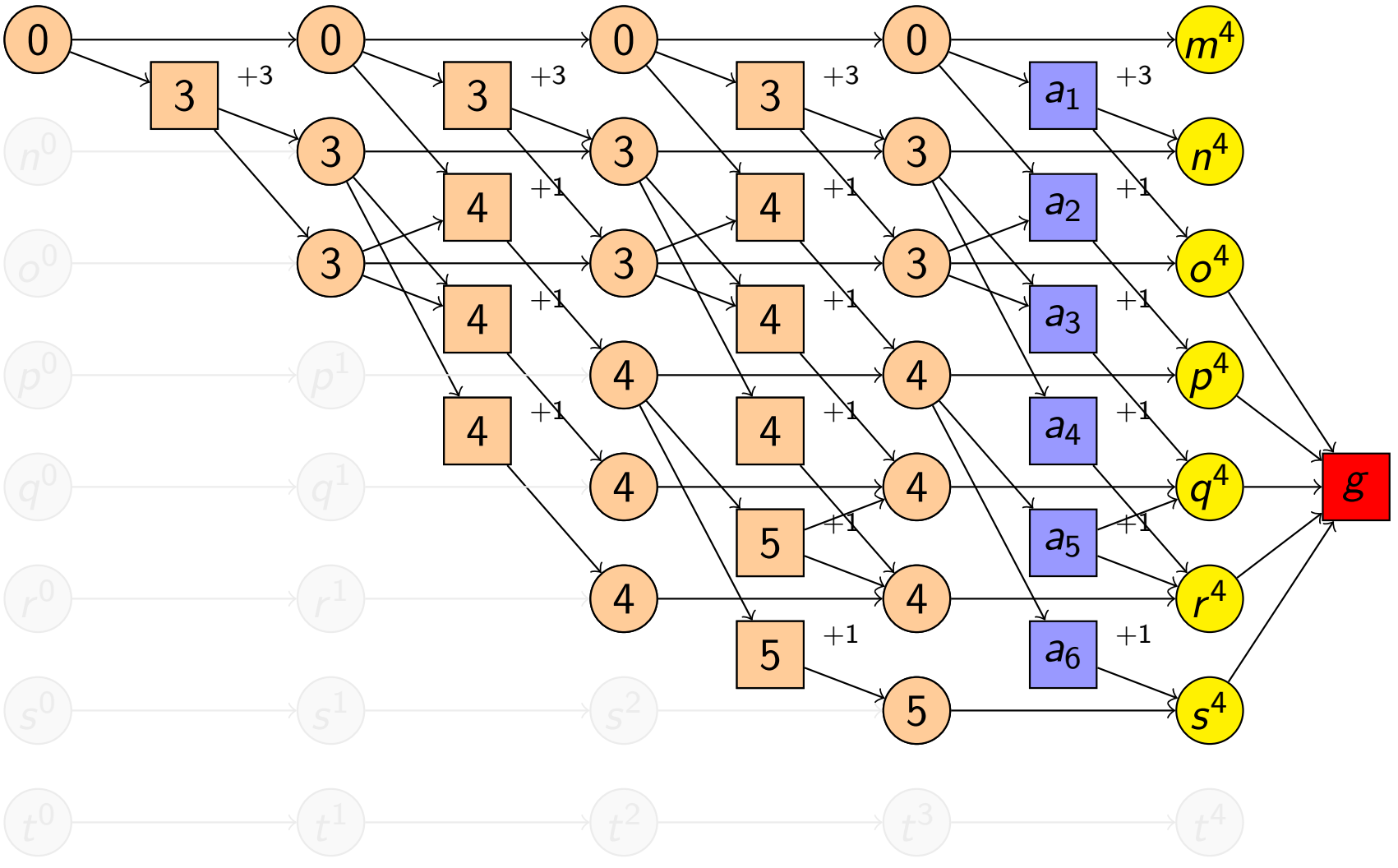
Illustrative Example: $h^{\max}$



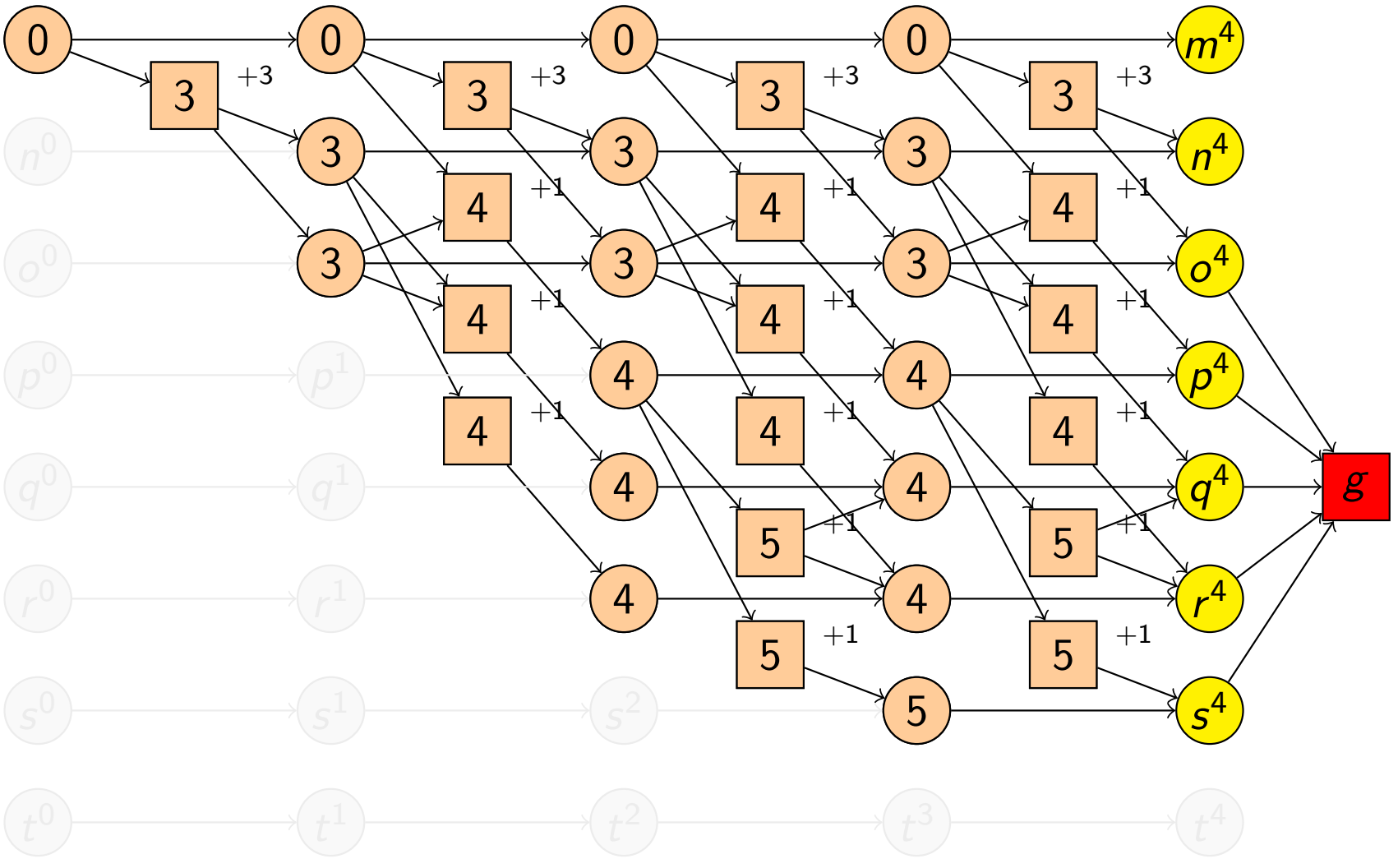
Illustrative Example: $h^{\max}$



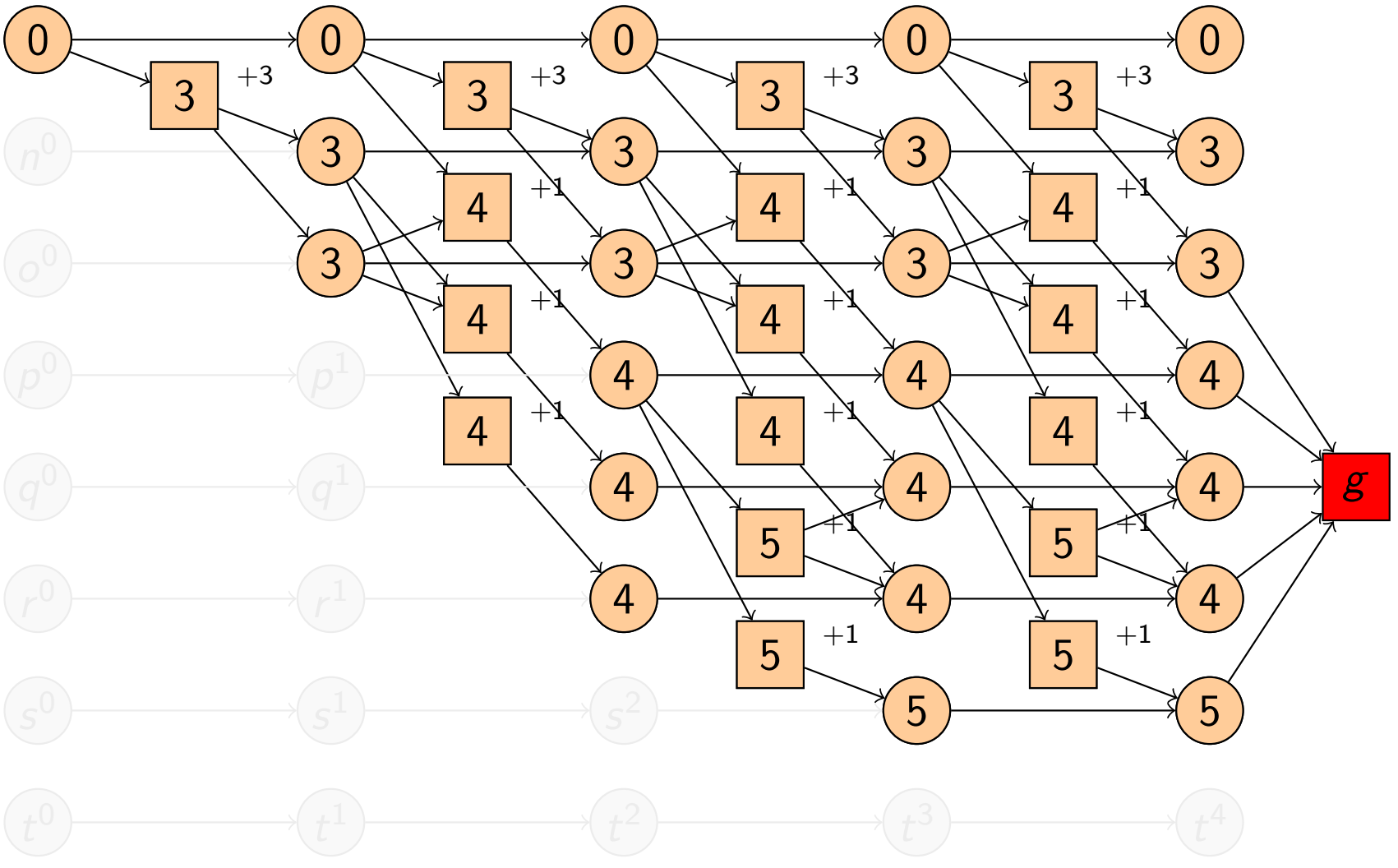
Illustrative Example: $h^{\max}$



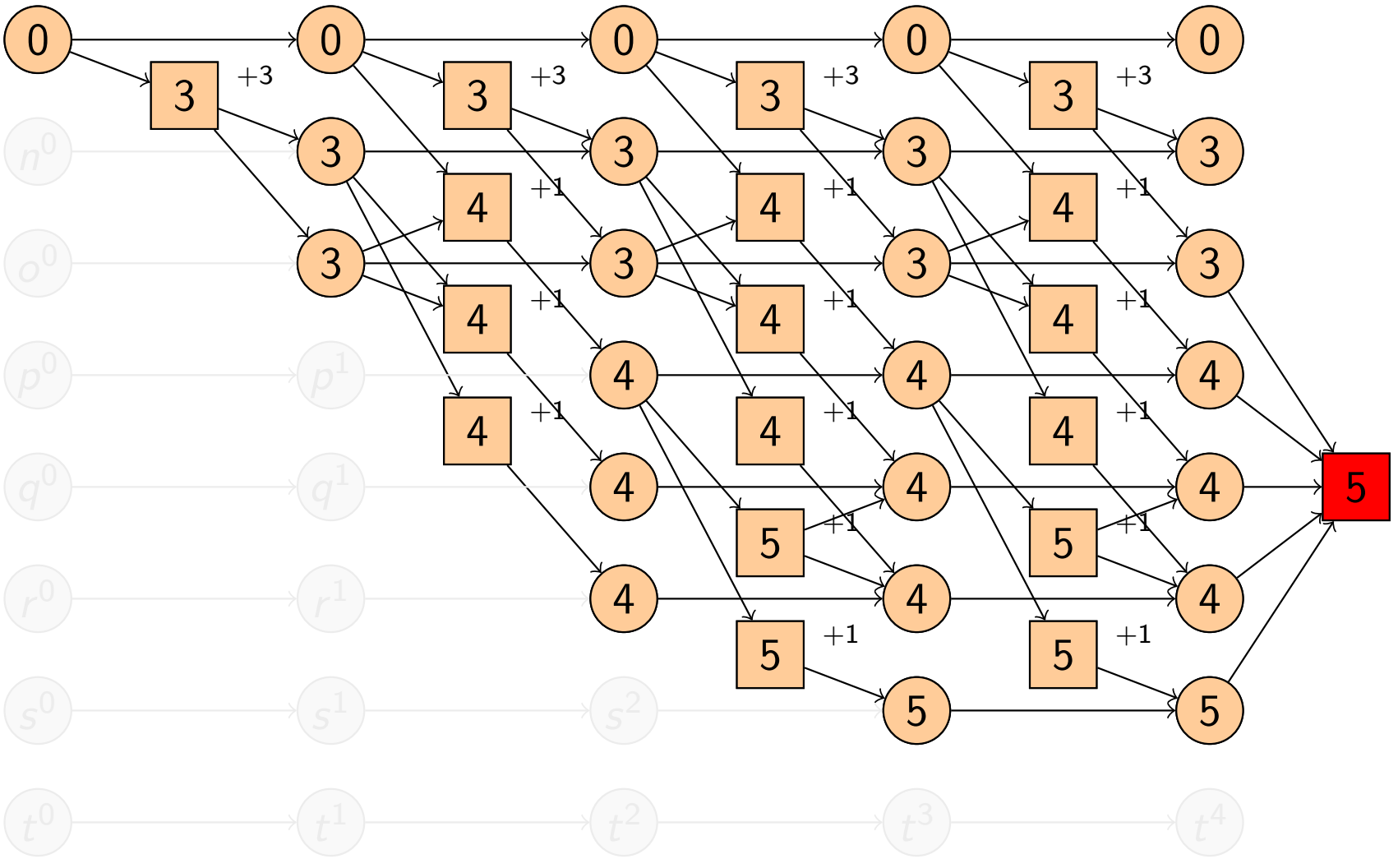
Illustrative Example: $h^{\max}$



Illustrative Example: $h^{\max}$

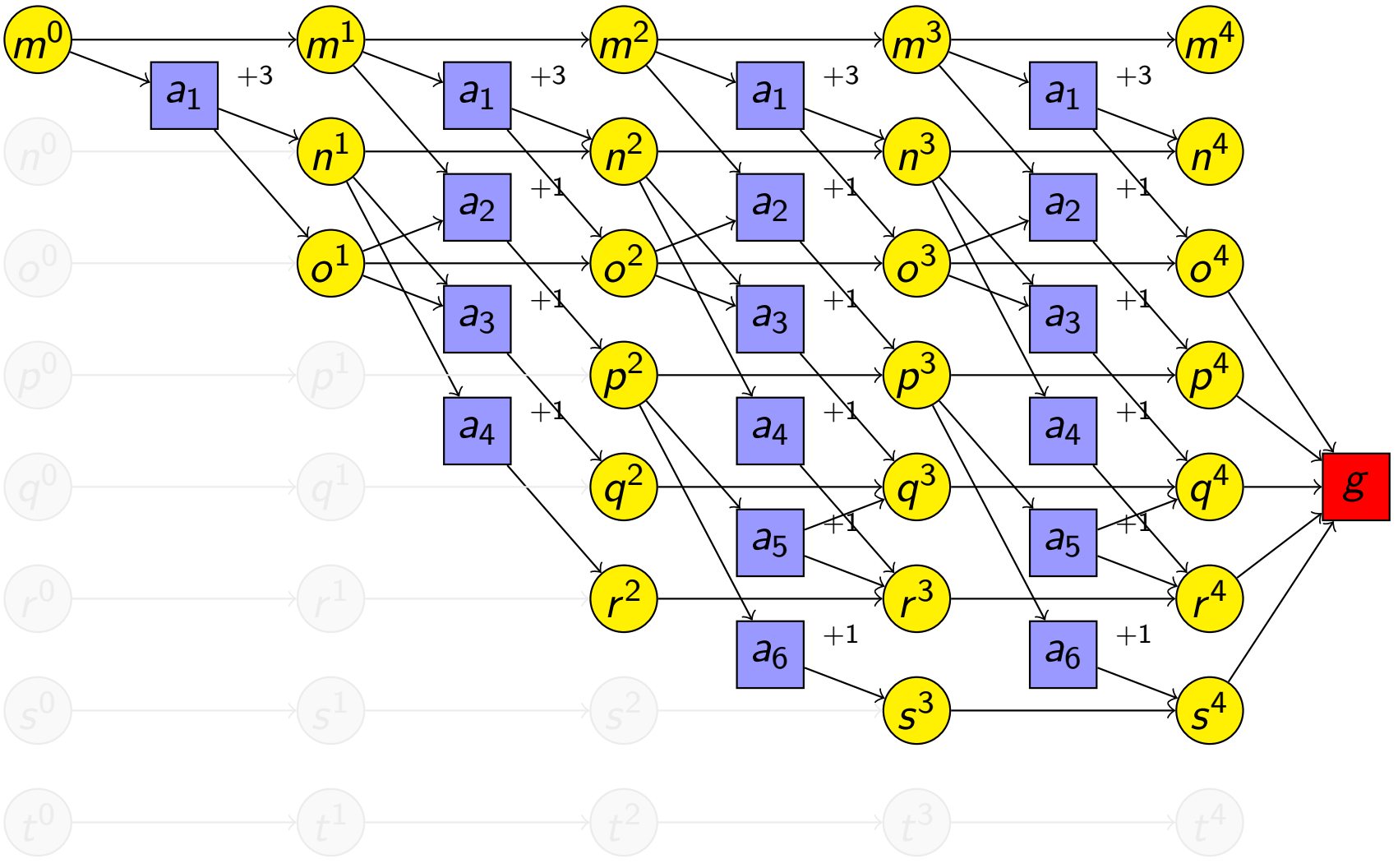


Illustrative Example: $h^{\max}$

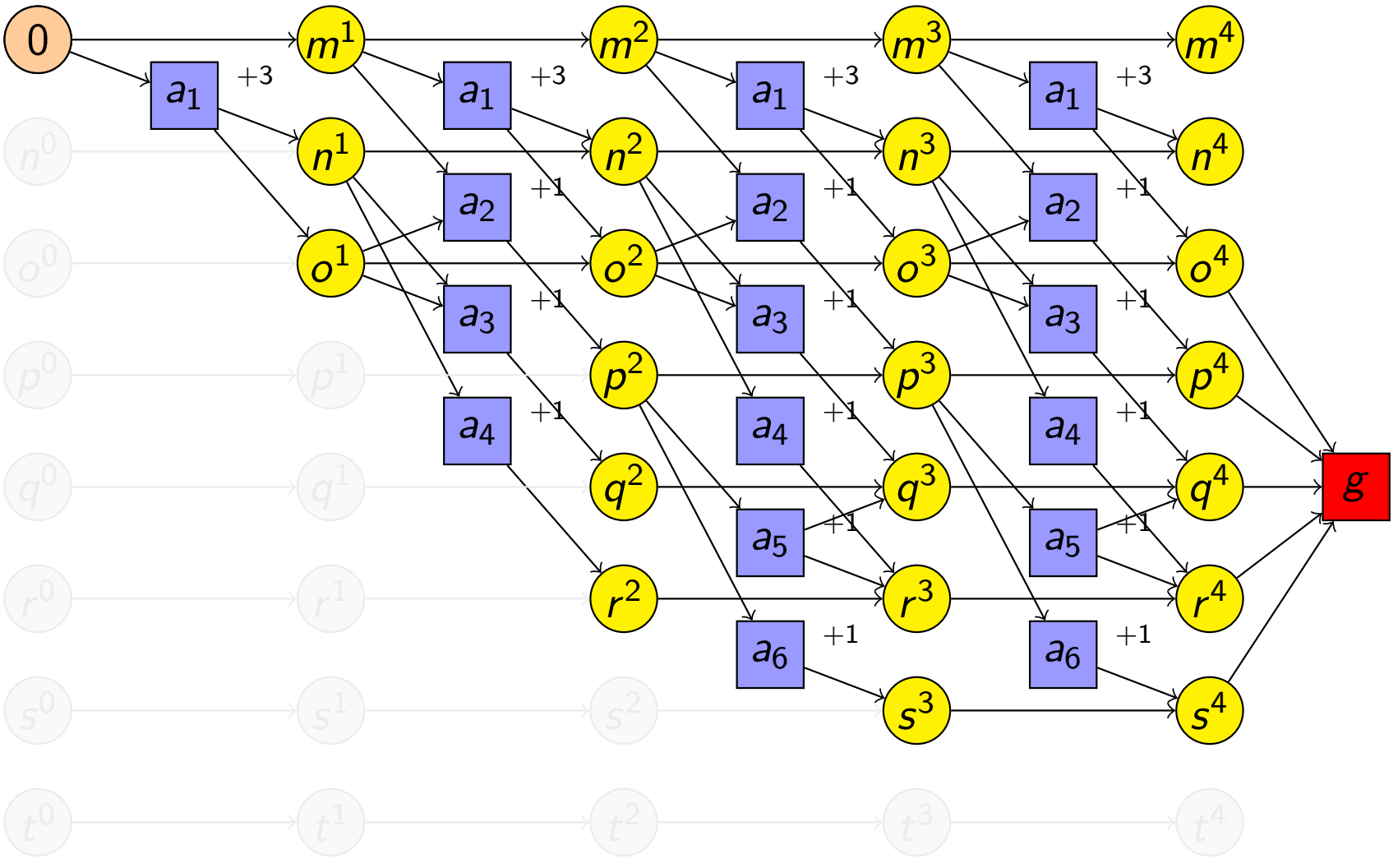


$h^{\max}(\{m\}) = 5$

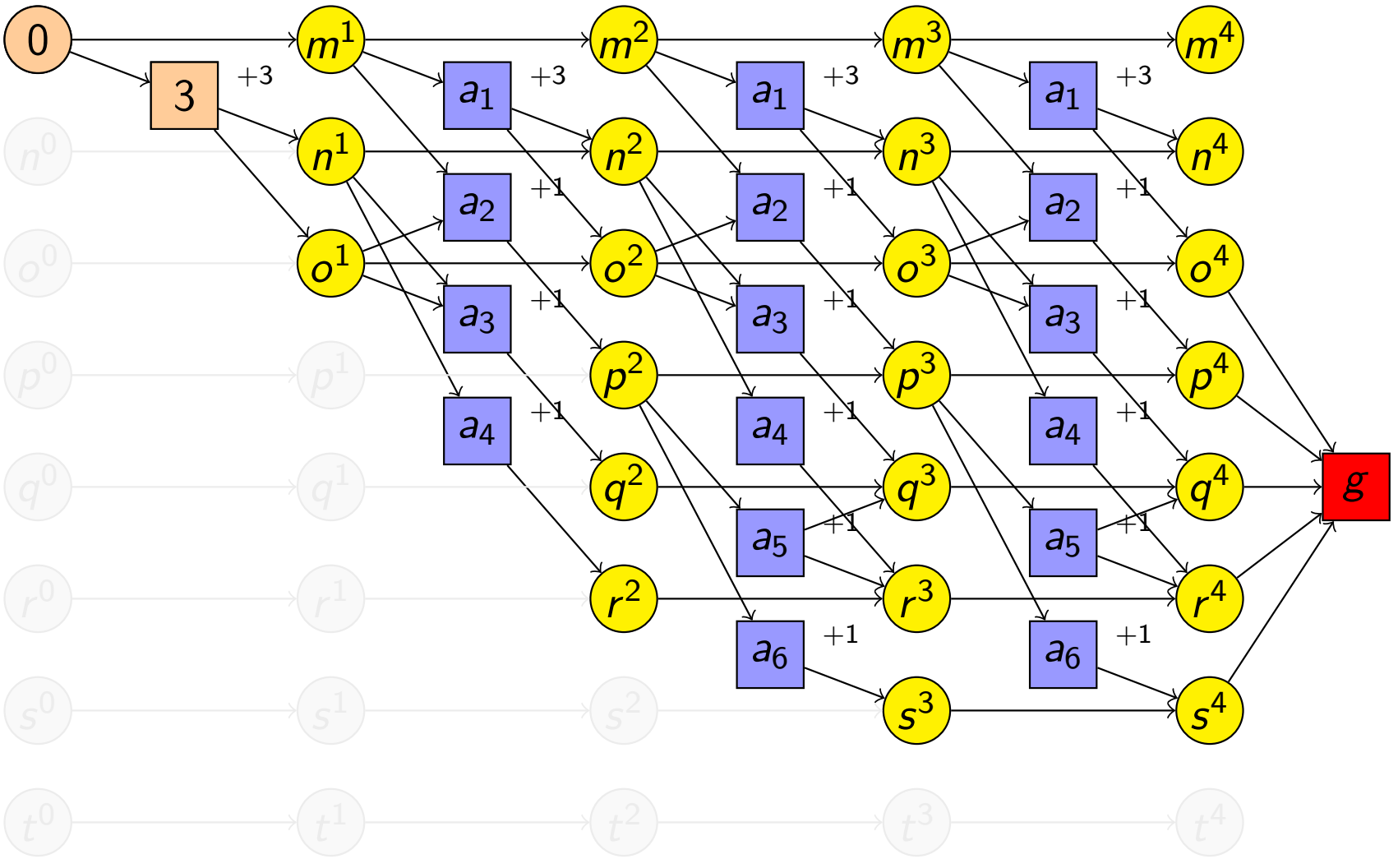
Illustrative Example: h^{add}



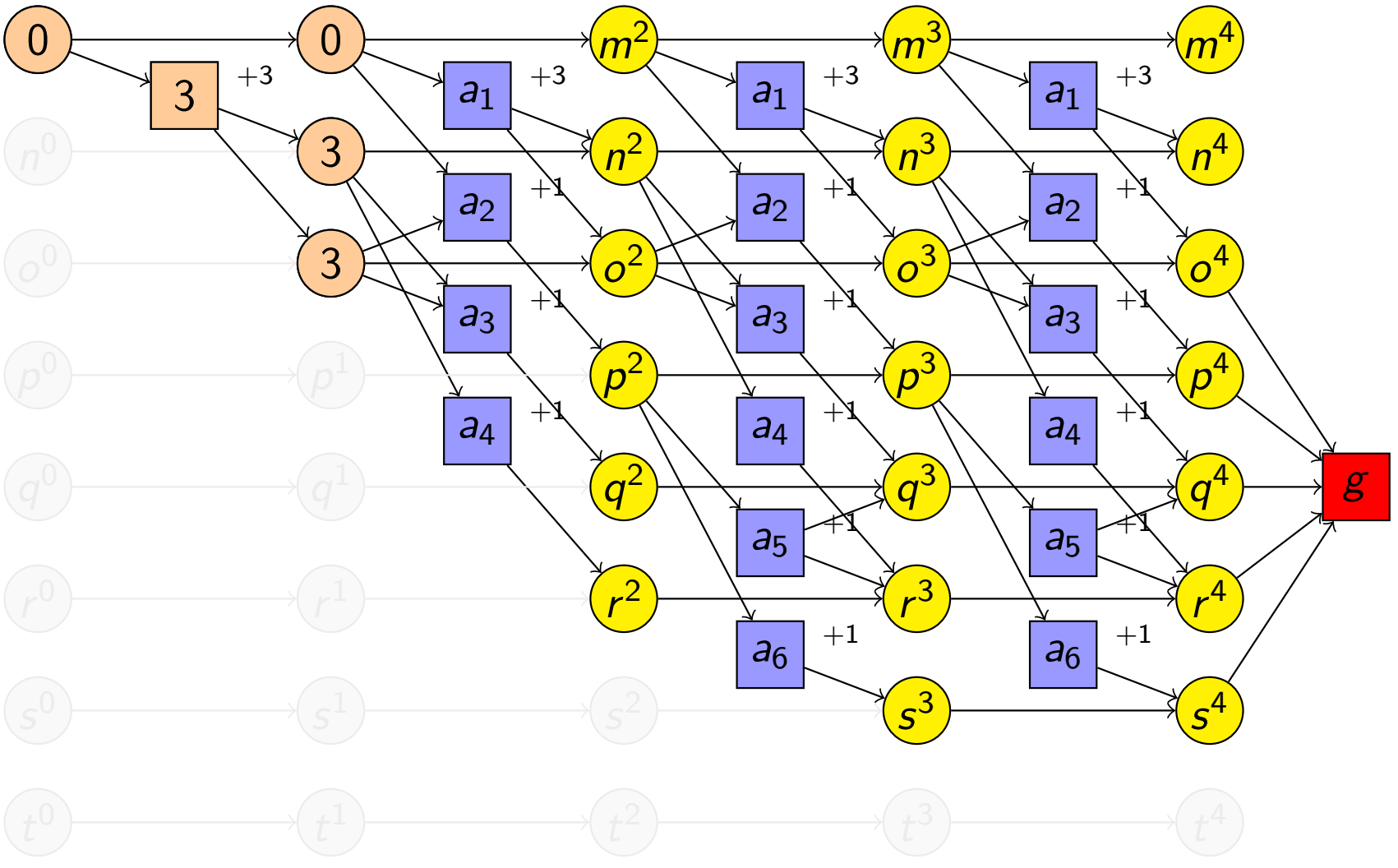
Illustrative Example: h^{add}



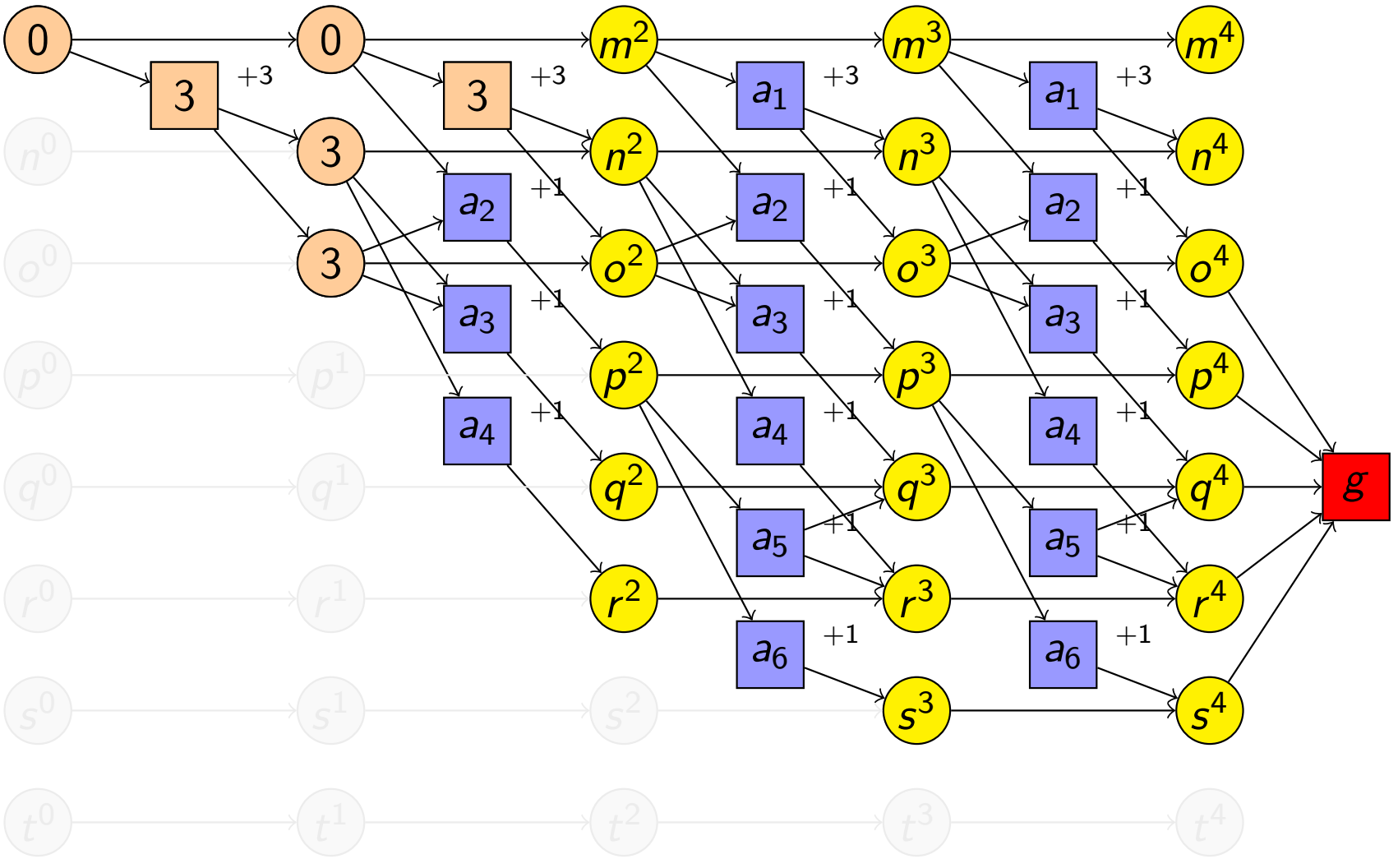
Illustrative Example: h^{add}



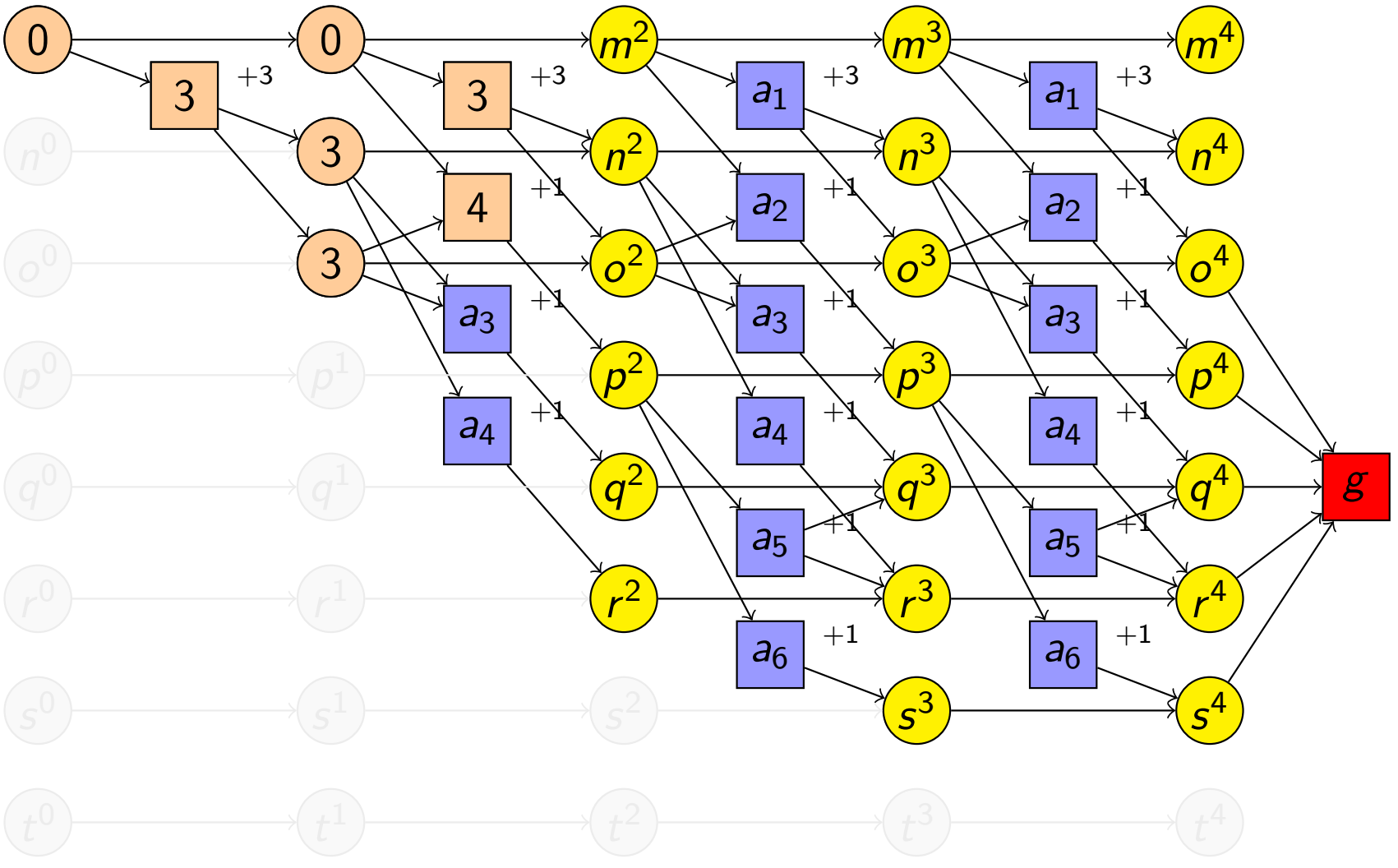
Illustrative Example: h^{add}



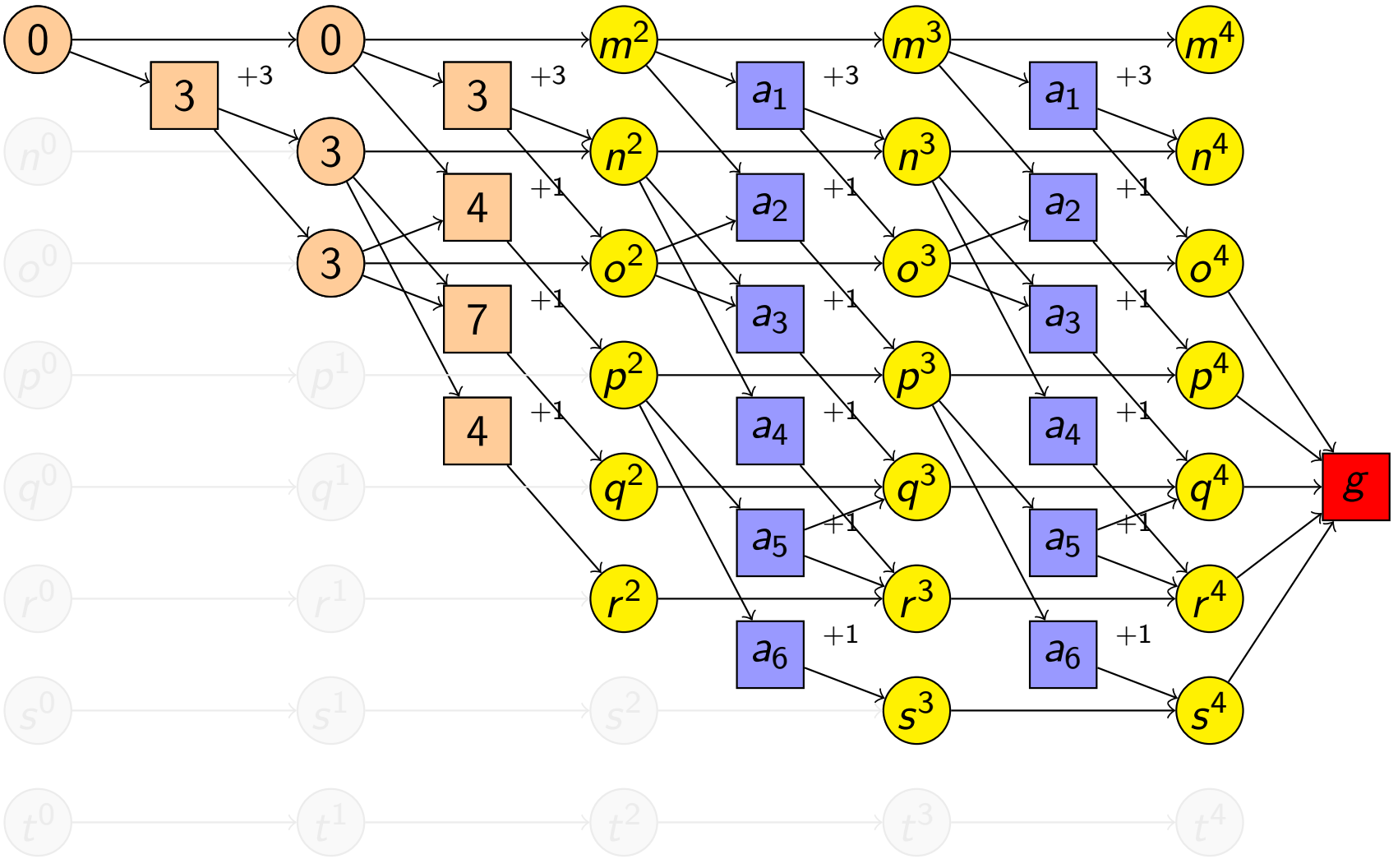
Illustrative Example: h^{add}



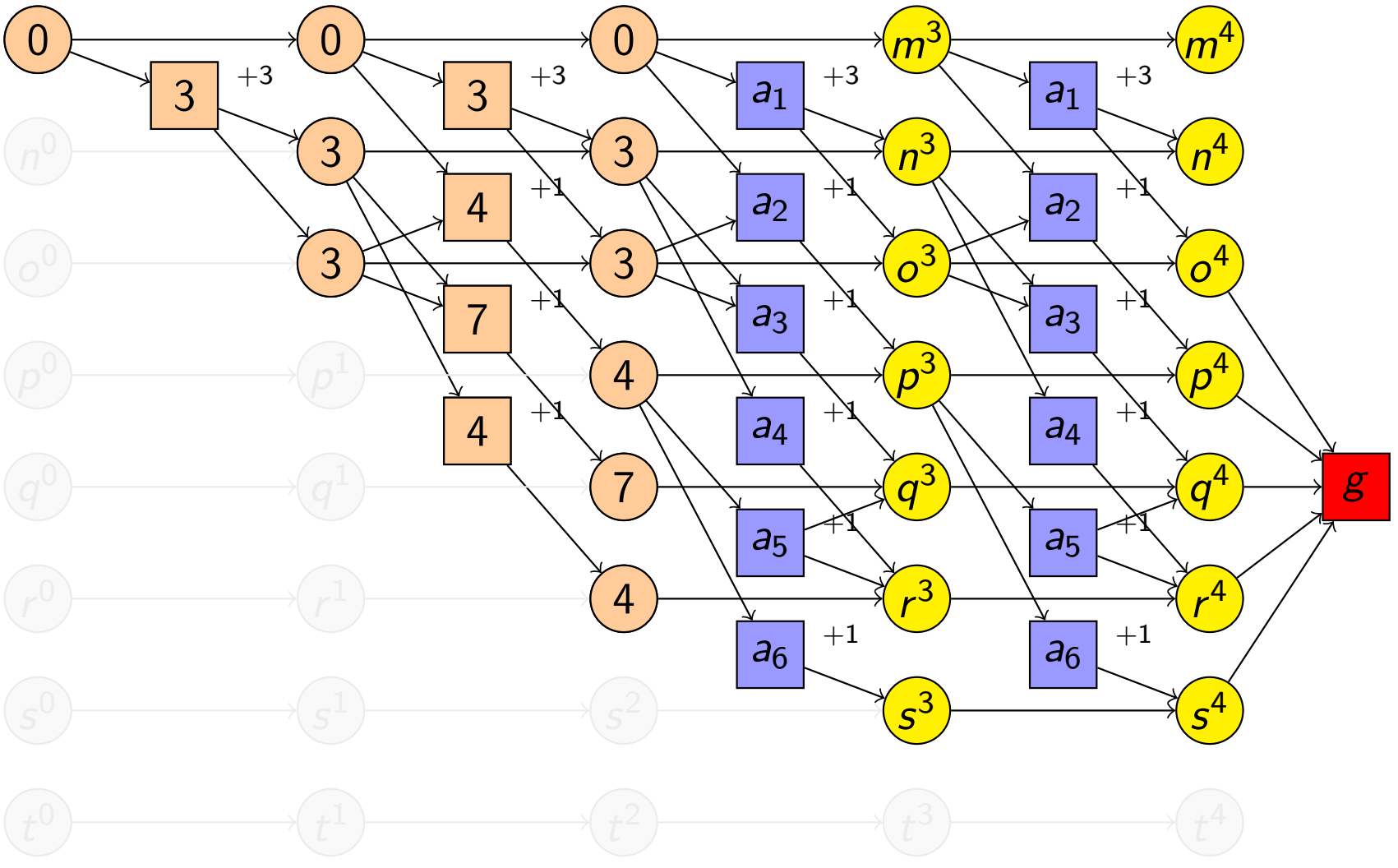
Illustrative Example: h^{add}



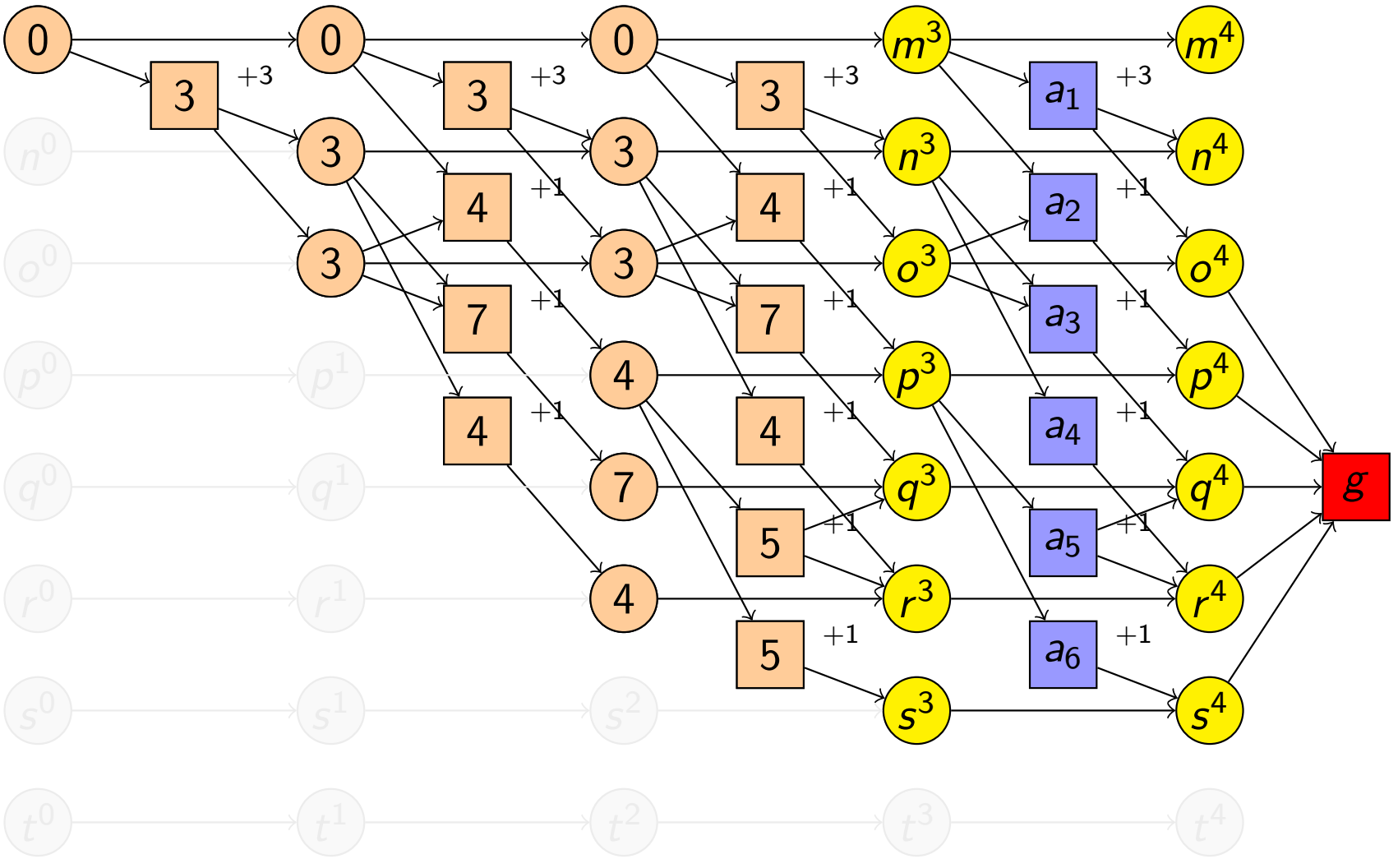
Illustrative Example: h^{add}



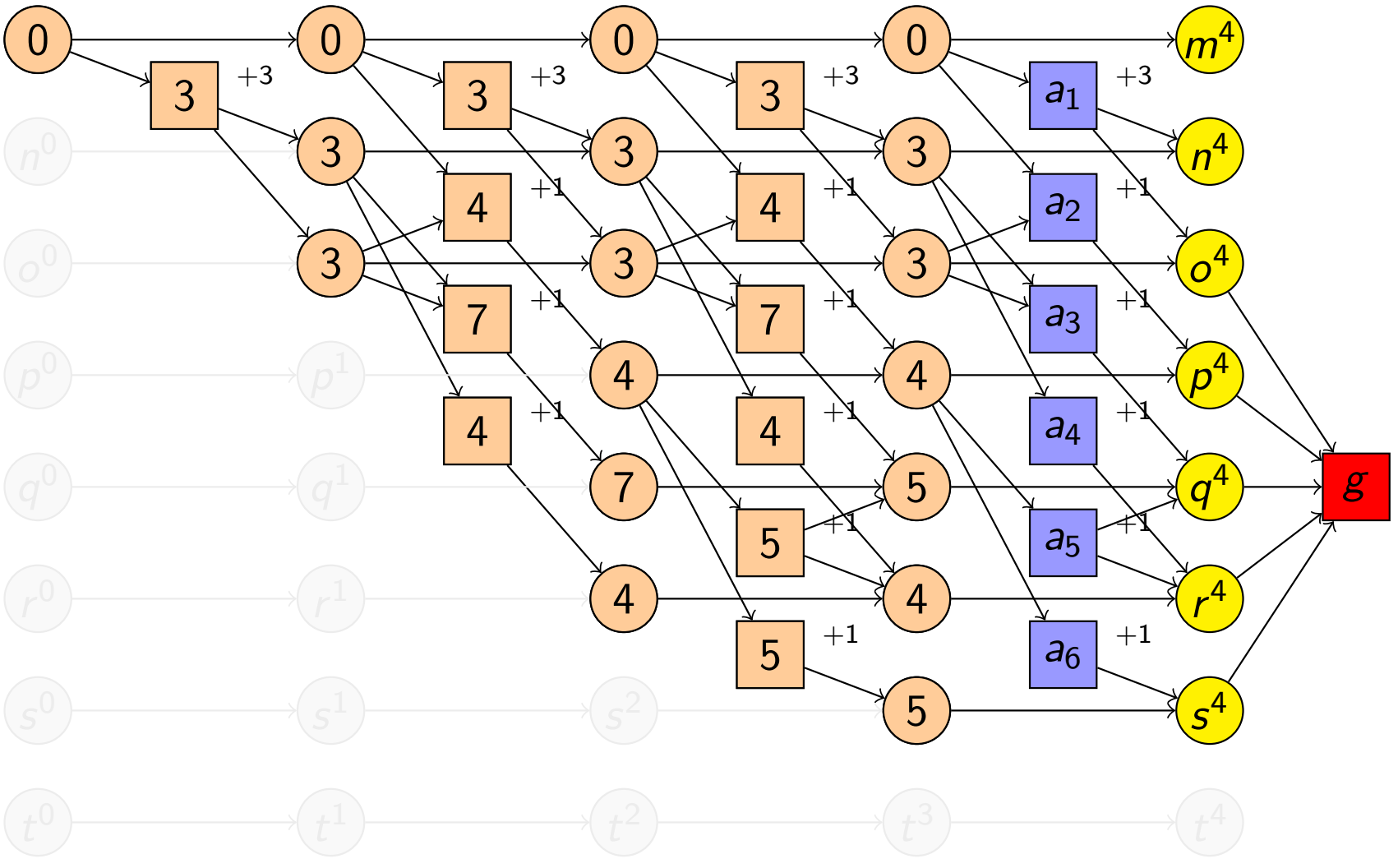
Illustrative Example: h^{add}



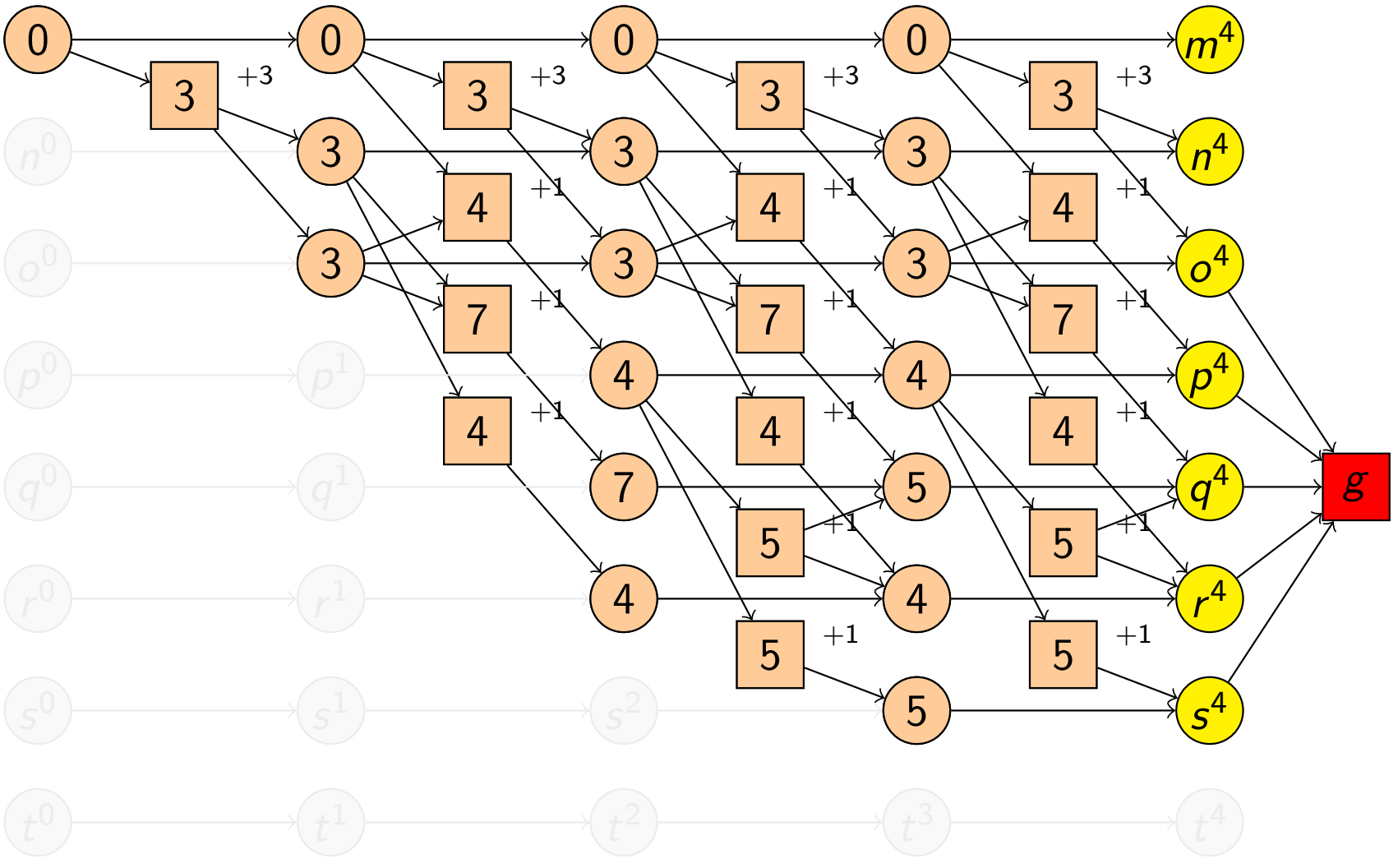
Illustrative Example: h^{add}



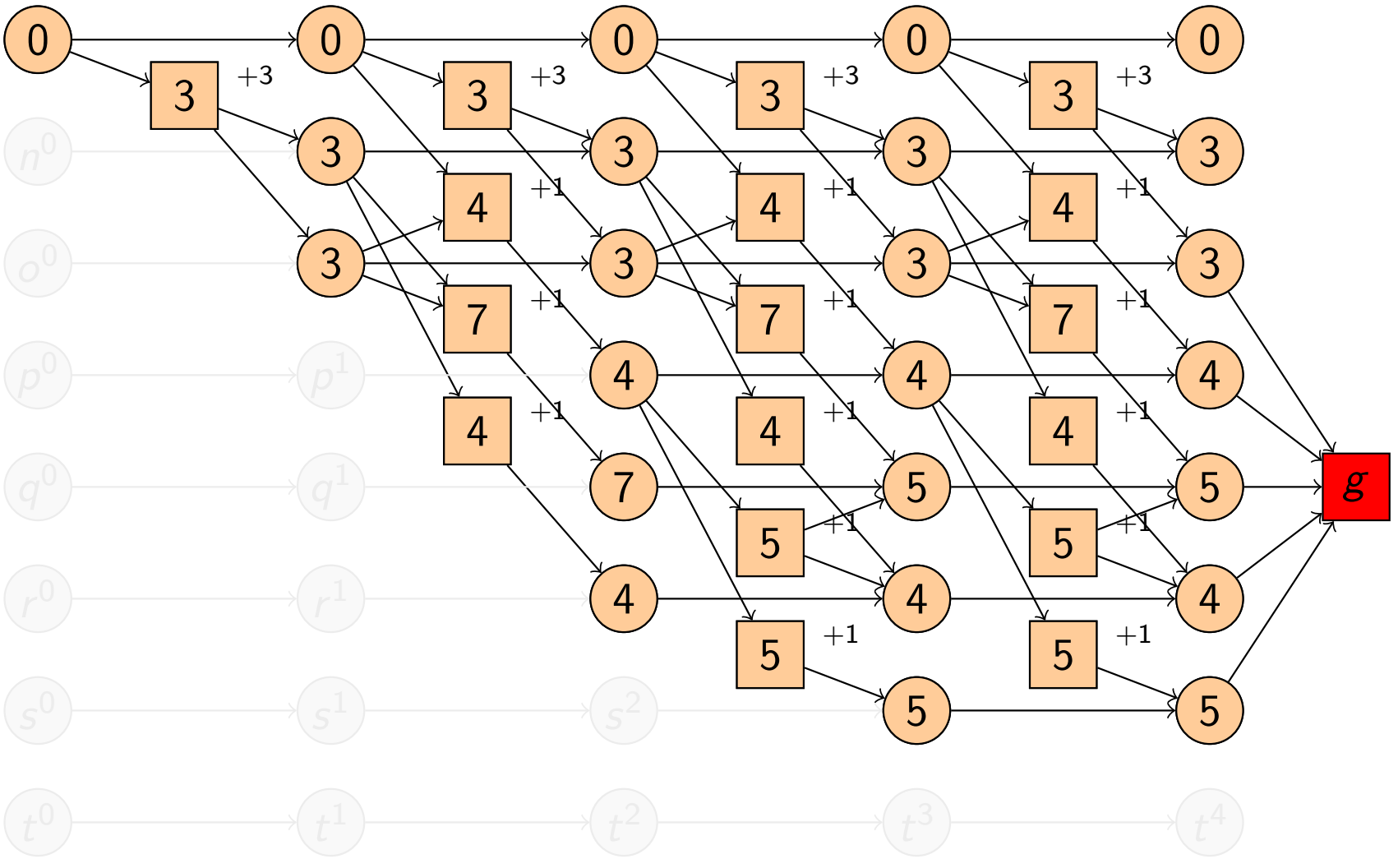
Illustrative Example: h^{add}



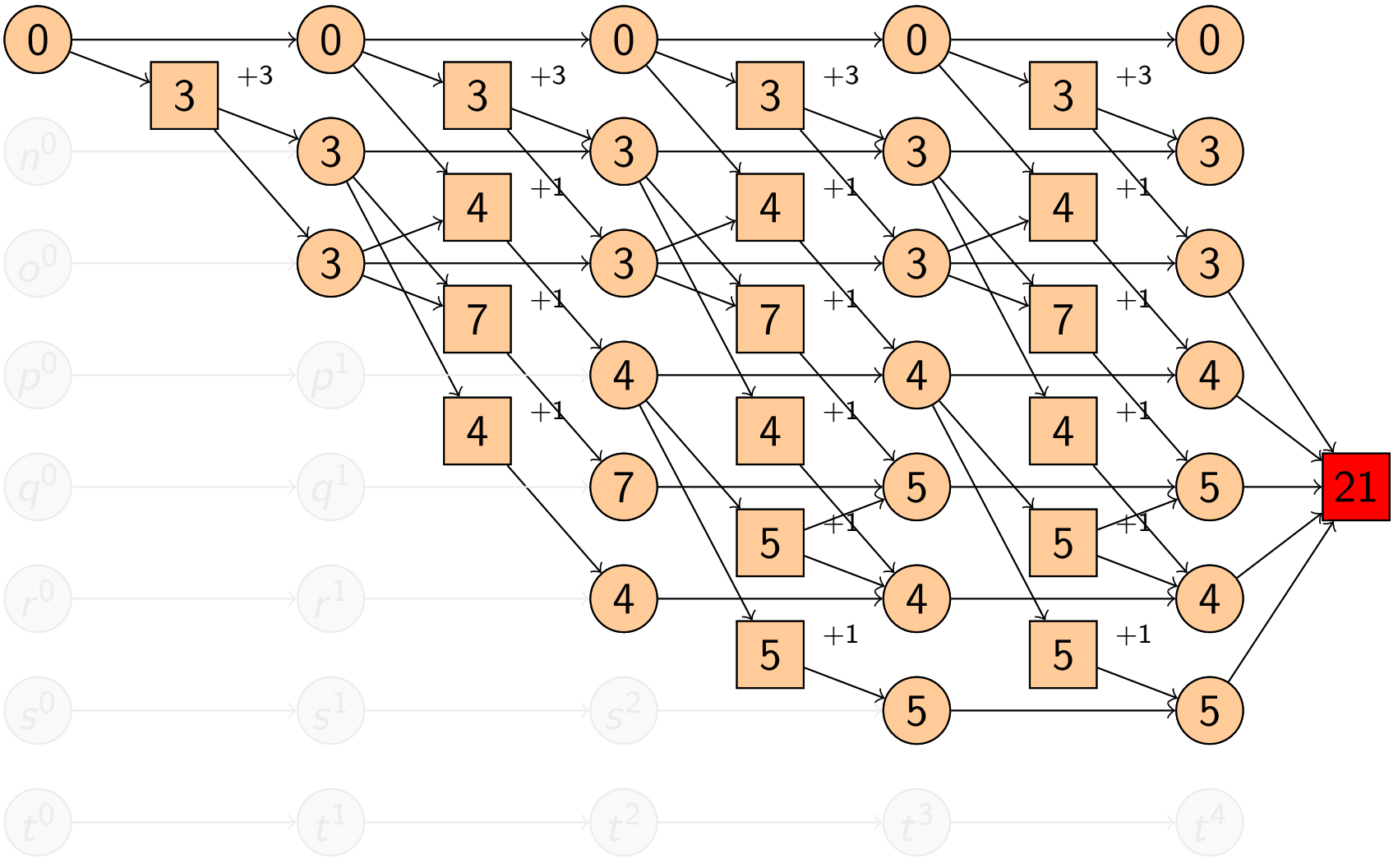
Illustrative Example: h^{add}



Illustrative Example: h^{add}



Illustrative Example: h^{add}



$h^{\text{add}}(\{m\}) = 21$

$h^{\max}$ and h^{add}: Remarks

comparison of $h^{\max}$ and h^{add}:

- both are safe and goal-aware
- $h^{\max}$ is admissible and consistent; h^{add} is neither.

$\rightsquigarrow$ h^{add} not suited for optimal planning

$h^{\max}$ and h^{add}: Remarks

comparison of $h^{\max}$ and h^{add}:

- both are safe and goal-aware
- $h^{\max}$ is admissible and consistent; h^{add} is neither.

$\rightsquigarrow$ h^{add} not suited for optimal planning

- However, h^{add} is usually much more informative than $h^{\max}$. Greedy best-first search with h^{add} is a decent algorithm.

$h^{\max}$ and h^{add}: Remarks

comparison of $h^{\max}$ and h^{add}:

- both are safe and goal-aware
- $h^{\max}$ is admissible and consistent; h^{add} is neither.

$\rightsquigarrow$ h^{add} not suited for optimal planning

- However, h^{add} is usually much more informative than $h^{\max}$. Greedy best-first search with h^{add} is a decent algorithm.
- Apart from not being admissible, h^{add} often vastly overestimates the actual costs because positive synergies between subgoals are not recognized.

$h^{\max}$ and h^{add}: Remarks

comparison of $h^{\max}$ and h^{add}:

- both are safe and goal-aware
- $h^{\max}$ is admissible and consistent; h^{add} is neither.

$\rightsquigarrow$ h^{add} not suited for optimal planning

- However, h^{add} is usually much more informative than $h^{\max}$. Greedy best-first search with h^{add} is a decent algorithm.
- Apart from not being admissible, h^{add} often vastly overestimates the actual costs because positive synergies between subgoals are not recognized.

$\rightsquigarrow$ FF heuristic

FF Heuristic

FF Heuristic

The FF Heuristic

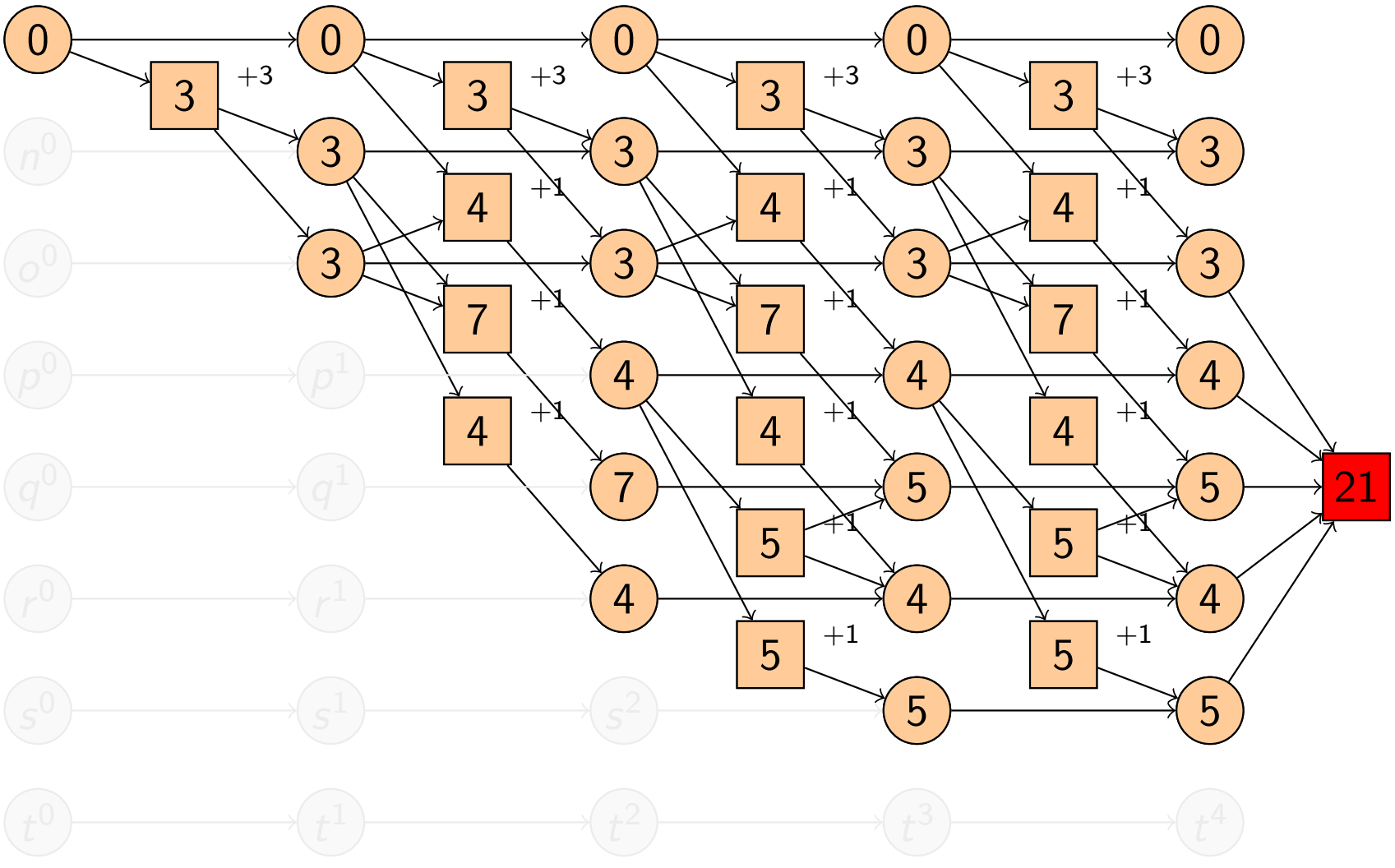
identical to h^{add}, but additional steps at the end:

- Mark goal vertex.
- Apply the following marking rules until nothing more to do:
  - marked action or goal vertex?
    $\rightsquigarrow$ mark all predecessors
  - marked variable vertex v^i in layer $i \geq 1$?
    $\rightsquigarrow$ mark one predecessor with minimal h^{add} value
    (tie-breaking: prefer variable vertices; otherwise arbitrary)

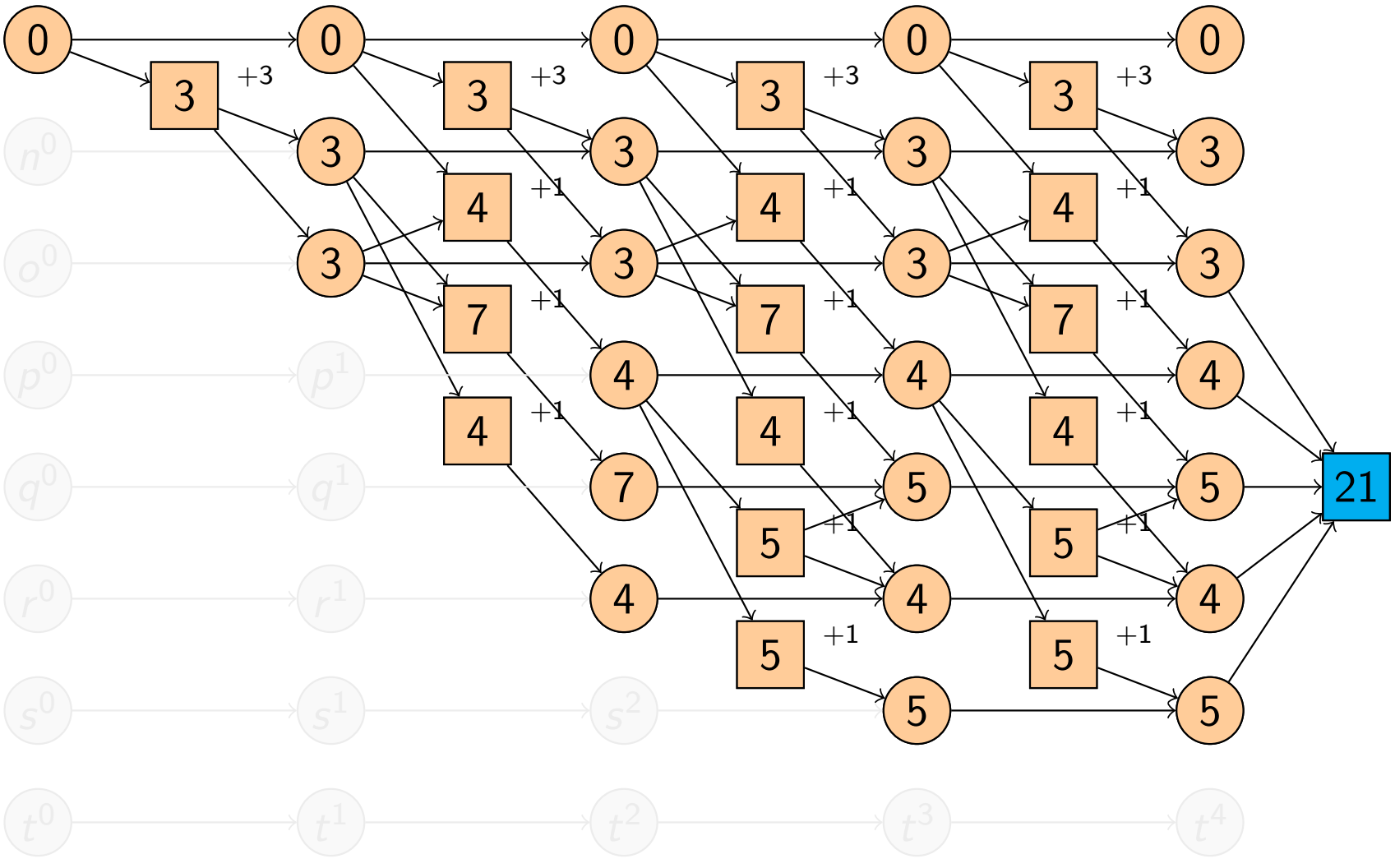
heuristic value:

- The actions corresponding to the marked action vertices build a relaxed plan.
- The cost of this plan is the heuristic value.

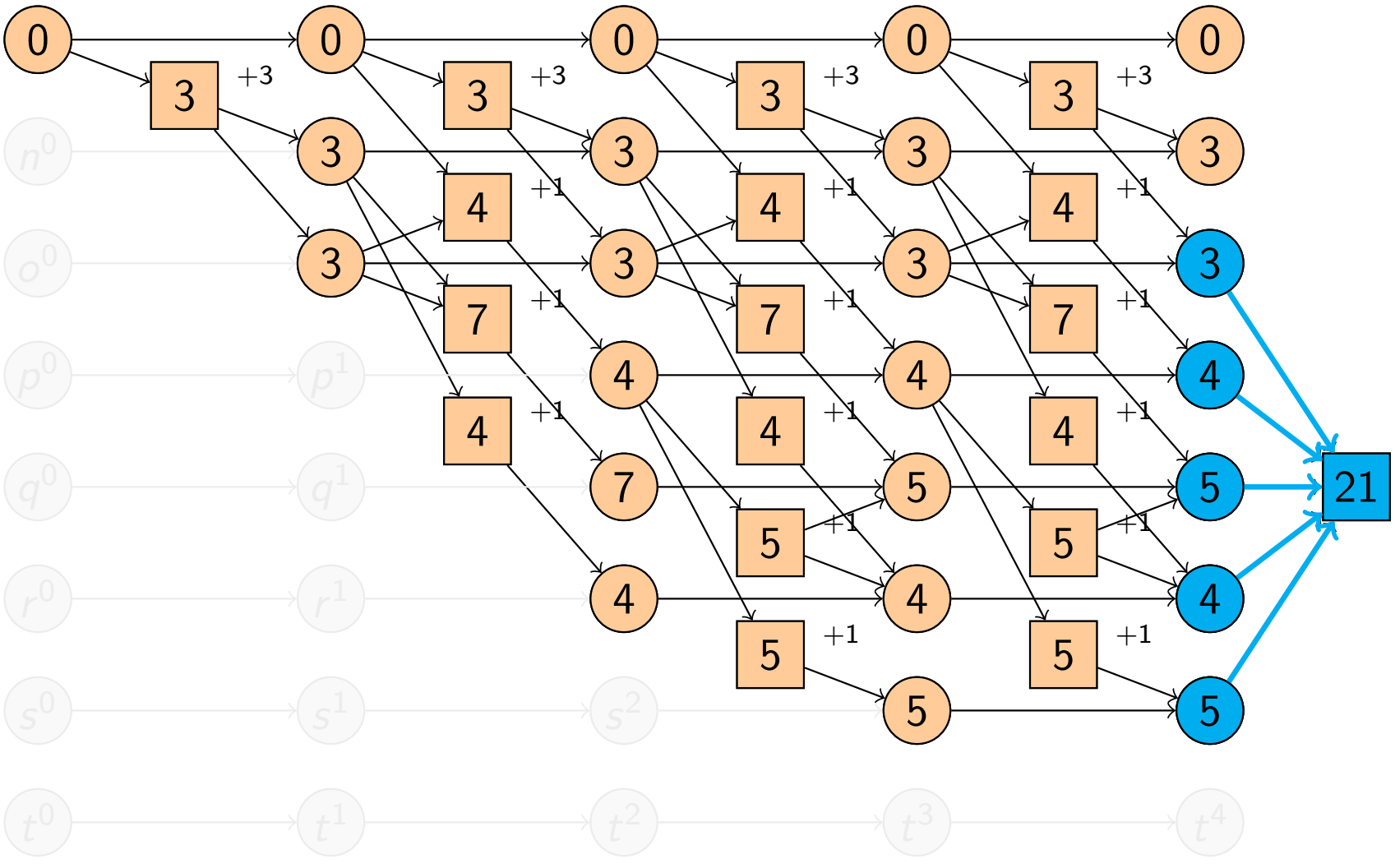
Illustrative Example: h^{FF}



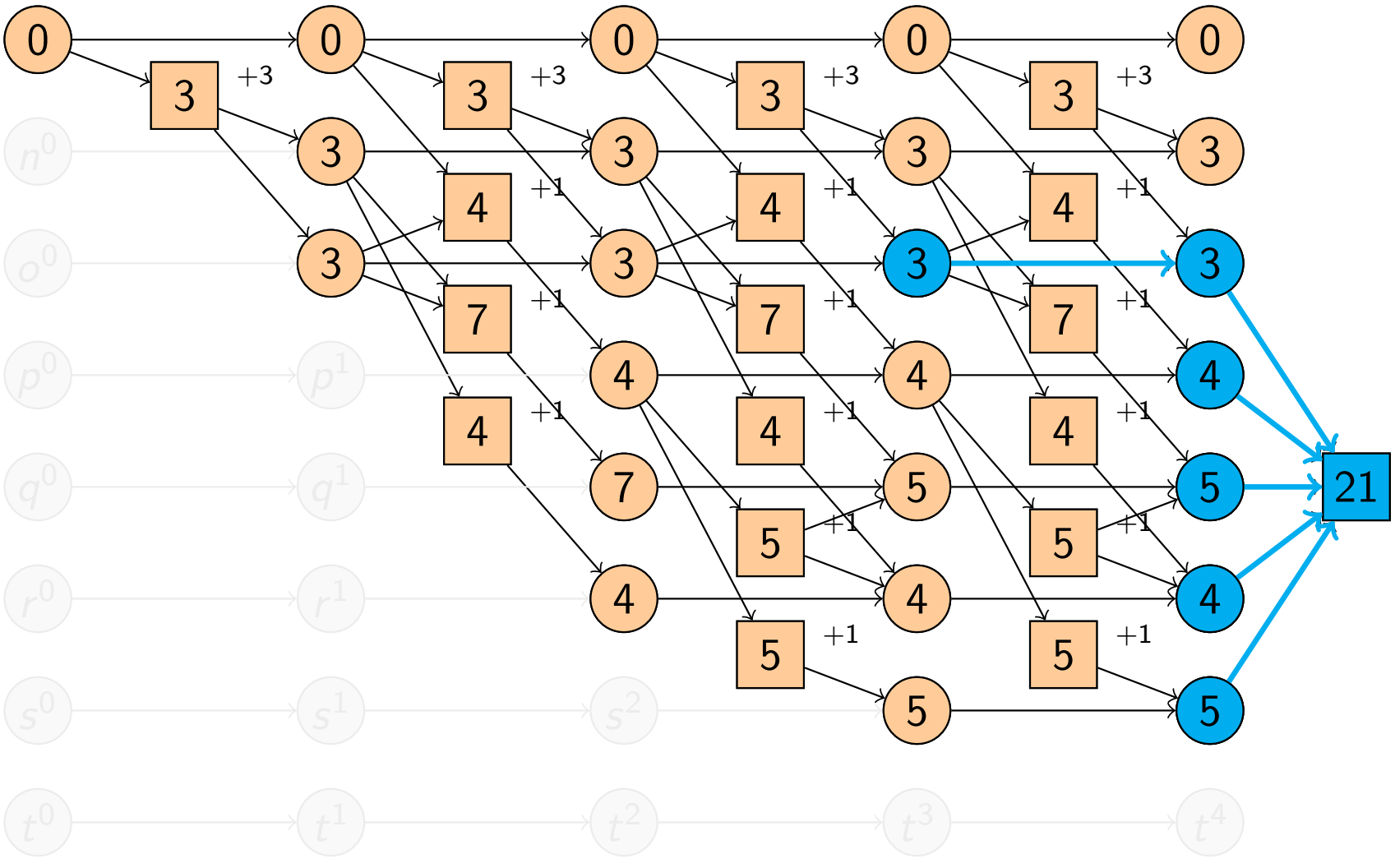
Illustrative Example: h^{FF}



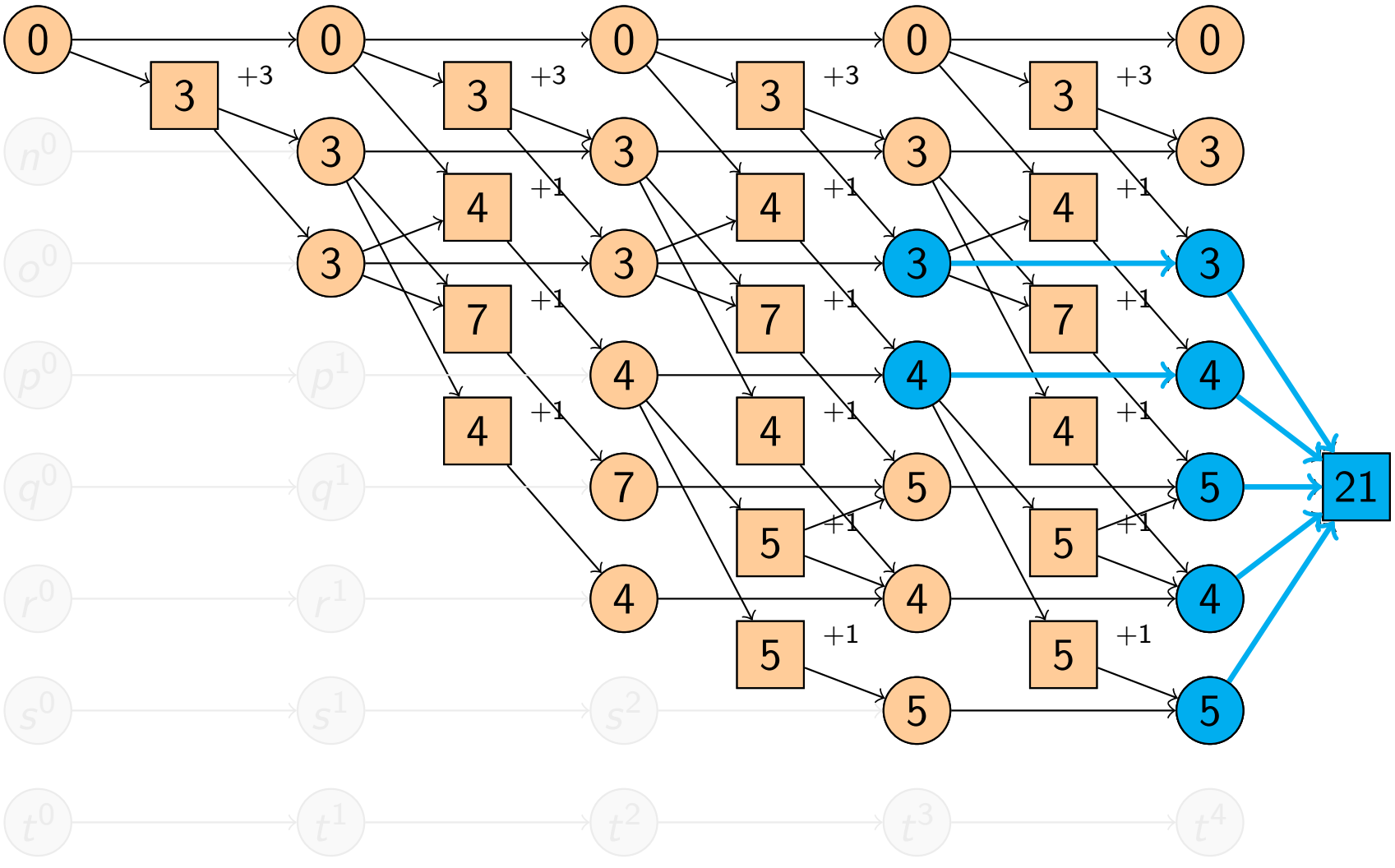
Illustrative Example: h^{FF}



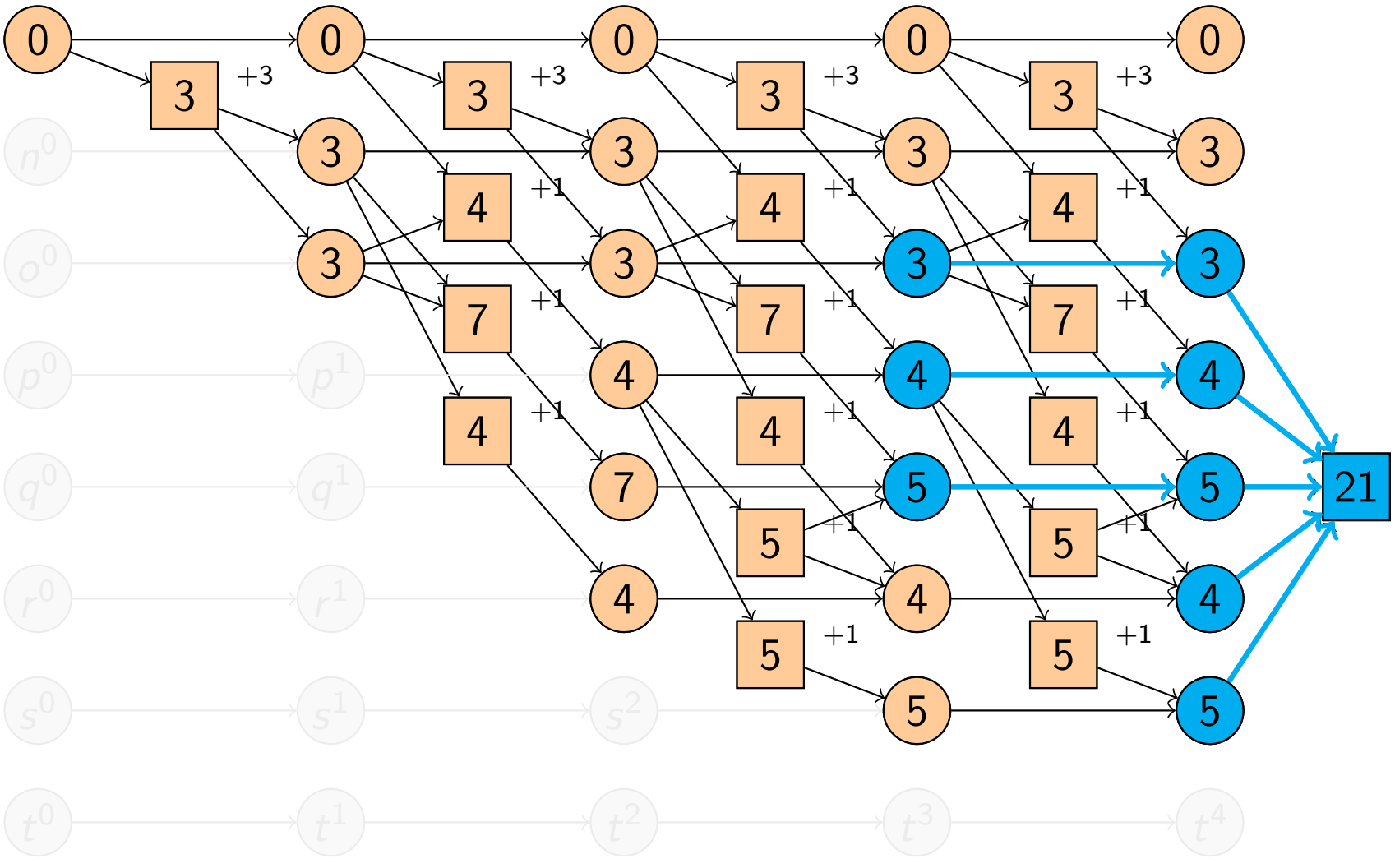
Illustrative Example: h^{FF}



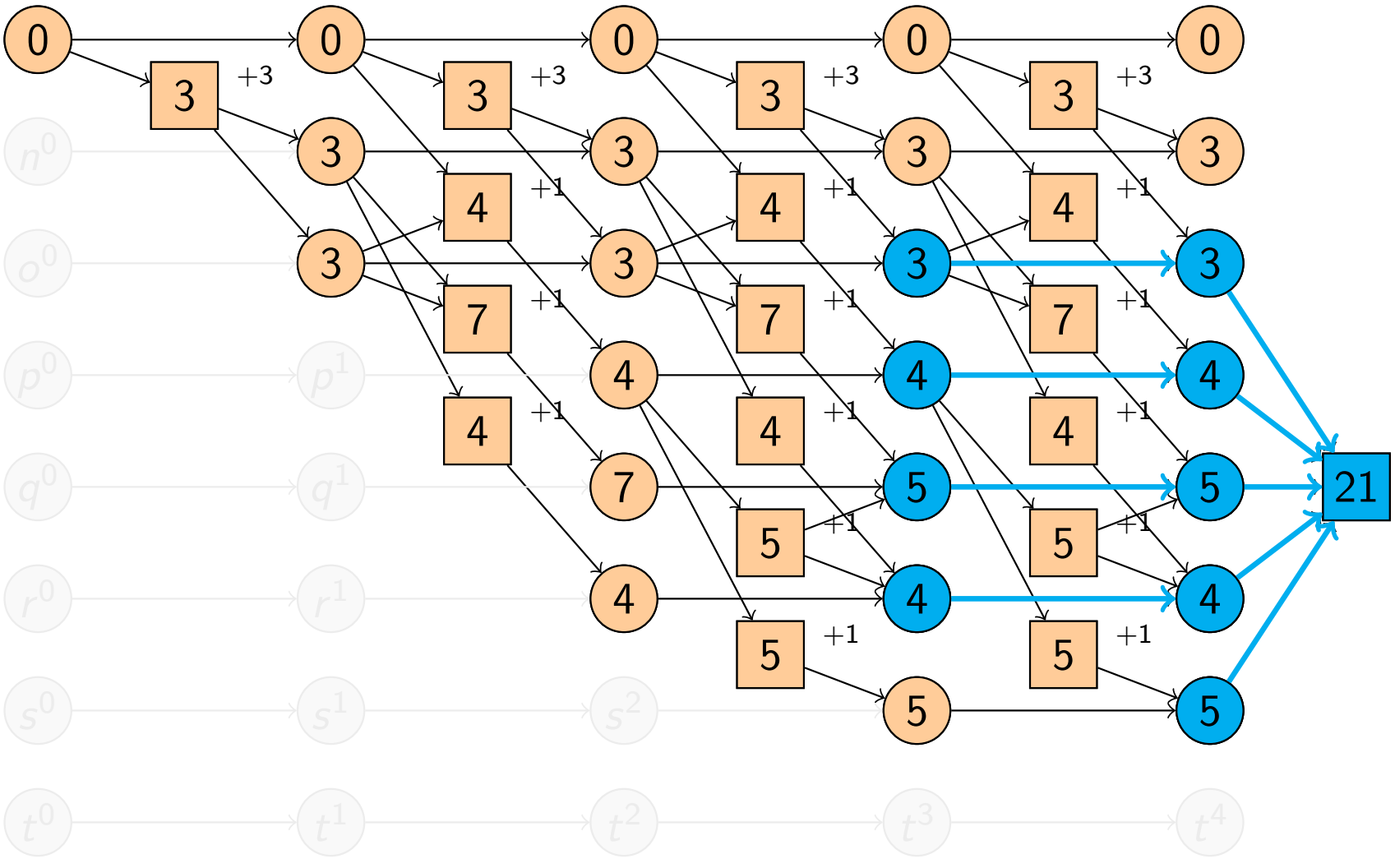
Illustrative Example: h^{FF}



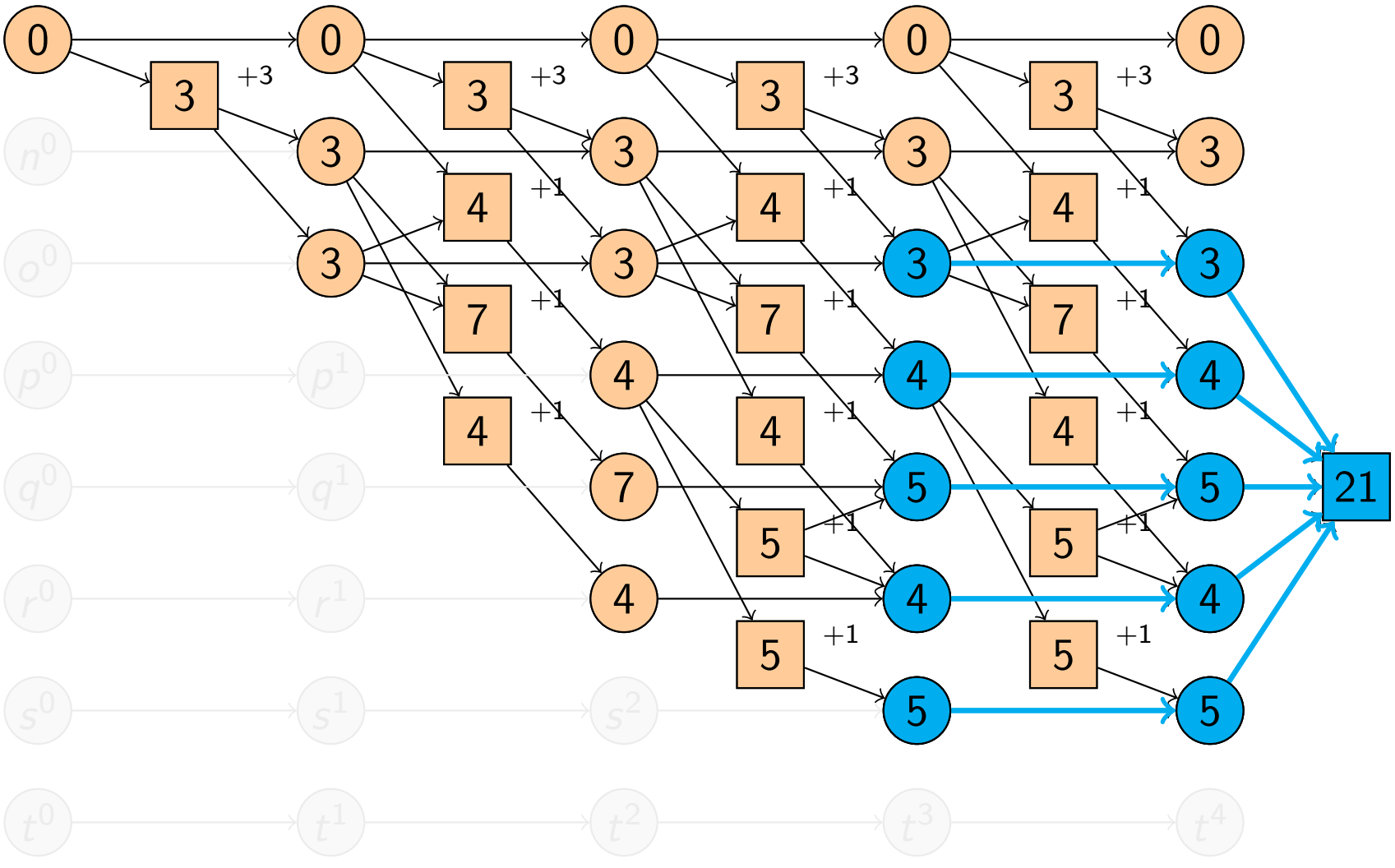
Illustrative Example: h^{FF}



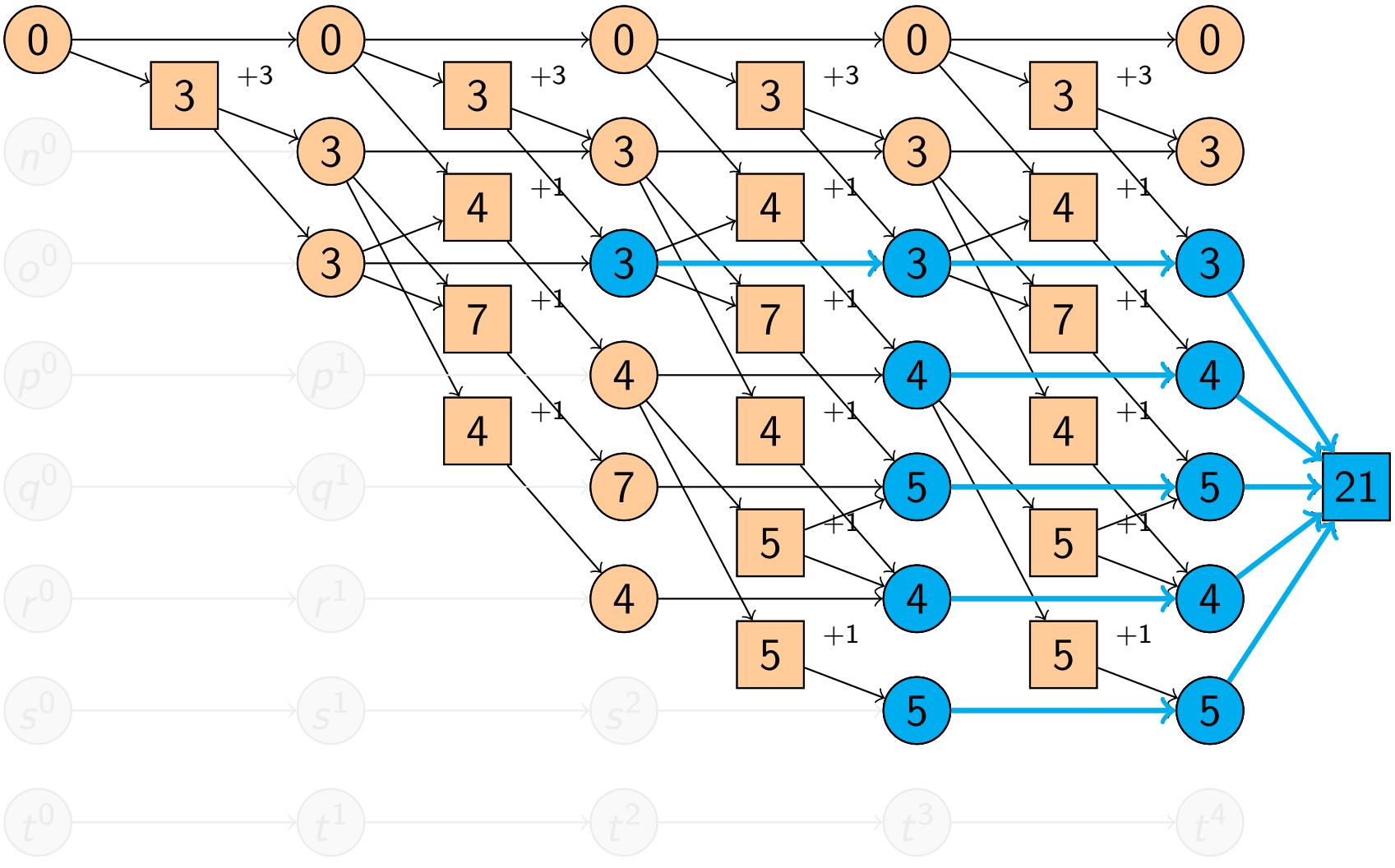
Illustrative Example: h^{FF}



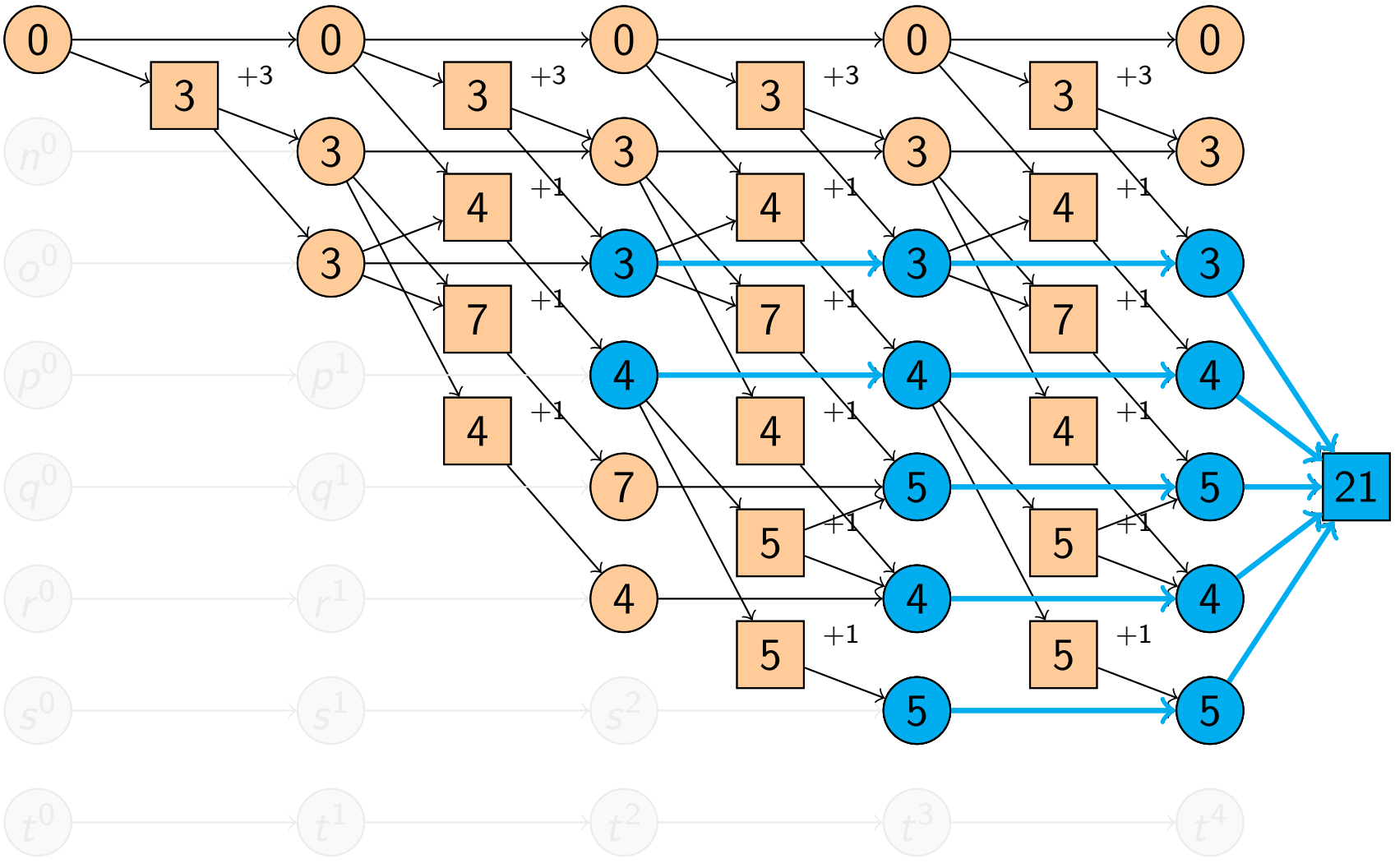
Illustrative Example: h^{FF}



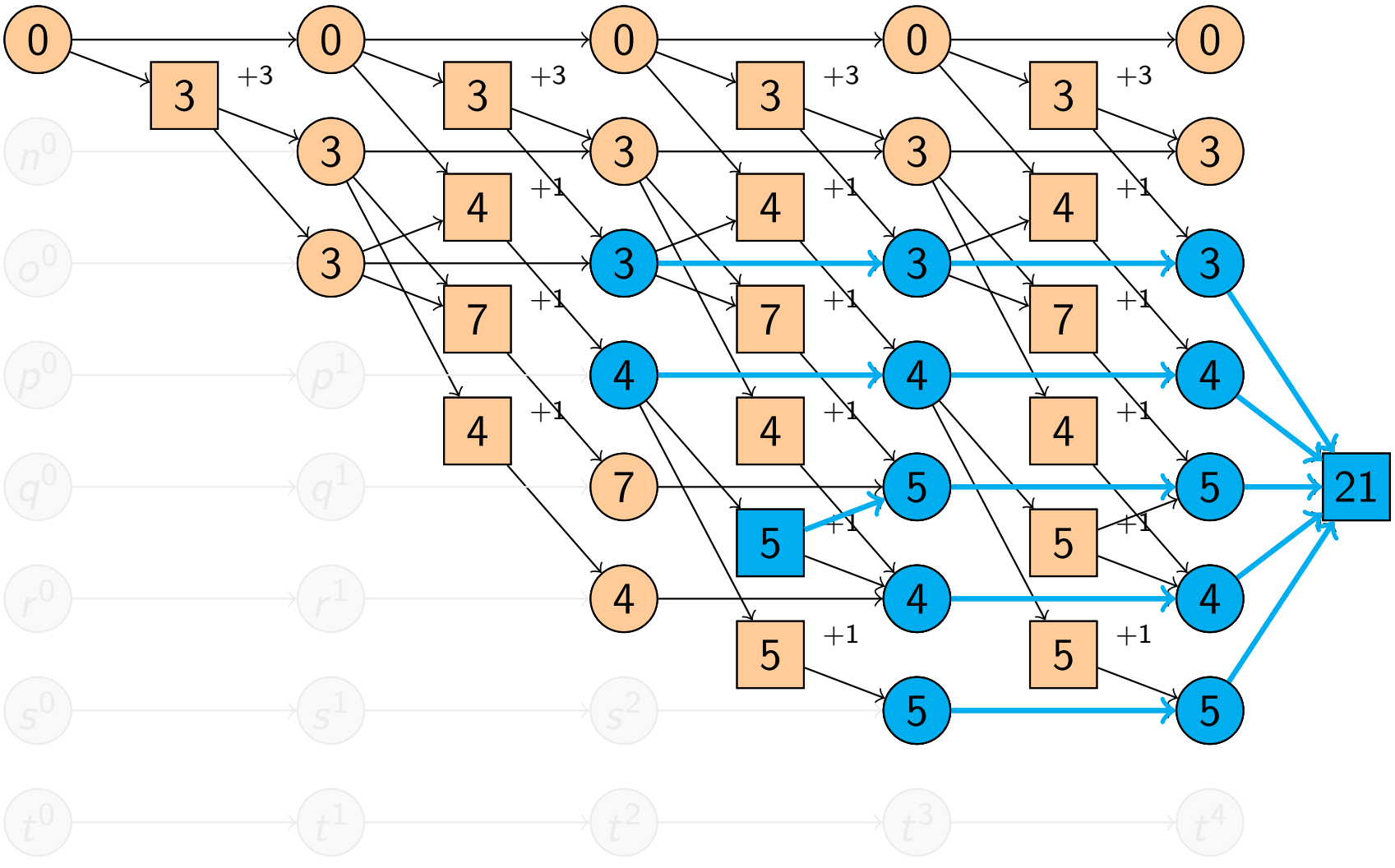
Illustrative Example: h^{FF}



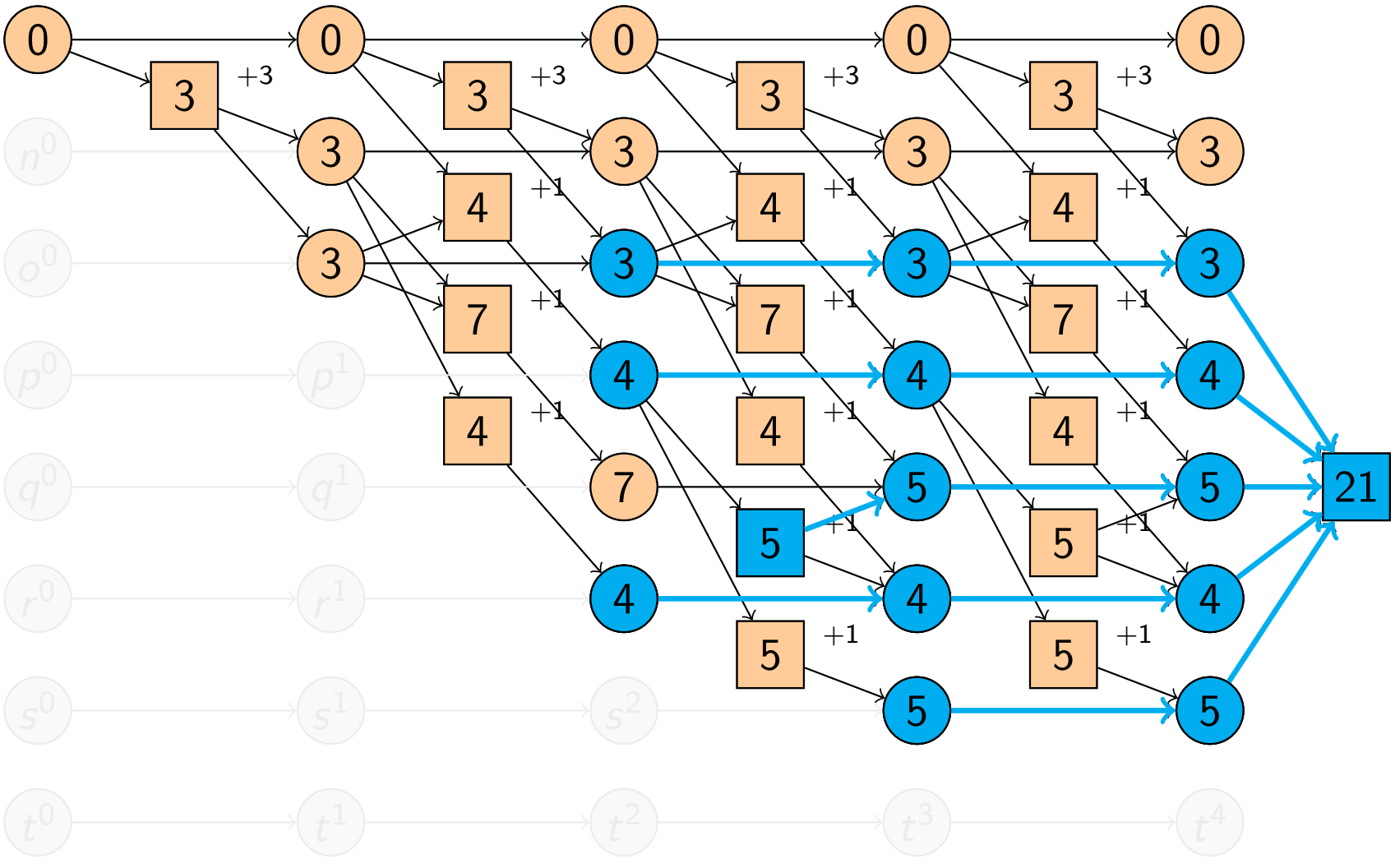
Illustrative Example: h^{FF}



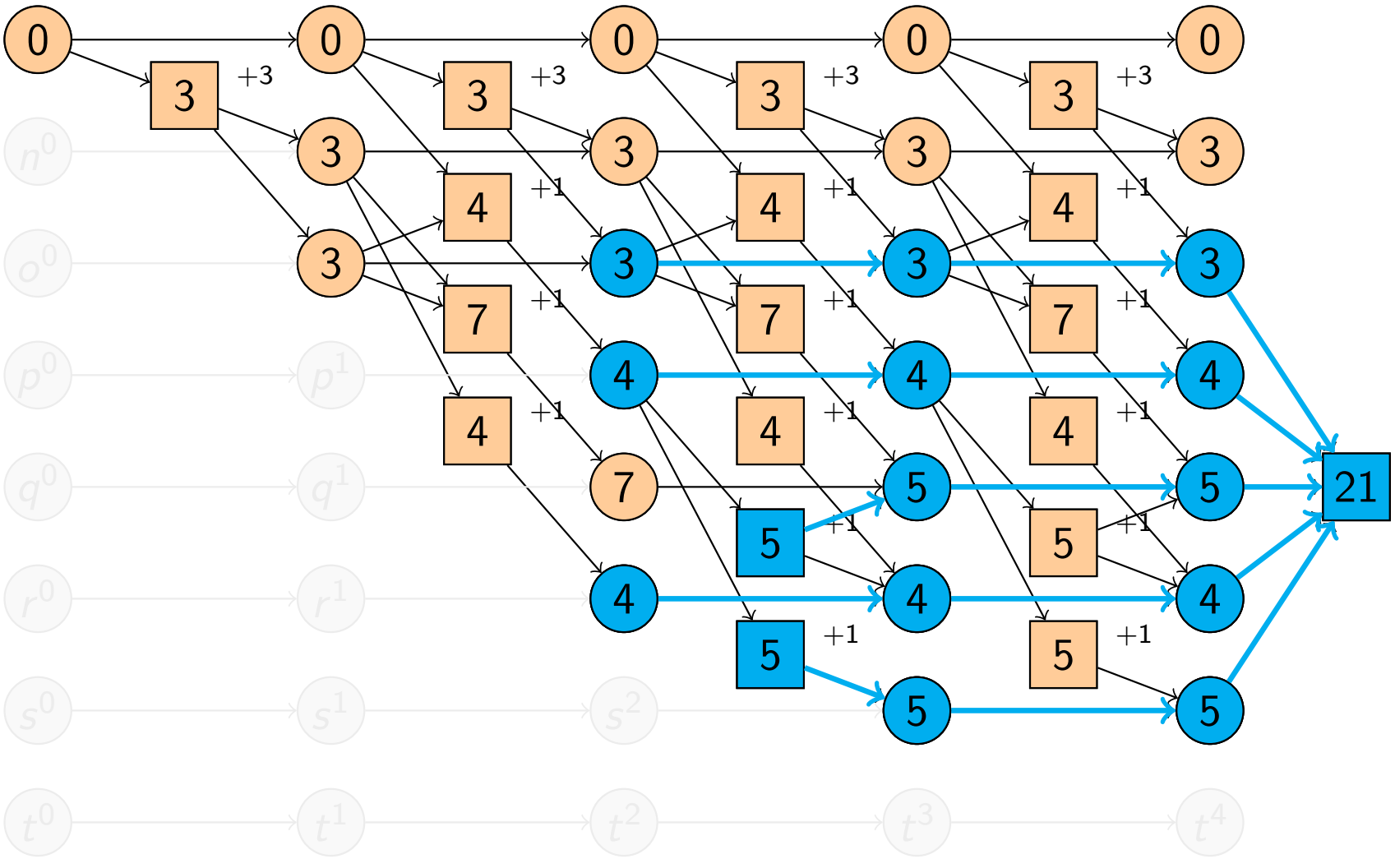
Illustrative Example: h^{FF}



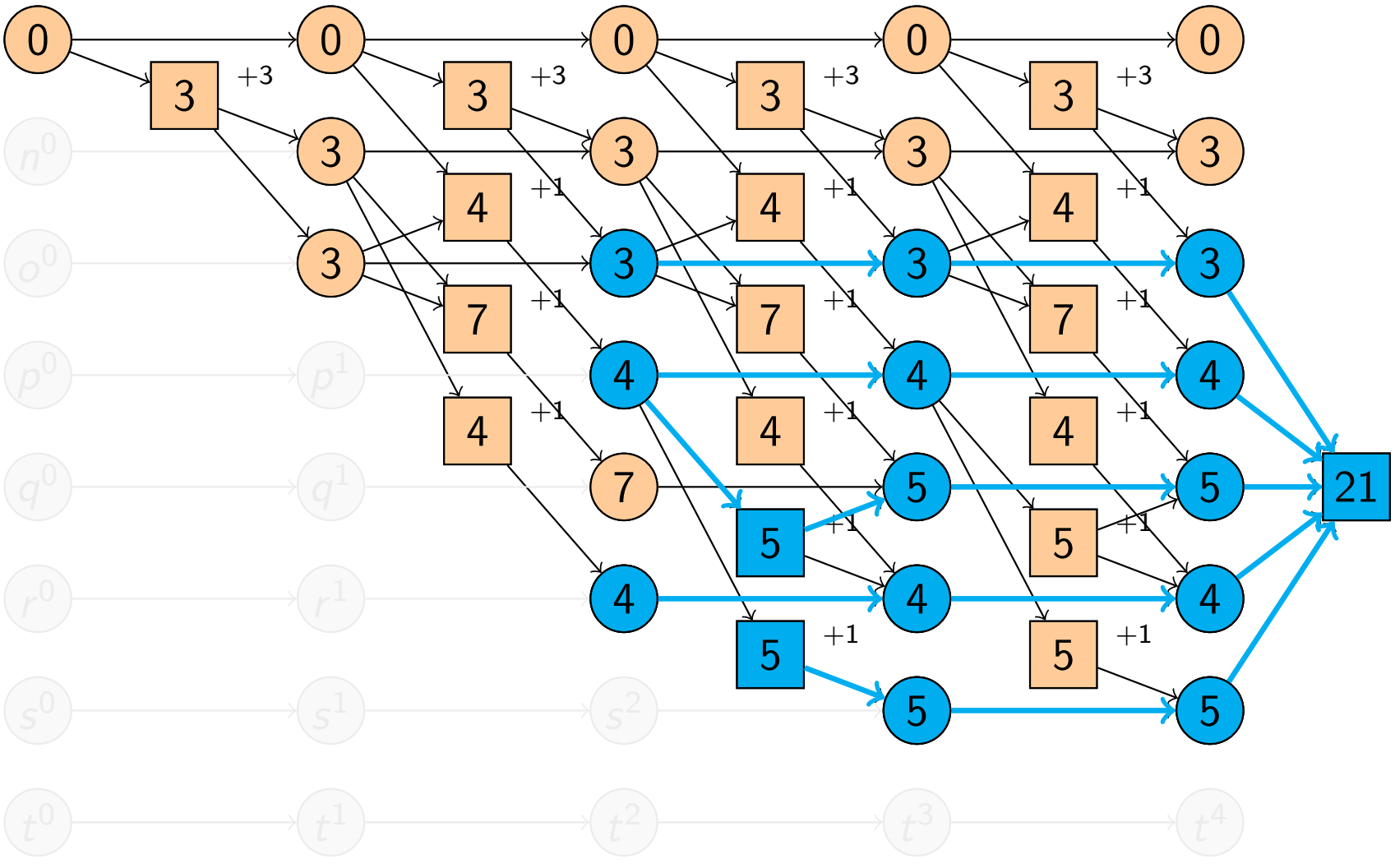
Illustrative Example: h^{FF}



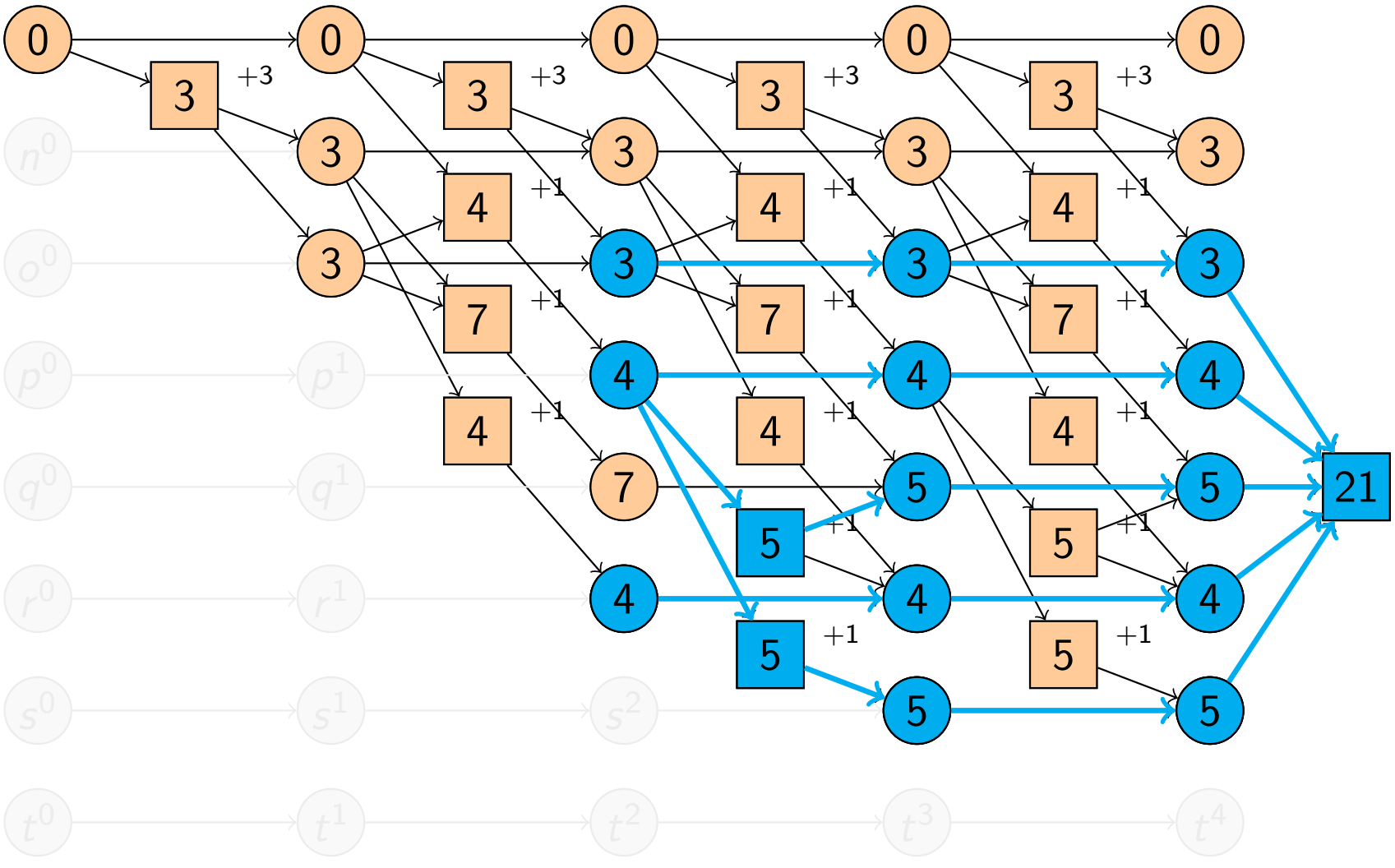
Illustrative Example: h^{FF}



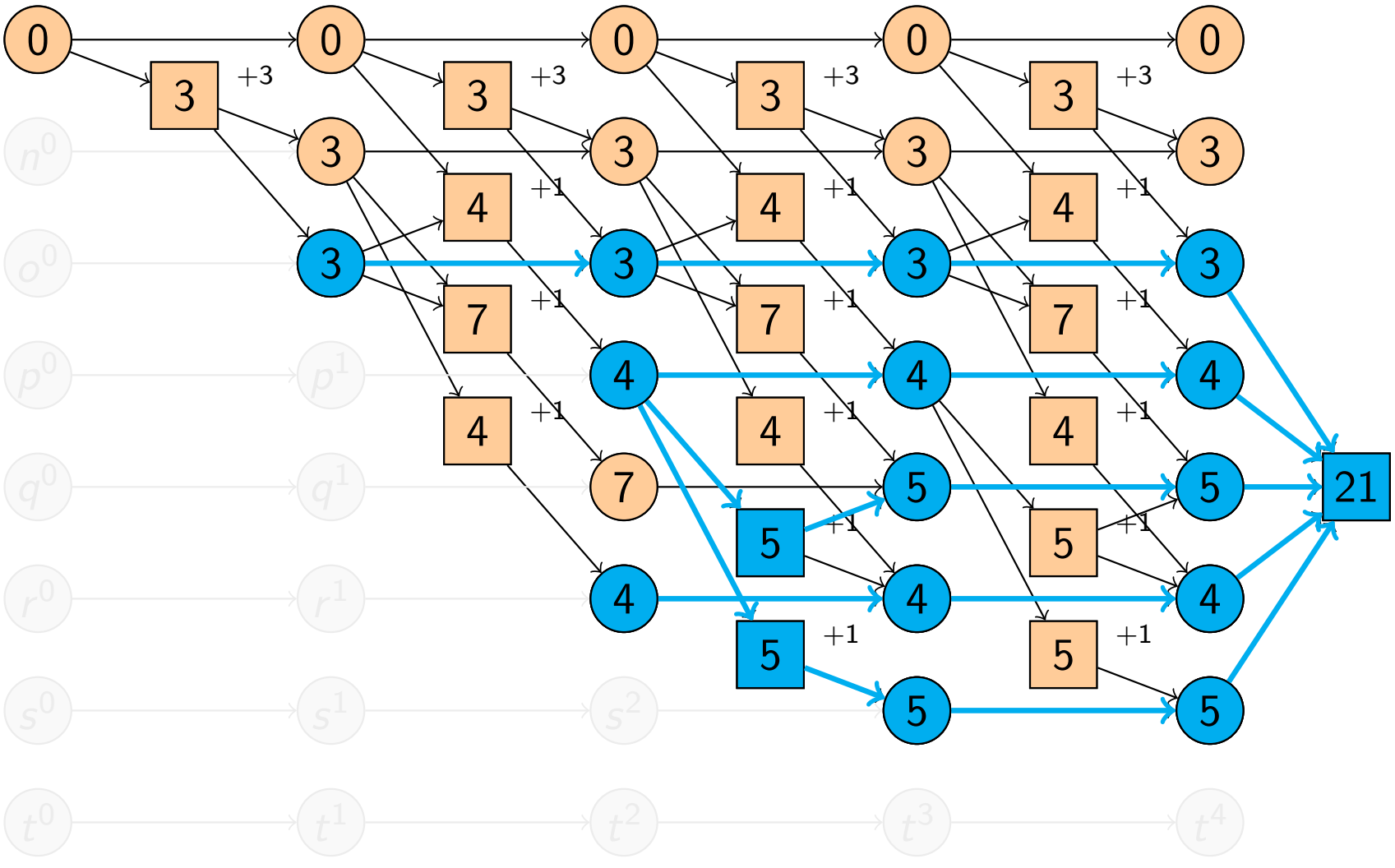
Illustrative Example: h^{FF}



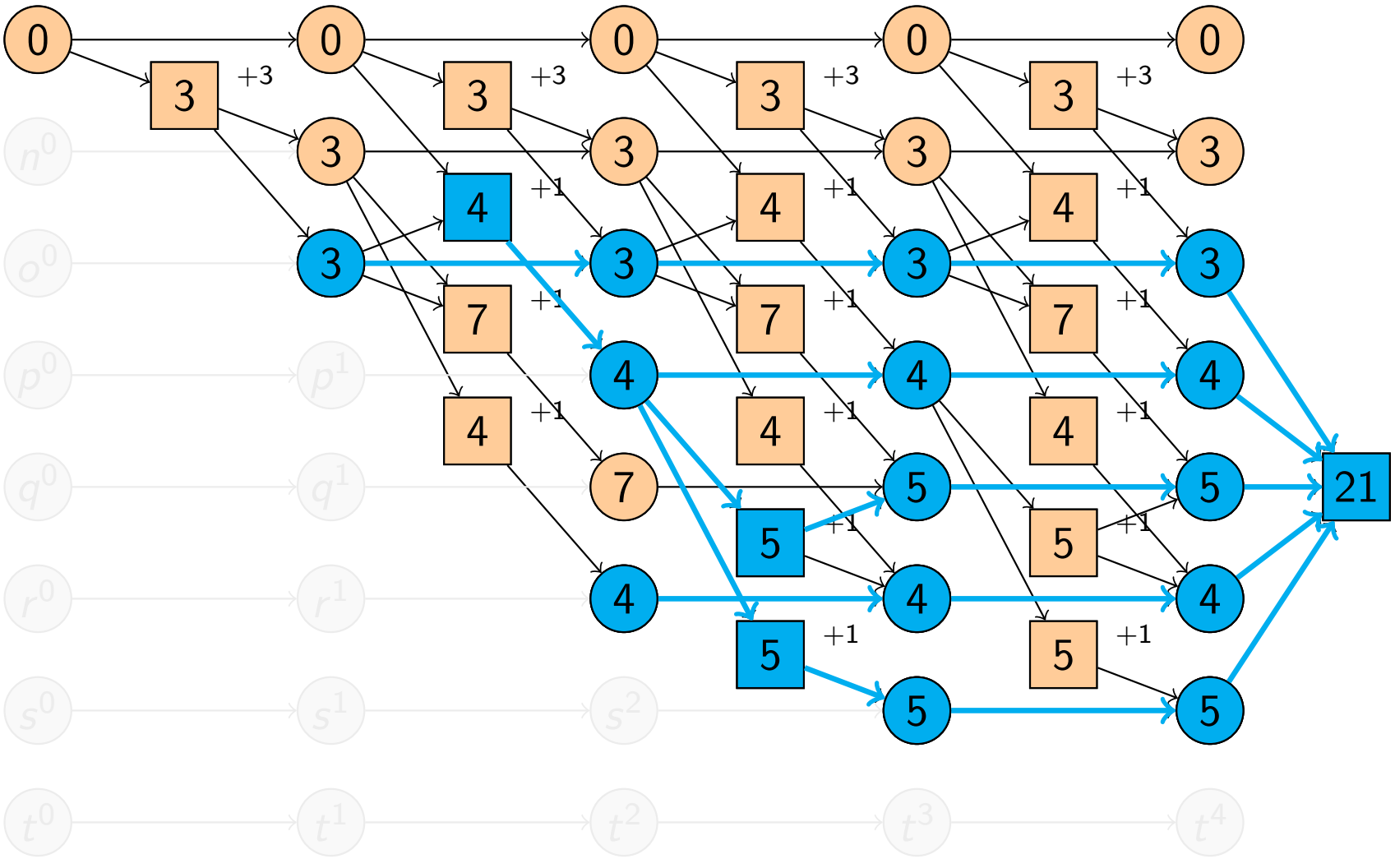
Illustrative Example: h^{FF}



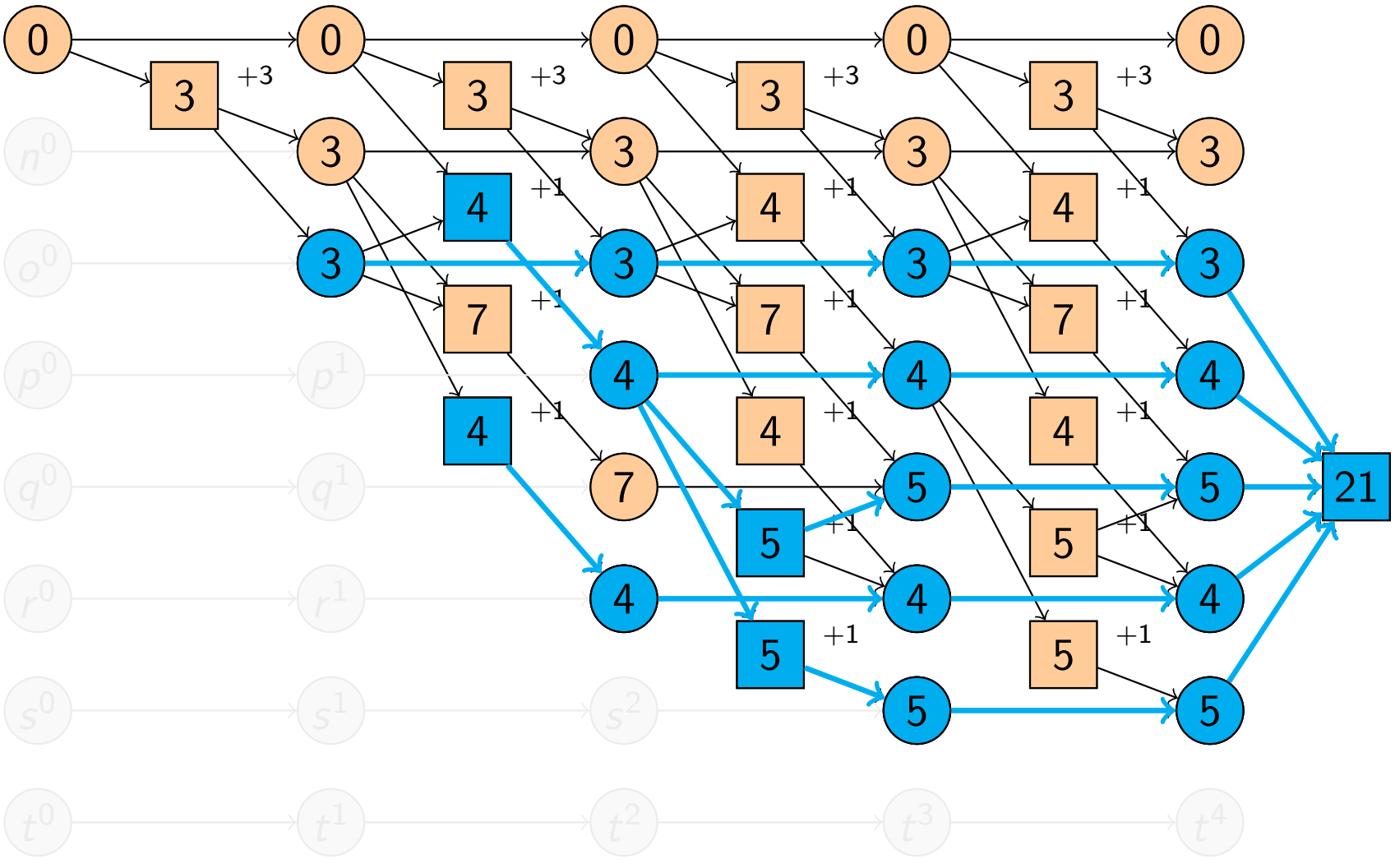
Illustrative Example: h^{FF}



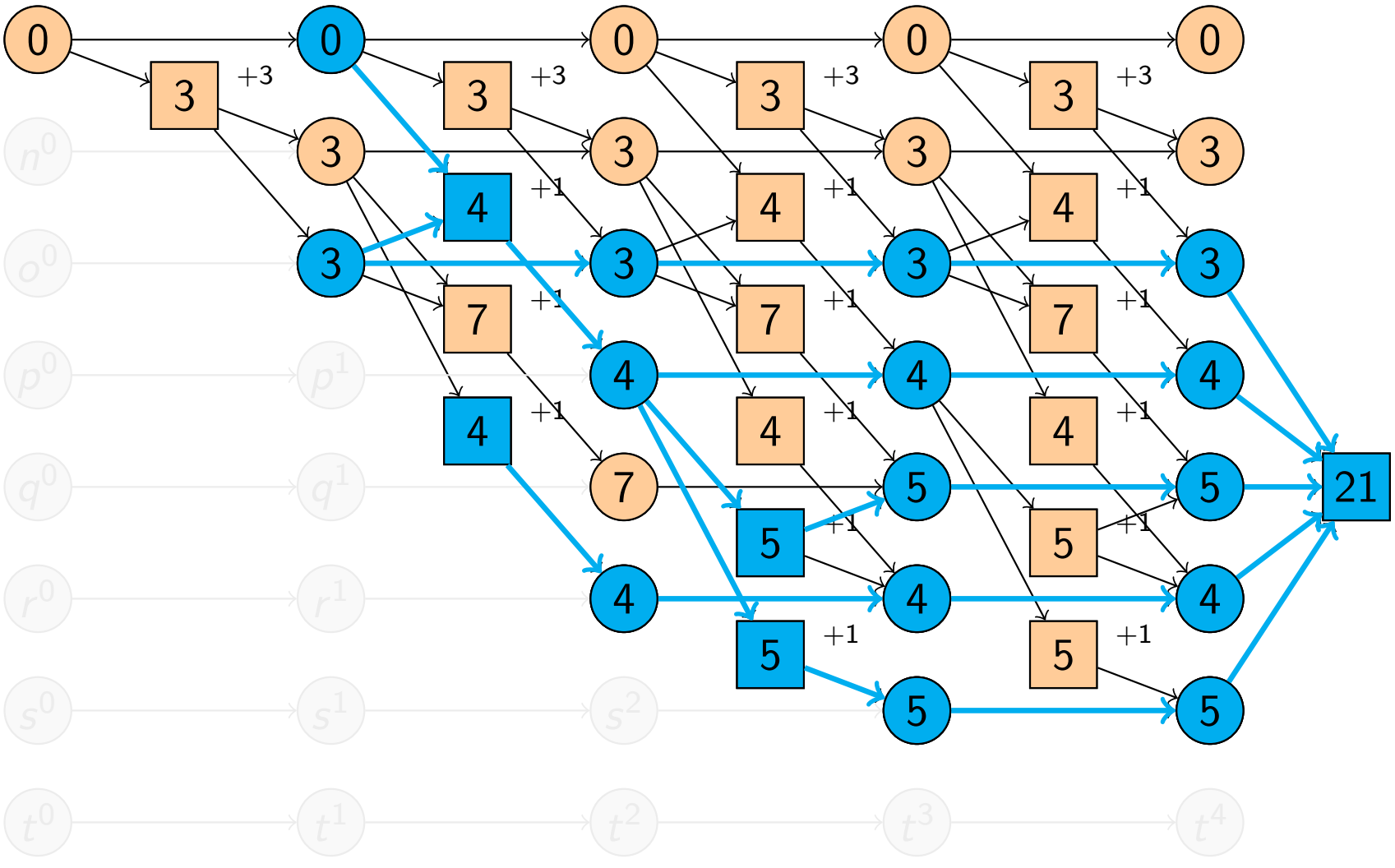
Illustrative Example: h^{FF}



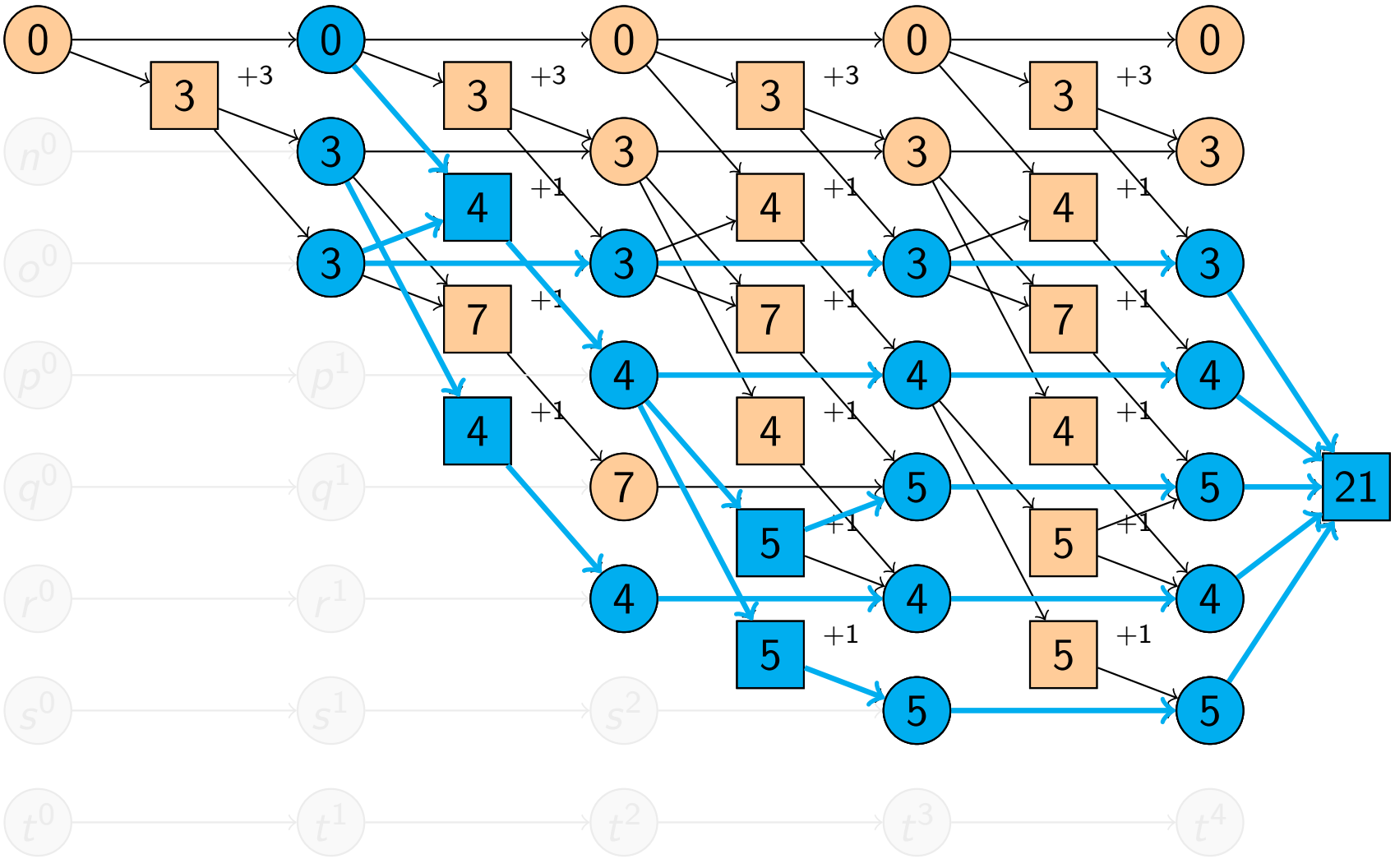
Illustrative Example: h^{FF}



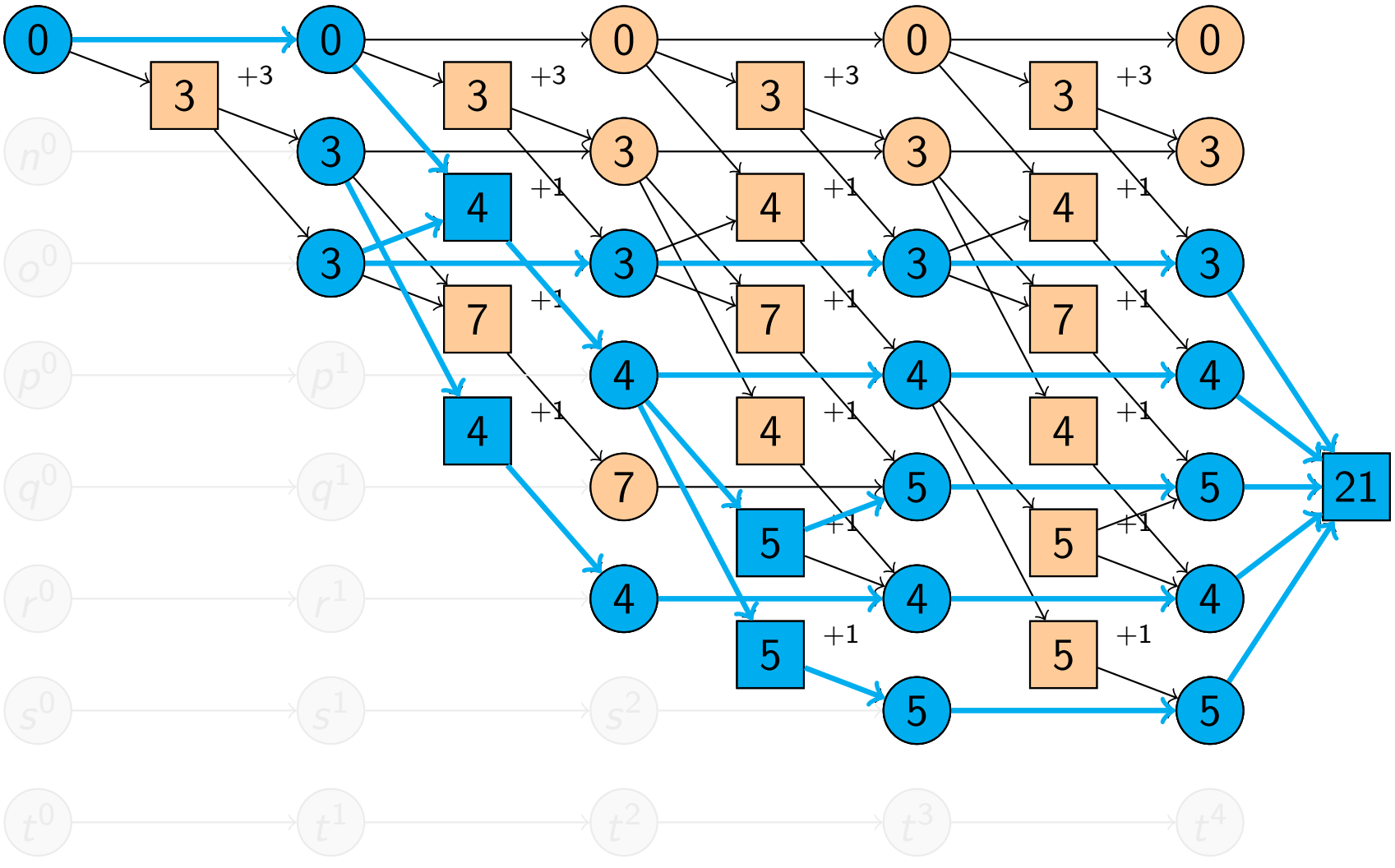
Illustrative Example: h^{FF}



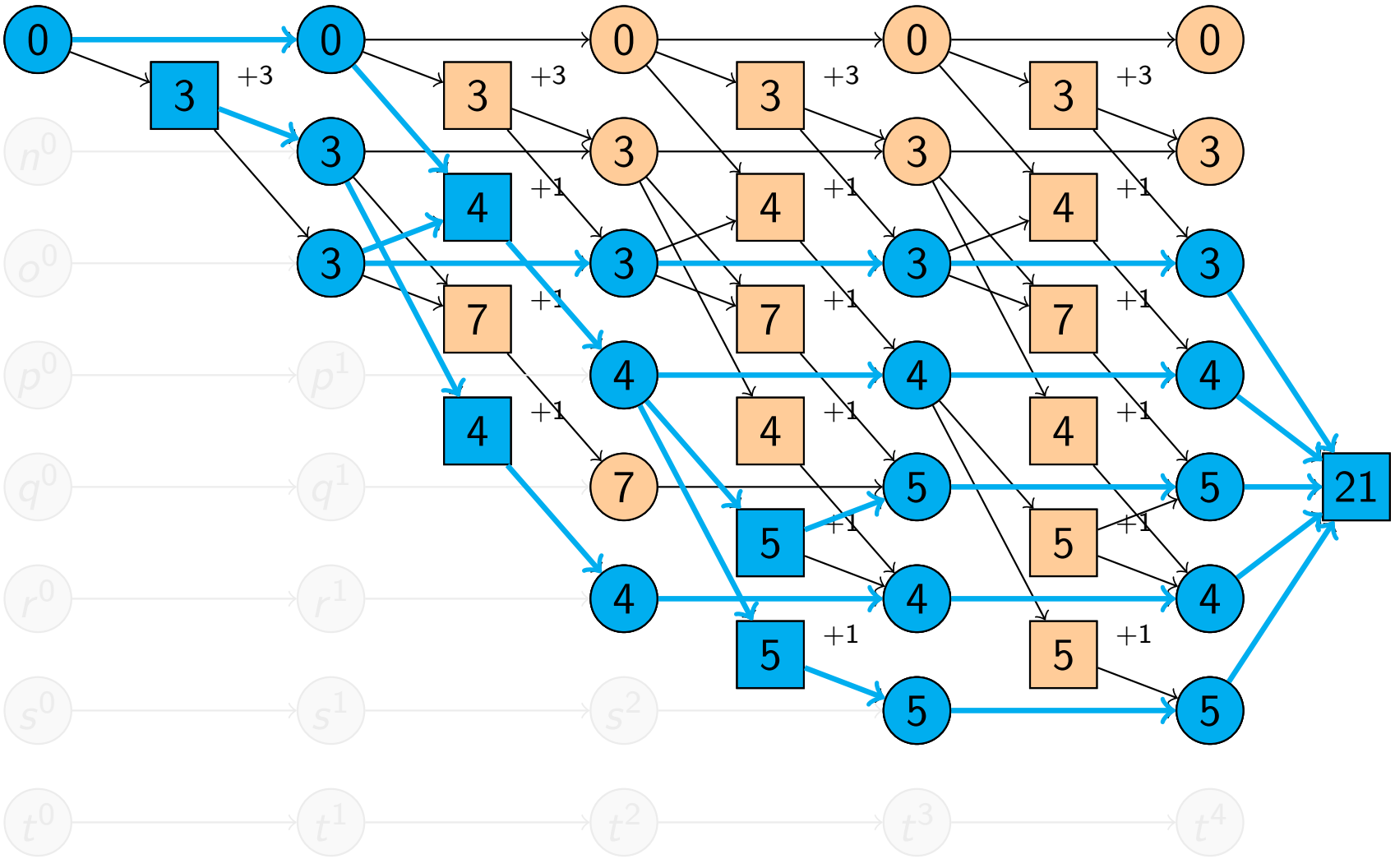
Illustrative Example: h^{FF}



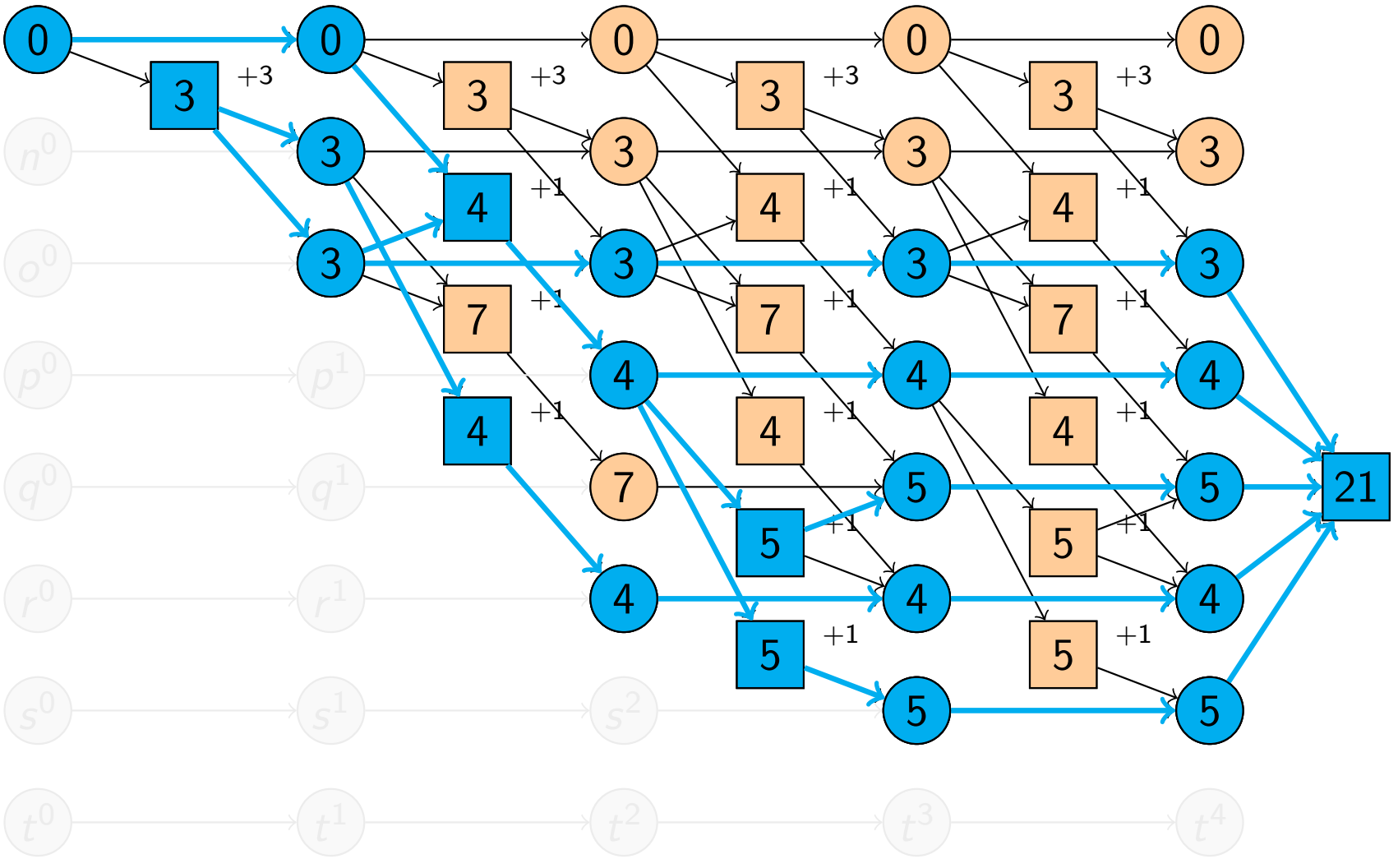
Illustrative Example: h^{FF}



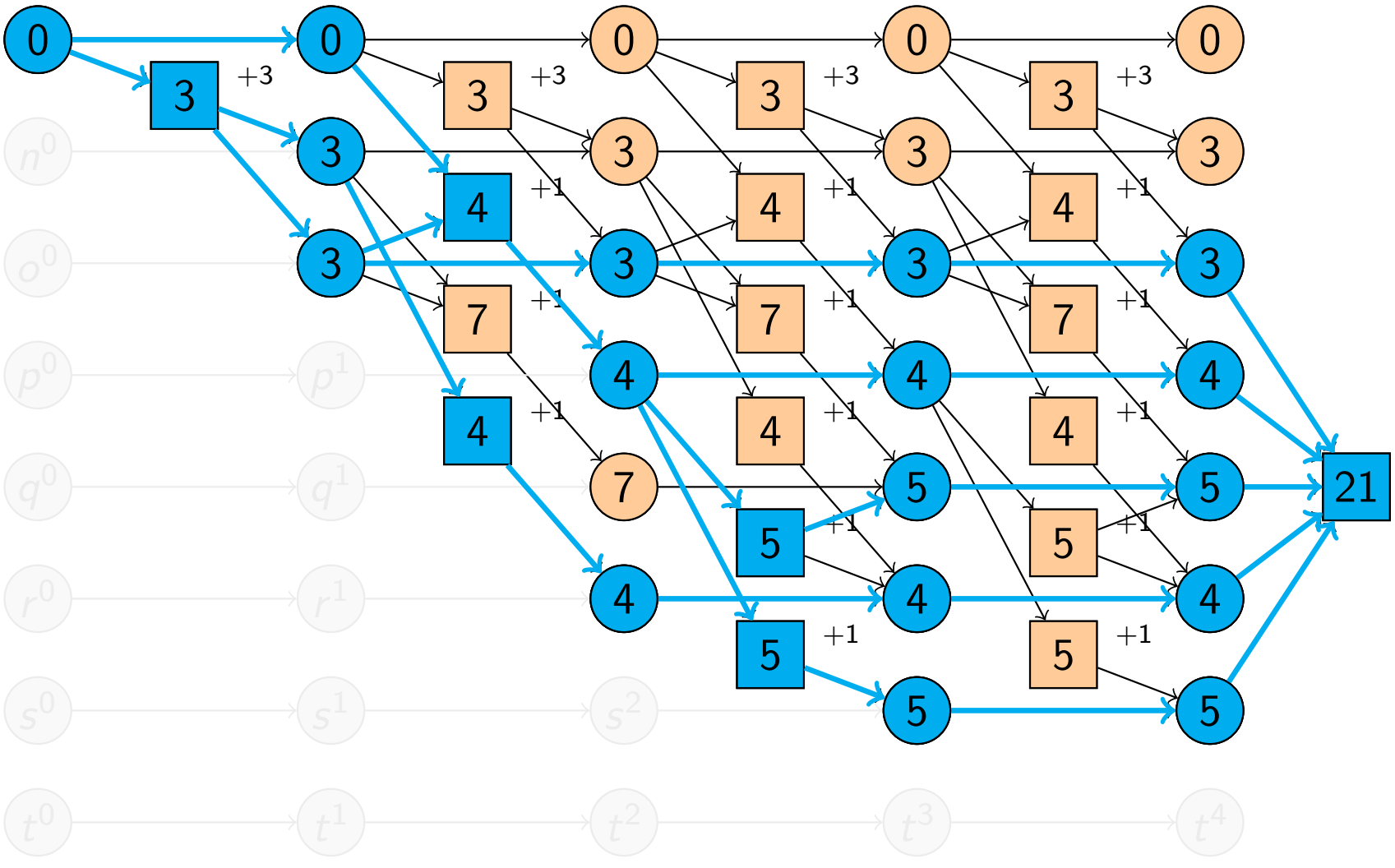
Illustrative Example: h^{FF}



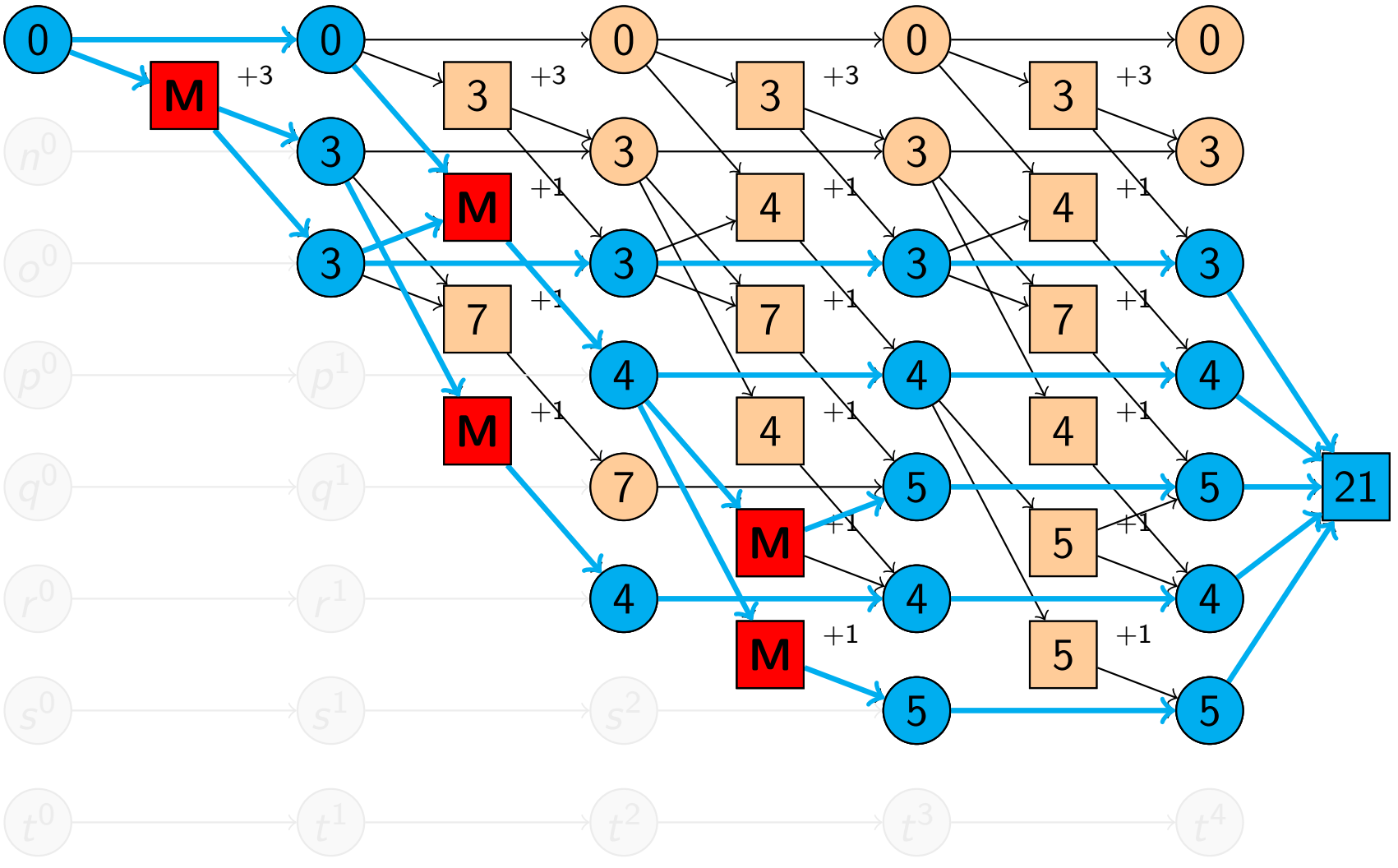
Illustrative Example: h^{FF}



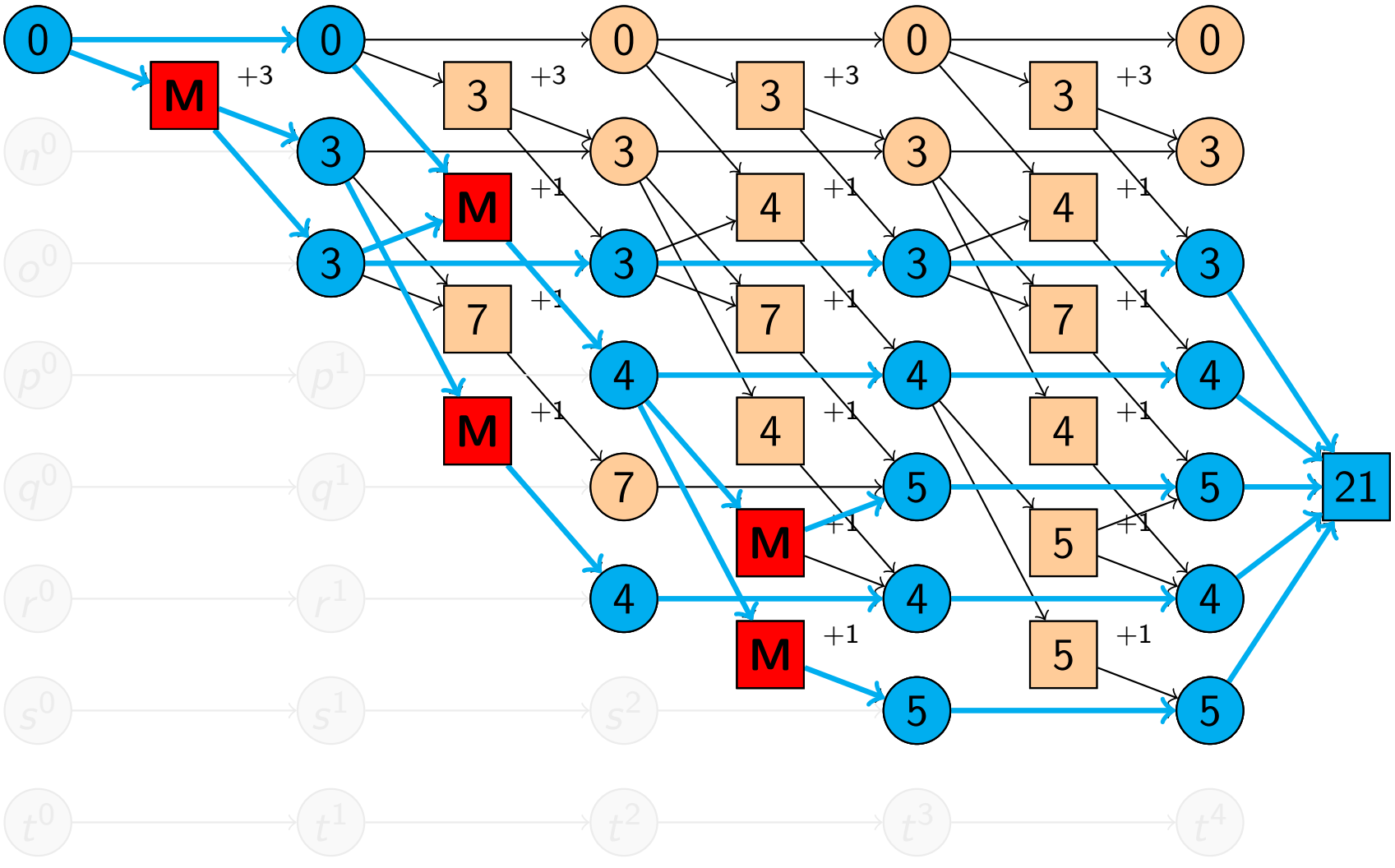
Illustrative Example: h^{FF}



Illustrative Example: h^{FF}



Illustrative Example: h^{FF}



$$h^{\mathrm{FF}}(\{m\}) = 3 + 1 + 1 + 1 + 1 = 7$$

FF Heuristic: Remarks

- Like h^{add}, h^{FF} is safe and goal-aware, but neither admissible nor consistent.
- approximation of h^+ which is always at least as good as h^{add}
- usually significantly better
- can be computed in almost linear time ($O(n \log n)$) in the size of the description of the planning task

FF Heuristic: Remarks

- Like h^{add}, h^{FF} is safe and goal-aware, but neither admissible nor consistent.
- approximation of h^+ which is always at least as good as h^{add}
- usually significantly better
- can be computed in almost linear time ($O(n \log n)$) in the size of the description of the planning task
- computation of heuristic value depends on tie-breaking of marking rules (h^{FF} not well-defined)
- one of the most successful planning heuristics

Comparison of Relaxation Heuristics

Relationships of Relaxation Heuristics

Let s be a state in the STRIPS planning task $\langle V, I, G, A \rangle$.

Then

- $h^{\max}(s) \leq h^{+}(s) \leq h^{*}(s)$
- $h^{\max}(s) \leq h^{+}(s) \leq h^{\mathrm{FF}}(s) \leq h^{\mathrm{add}}(s)$
- h^{*} and h^{FF} are incomparable
- h^{*} and h^{add} are incomparable

further remarks:

- For non-admissible heuristics, it is generally neither good nor bad to compute higher values than another heuristic.
- For relaxation heuristics, the objective is to approximate h^{+} as closely as possible.

Summary

Summary

- Many delete relaxation heuristics can be viewed as computations on relaxed planning graphs (RPGs).
- examples: $h^{\max}$, h^{add}, h^{FF}
- $h^{\max}$ and h^{add} propagate numeric values in the RPGs
  - difference: $h^{\max}$ computes the maximum of predecessor costs for action and goal vertices; h^{add} computes the sum
- h^{FF} marks vertices and sums the costs of marked action vertices.
- generally: $h^{\max}(s) \leq h^{+}(s) \leq h^{\text{FF}}(s) \leq h^{\text{add}}(s)$

Foundations of Artificial Intelligence

F5. Automated Planning: Abstraction

Malte Helmert

University of Basel

May 7, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

Planning Heuristics

We consider two basic ideas for general heuristics:

- Delete Relaxation
- Abstraction $\rightsquigarrow$ this chapter

Planning Heuristics

We consider two basic ideas for general heuristics:

- Delete Relaxation
- Abstraction $\rightsquigarrow$ this chapter

Abstraction: Idea

Estimate solution costs by considering a smaller planning task.

SAS$^+$

SAS$^+$ Encoding

- in this chapter: SAS$^+$ encoding
  instead of STRIPS (see Chapter F2)
- difference: state variables v not binary,
  but with finite domain $\mathrm{dom}(v)$
- accordingly, preconditions, effects, goals
  specified as partial assignments
- everything else equal to STRIPS

(In practice, planning systems convert automatically between STRIPS and SAS$^+$.)

SAS$^+$ Planning Task

Definition (SAS$^+$ planning task)

A SAS$^+$ planning task is a 5-tuple $\Pi = \langle V, \mathrm{dom}, I, G, A \rangle$ with the following components:

- V: finite set of state variables
- dom: domain; $\mathrm{dom}(v)$ finite and non-empty for all $v \in V$
  - states: total assignments for V according to dom
- I: the initial state (state = total assignment)
- G: goals (partial assignment)
- A: finite set of actions a with
  - $pre(a)$: its preconditions (partial assignment)
  - $eff(a)$: its effects (partial assignment)
  - $cost(a) \in \mathbb{N}_0$: its cost

German: SAS$^+$-Planungsaufgabe

State Space of SAS$^+$ Planning Task

Definition (state space induced by SAS$^+$ planning task)

Let $\Pi = \langle V, \mathrm{dom}, I, G, A \rangle$ be a SAS$^+$ planning task.
Then Π induces the state space $\mathcal{S}(\Pi) = \langle S, A, \mathit{cost}, T, s_\mathsf{I}, S_\mathsf{G} \rangle$:

- set of states: total assignments of V according to dom
- actions: actions A defined as in Π
- action costs: cost as defined in Π
- transitions: $s \xrightarrow{a} s'$ for states s, s' and action a iff
  - $\mathit{pre}(a)$ agrees with s (precondition satisfied)
  - s' agrees with $\mathit{eff}(a)$ for all variables mentioned in eff; agrees with s for all other variables (effects are applied)
- initial state: $s_\mathsf{I} = I$
- goal states: $s \in S_\mathsf{G}$ for state s iff G agrees with s

German: durch SAS$^+$-Planungsaufgabe induzierter Zustandsraum

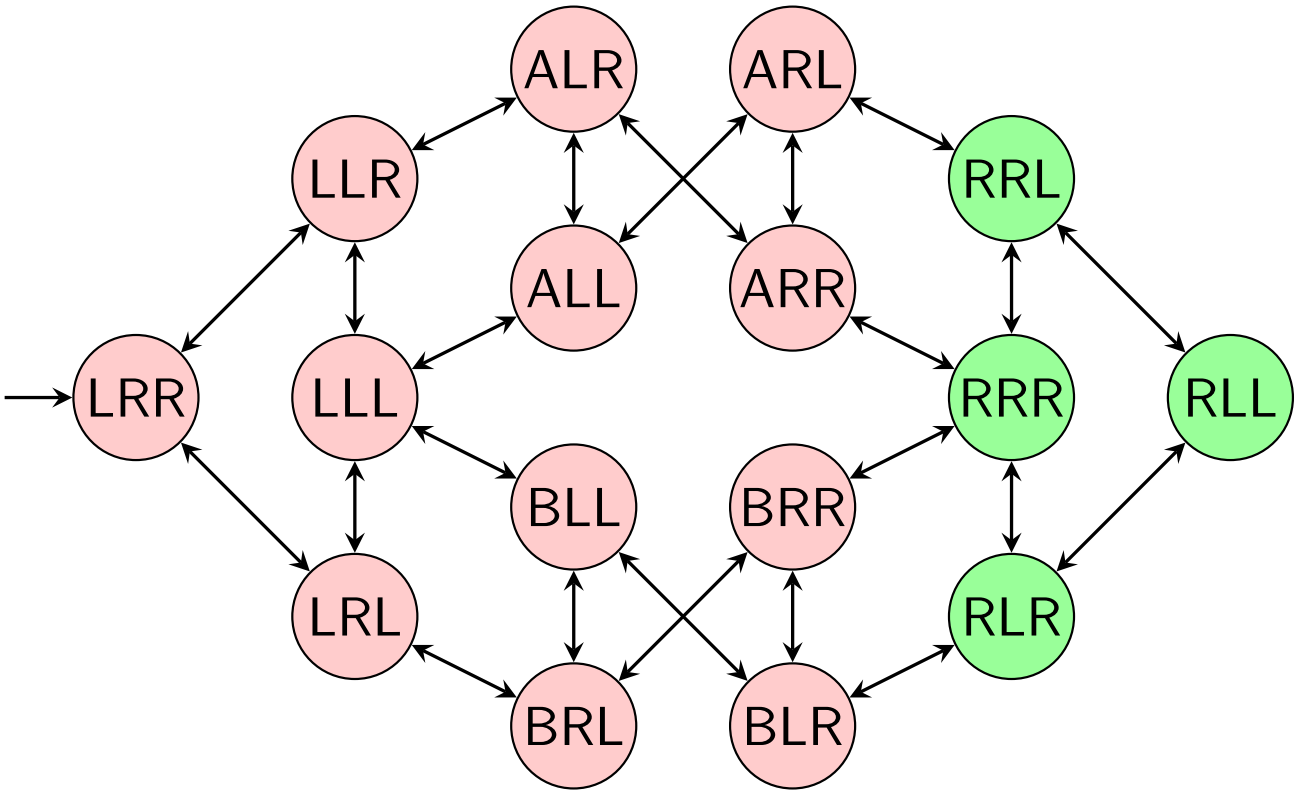
Example: Logistics Task with One Package, Two Trucks

Example (one package, two trucks)

Consider the SAS$^+$ planning task $\langle V, \mathrm{dom}, I, G, A \rangle$ with:

- $V = \{p, t_\mathrm{A}, t_\mathrm{B}\}$
- $\mathrm{dom}(p) = \{\mathrm{L}, \mathrm{R}, \mathrm{A}, \mathrm{B}\}$ and $\mathrm{dom}(t_\mathrm{A}) = \mathrm{dom}(t_\mathrm{B}) = \{\mathrm{L}, \mathrm{R}\}$
- $I = \{p \mapsto \mathrm{L}, t_\mathrm{A} \mapsto \mathrm{R}, t_\mathrm{B} \mapsto \mathrm{R}\}$
- $G = \{p \mapsto \mathrm{R}\}$
- $A = \{\mathit{load}_{i,j} \mid i \in \{\mathrm{A}, \mathrm{B}\}, j \in \{\mathrm{L}, \mathrm{R}\}\}$
  $\cup \{\mathit{unload}_{i,j} \mid i \in \{\mathrm{A}, \mathrm{B}\}, j \in \{\mathrm{L}, \mathrm{R}\}\}$
  $\cup \{\mathit{move}_{i,j,j'} \mid i \in \{\mathrm{A}, \mathrm{B}\}, j, j' \in \{\mathrm{L}, \mathrm{R}\}, j \neq j'\}$ with:
  - $\mathit{load}_{i,j}$ has preconditions $\{t_i \mapsto j, p \mapsto j\}$, effects $\{p \mapsto i\}$
  - $\mathit{unload}_{i,j}$ has preconditions $\{t_i \mapsto j, p \mapsto i\}$, effects $\{p \mapsto j\}$
  - $\mathit{move}_{i,j,j'}$ has preconditions $\{t_i \mapsto j\}$, effects $\{t_i \mapsto j'\}$
  - All actions have cost 1.

State Space for Example Task



- state $\{p \mapsto i, t_A \mapsto j, t_B \mapsto k\}$ denoted as ijk
- annotations of edges not shown for simplicity
- for example, edge from LLL to ALL has annotation $load_{A,L}$

Abstractions

State Space Abstraction

State space abstractions drop distinctions between certain states, but preserve the state space behavior as well as possible.

- An abstraction of a state space $\mathcal{S}$ is defined by an abstraction function α that determines which states can be distinguished in the abstraction.
- Based on $\mathcal{S}$ and α, we compute the abstract state space $\mathcal{S}^\alpha$ which is "similar" to $\mathcal{S}$ but smaller.
- main idea: use optimal solution cost in $\mathcal{S}^\alpha$ as heuristic

German: Abstraktionsfunktion, abstrakter Zustandsraum

Induced Abstraction

Definition (induced abstraction)

Let $\mathcal{S} = \langle S, A, \mathit{cost}, T, s_\mathsf{I}, S_\mathsf{G} \rangle$ be a state space,
and let $\alpha : S \to S'$ be a surjective function.

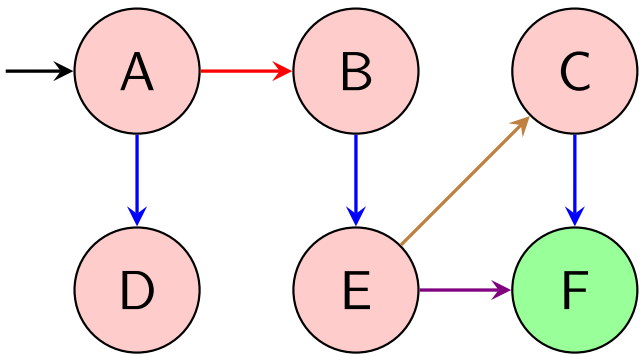
The abstraction of $\mathcal{S}$ induced by α, denoted as $\mathcal{S}^\alpha$,
is the state space $\mathcal{S}^\alpha = \langle S', A, \mathit{cost}, T', s'_\mathsf{I}, S'_\mathsf{G} \rangle$ with:

- $T' = \{\langle \alpha(s), a, \alpha(t) \rangle \mid \langle s, a, t \rangle \in T\}$
- $s'_\mathsf{I} = \alpha(s_\mathsf{I})$
- $S'_\mathsf{G} = \{\alpha(s) \mid s \in S_\mathsf{G}\}$

German: induzierte Abstraktion

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



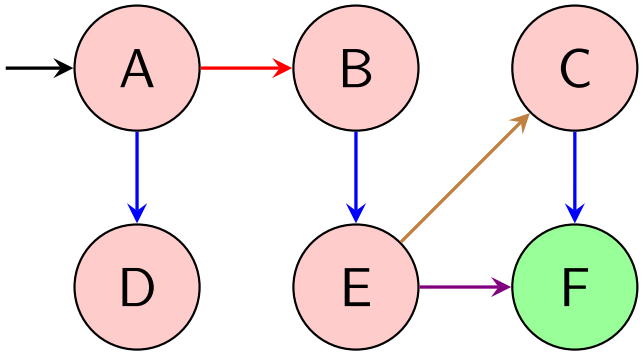
abstract state space with states $S^\alpha = \{W, X, Y, Z\}$

abstraction function $\alpha : S \to S^\alpha$

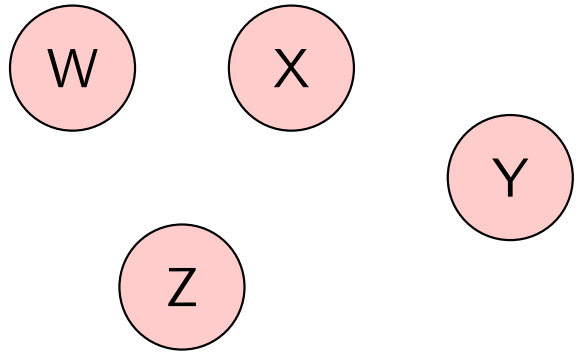
$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$

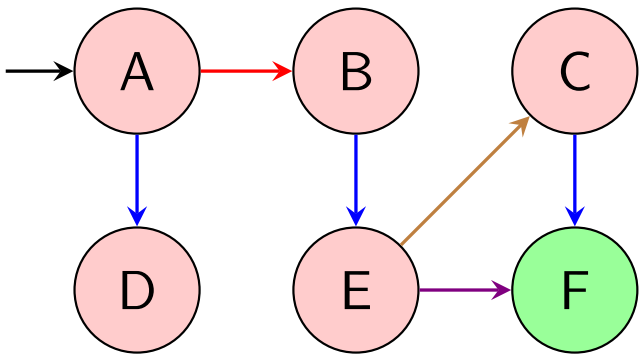


abstraction function $\alpha : S \to S^\alpha$

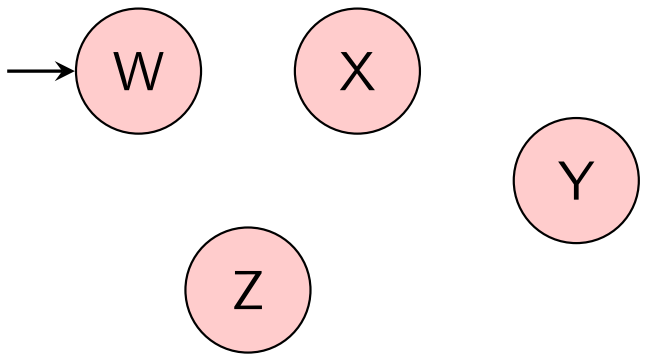
$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



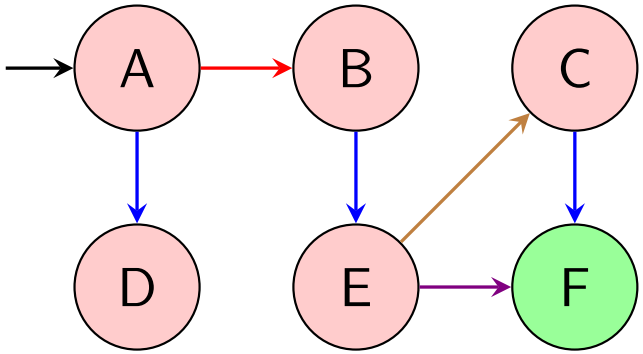
abstraction function $\alpha : S \to S^\alpha$

$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$

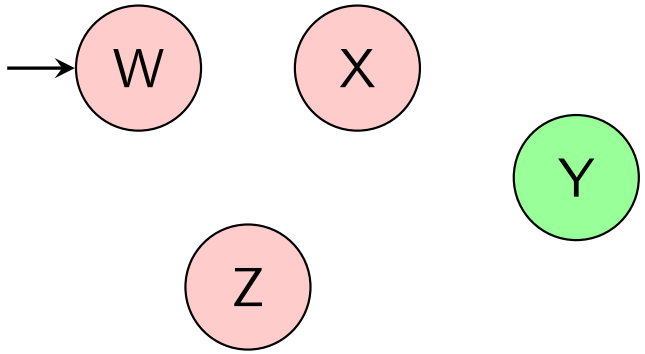
$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$

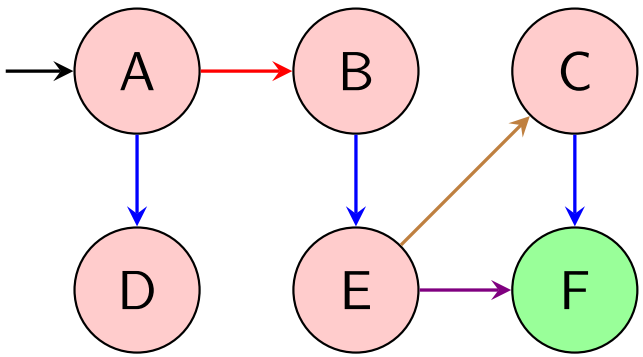


abstraction function $\alpha : S \to S^\alpha$

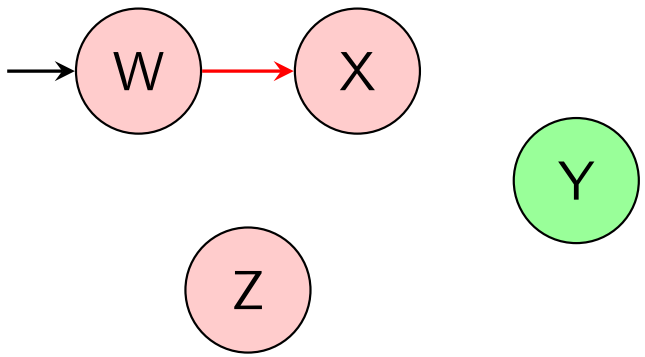
$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



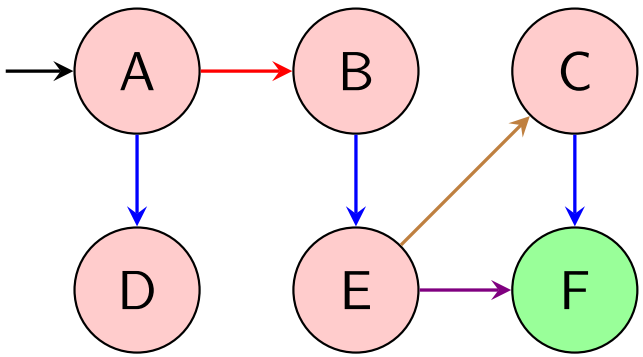
abstraction function $\alpha : S \to S^\alpha$

$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$

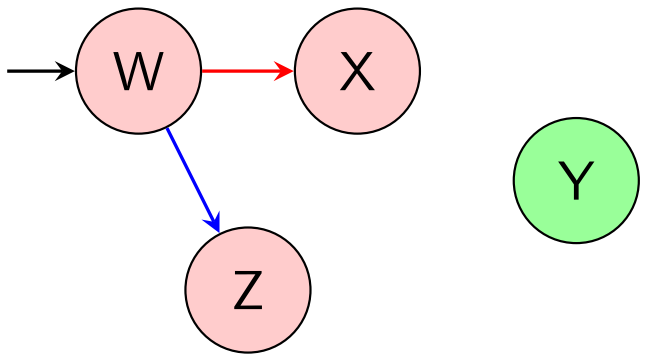
$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



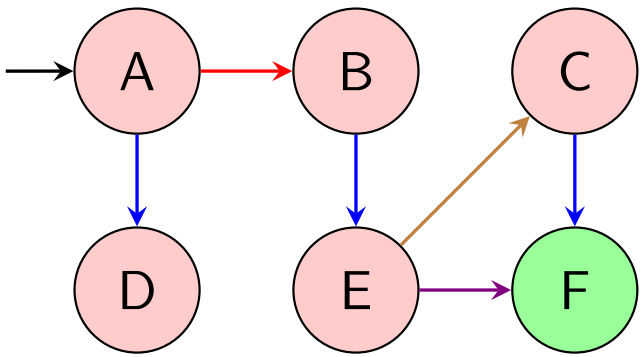
abstraction function $\alpha : S \to S^\alpha$

$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$

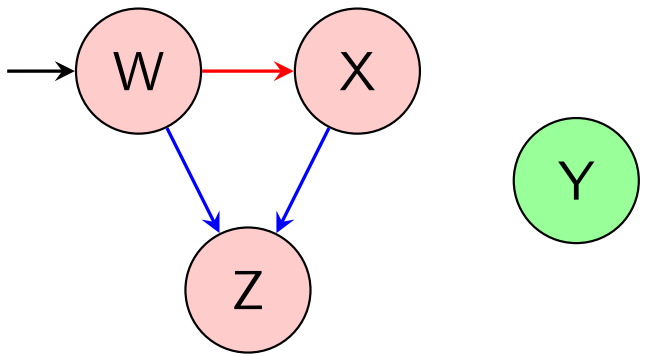
$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$

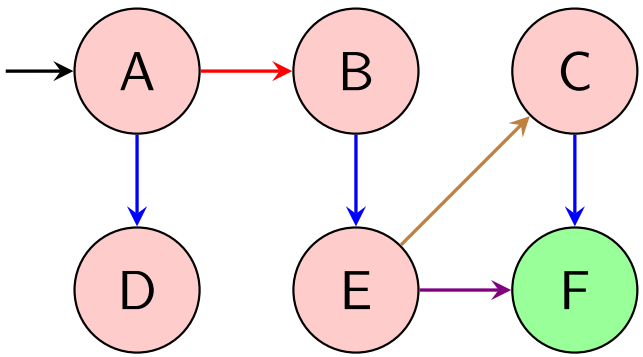


abstraction function $\alpha : S \to S^\alpha$

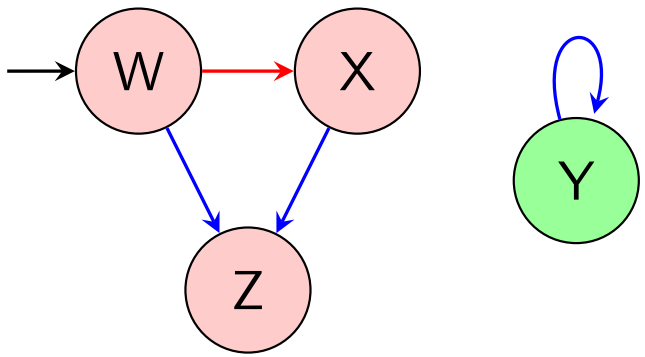
$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$

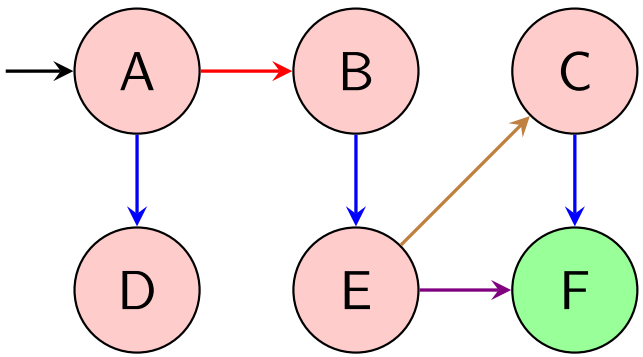


abstraction function $\alpha : S \to S^\alpha$

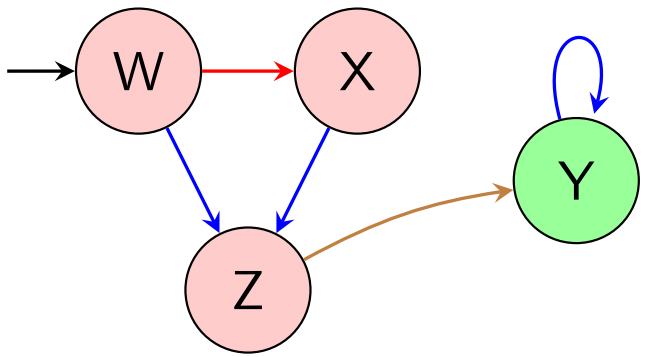
$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



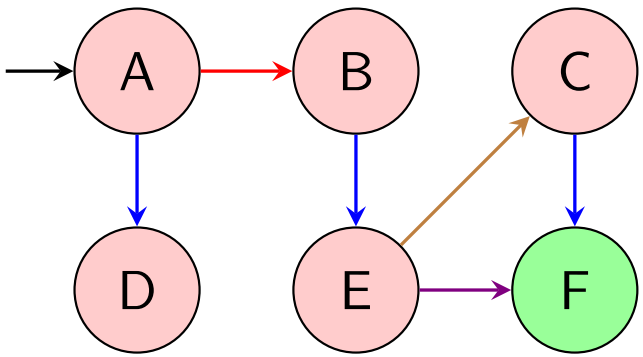
abstraction function $\alpha : S \to S^\alpha$

$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$

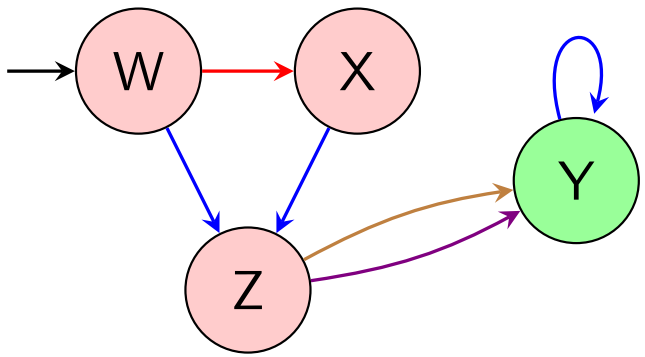
$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



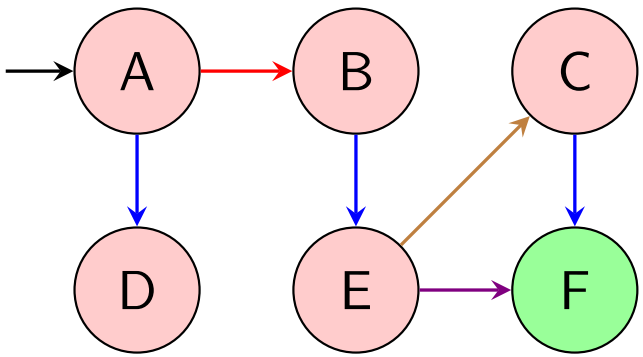
abstraction function $\alpha : S \to S^\alpha$

$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$

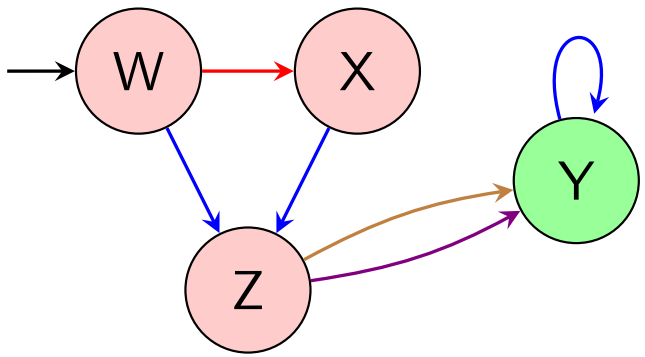
$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$

Abstraction: Example

concrete state space with states $S = \{A, B, C, D, E, F\}$



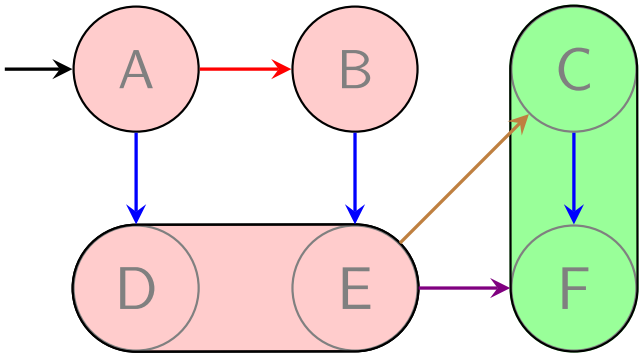
abstract state space with states $S^\alpha = \{W, X, Y, Z\}$



abstraction function $\alpha : S \to S^\alpha$

$$\alpha(A) = W \quad \alpha(B) = X \quad \alpha(C) = Y$$
$$\alpha(D) = Z \quad \alpha(E) = Z \quad \alpha(F) = Y$$

intuition: grouping states



Summary

Summary

- basic idea of abstractions: simplify state space by considering a smaller version
- formally: abstraction function α maps states to abstract states and thus defines which states can be distinguished by the resulting abstraction
- induces abstract state space

Foundations of Artificial Intelligence

F6. Automated Planning: Abstraction Heuristics

Malte Helmert

University of Basel

May 7, 2025

Automated Planning: Overview

Chapter overview: automated planning

- F1. Introduction
- F2. Planning Formalisms
- F3. Delete Relaxation
- F4. Delete Relaxation Heuristics
- F5. Abstraction
- F6. Abstraction Heuristics

Abstraction Heuristics

Abstraction Heuristic

Given an abstraction function α for a state space $\mathcal{S}$,
use abstract solution cost (solution cost of $\alpha(s)$ in $\mathcal{S}^\alpha$) as heuristic for concrete solution cost (solution cost of s in $\mathcal{S}$).

Definition (abstraction heuristic)

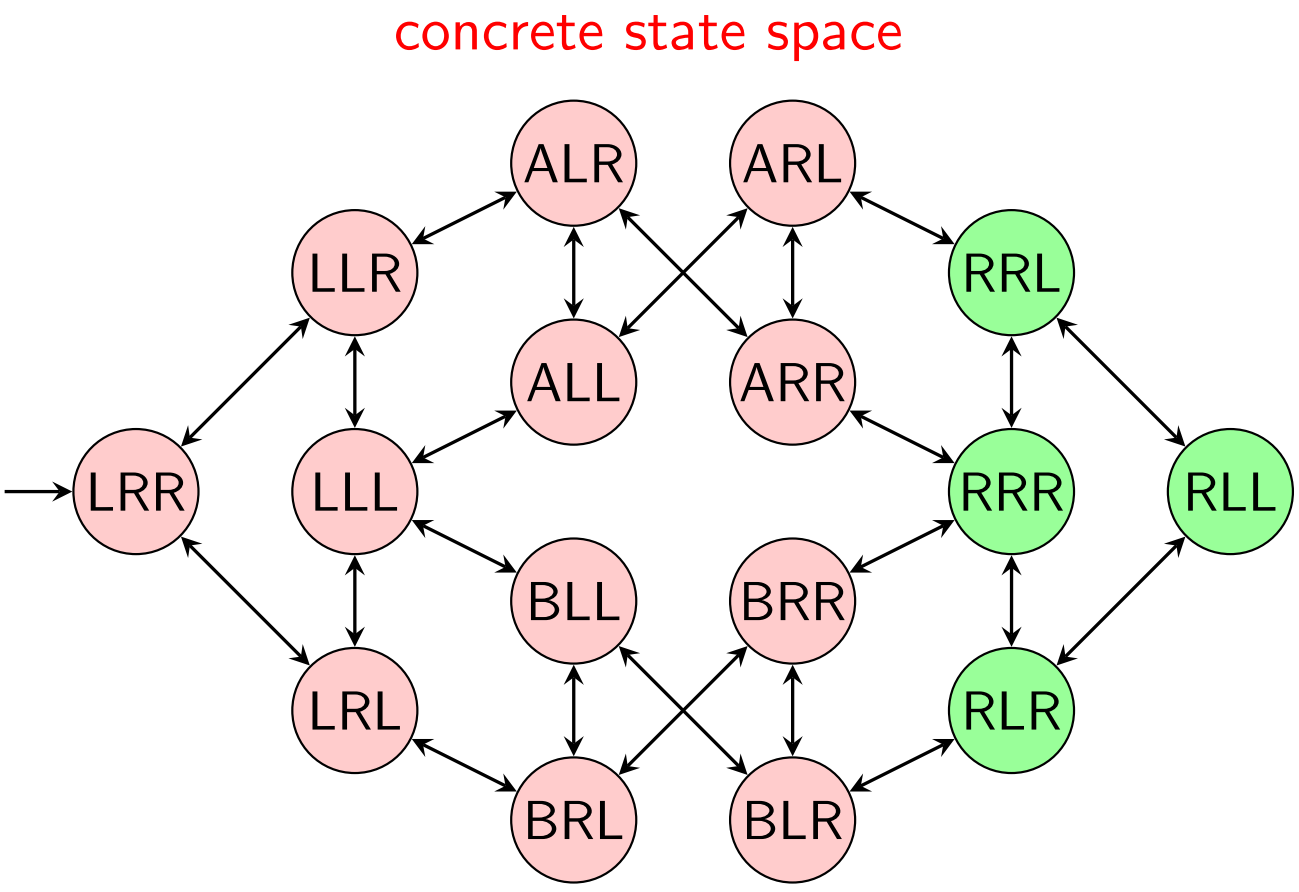
The abstraction heuristic for abstraction α maps each state s to its abstract solution cost

$$h^\alpha(s) = h^*_{\mathcal{S}^\alpha}(\alpha(s)),$$

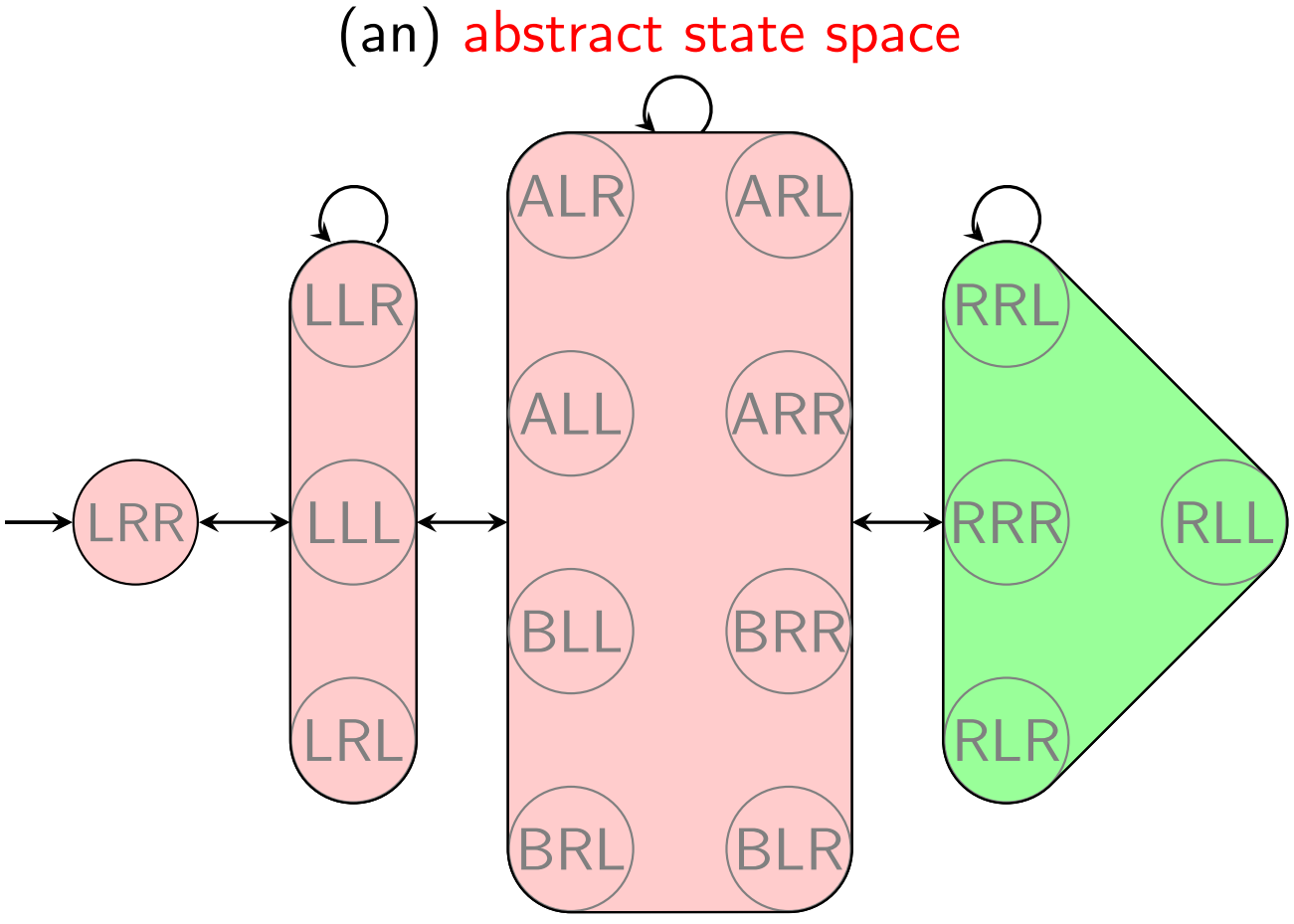
where $h^*_{\mathcal{S}^\alpha}$ is the perfect heuristic in $\mathcal{S}^\alpha$.

German: abstrakte/konkrete Zielabstände, Abstraktionsheuristik

Abstraction: Example

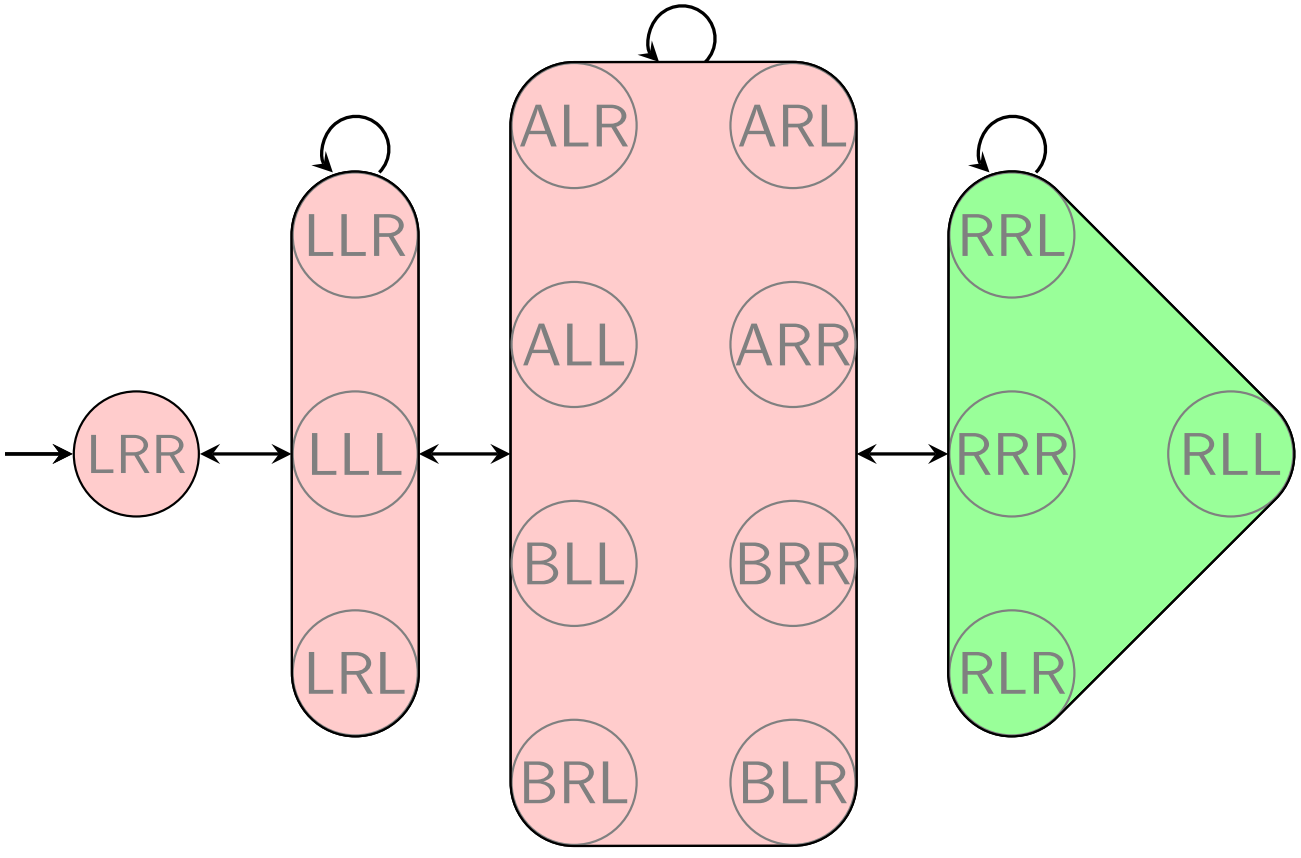


Abstraction: Example



Remark: Most arcs correspond to several (parallel) transitions with different labels.

Abstraction Heuristic: Example



$$h^{\alpha}(\{p \mapsto \mathrm{L}, t_{\mathrm{A}} \mapsto \mathrm{R}, t_{\mathrm{B}} \mapsto \mathrm{R}\}) = 3$$

Abstraction Heuristics: Discussion

- Every abstraction heuristic is admissible and consistent. (proof idea?)
- The choice of the abstraction function α is very important.
  - Every α yields an admissible and consistent heuristic.
  - But most α lead to poor heuristics.
- An effective α must yield an informative heuristic ...
- ... as well as being efficiently computable.
- How to find a suitable α?

Automatic Computation of Suitable Abstractions

Main Problem with Abstraction Heuristics

How to find a good abstraction?

Several successful methods:

- pattern databases (PDBs) $\rightsquigarrow$ this course
  (Culberson & Schaeffer, 1996)
- merge-and-shrink abstractions
  (Dräger, Finkbeiner & Podelski, 2006)
- Cartesian abstractions (Seipp & Helmert, 2013)
- domain abstractions (Kreft et al., 2023)

German: Pattern Databases, Merge-and-Shrink-Abstraktionen, Kartesische Abstraktionen, Domänenabstraktionen

Pattern Databases

Pattern Databases: Background

- The most common abstraction heuristics are pattern database heuristics.
- originally introduced for the 15-puzzle (Culberson & Schaeffer, 1996) and for Rubik's Cube (Korf, 1997)
- introduced for automated planning by Edelkamp (2001)
- for many search problems the best known heuristics
- many many research papers studying
  - theoretical properties
  - efficient implementation and application
  - pattern selection
  - ...

Pattern Databases: Projections

A PDB heuristic for a planning task is an abstraction heuristic where

- some aspects (= state variables) of the task are preserved with perfect precision while
- all other aspects are not preserved at all.

formalized as projections to a pattern $P \subseteq V$:

$$\pi_P(s) = \{v \mapsto s(v) \mid v \in P\}$$

example:

- $s = \{p \mapsto L, t_A \mapsto R, t_B \mapsto R\}$
- projection on $P = \{p\}$ (= ignore trucks):
  $\pi_P(s) = \{p \mapsto L\}$
- projection on $P = \{p, t_A\}$ (= ignore truck B):
  $\pi_P(s) = \{p \mapsto L, t_A \mapsto R\}$

German: Projektionen

Pattern Databases: Definition

Definition (pattern database heuristic)

Let P be a subset of the variables of a planning task.

The abstraction heuristic induced by the projection π_P on P is called pattern database heuristic (PDB heuristic) with pattern P.

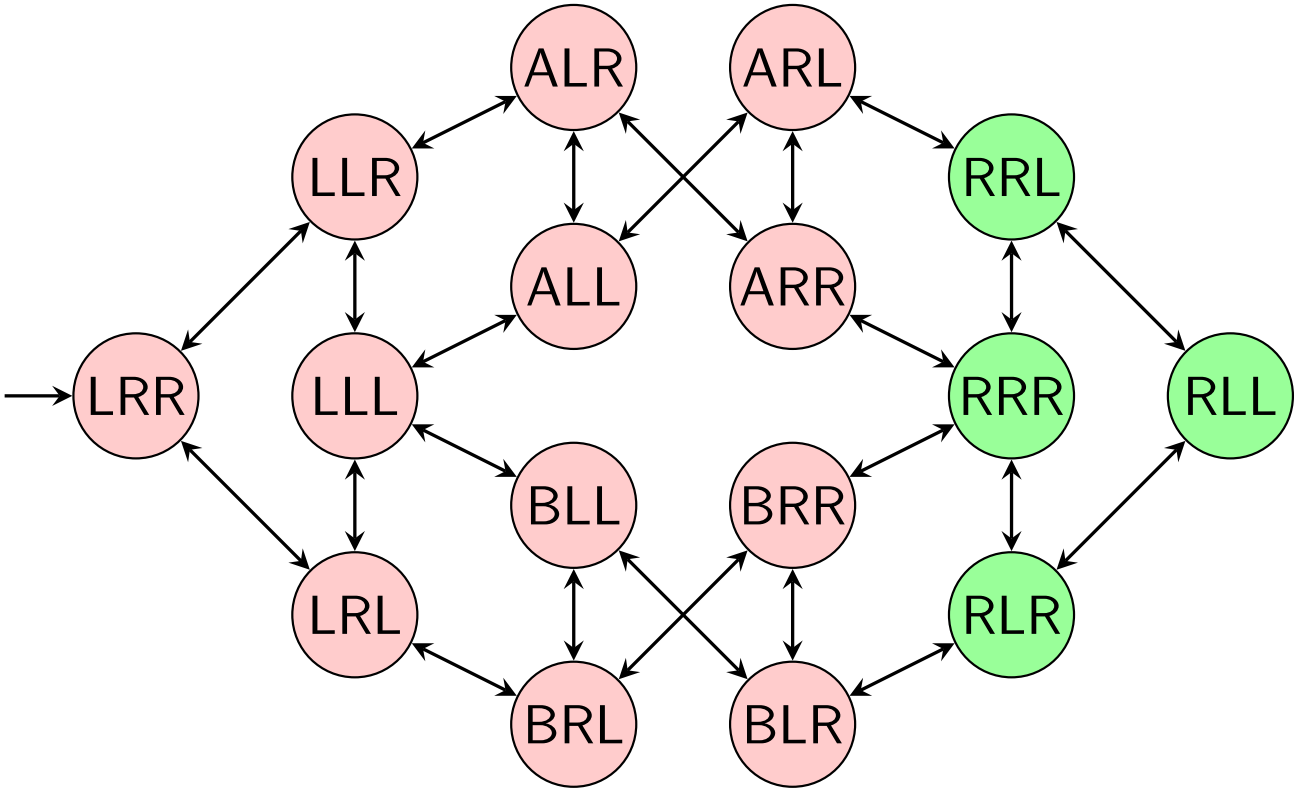
abbreviated notation: h^P for h^{π_P}

German: Pattern-Database-Heuristik

remark:

- "pattern databases" in analogy to endgame databases (which have been successfully applied in 2-person-games)

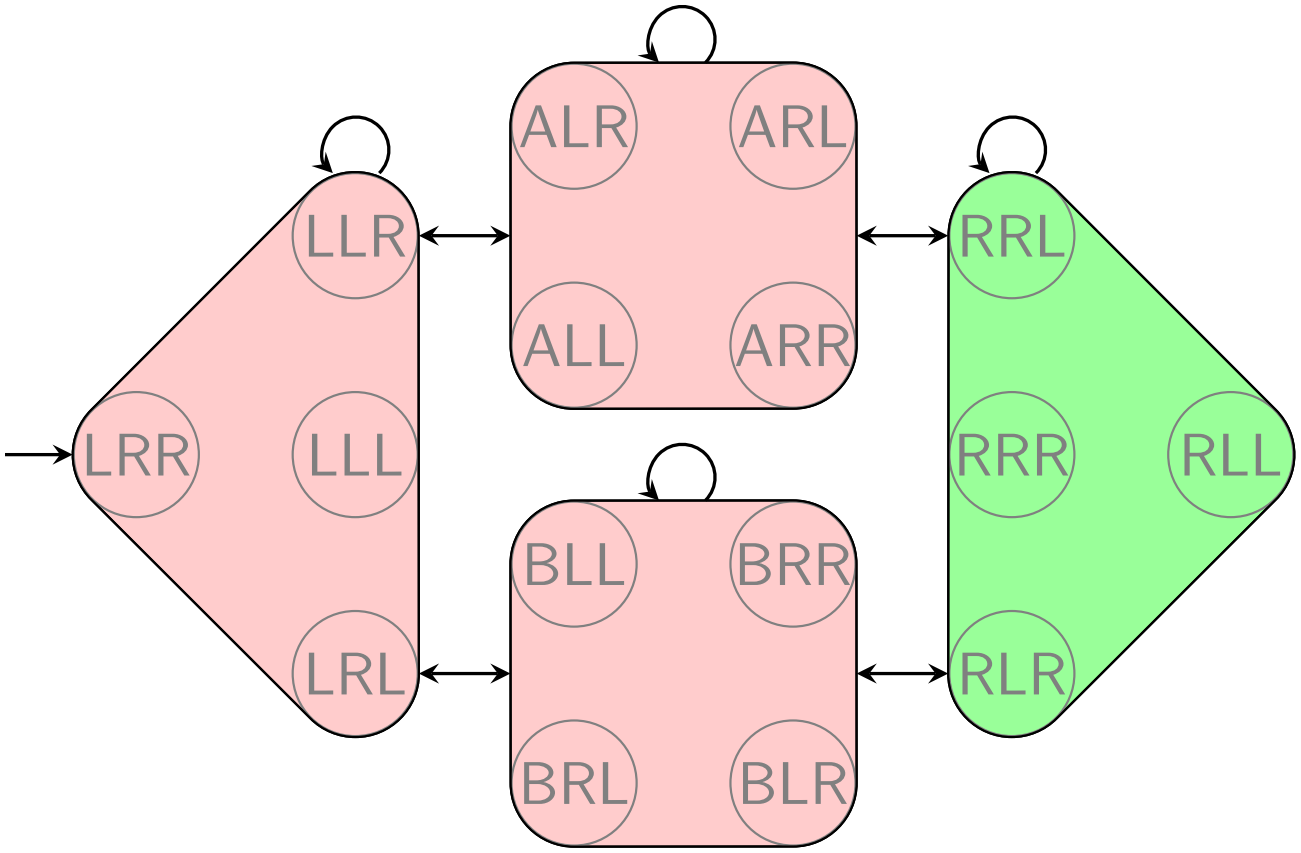
Example: Concrete State Space



- state variable *package*: $\{L, R, A, B\}$
- state variable *truck A*: $\{L, R\}$
- state variable *truck B*: $\{L, R\}$

Example: Projection (1)

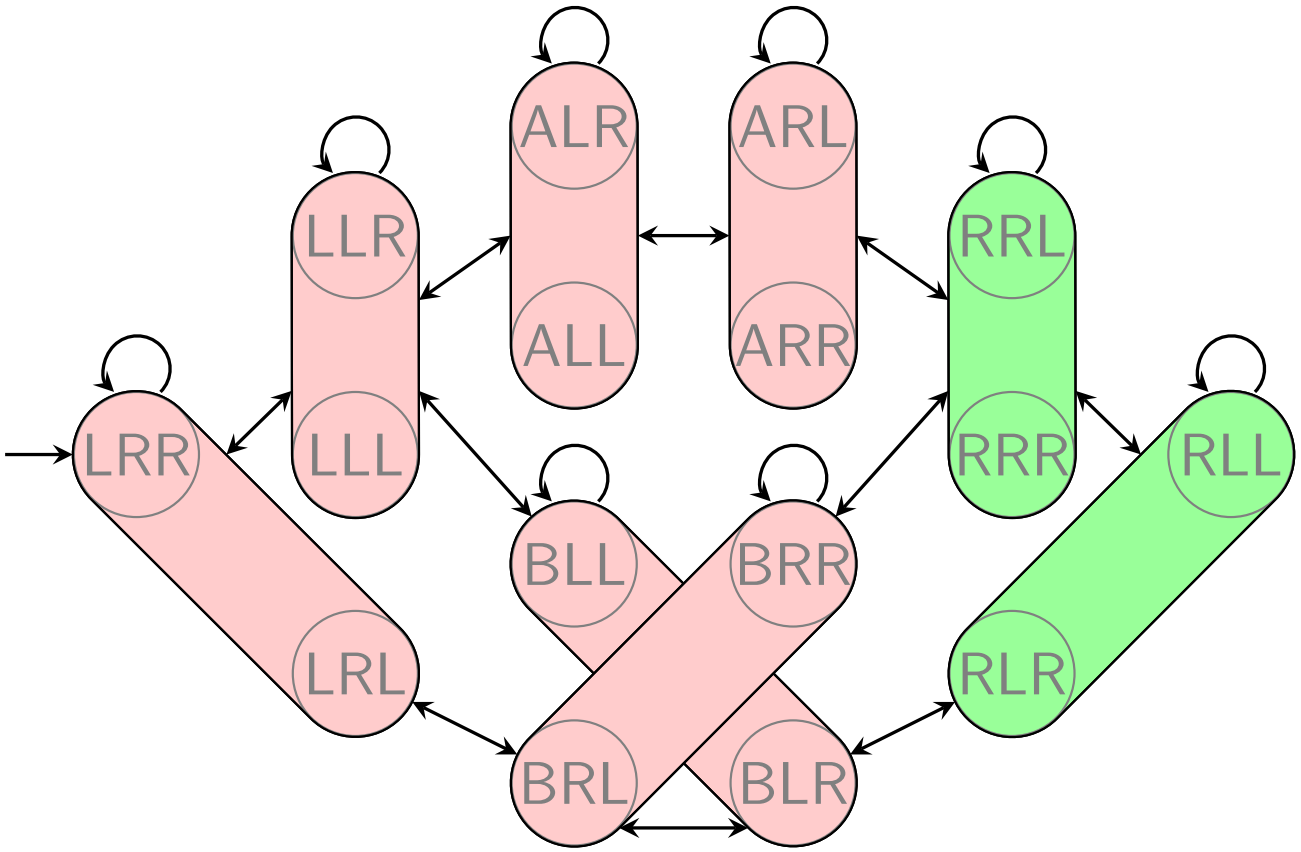
abstraction induced by $\pi_{\{package\}}$:



$h^{\{package\}}(\text{LRR}) = 2$

Example: Projection (2)

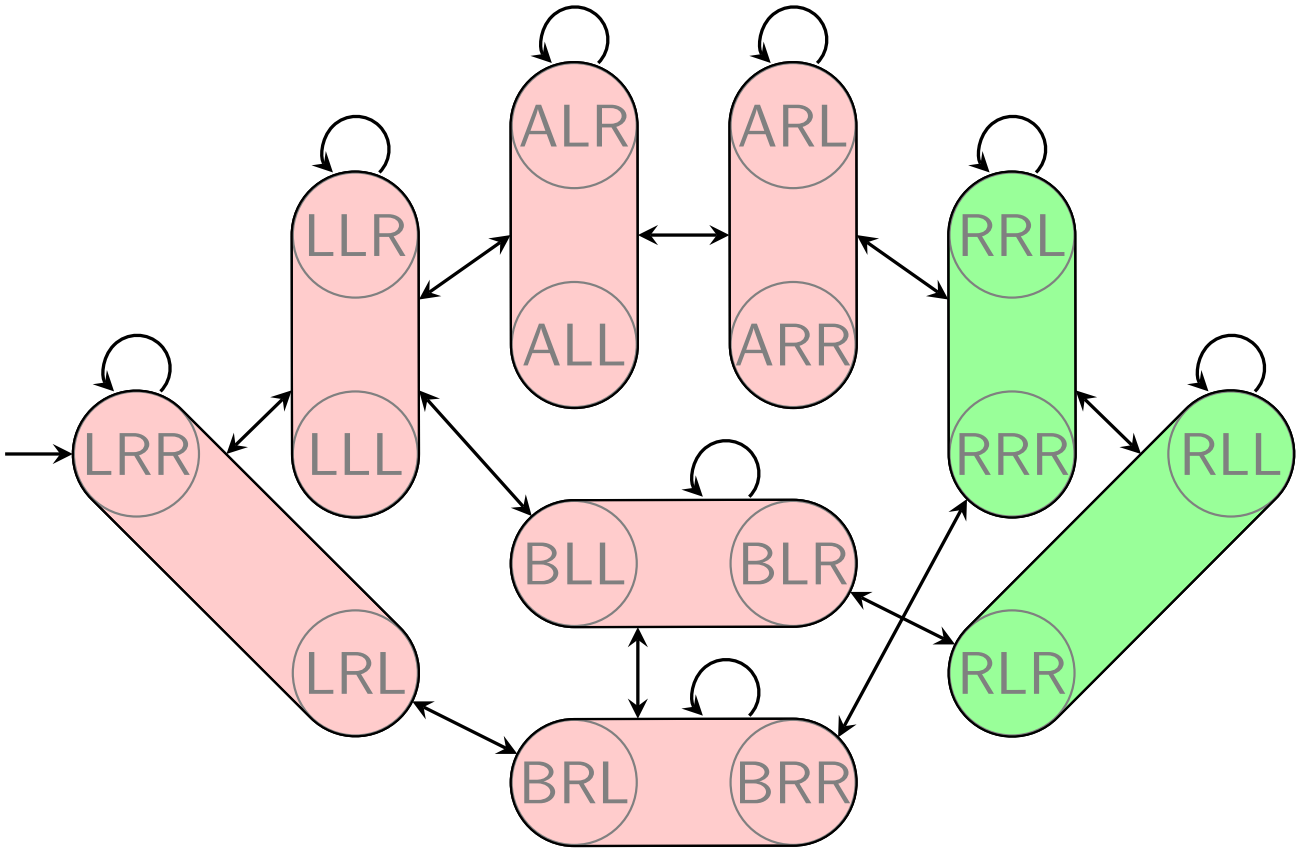
abstraction induced by $\pi_{\{package, truck\ A\}}$:



$h^{\{package, truck\ A\}}(\mathrm{LRR}) = 2$

Example: Projection (2)

abstraction induced by $\pi_{\{package, truck\ A\}}$:



$h^{\{package, truck\ A\}}(\mathrm{LRR}) = 2$

Pattern Databases in Practice

practical aspects which we do not discuss in detail:

- How to automatically find good patterns?
- How to combine multiple PDB heuristics?
- How to implement PDB heuristics efficiently?
  - good implementations efficiently handle abstract state spaces with 10^7, 10^8 or more abstract states
  - effort independent of the size of the concrete state space
  - usually all heuristic values are precomputed
    $\rightsquigarrow$ space complexity $=$ number of abstract states

Summary

Summary

- basic idea of abstraction heuristics: estimate solution cost by considering a smaller planning task.
- formally: abstraction function α maps states to abstract states and thus defines which states can be distinguished by the resulting heuristic.
- induces abstract state space whose solution costs are used as heuristic
- Pattern database heuristics are abstraction heuristics based on projections onto state variable subsets (patterns): states are distinguishable iff they differ on the pattern.

Foundations of Artificial Intelligence

G1. Board Games: Introduction and State of the Art

Malte Helmert

University of Basel

May 14, 2025

Board Games: Overview

chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Introduction

Why Board Games?

Board games are one of the oldest areas of AI (Shannon 1950; Turing 1950).

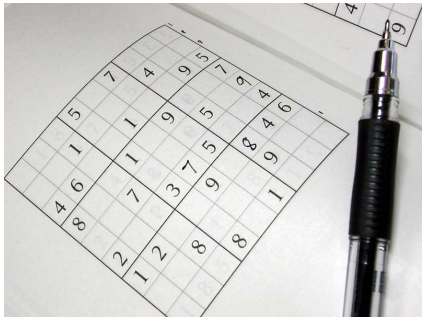
- abstract class of problems, easy to formalize
- obviously "intelligence" is needed (really?)
- dream of an intelligent machine capable of playing chess is older than electronic computers
- cf. von Kempelen's "Schachtürke" (1769), Torres y Quevedo's "El Ajedrecista" (1912)

Board Games

algorithms considered previously:

agent has full control over environment:

- agent is only acting entity
- effects of actions fully predictable

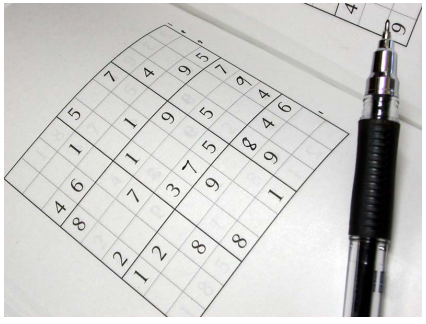


Board Games

algorithms considered previously:

agent has full control over environment:

- agent is only acting entity
- effects of actions fully predictable



games considered now (Chapters G1–G3):

environment changes independently of agent:

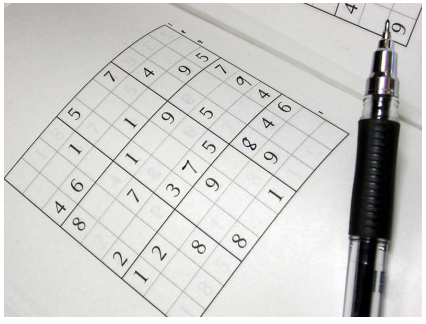
- other agent (with opposing objective) acts

Board Games

algorithms considered previously:

agent has full control over environment:

- agent is only acting entity
- effects of actions fully predictable

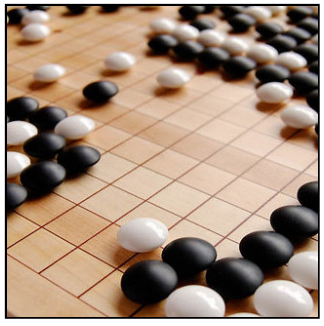
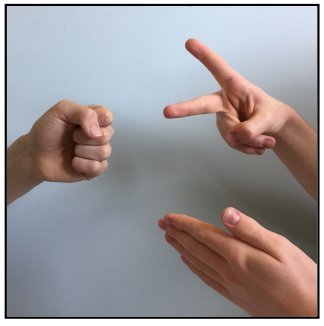


games considered later (Chapter G4):

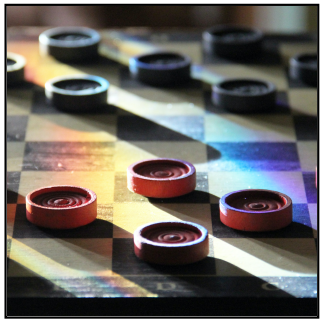
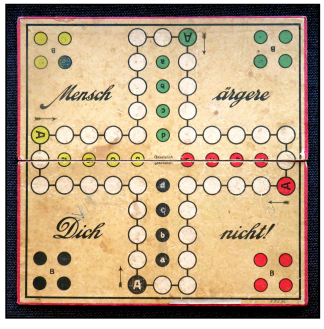
environment changes independently of agent:

- other agent (with opposing objective) acts
- effects of actions underly chance

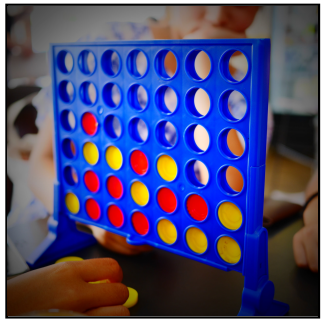
Applications



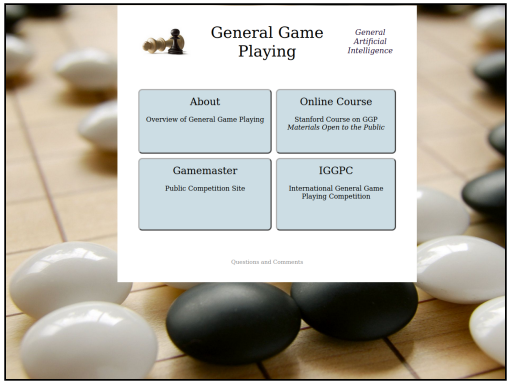




Game Applications Beyond Specific Board Games

video games

general game playing



cyber security

wildlife preservation

generative adversarial networks

auctions

Game Environments

game environments cover entire spectrum of properties
$\rightsquigarrow$ need some restrictions

important classes of games that we do not consider:

- with randomness (e.g., backgammon) ($\rightsquigarrow$ Chapter G4)
- with more than two players (e.g., poker)
- with hidden information (e.g., scrabble)
- with simultaneous moves (e.g., rock-paper-scissors)
- without turns (e.g., many video games)
- without zero-sum property (e.g., auctions)
- ...

many of these can be handled with similar/generalized algorithms

Properties of Games Considered (for Now)

- current situation representable by finite set of positions
- there is a finite set of moves players can play
- effects of actions are deterministic
- the game ends when a terminal position is reached
- a terminal position is reached after a finite number of steps (*)
- terminal positions yield a utility
- no randomness, no hidden information

(*) Our definitions do not enforce this, and there are some subtleties associated with this requirement which we ignore.

Properties of Games Considered (for Now)

- there are exactly two players called MAX and MIN
- both players observe the entire position (perfect information)
- it is the turn of exactly one player in each non-terminal position
- utility for MAX is opposite of utility for MIN (zero-sum game)
- MAX aims to maximize utility, MIN aims to minimize utility

Classification

classification:

Board Games

environment:

- static vs. dynamic
- deterministic vs. nondeterministic vs. stochastic
- fully observable vs. partially observable
- discrete vs. continuous
- single-agent vs. multi-agent (adversarial)

problem solving method:

- problem-specific vs. general vs. learning

Informal Description

objective of the agent:

- compute a strategy
- that determines which move to execute
- in the current position or in any (reachable) position

performance measure:

- maximize utility (given available resources)

To study board games, we need a formal model.

Games

Example: Chess

Example (Chess)

- positions described by:
  - configuration of pieces
  - whose turn it is
  - en-passant and castling rights
- turns alternate
- terminal positions: checkmate and stalemate positions
- utility of terminal position for first player (white):
  - $+1$ if black is checkmated
  - 0 if stalemate position
  - -1 if white is checkmated

Terminology Compared to State-Space Search

Many concepts for board games are similar to state-space search.
Terminology differs, but is often in close correspondence:

- state $\rightsquigarrow$ position
- goal state $\rightsquigarrow$ terminal position
- action $\rightsquigarrow$ move
- search tree $\rightsquigarrow$ game tree

Definition

Definition (game)

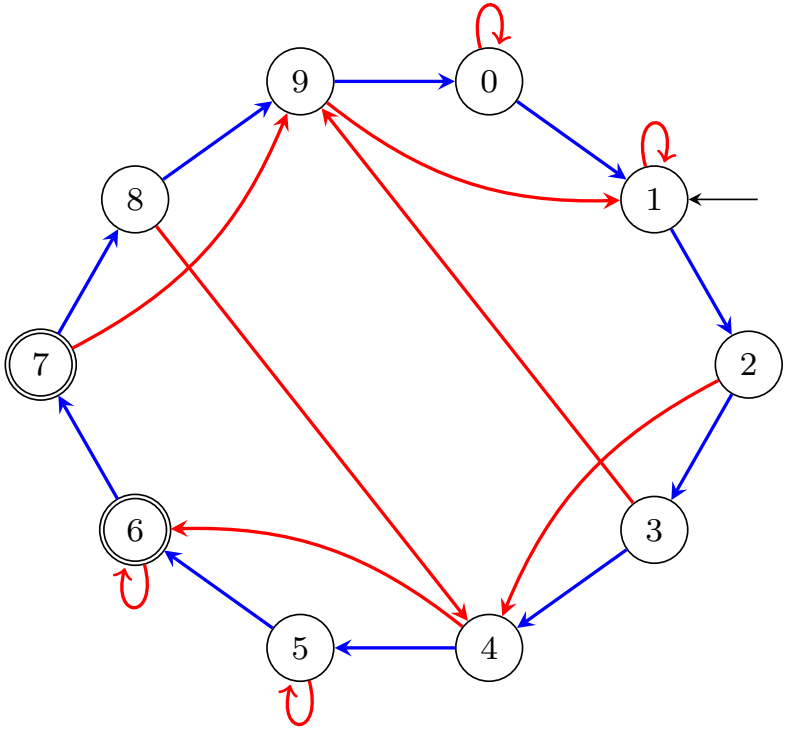
A game is a 7-tuple $\mathcal{S} = \langle S, A, T, s_I, S_G, \mathit{utility}, \mathit{player} \rangle$ with

- finite set of positions S
- finite set of moves A
- deterministic transition relation $T \subseteq S \times A \times S$
- initial position $s_I \in S$
- set of terminal positions $S_G \subseteq S$
- utility function $\mathit{utility} : S_G \to \mathbb{R}$
- player function $\mathit{player} : S \setminus S_G \to \{\mathrm{MAX}, \mathrm{MIN}\}$

Reminder: Bounded Inc-and-Square Search Problem

informal description:

- find a sequence of
  - increment-mod10 (*inc*) and
  - square-mod10 (*sqr*) actions
- on the natural numbers from 0 to 9
- that reaches the number 6 or 7
- starting from the number 1
- assuming each action costs 1.



Running Example: Bounded Inc-and-Square Game

informal description:

- Players alternatingly apply a
  - increment-mod10 (*inc*) or
  - square-mod10 (*sqr*) move
- on the natural numbers from 0 to 9
- starting from the number 1;
- if the game reaches the number 6 or 7
- or after a fixed number of 4 moves
- MAX obtains utility u (MIN: $-u$)
  where u is the current number.

Running Example: Bounded Inc-and-Square Game

informal description:

- Players alternatingly apply a
  - increment-mod10 (*inc*) or
  - square-mod10 (*sqr*) move
- on the natural numbers from 0 to 9
- starting from the number 1;
- if the game reaches the number 6 or 7
- or after a fixed number of 4 moves
- MAX obtains utility u (MIN: $-u$) where u is the current number.

formal model:

- $S = \{s_i^k \mid 0 \leq i \leq 9, 0 \leq k \leq 4\}$
- $A = \{inc, sqr\}$
- for $0 \leq i \leq 9$ and $0 \leq k < 4$:
  - $\langle s_i^k, inc, s^{k+1}_{(i+1) \bmod 10} \rangle \in T$
  - $\langle s_i^k, sqr, s^{k+1}_{i^2 \bmod 10} \rangle \in T$
- $s_{\mathsf{I}} = s_1^0$
- $S_{\mathsf{G}} = \{s_i^k \mid i \in \{6,7\} \vee k = 4\}$
- $utility(s_i^k) = i$ for all $s_i^k \in S_{\mathsf{G}}$
- $player(s_i^k) =$ MAX if k even and $player(s_i^k) =$ MIN otherwise

Why are Board Games Difficult?

As in classical search problems, the number of positions of (interesting) board games is huge:

- Chess: roughly 10^{40} reachable positions;
  game with 50 moves/player and branching factor 35:
  tree size roughly $35^{100} \approx 10^{154}$
- Go: more than 10^{100} positions;
  game with roughly 300 moves and branching factor 200:
  tree size roughly $200^{300} \approx 10^{690}$

In addition, it is not sufficient to find a solution path:

- We need a strategy reacting to all possible opponent moves.
- Usually, such a strategy is implemented as an algorithm that provides the next move on the fly (i.e., not precomputed).

Strategies

Definition (strategy, partial strategy)

Let $\mathcal{S} = \langle S, A, T, s_I, S_G, \mathit{utility}, \mathit{player} \rangle$ be a game and let $S_{\mathrm{MAX}} = \{s \in S \mid \mathit{player}(s) = \mathrm{MAX}\}$.
A partial strategy for player MAX is a function

$$\pi : S'_{\mathrm{MAX}} \mapsto A$$

with $S'_{\mathrm{MAX}} \subseteq S_{\mathrm{MAX}}$ and $\pi(s) = a$ implies that a is applicable in s.

If $S'_{\mathrm{MAX}} = S_{\mathrm{MAX}}$, then π is also called total strategy (or strategy).

We always take the viewpoint of MAX, but S_{MIN} and a (partial/total) strategy for MIN are defined accordingly.

Specific vs. General Algorithms

- We consider approaches that must be tailored to a specific board game for good performance, e.g., by using a suitable evaluation function.

⇝ see chapters on informed search methods

- Analogously to the generalization of search methods to declaratively described problems (automated planning), board games can be considered in a more general setting, where game rules (state spaces) are part of the input.

⇝ general game playing: regular competitions since 2005

Algorithms for Board Games

properties of good algorithms for board games:

- look ahead as far as possible (deep search)
- consider only interesting parts of the game tree
  (selective search, analogously to heuristic search algorithms)
- evaluate current position as accurately as possible
  (evaluation functions, analogously to heuristics)

State of the Art

State of the Art

some well-known board games:

- Chess, Go: $\rightsquigarrow$ next slides
- Othello: Logistello defeated human world champion in 1997;
  best computer players significantly stronger than best humans
- Checkers: Chinook official world champion (since 1994);
  proved in 2007 that it cannot be defeated
  and perfect game play results in a draw (game "solved")

Computer Chess

World champion Garry Kasparov was defeated by Deep Blue in 1997 (6 matches, result 3.5–2.5).

- specialized chess hardware (30 cores with 16 chips each)
- alpha-beta search (⇝ Chapter G3) with extensions
- database of opening moves from millions of chess games

Nowadays, chess programs on standard PCs are much stronger than all human players.

Computer Chess: Quotes

Claude Shannon (1950)

The chess machine is an ideal one to start with, since

1. the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate),
2. it is neither so simple as to be trivial nor too difficult for satisfactory solution,
3. chess is generally considered to require "thinking" for skillful play, [. . .]
4. the discrete structure of chess fits well into the digital nature of modern computers.

Computer Chess: Quotes

Claude Shannon (1950)

The chess machine is an ideal one to start with, since

1. the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate),
2. it is neither so simple as to be trivial nor too difficult for satisfactory solution,
3. chess is generally considered to require "thinking" for skillful play, [. . .]
4. the discrete structure of chess fits well into the digital nature of modern computers.

Alexander Kronrod (1965)

Chess is the drosophila of Artificial Intelligence.

Computer Chess: Another Quote

John McCarthy (1997)

In 1965, the Russian mathematician Alexander Kronrod said, "Chess is the drosophila of artificial intelligence."

Computer Chess: Another Quote

John McCarthy (1997)

In 1965, the Russian mathematician Alexander Kronrod said, "Chess is the drosophila of artificial intelligence."

However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing drosophilae. We would have some science, but mainly we would have very fast fruit flies.

Computer Go

Computer Go

- The best Go programs use Monte-Carlo techniques (UCT).
- Until autumn 2015, leading programs Zen, Mogo, Crazystone played on the level of strong amateurs (1 kyu/1 dan).
- Until then, Go was considered as one of the "last" games that are too complex for computers.
- In October 2015, Deep Mind's AlphaGo defeated the European Champion Fan Hui (2p dan) with 5:0.
- In March 2016, AlphaGo defeated world-class player Lee Sedol (9p dan) with 4:1. The prize for the winner was 1 million US dollars.

Summary

Summary

- Board games can be considered as classical search problems extended by an opponent.
- Both players try to reach a terminal position with (for the respective player) maximal utility.
- very successful for a large number of popular games
- Deep Blue defeated the world chess champion in 1997. Today, chess programs play vastly more strongly than humans.
- AlphaGo defeated one of the world's best players in the game of Go in 2016.

Foundations of Artificial Intelligence

G2. Board Games: Minimax Search and Evaluation Functions

Malte Helmert

University of Basel

May 14, 2025

Board Games: Overview

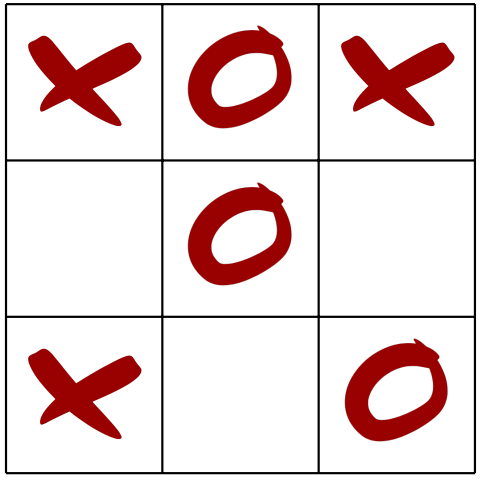
chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Minimax Search

Example: Tic-Tac-Toe

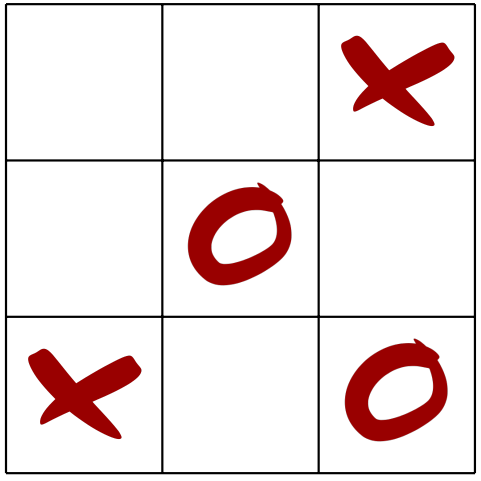
consider it's the turn of player ✕:



If the utility for win/draw/lose for player ✕ is +1/0/-1,
what is an appropriate utility value for the depicted position?

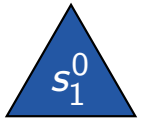
Example: Tic-Tac-Toe

consider it's the turn of player ✕:



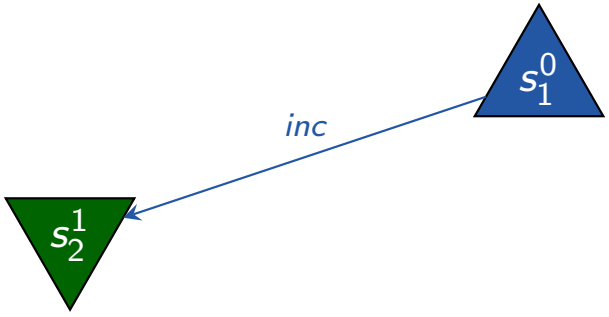
And what about this one?

Idea and Example



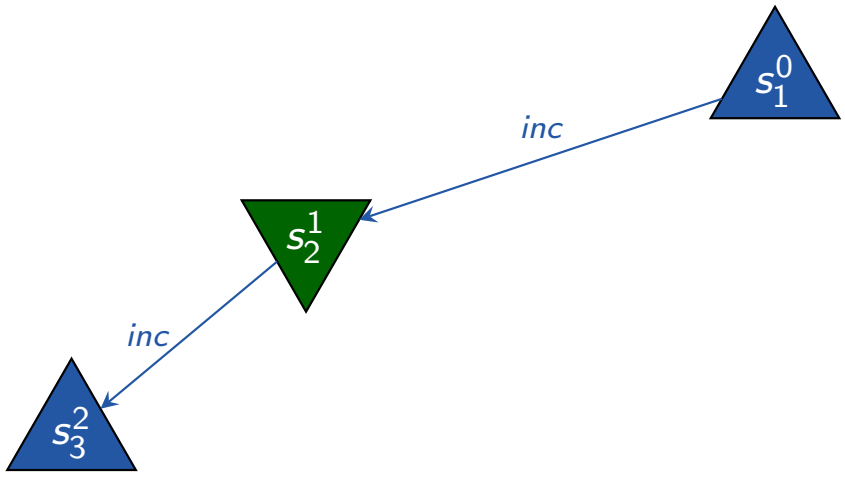
- depth-first search in game tree

Idea and Example



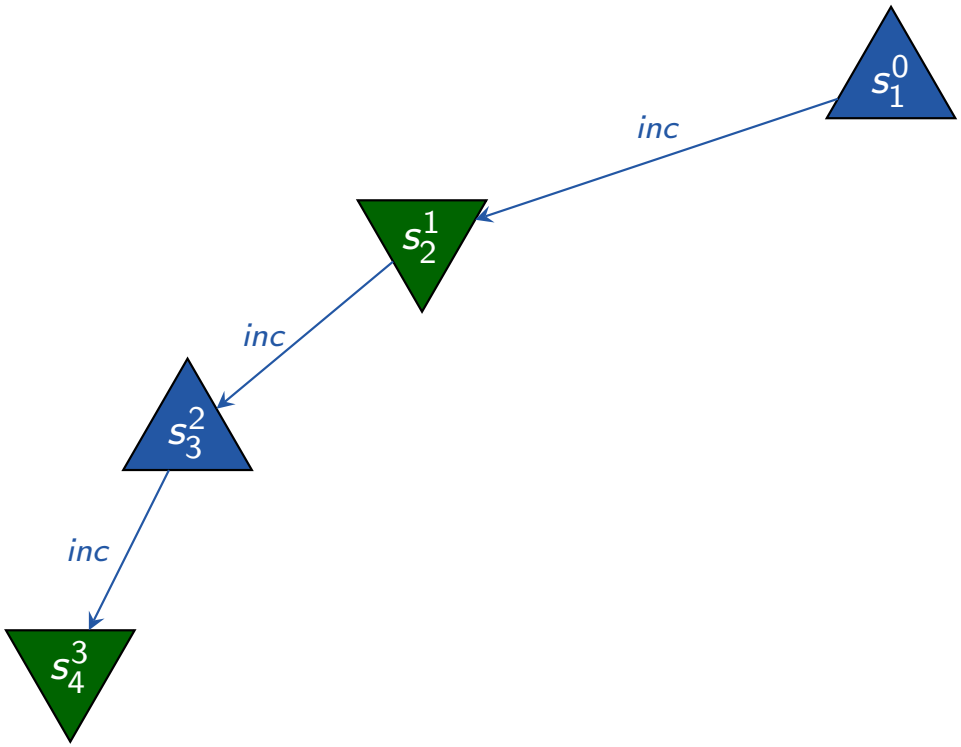
- depth-first search in game tree

Idea and Example



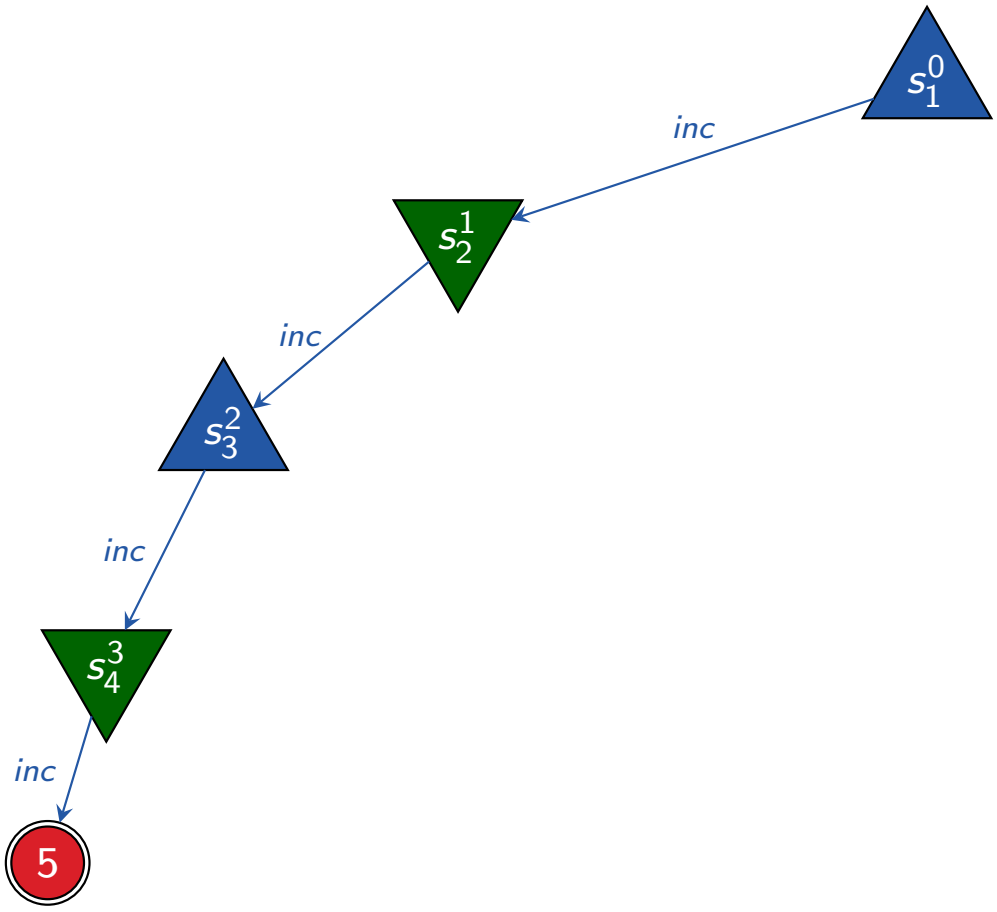
- depth-first search in game tree

Idea and Example



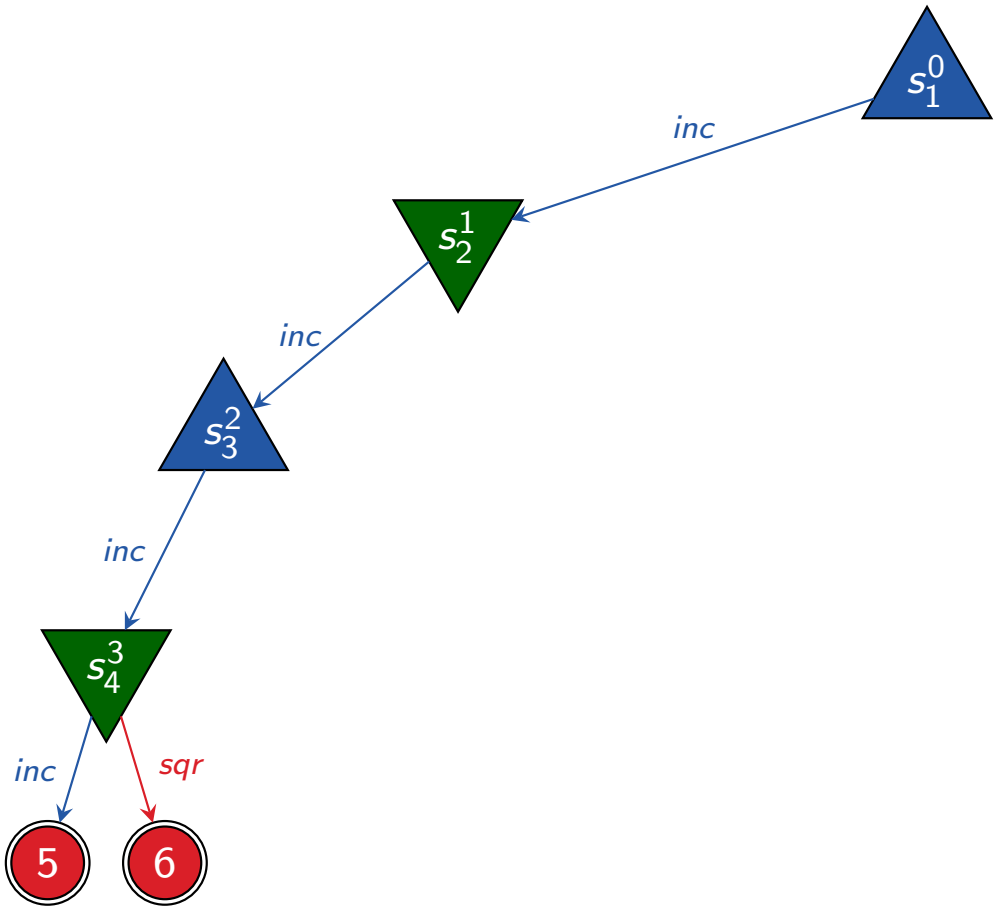
- depth-first search in game tree

Idea and Example



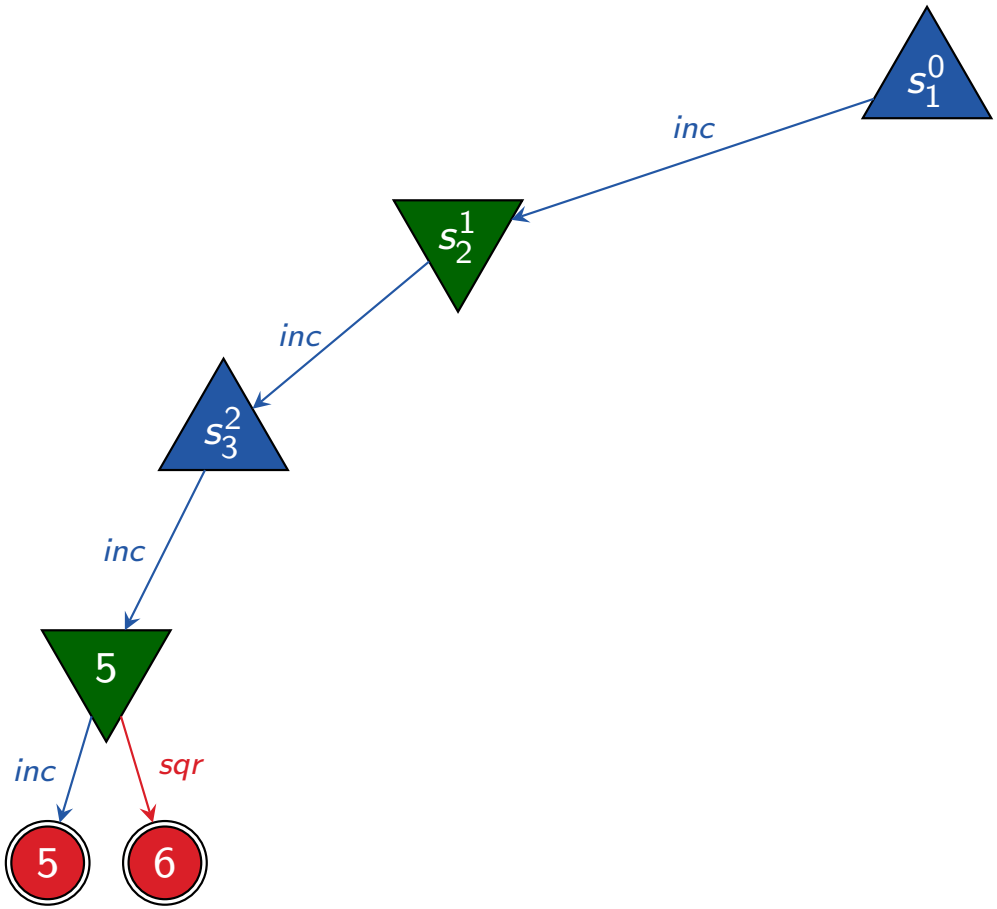
- depth-first search in game tree
- determine utility value of terminal position with utility function

Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function

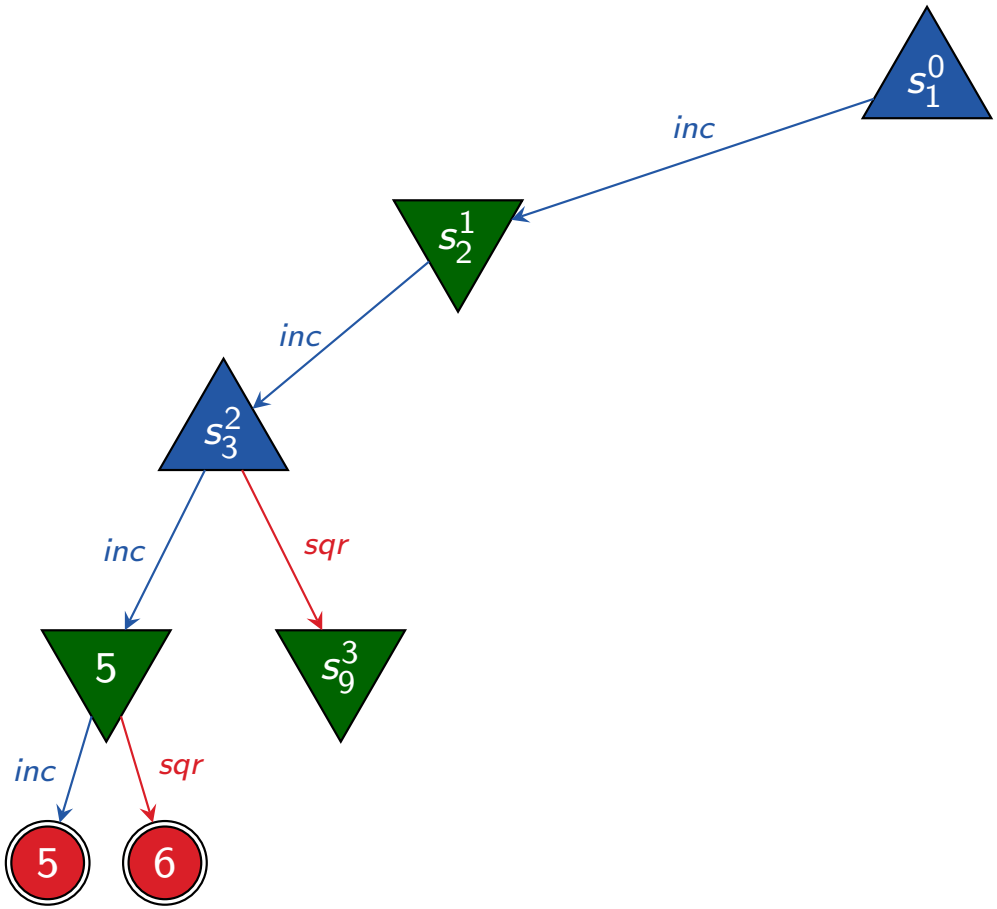
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

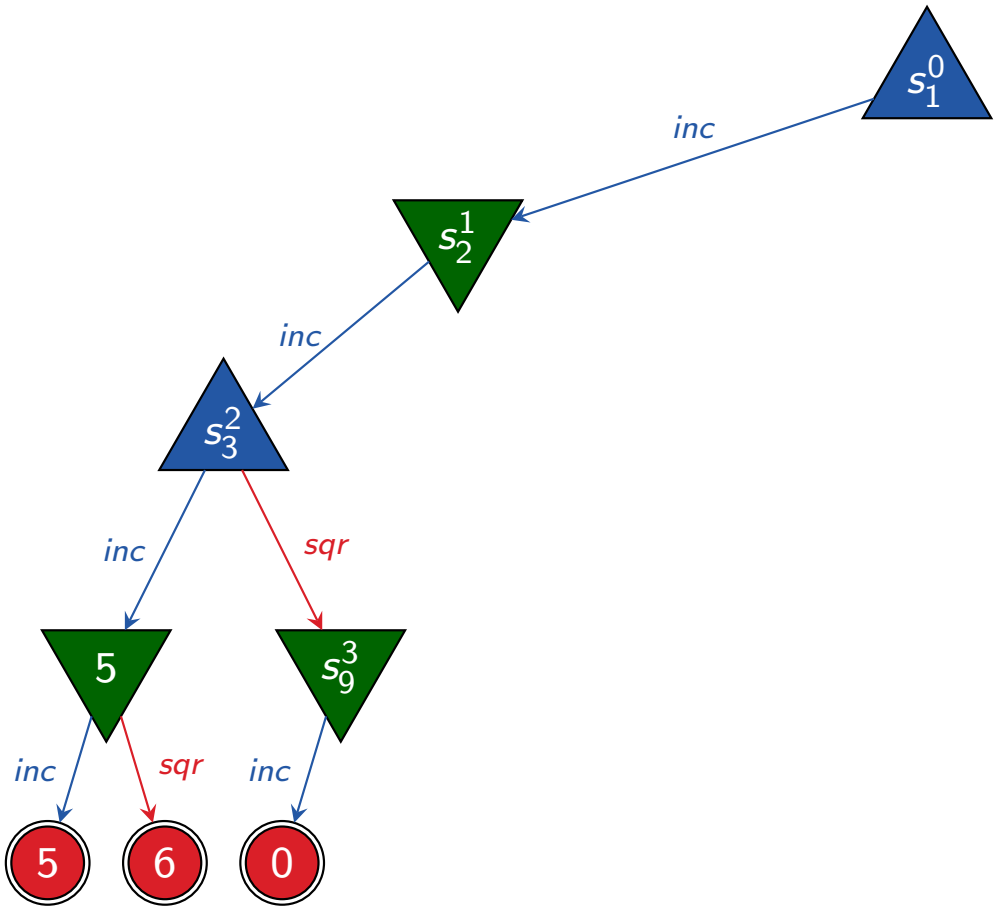
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

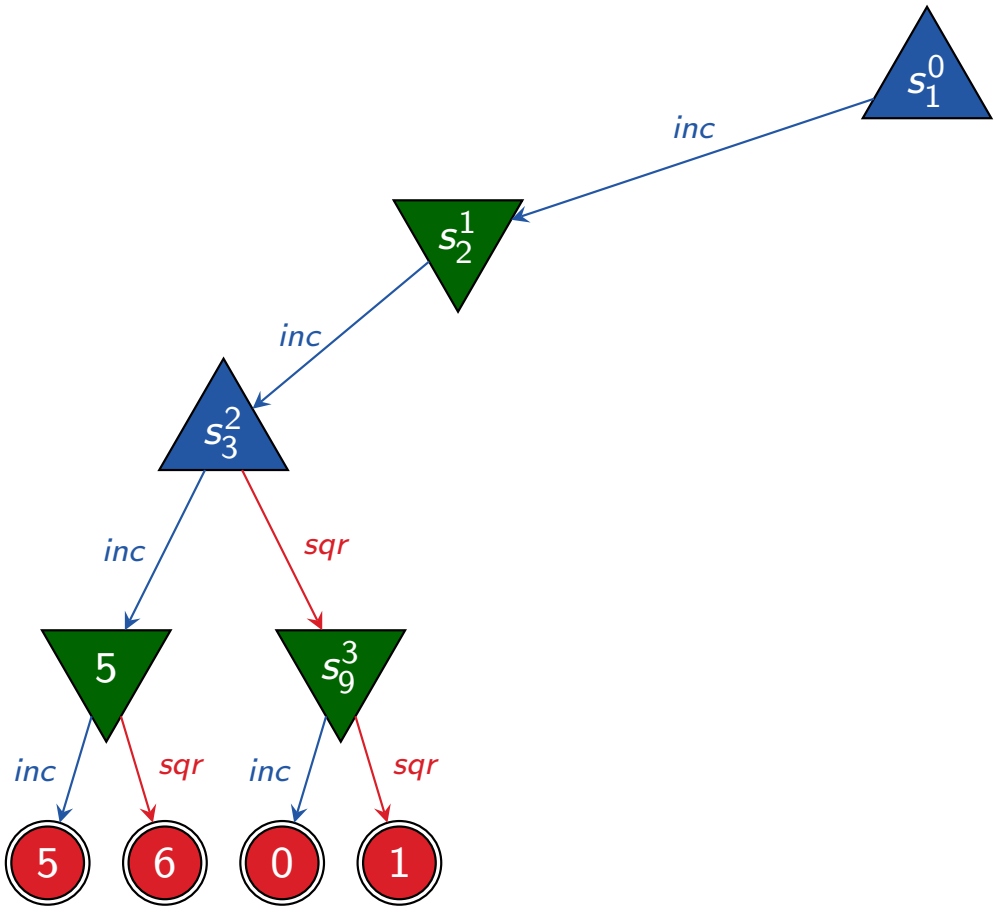
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

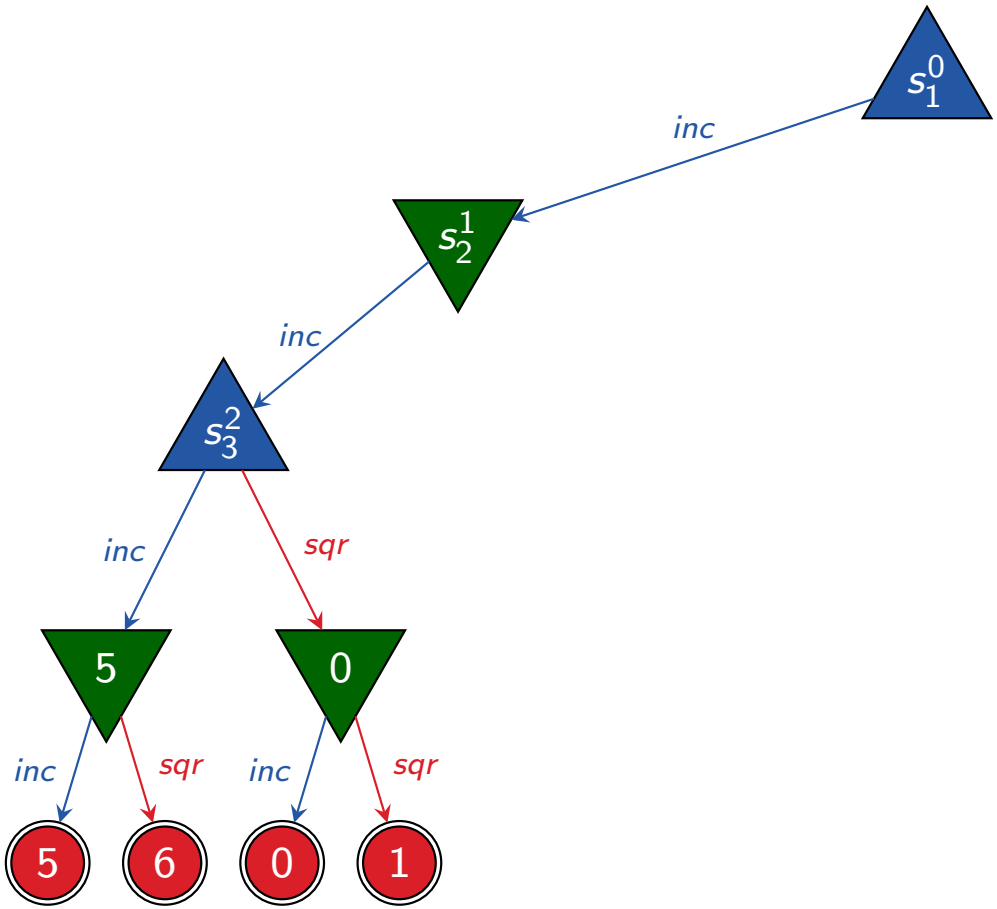
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

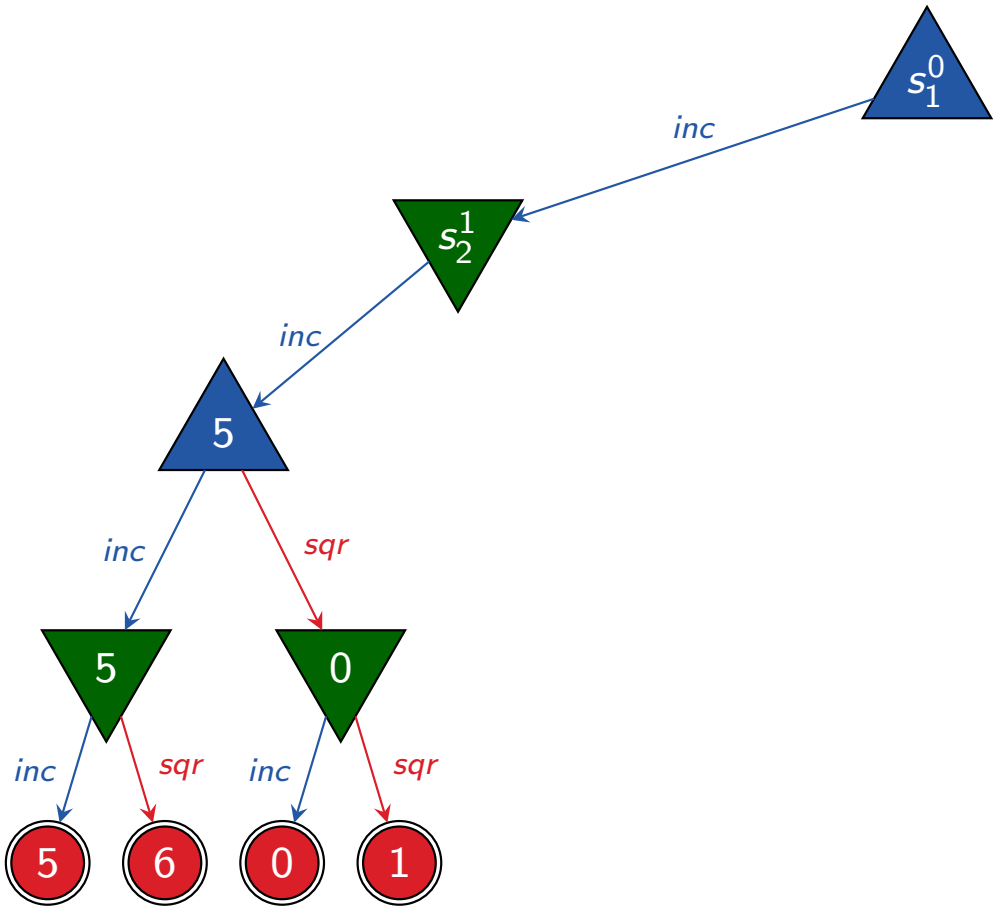
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

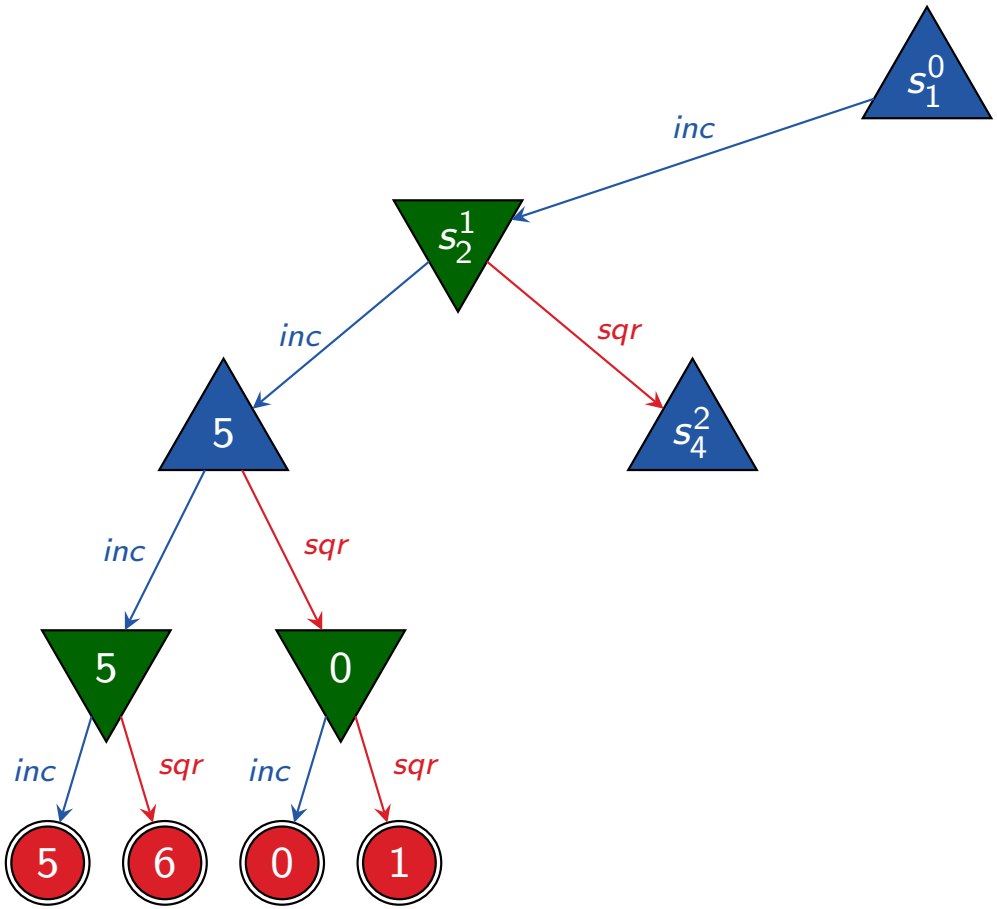
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

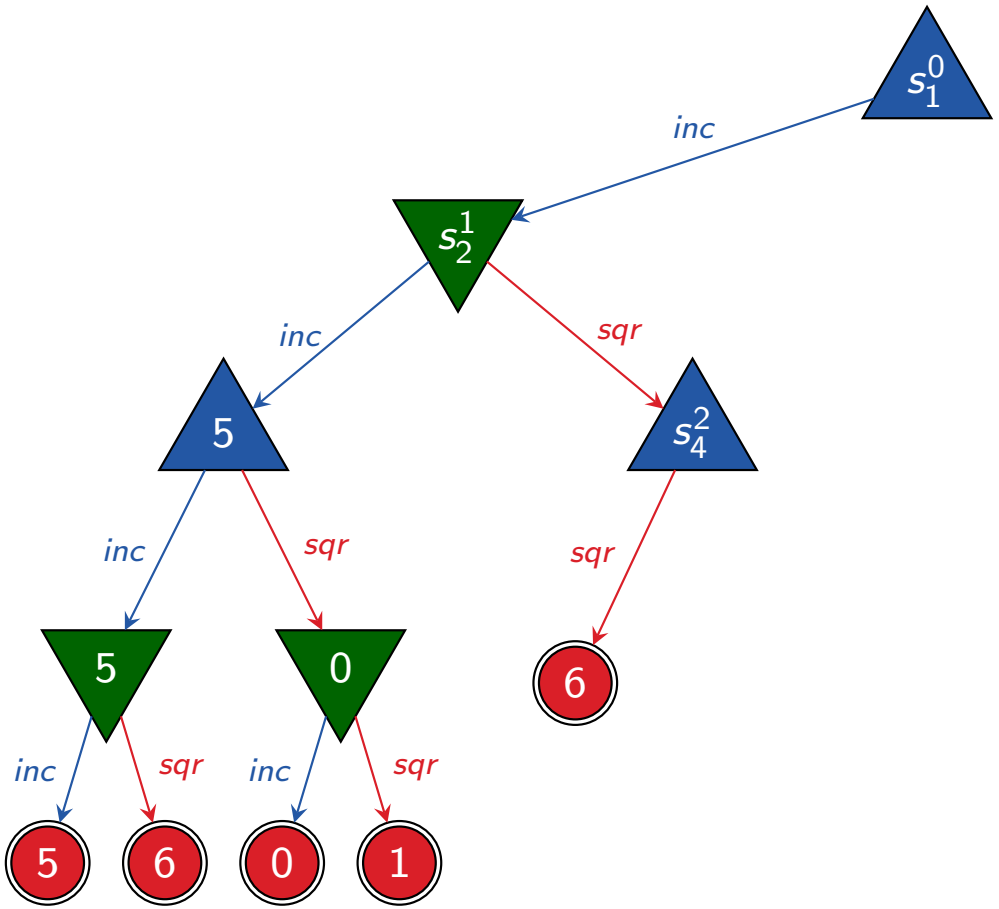
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

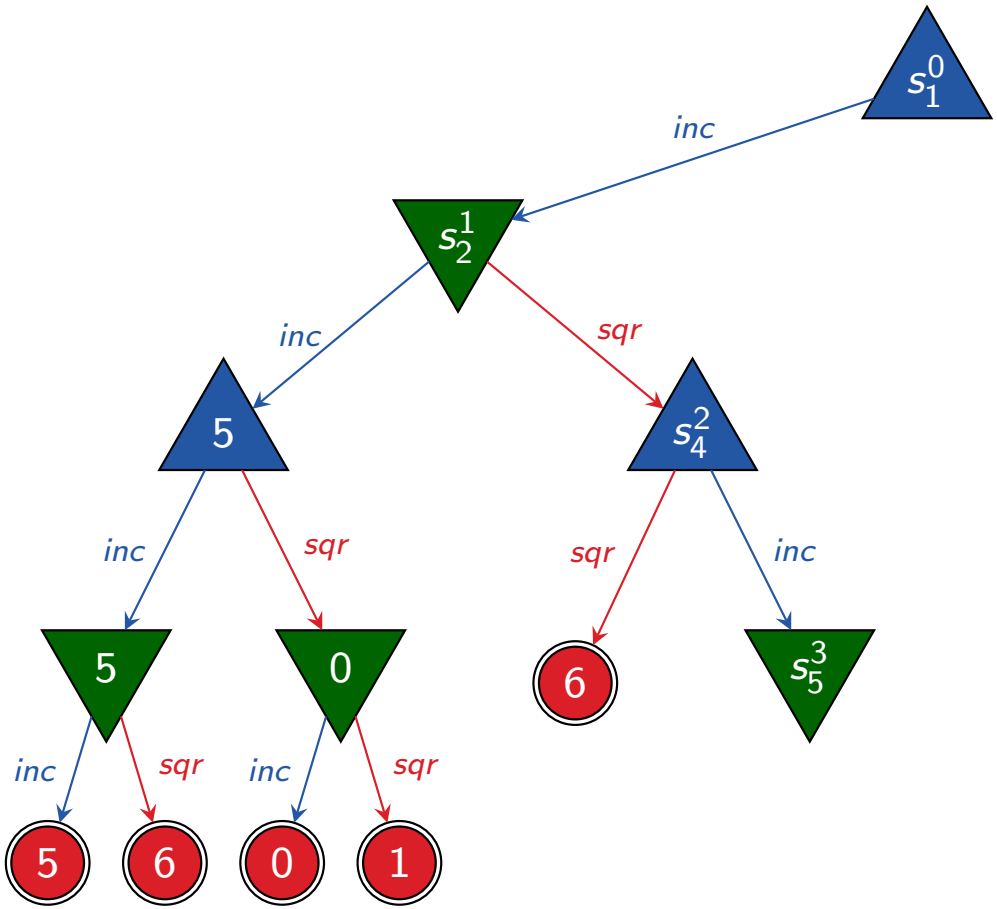
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

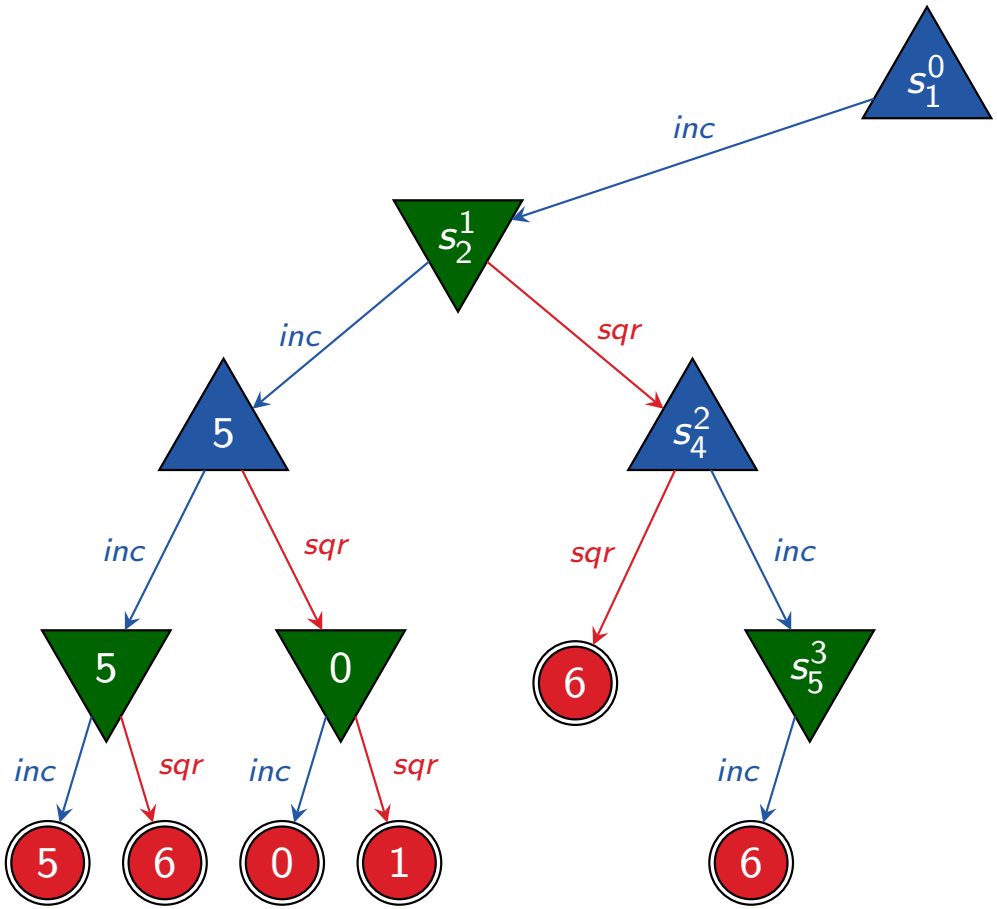
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

Idea and Example

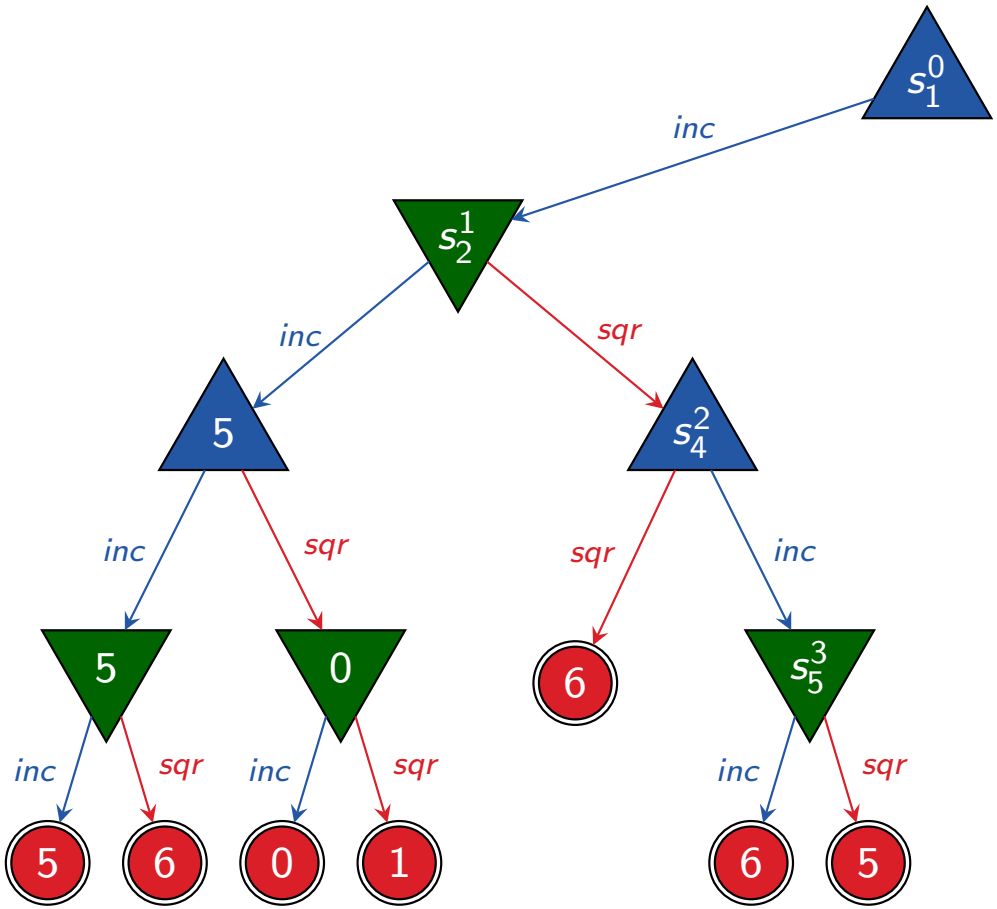


- depth-first search in game tree
- determine utility value of terminal position with utility function

- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

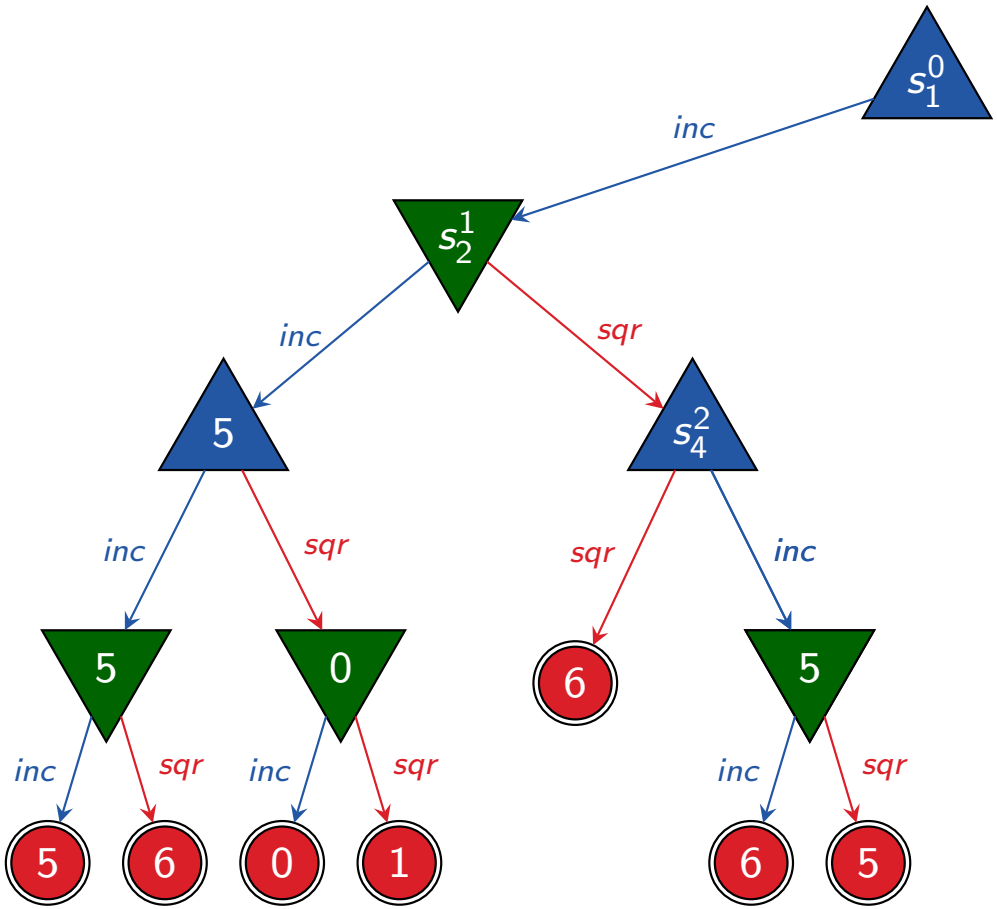
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

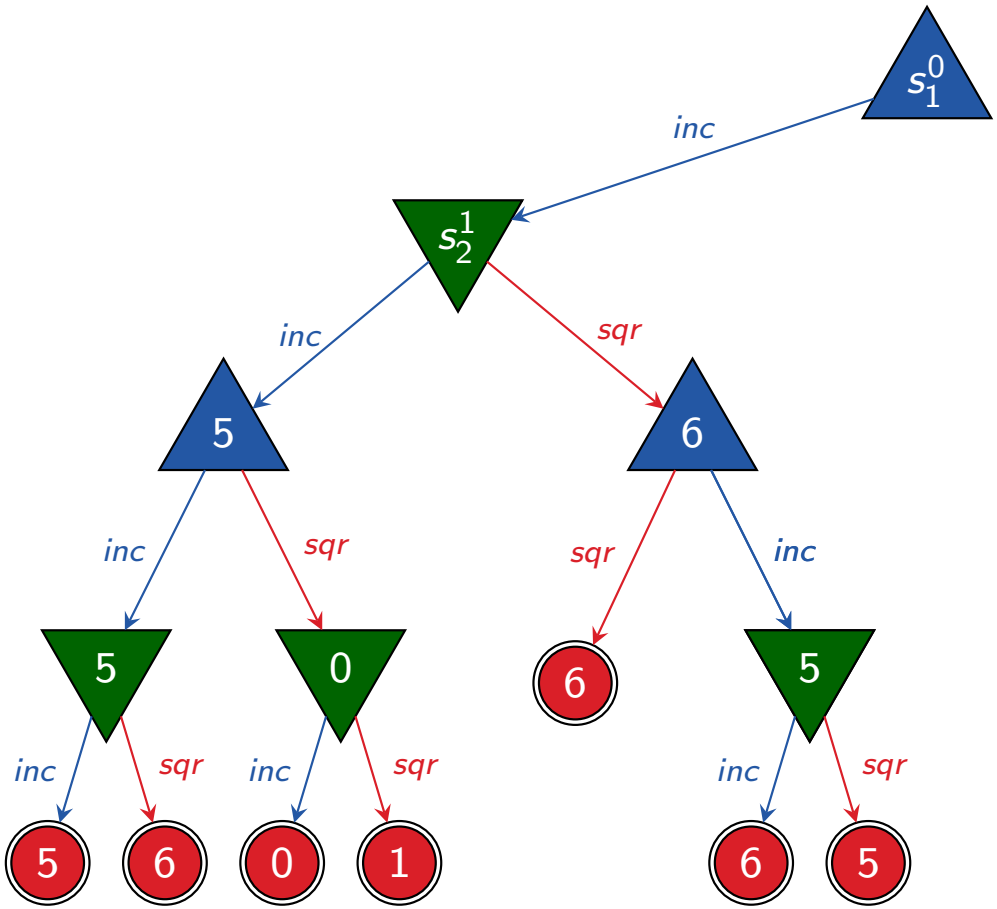
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

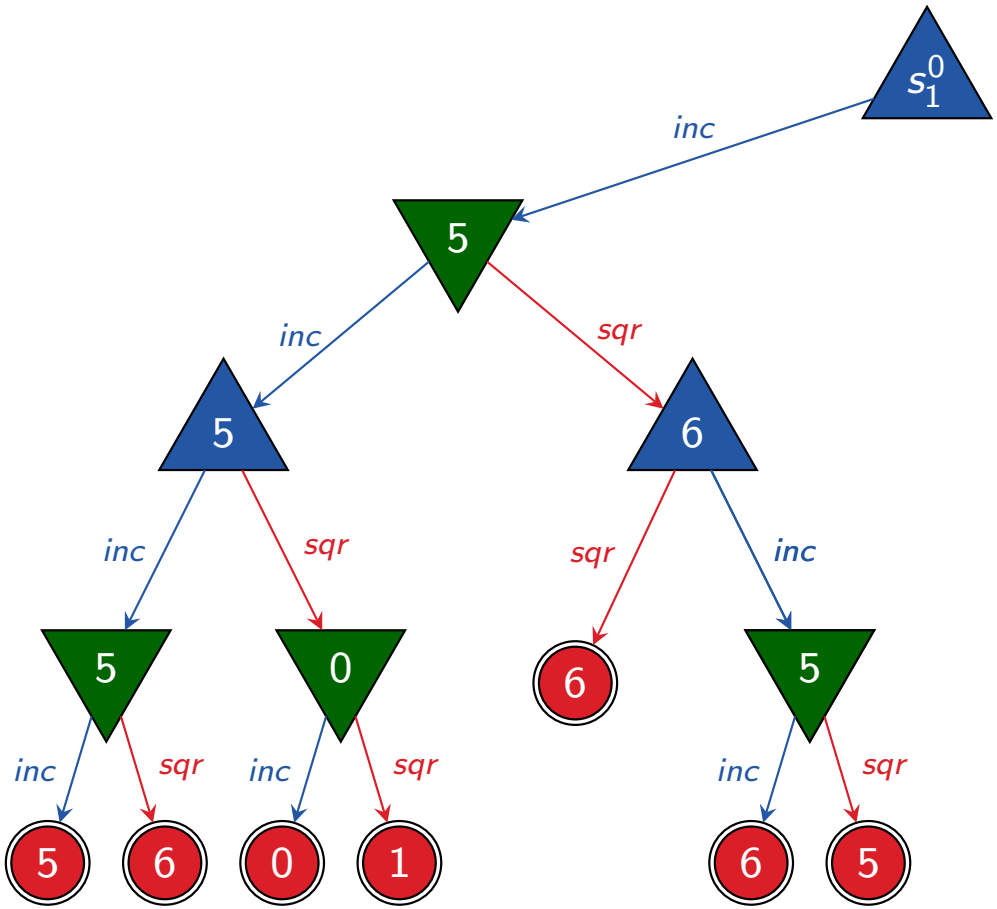
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

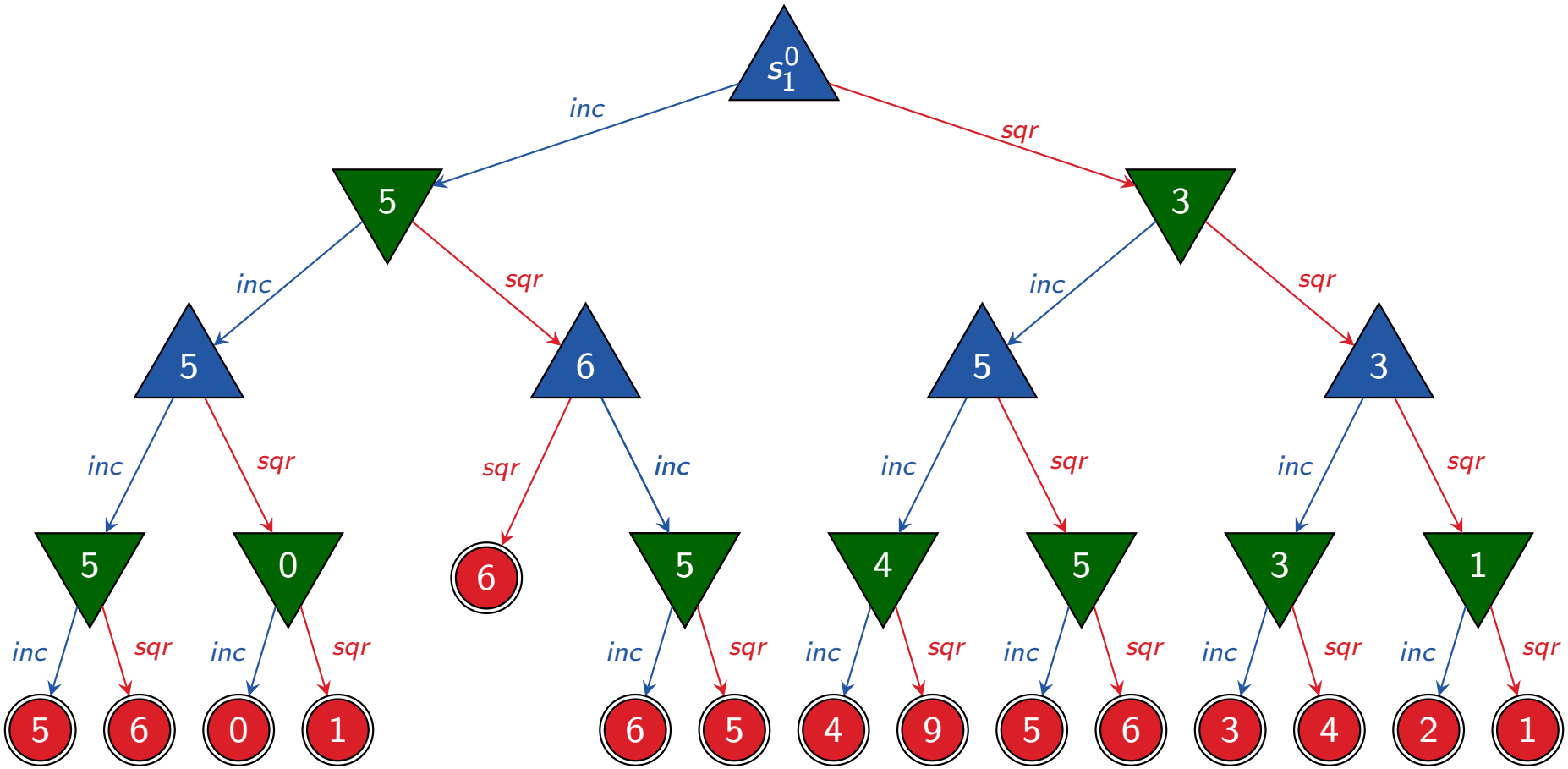
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

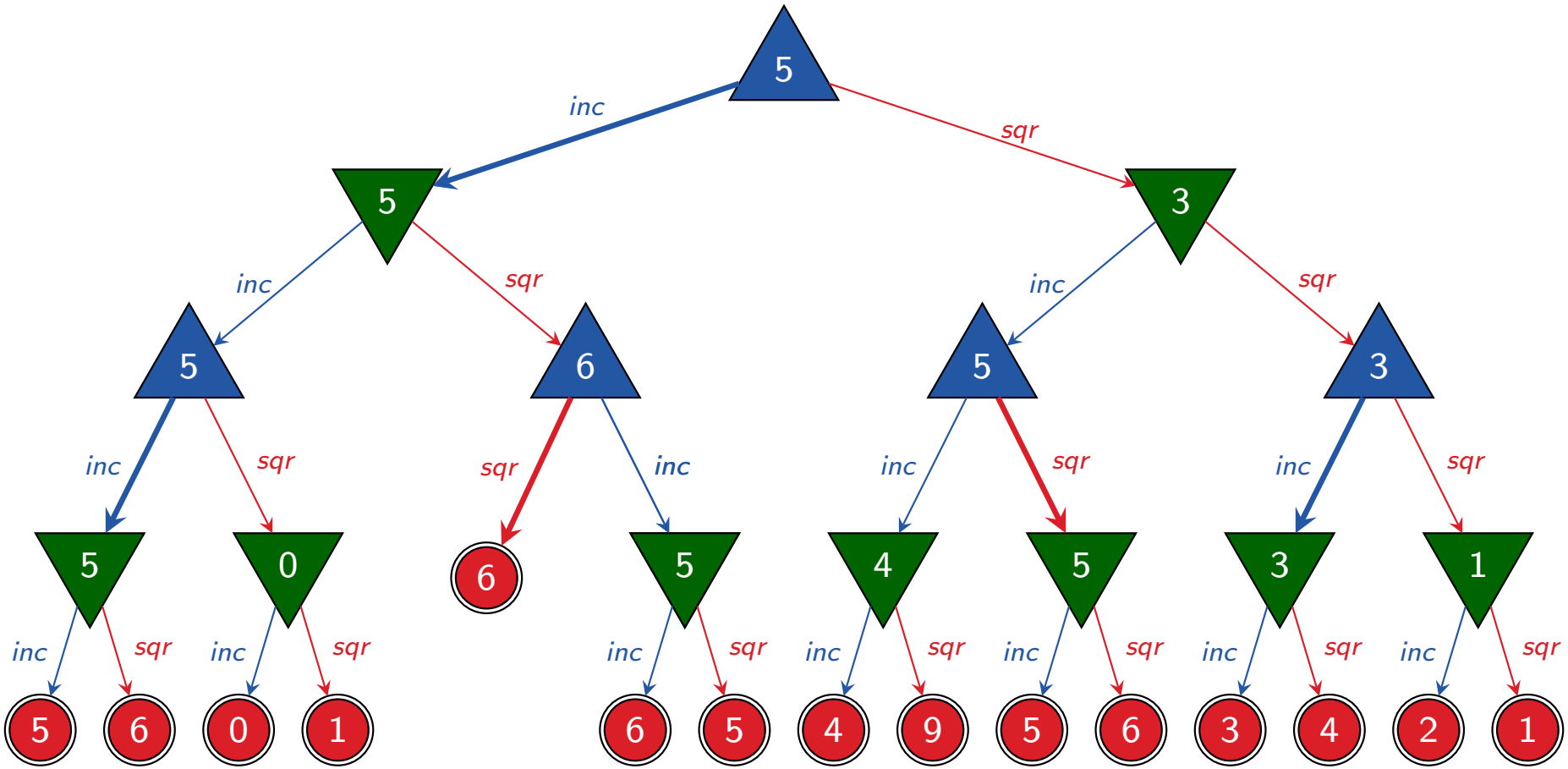
Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

Idea and Example



- depth-first search in game tree
- determine utility value of terminal position with utility function
- strategy: action that maximizes utility value (minimax decision)
- compute utility value of inner nodes

  from below to above through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children

Minimax: Pseudo-Code

function minimax(p)

if p is terminal position:
    return $\langle utility(p), \textbf{none} \rangle$
$best\_move := \textbf{none}$
if $player(p) = \text{MAX}$:
    $v := -\infty$
else:
    $v := \infty$
for each $\langle move, p' \rangle \in \text{succ}(p)$:
    $\langle v', best\_move' \rangle := minimax(p')$
    if ($player(p) = \text{MAX}$ **and** $v' > v$) **or**
       ($player(p) = \text{MIN}$ **and** $v' < v$):
        $v := v'$
        $best\_move := move$
return $\langle v, best\_move \rangle$

Discussion

- minimax is the simplest (decent) search algorithm for games
- yields optimal strategy (in the game-theoretic sense, i.e., under the assumption that the opponent plays perfectly)
- MAX obtains at least the utility value computed for the root, no matter how MIN plays
- if MIN plays perfectly, MAX obtains exactly the computed value

Limitations of Minimax

What if the size of the game tree is too big for minimax?

$\rightsquigarrow$ heuristic alpha-beta search

- heuristics (evaluation functions): rest of this chapter
- alpha-beta search: next chapter

Evaluation Functions

Evaluation Functions

Definition (evaluation function)

Let $\mathcal{S}$ be a game with set of positions S.
An evaluation function for $\mathcal{S}$ is a function

$$h : S \to \mathbb{R}$$

which assigns a real-valued number to each position $s \in S$.

Looks familiar? Commonalities? Differences?

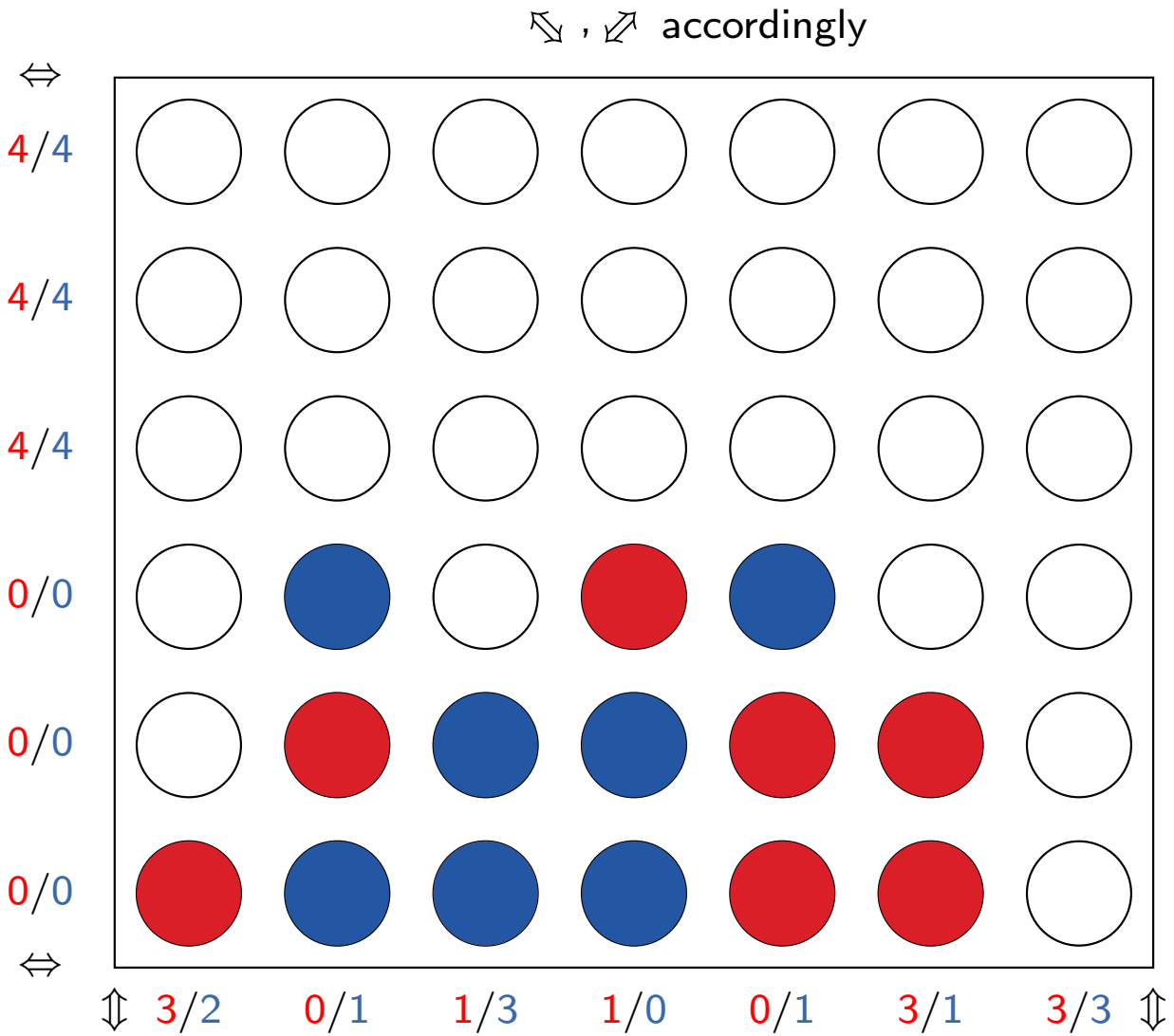
Intuition

- problem: game tree too big
- idea: search only up to predefined depth
- depth reached: estimate the utility value according to heuristic criteria (as if terminal position had been reached)

accuracy of evaluation function is crucial

- high values should relate to high "winning chances"
- at the same time, the evaluation should be efficiently computable in order to be able to search deeply

Example: Connect Four



evalution function: difference of number of possible lines of four

General Method: Linear Evaluation Functions

expert knowledge often represented with weighted linear functions:

$$h(s) = w_0 + w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s),$$

where w_i are weights and f_i are features.

General Method: Linear Evaluation Functions

expert knowledge often represented with weighted linear functions:

$$h(s) = w_0 + w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s),$$

where w_i are weights and f_i are features.

- assumes that feature contributions are mutually independent (usually wrong but acceptable assumption)
- features are (usually) provided by human experts
- weights provided by human experts or learned automatically

General Method: Linear Evaluation Functions

expert knowledge often represented with weighted linear functions:

$$h(s) = w_0 + w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s),$$

where w_i are weights and f_i are features.

example: evaluation function in chess (cf. Lolli 1763)

feature	f_p^{player}	f_k^{player}	f_b^{player}	f_r^{player}	f_q^{player}
no. of pieces	pawn	knight	bishop	rook	queen
weight for MAX	1	3	3	5	9
weight for MIN	−1	−3	−3	−5	−9

often additional features based on pawn structure, mobility, ...

$$\rightsquigarrow h(s) = f_p^{\text{MAX}}(s) + 3f_k^{\text{MAX}}(s) + 3f_b^{\text{MAX}}(s) + 5f_r^{\text{MAX}}(s) + 9f_q^{\text{MAX}}(s) - f_p^{\text{MIN}}(s) - 3f_k^{\text{MIN}}(s) - 3f_b^{\text{MIN}}(s) - 5f_r^{\text{MIN}}(s) - 9f_q^{\text{MIN}}(s)$$

General Method: State Value Networks

alternative: evaluation functions based on neural networks

- value network takes position features as input (usually provided by human experts)
- and outputs utility value prediction
- weights of network learned automatically

General Method: State Value Networks

alternative: evaluation functions based on neural networks

- value network takes position features as input
  (usually provided by human experts)
- and outputs utility value prediction
- weights of network learned automatically

example: value network of AlphaGo

- start with policy network trained on human expert games
- train sequence of policy networks by self-play against earlier version
- final step: convert to utility value network
  (slightly worse informed but much faster)

$\rightsquigarrow$ Mastering the game of Go with deep neural networks and tree search
(Silver et al., 2016)

How Deep Shall We Search?

- objective: search as deeply as possible within a given time
- problem: search time difficult to predict
- solution: iterative deepening
  - sequence of searches of increasing depth
  - time expires: return result of previously finished search
  - overhead acceptable (⇝ Chapter B8)
- refinement: search deeper in "turbulent" states
  (i.e., with strong fluctuations of the evaluation function)
  ⇝ quiescence search
  - example chess: deepen the search after capturing moves

Summary

Summary

- Minimax is a tree search algorithm that plays perfectly (in the game-theoretic sense), but its complexity is $O(b^d)$ (branching factor b, search depth d).
- In practice, the search depth must be bounded
  $\rightsquigarrow$ apply evaluation functions.

Foundations of Artificial Intelligence

G3. Board Games: Alpha-Beta Search

Malte Helmert

University of Basel

May 19, 2025

Board Games: Overview

chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Configurations

Limitations of Minimax

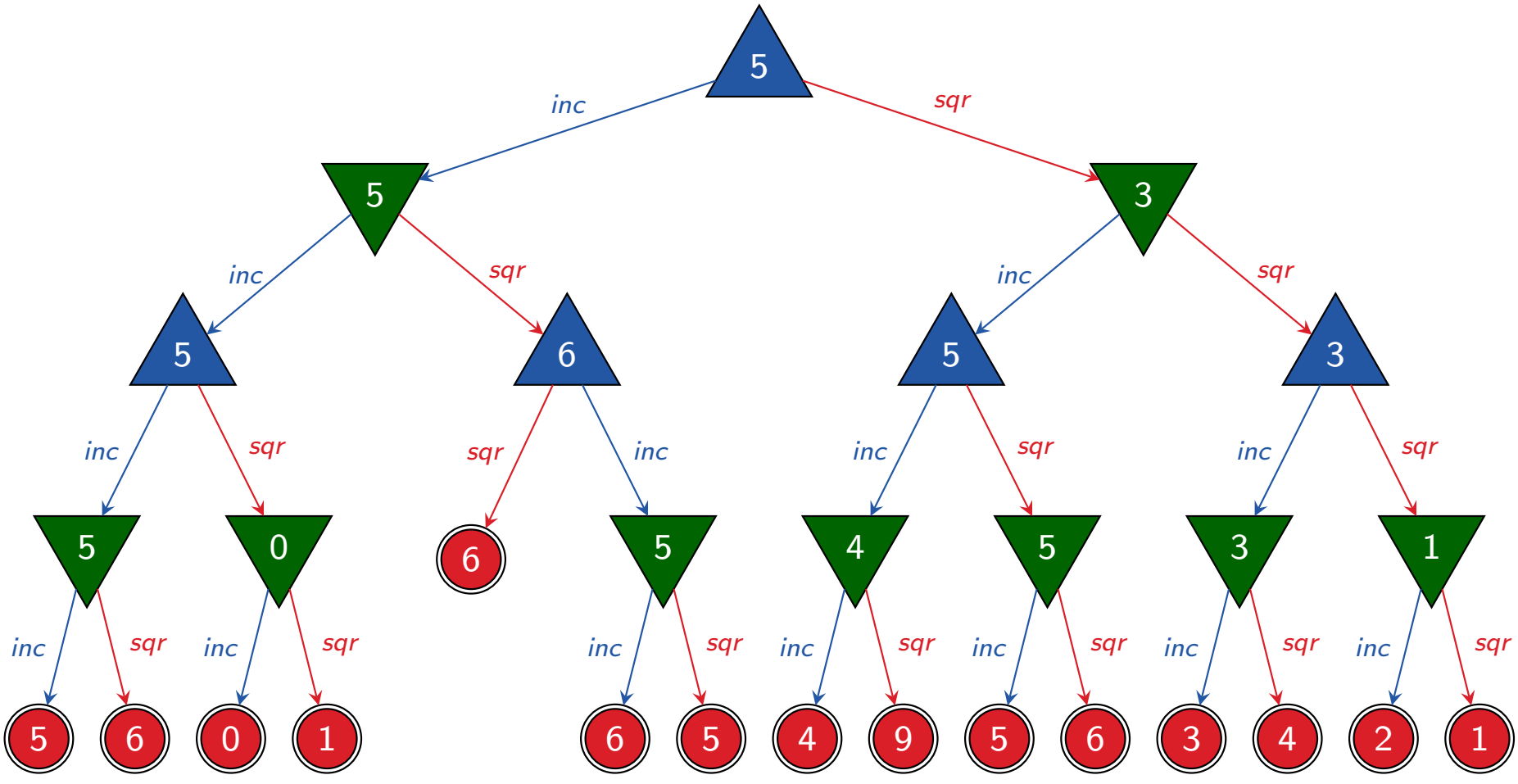
What if the size of the game tree is too big for minimax?

$\rightsquigarrow$ heuristic alpha-beta search

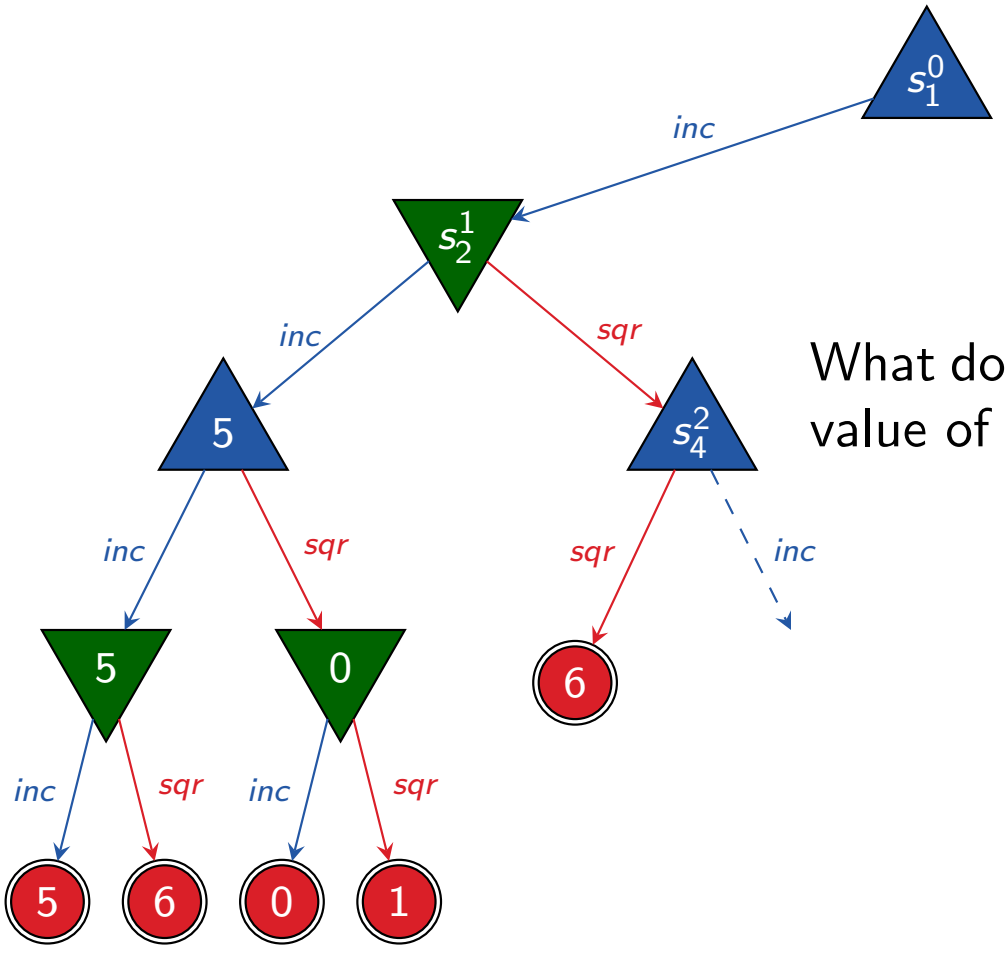
- heuristics (evaluation functions): previous chapter
- alpha-beta search: this chapter

Alpha-Beta Search

Can We Save Search Effort?



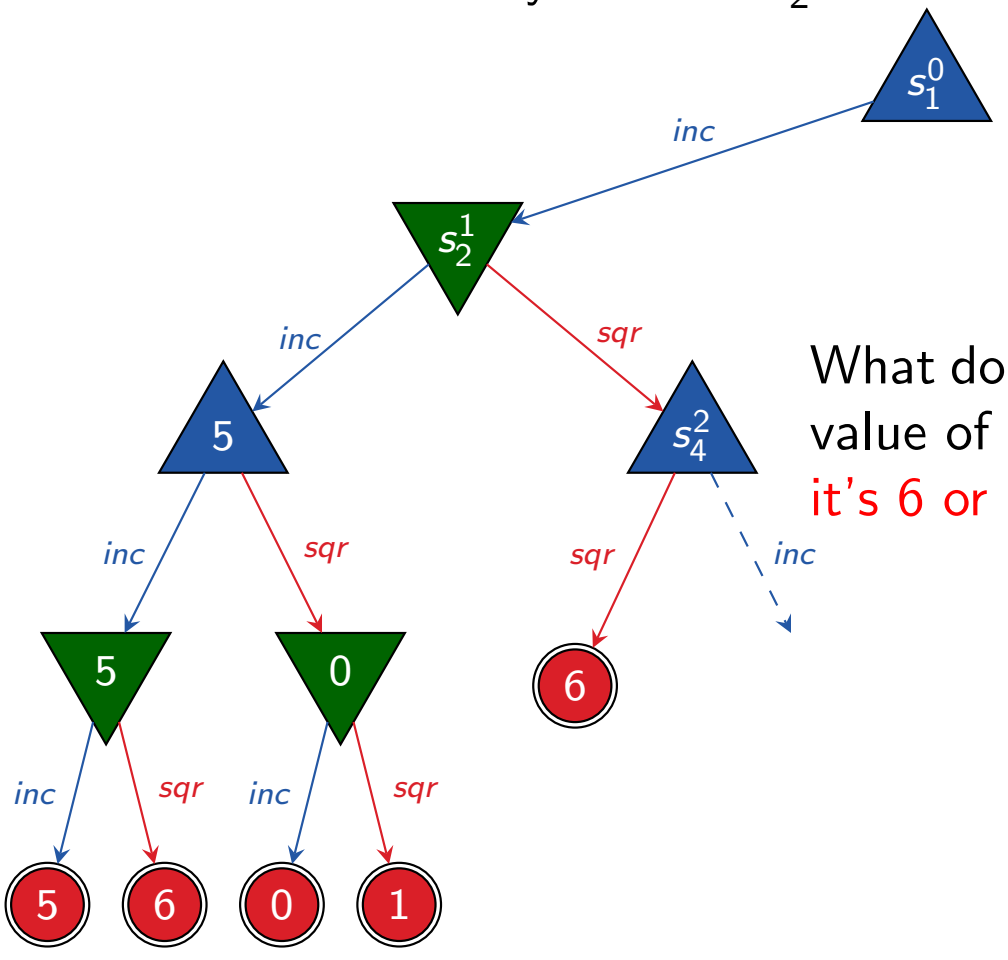
Can We Save Search Effort?



What do we know about the utility value of s_4^2 in this situation?

Can We Save Search Effort?

And about the utility value of s_2^1?

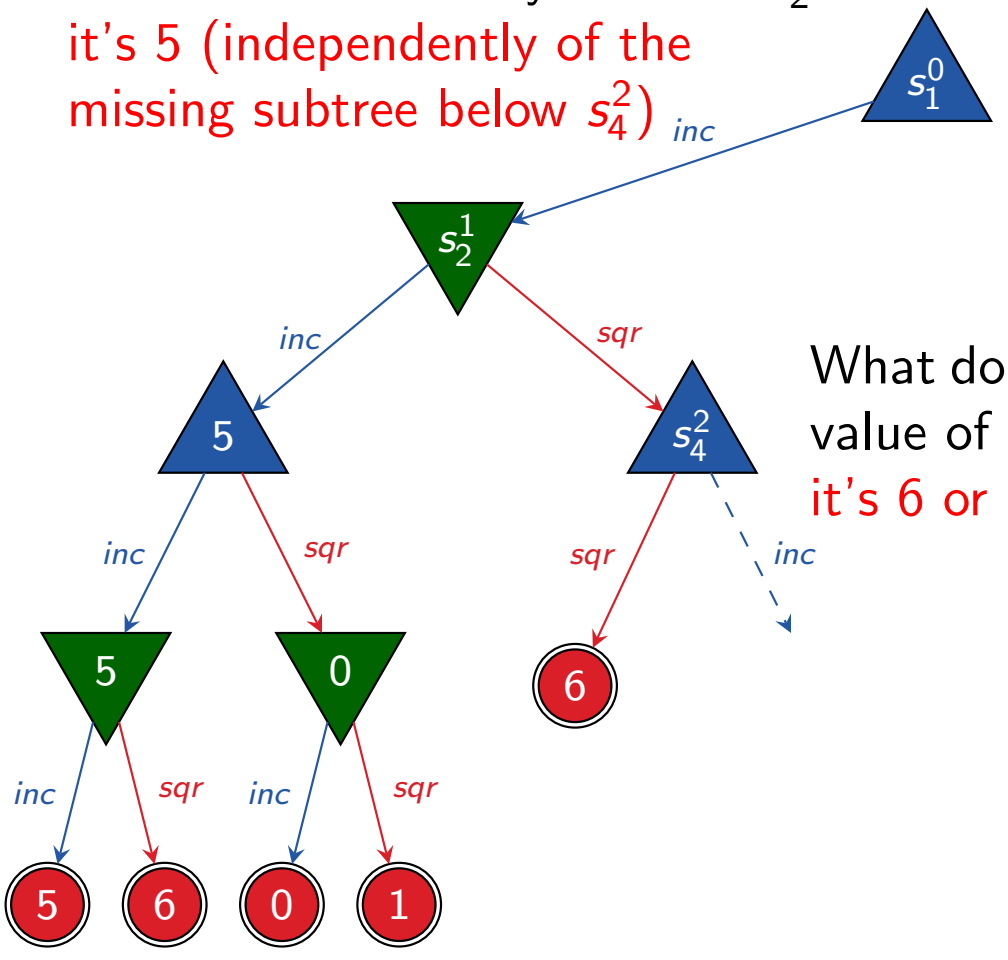


What do we know about the utility value of s_4^2 in this situation?

it's 6 or higher

Can We Save Search Effort?

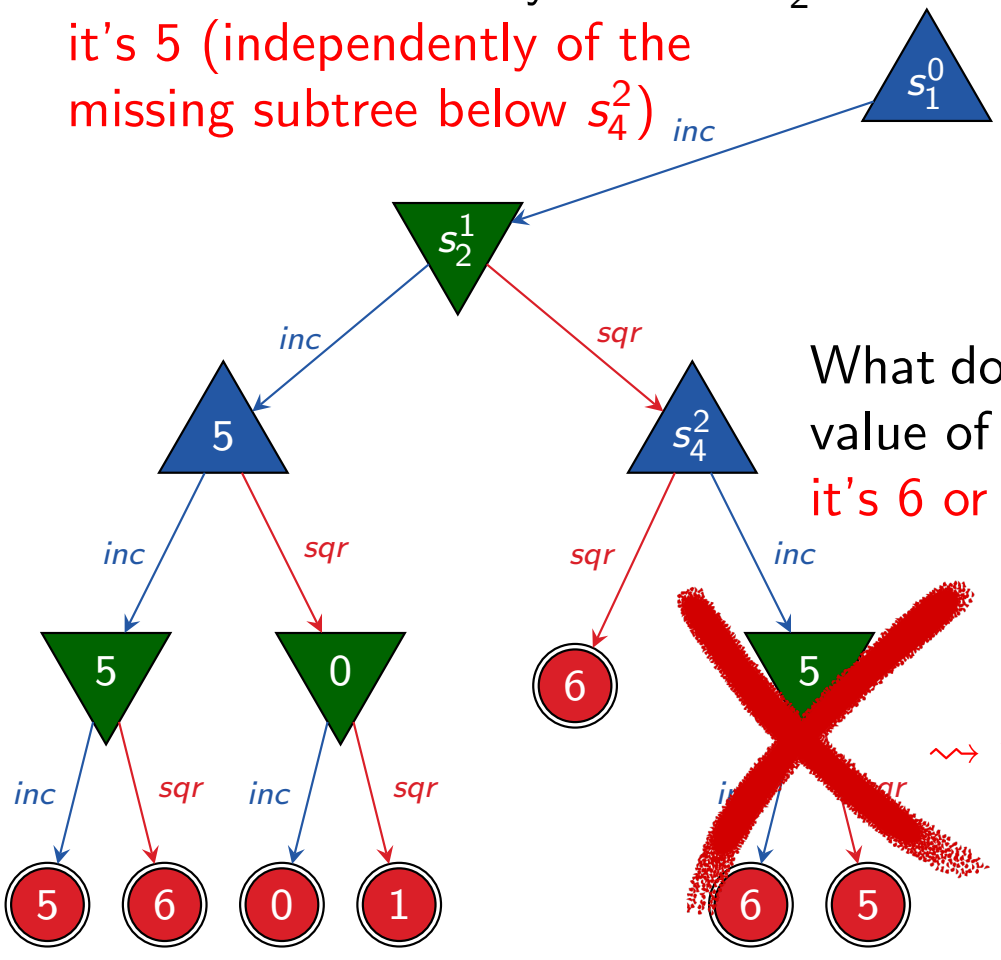
And about the utility value of s_2^1?
it's 5 (independently of the missing subtree below s_4^2)



What do we know about the utility value of s_4^2 in this situation?
it's 6 or higher

Can We Save Search Effort?

And about the utility value of s_2^1?
it's 5 (independently of the missing subtree below s_4^2)



What do we know about the utility value of s_4^2 in this situation?
it's 6 or higher

$\rightsquigarrow$ we don't have to look at this

Idea

idea: for every search node, use two values α and β such that we know that the subtree rooted at the node

- is irrelevant if its utility is $\leq \alpha$
  because MAX will prevent entering it when playing optimally
- is irrelevant if its utility is $\geq \beta$
  because MIN will prevent entering it when playing optimally

We can prune every node with $\alpha \geq \beta$
because it must be irrelevant (no matter what its utility is).

Alpha-Beta Search: Pseudo Code

- algorithm skeleton the same as minimax
- function signature extended by two variables α and β

function alpha-beta-main(p)

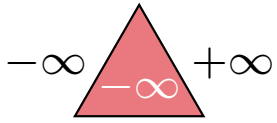
$\langle v, move \rangle$:= alpha-beta($p, -\infty, +\infty$)
return *move*

Alpha-Beta Search: Pseudo-Code

function alpha-beta(p, α, β)

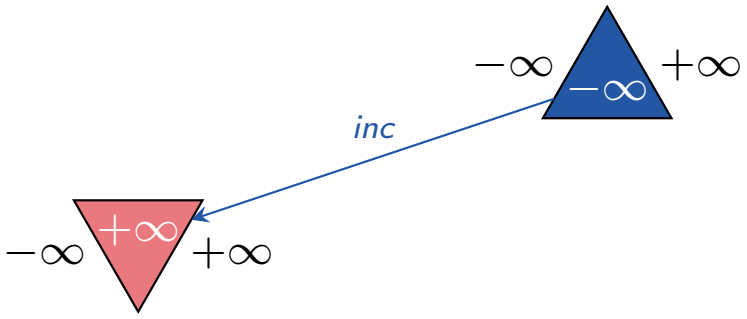
if p is terminal position:
    return $\langle utility(p), \textbf{none} \rangle$
initialize v and *best_move* [as in minimax]
for each $\langle move, p' \rangle \in succ(p)$:
    $\langle v', best\_move' \rangle := alpha\text{-}beta(p', \alpha, \beta)$
    update v and *best_move* [as in minimax]
    if $player(p) = \text{MAX}$:
        if $v \geq \beta$:
            return $\langle v, \textbf{none} \rangle$
        $\alpha := \max\{\alpha, v\}$
    if $player(p) = \text{MIN}$:
        if $v \leq \alpha$:
            return $\langle v, \textbf{none} \rangle$
        $\beta := \min\{\beta, v\}$
return $\langle v, best\_move \rangle$

Example



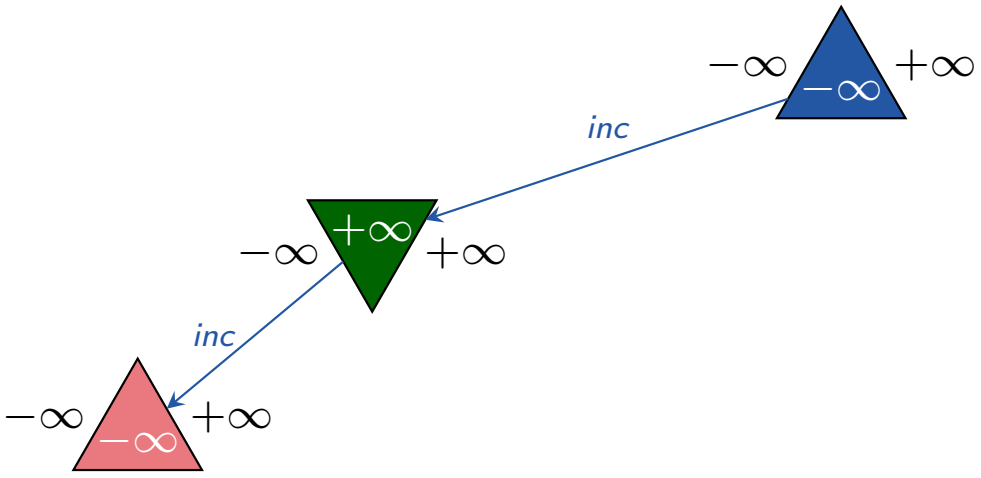
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



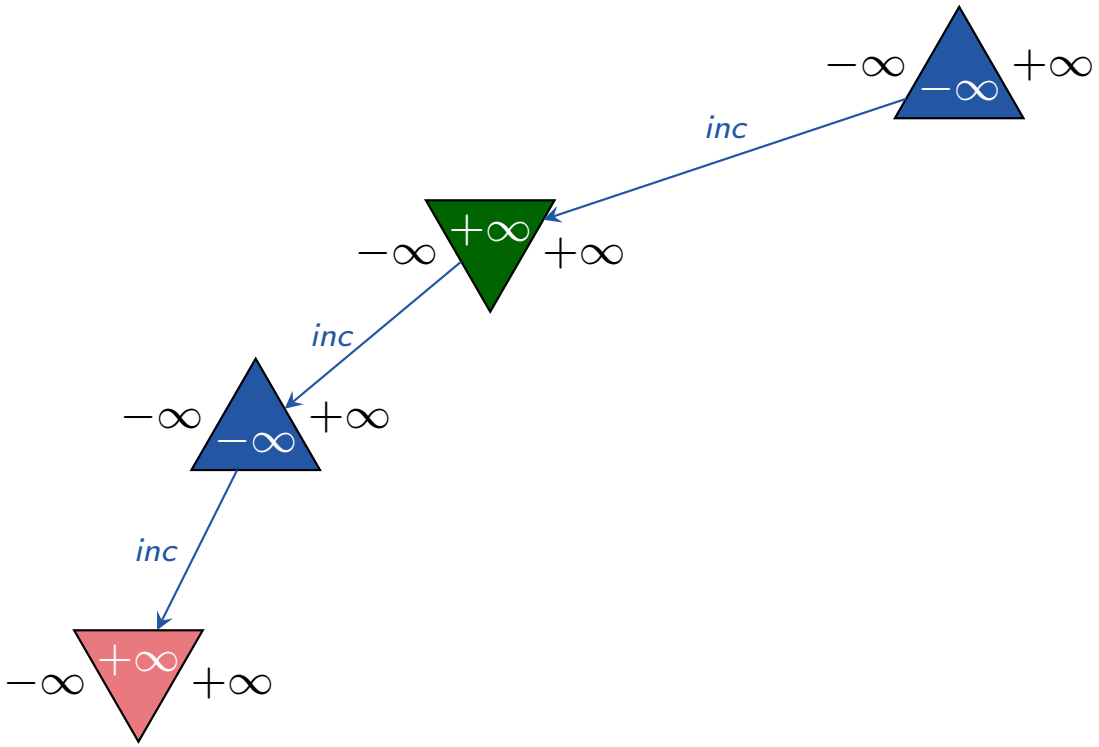
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



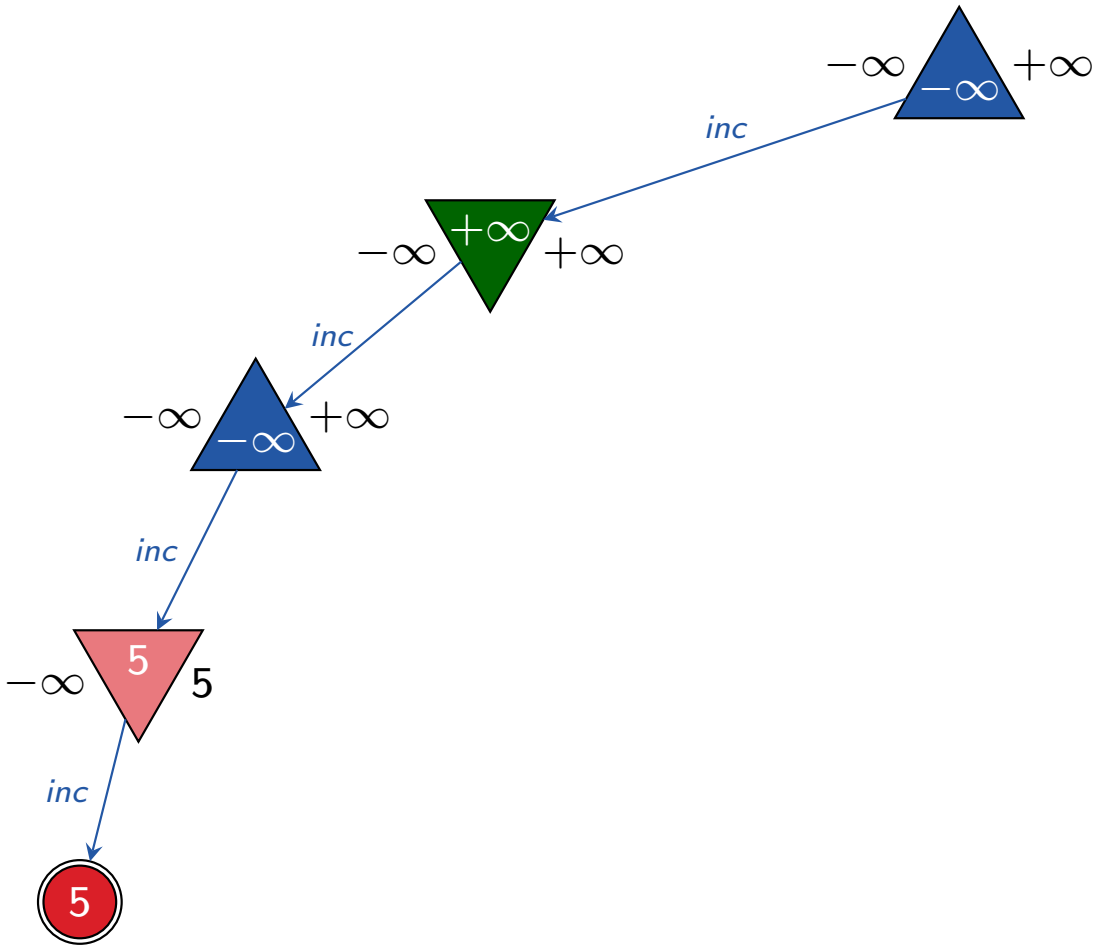
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



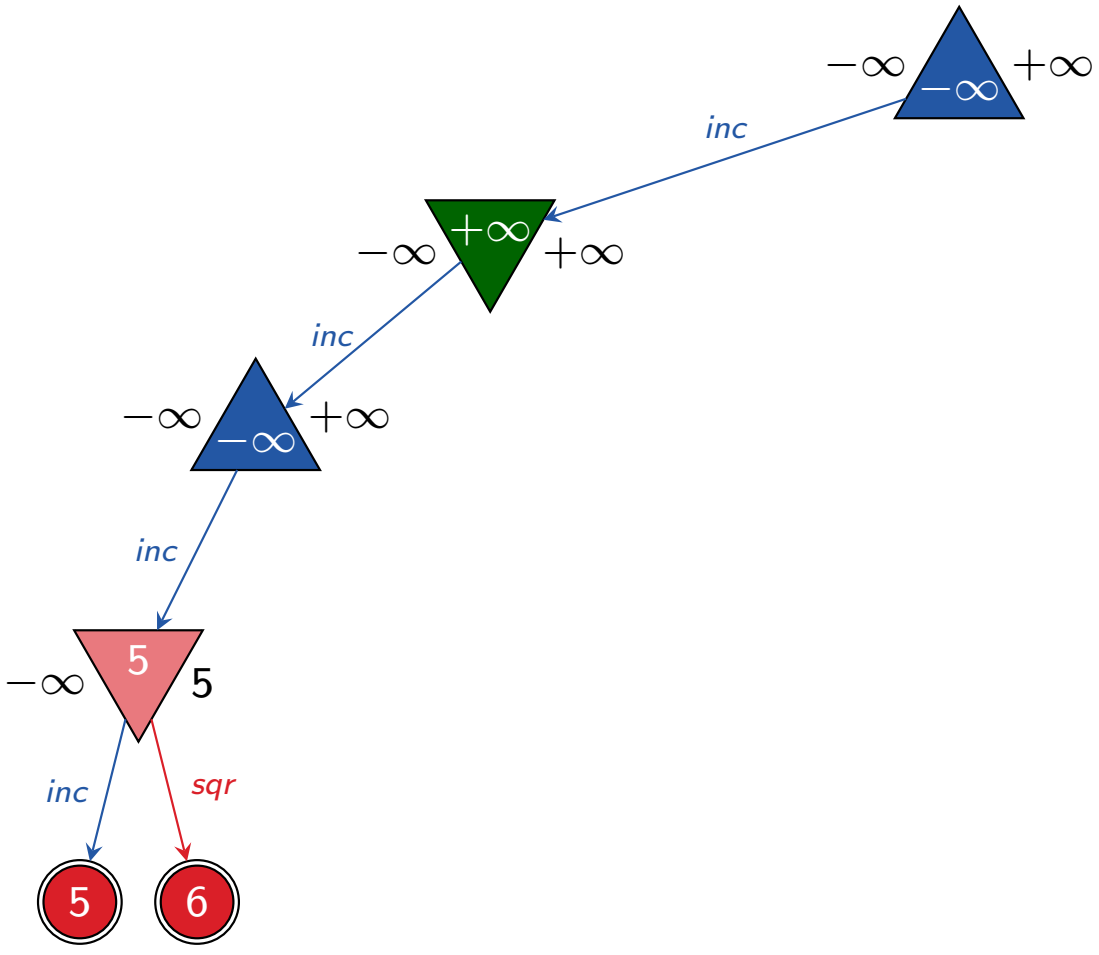
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



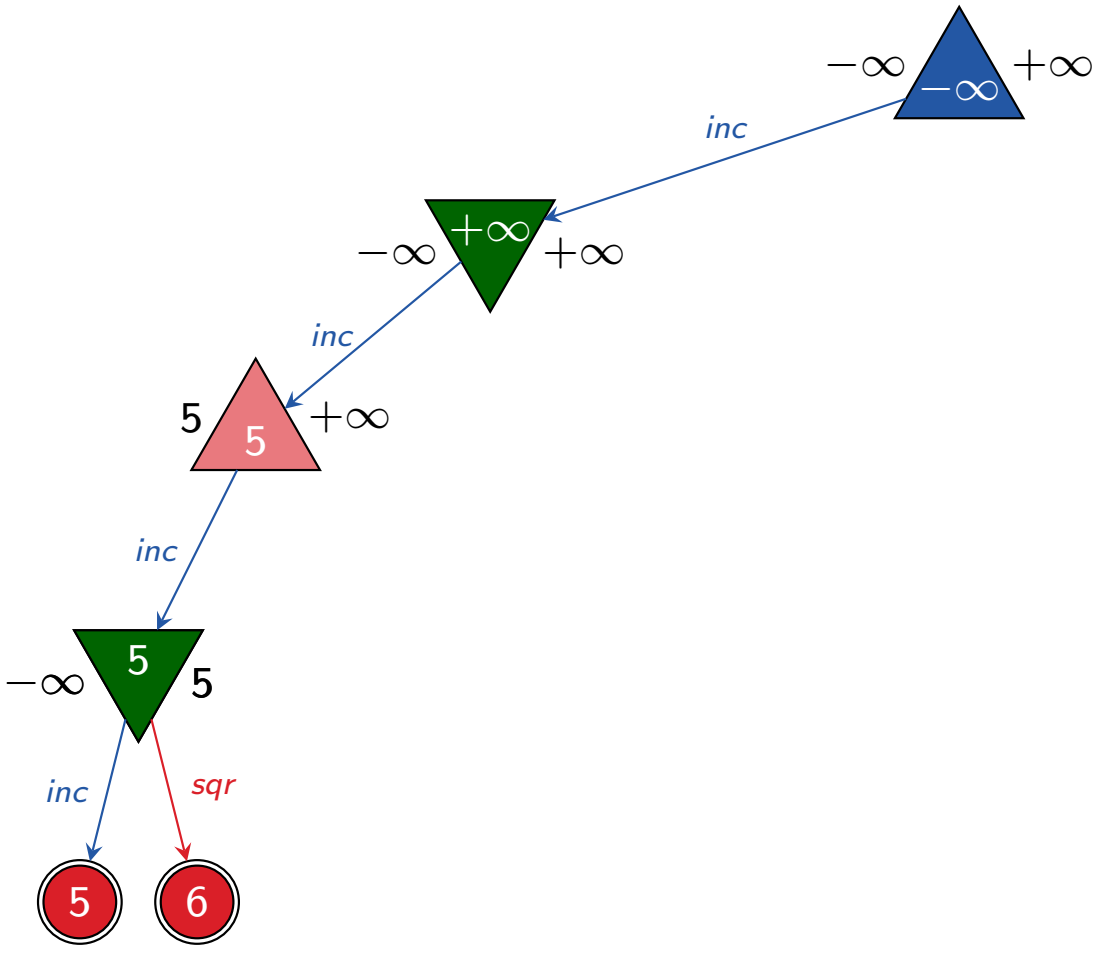
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



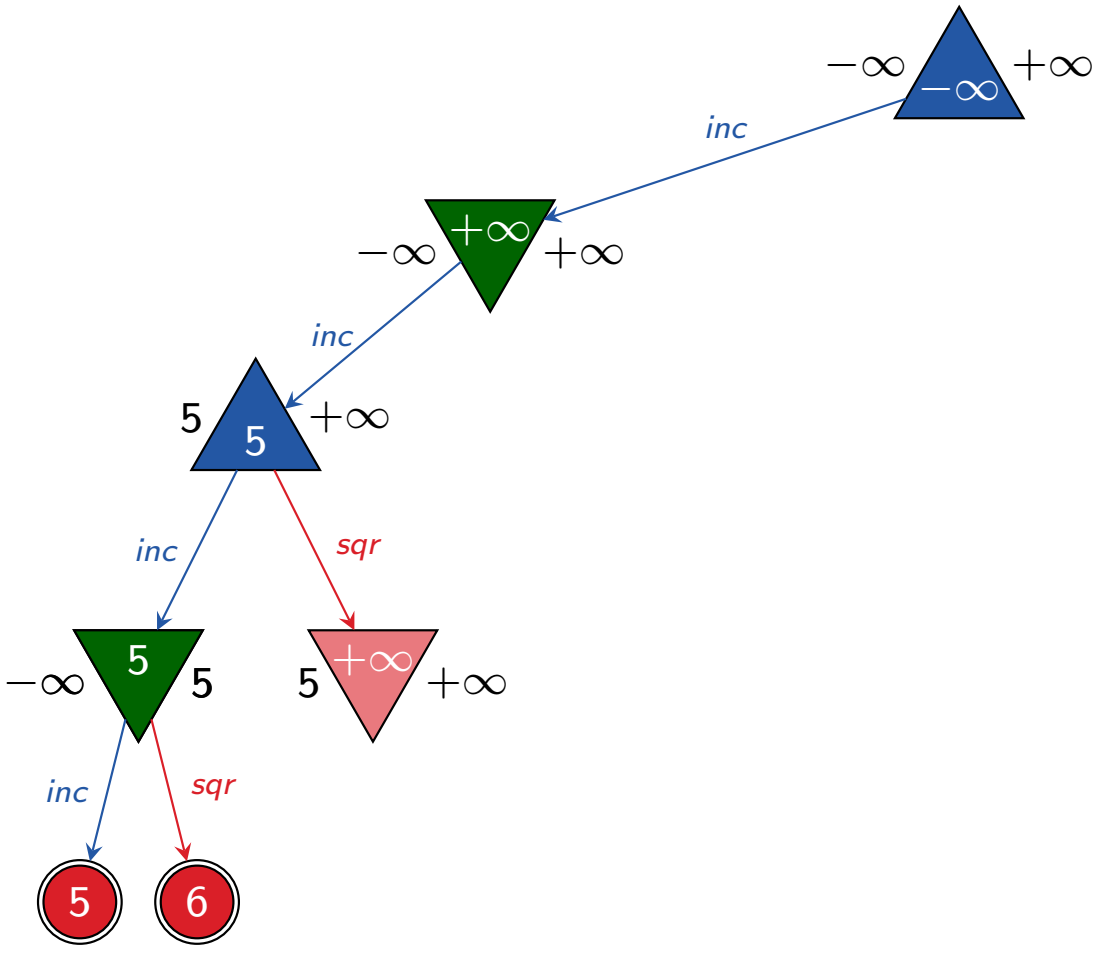
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



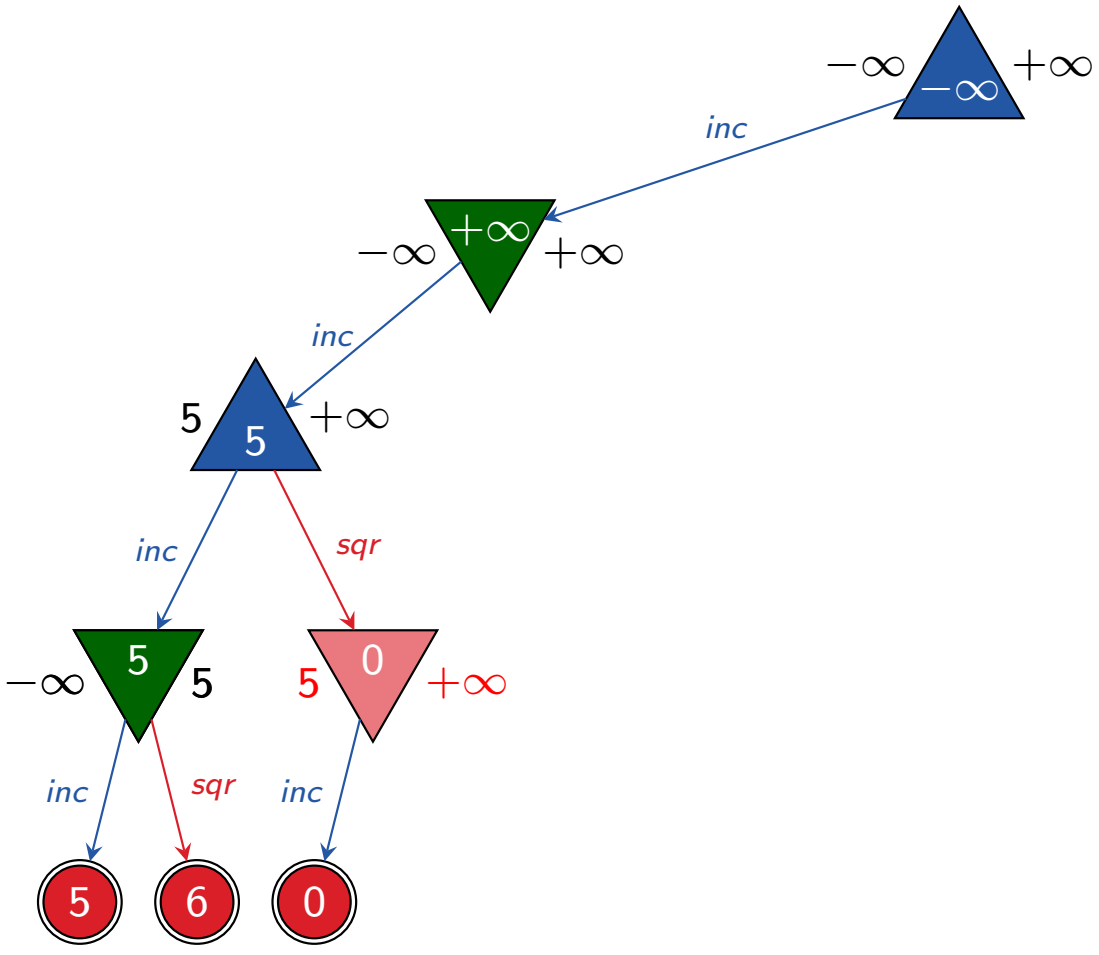
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



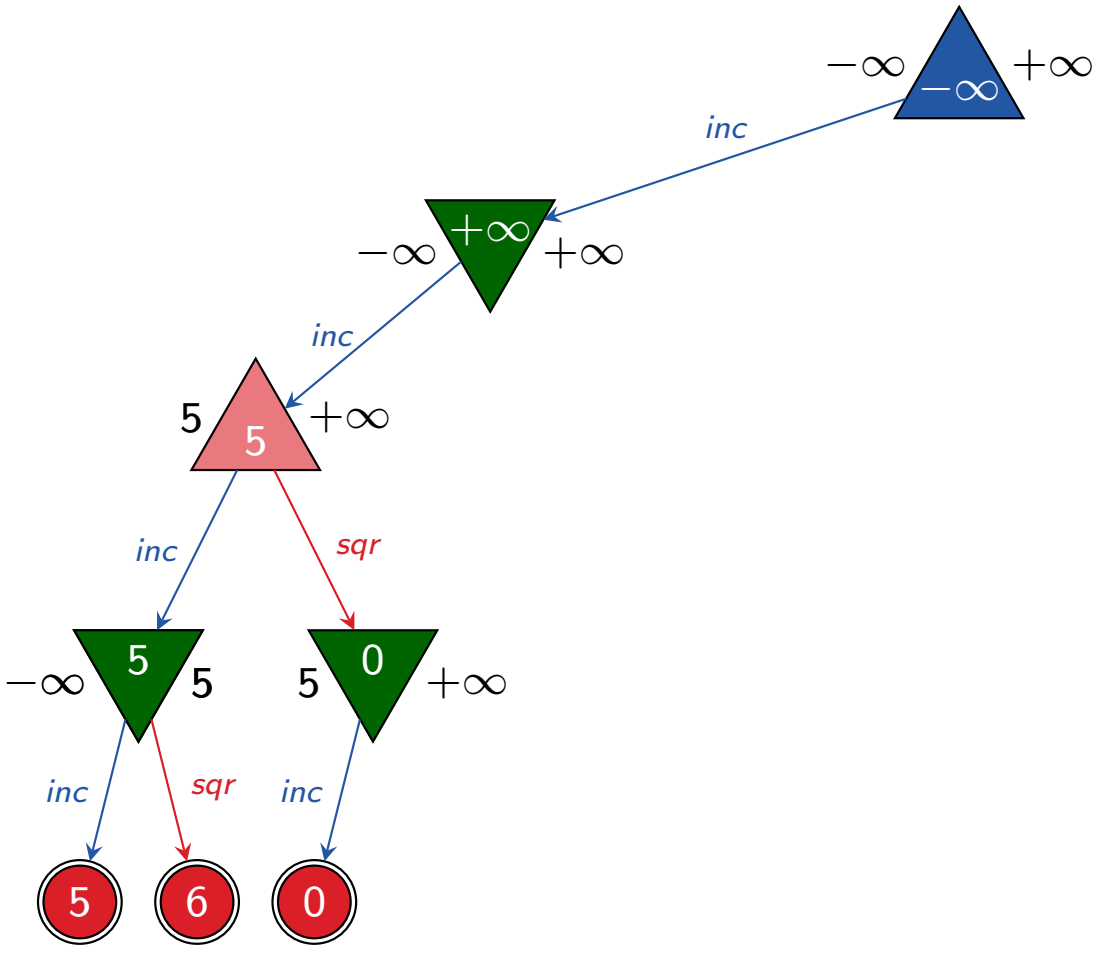
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



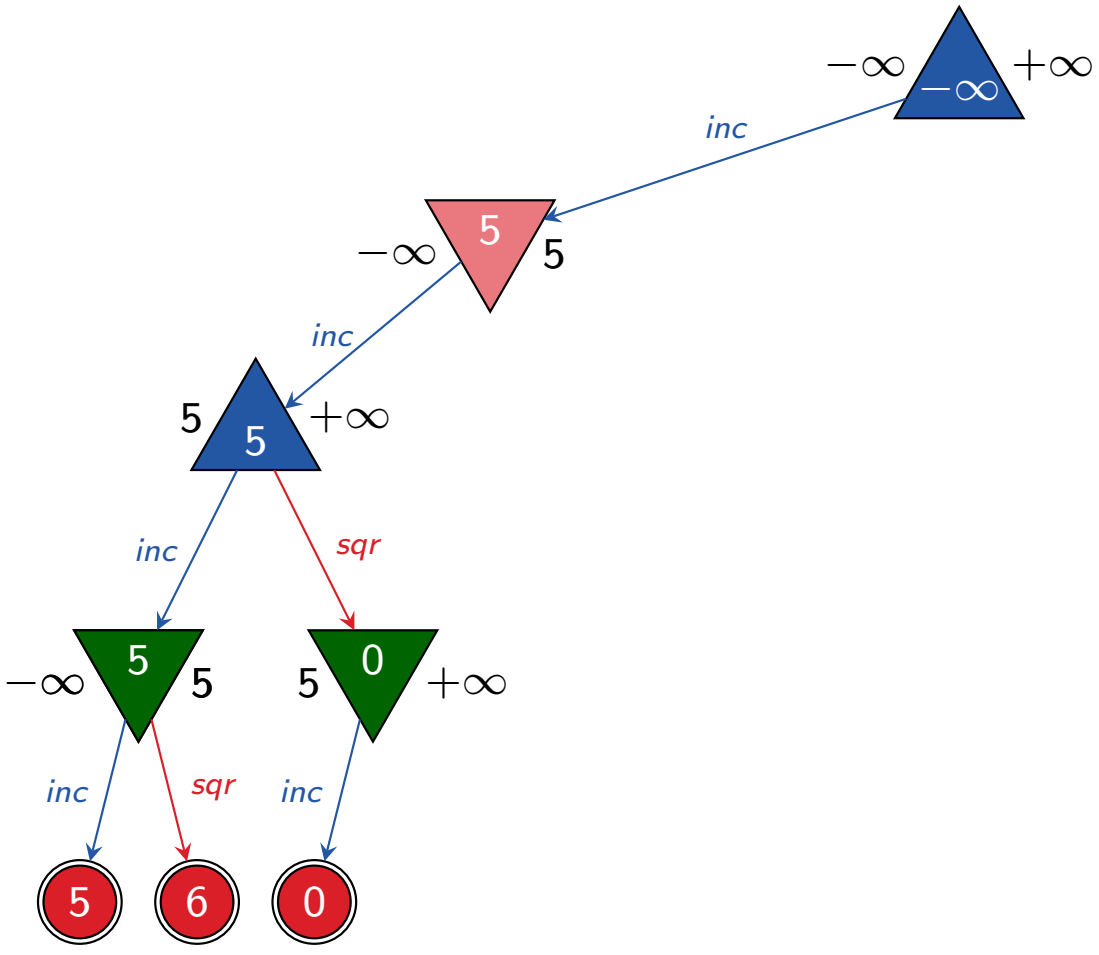
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



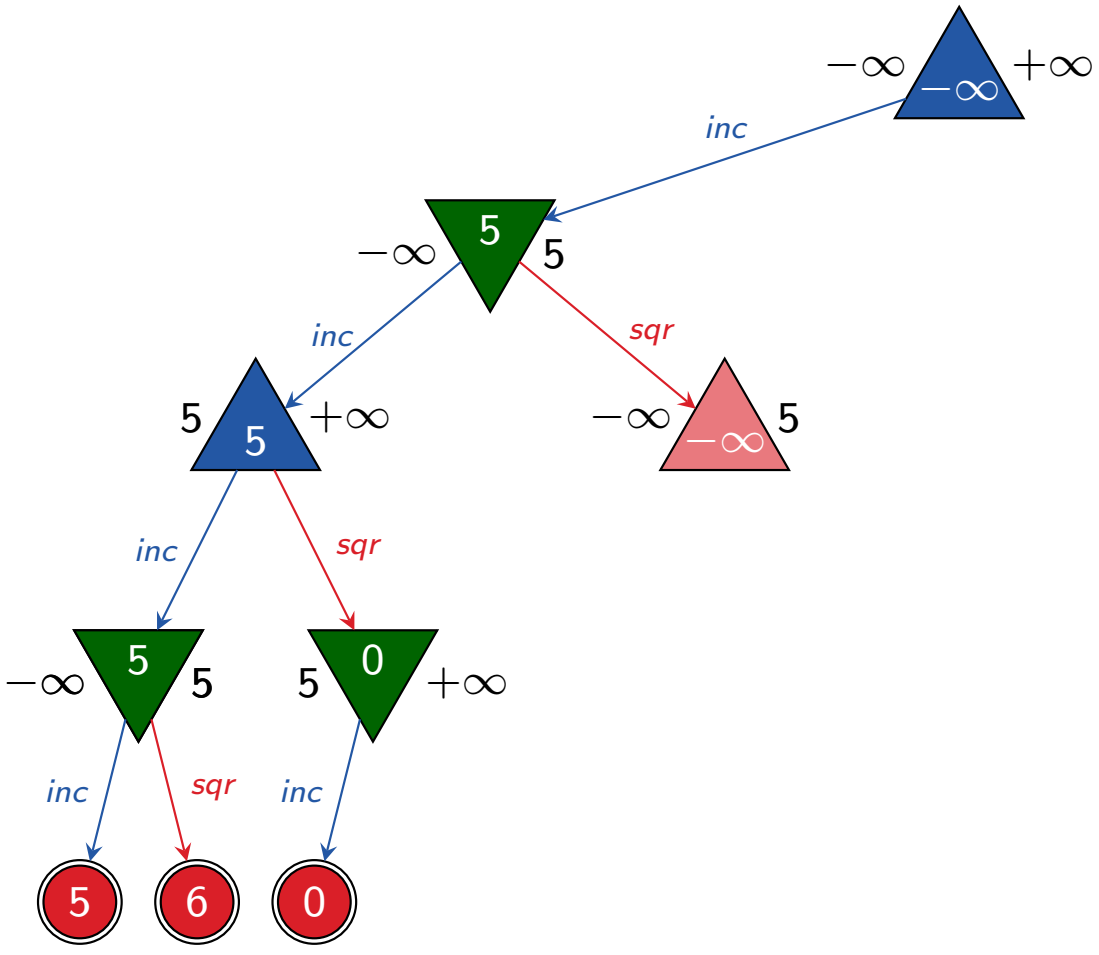
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



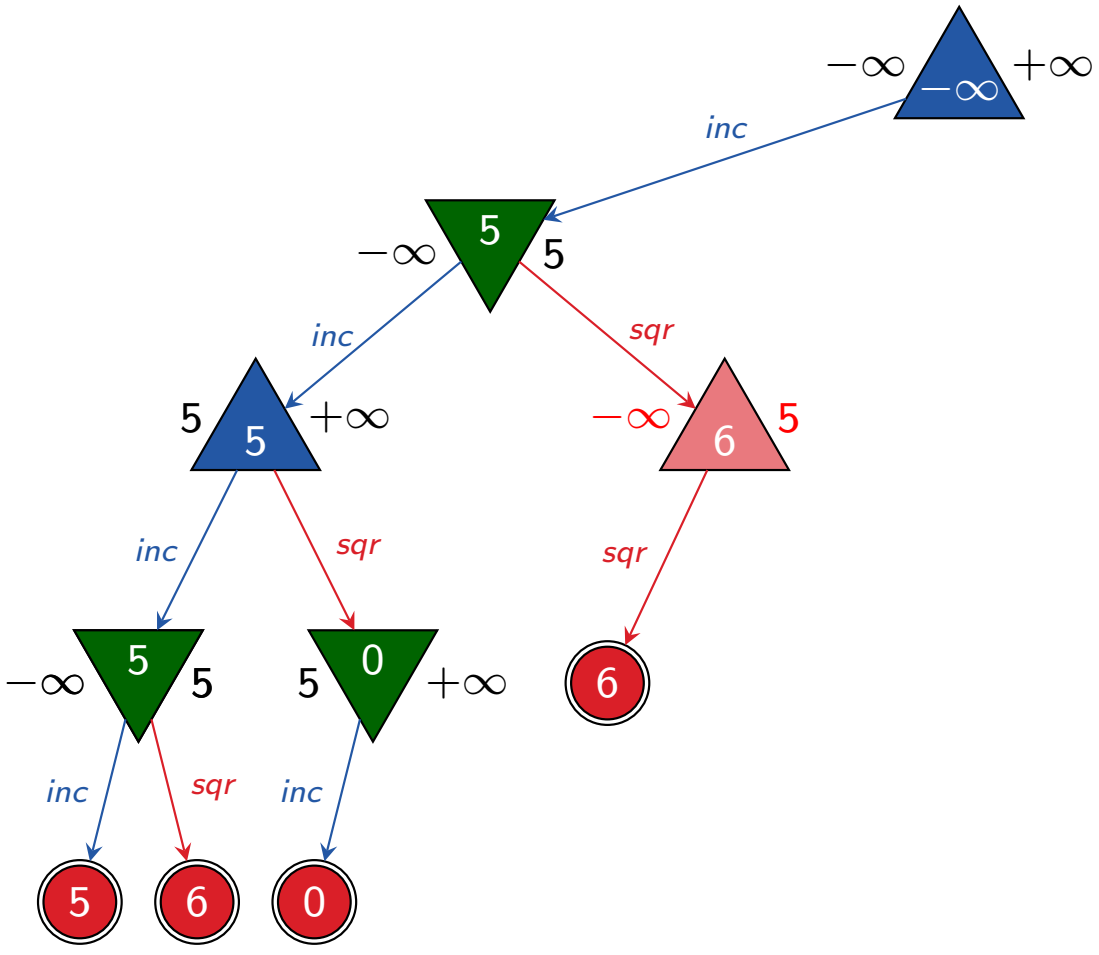
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



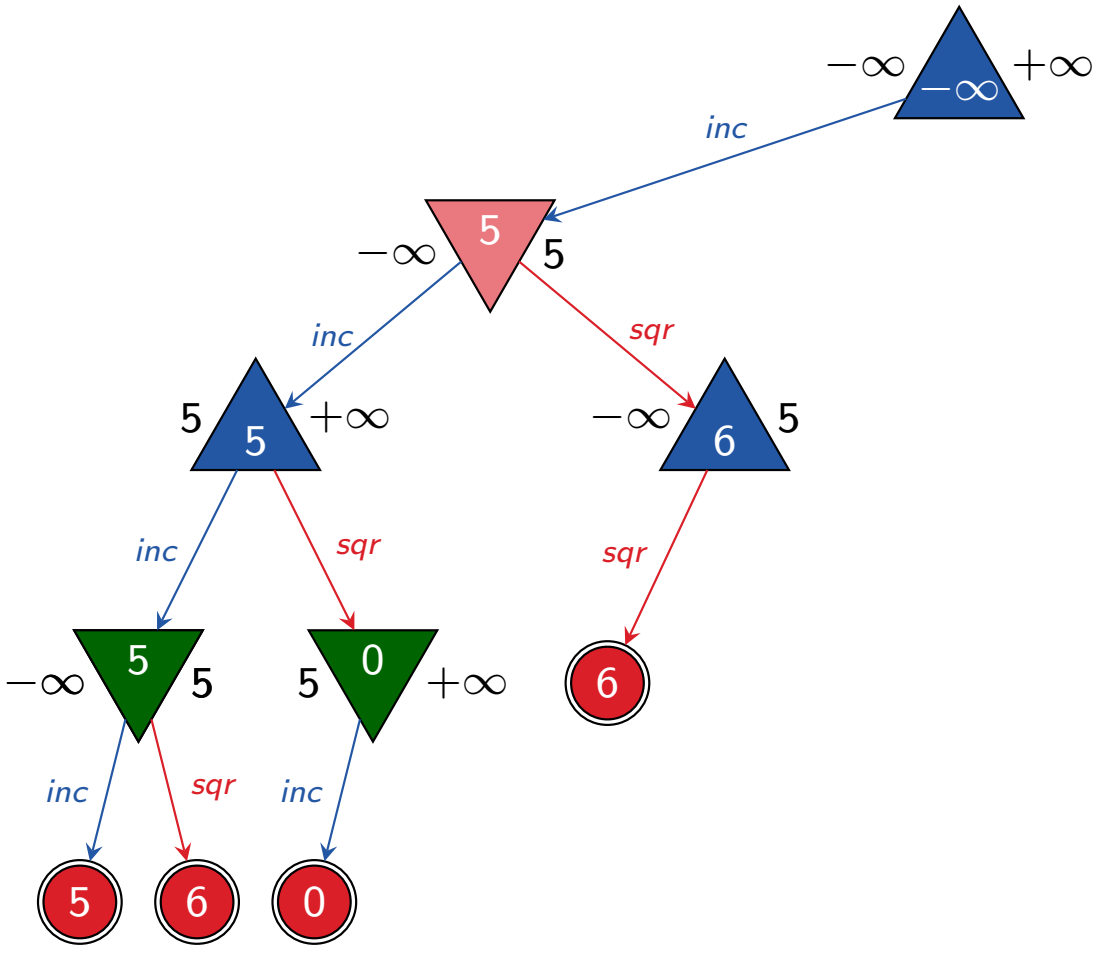
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



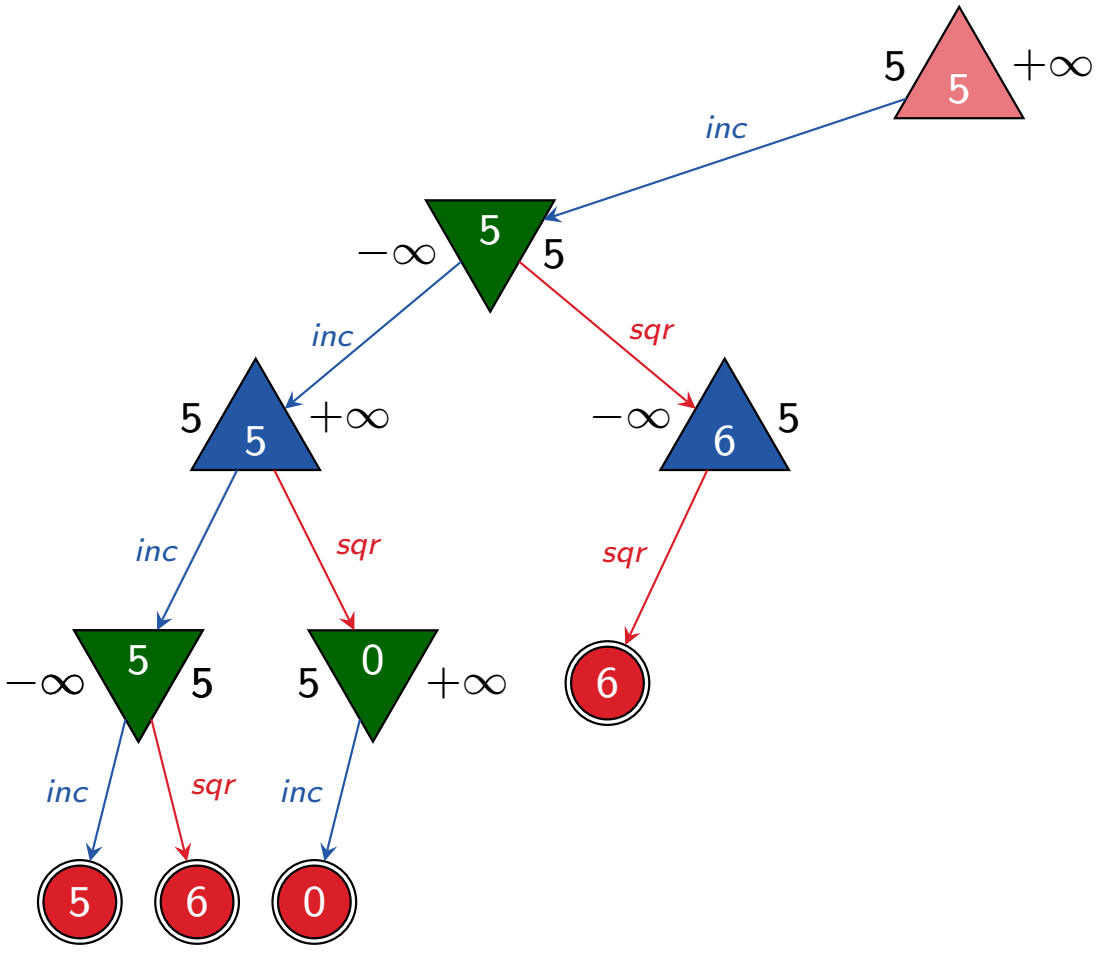
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



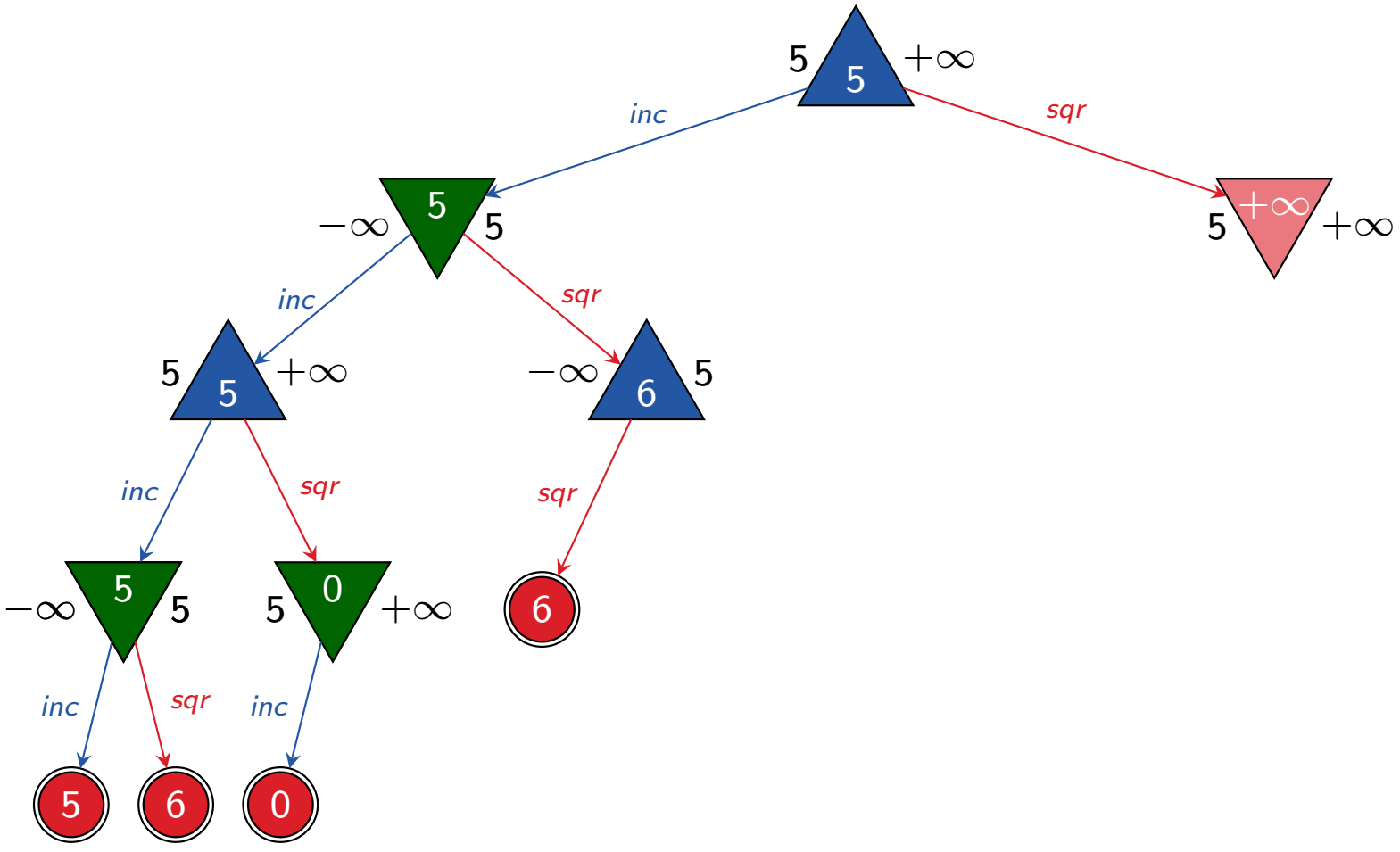
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



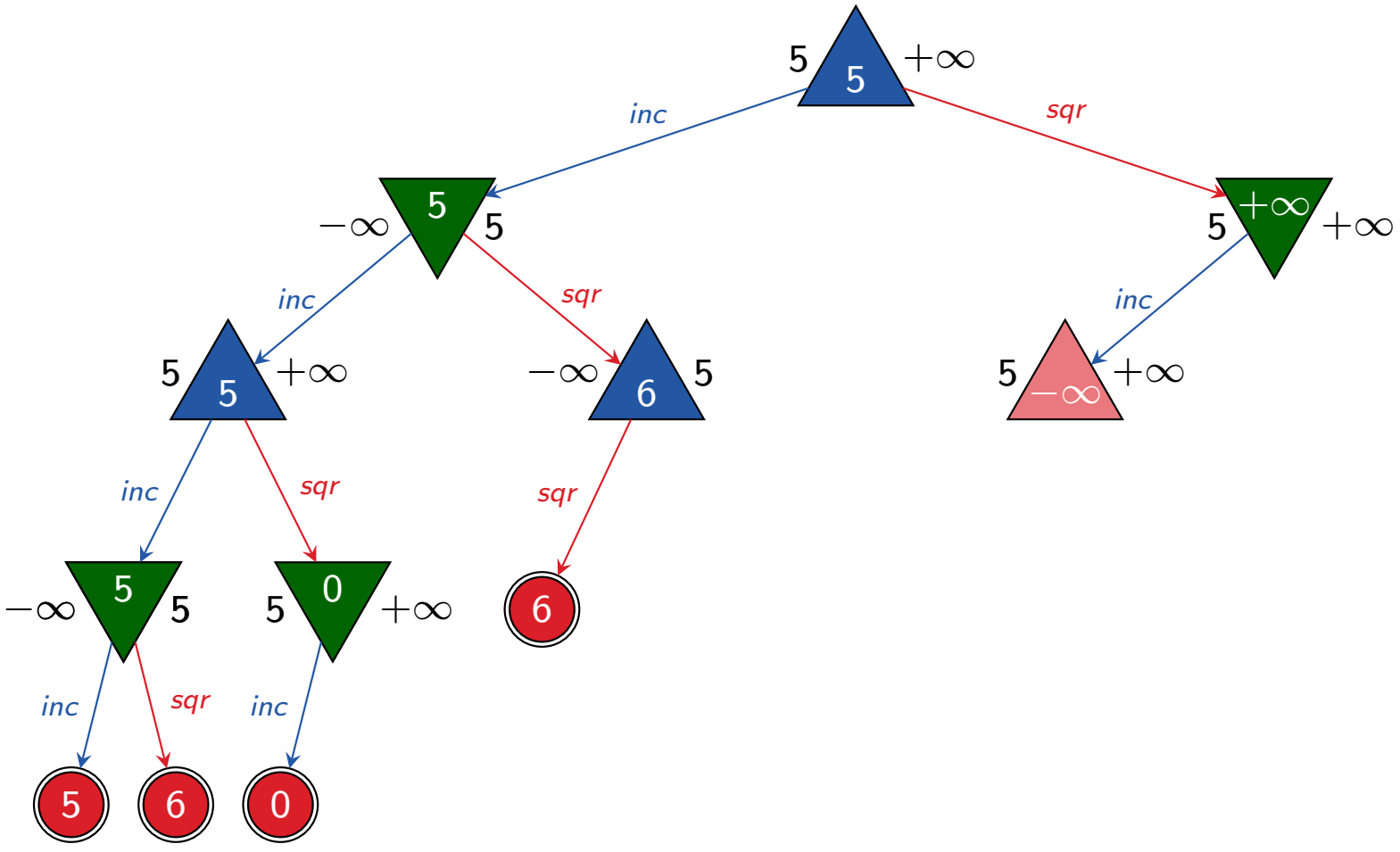
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



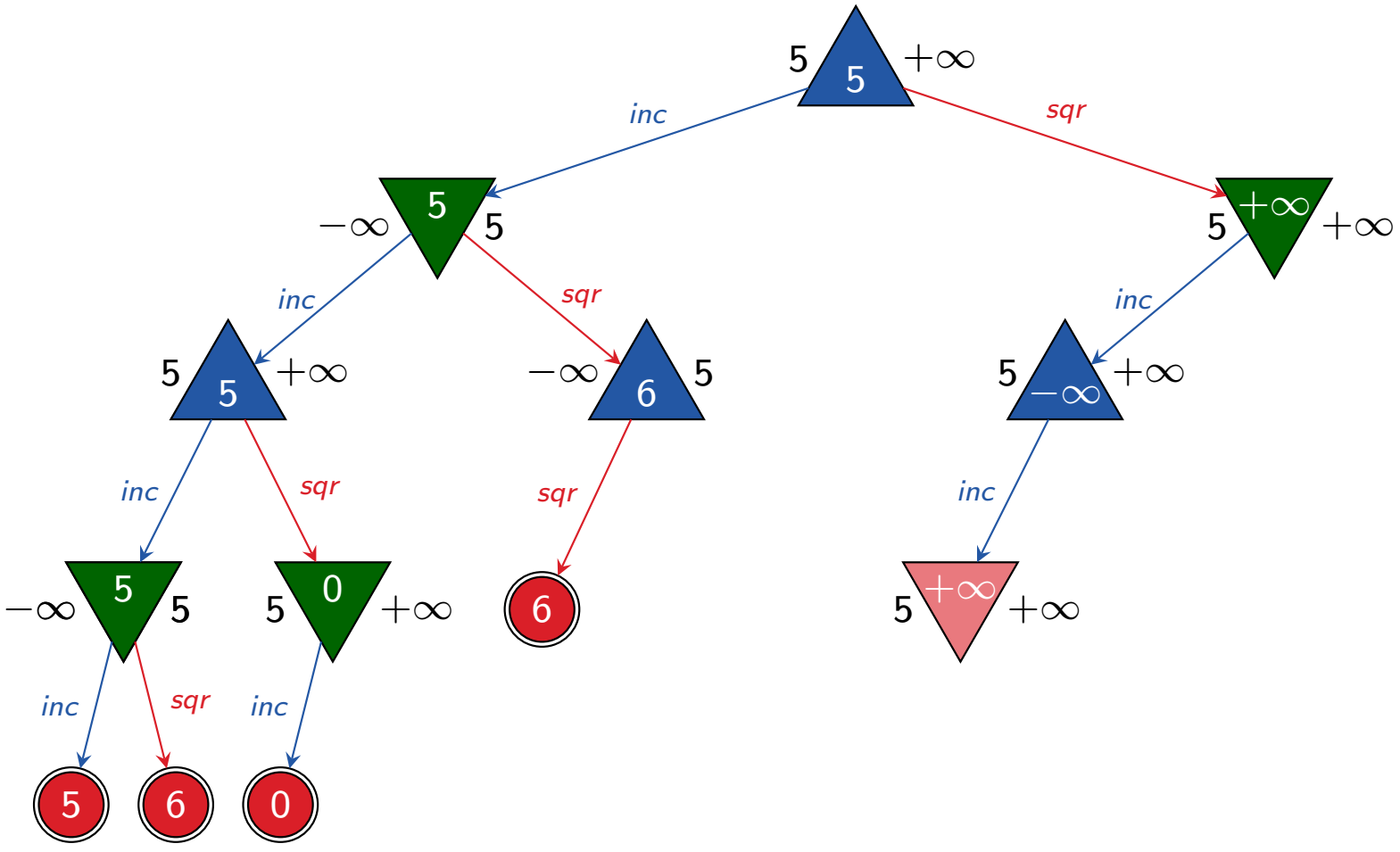
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



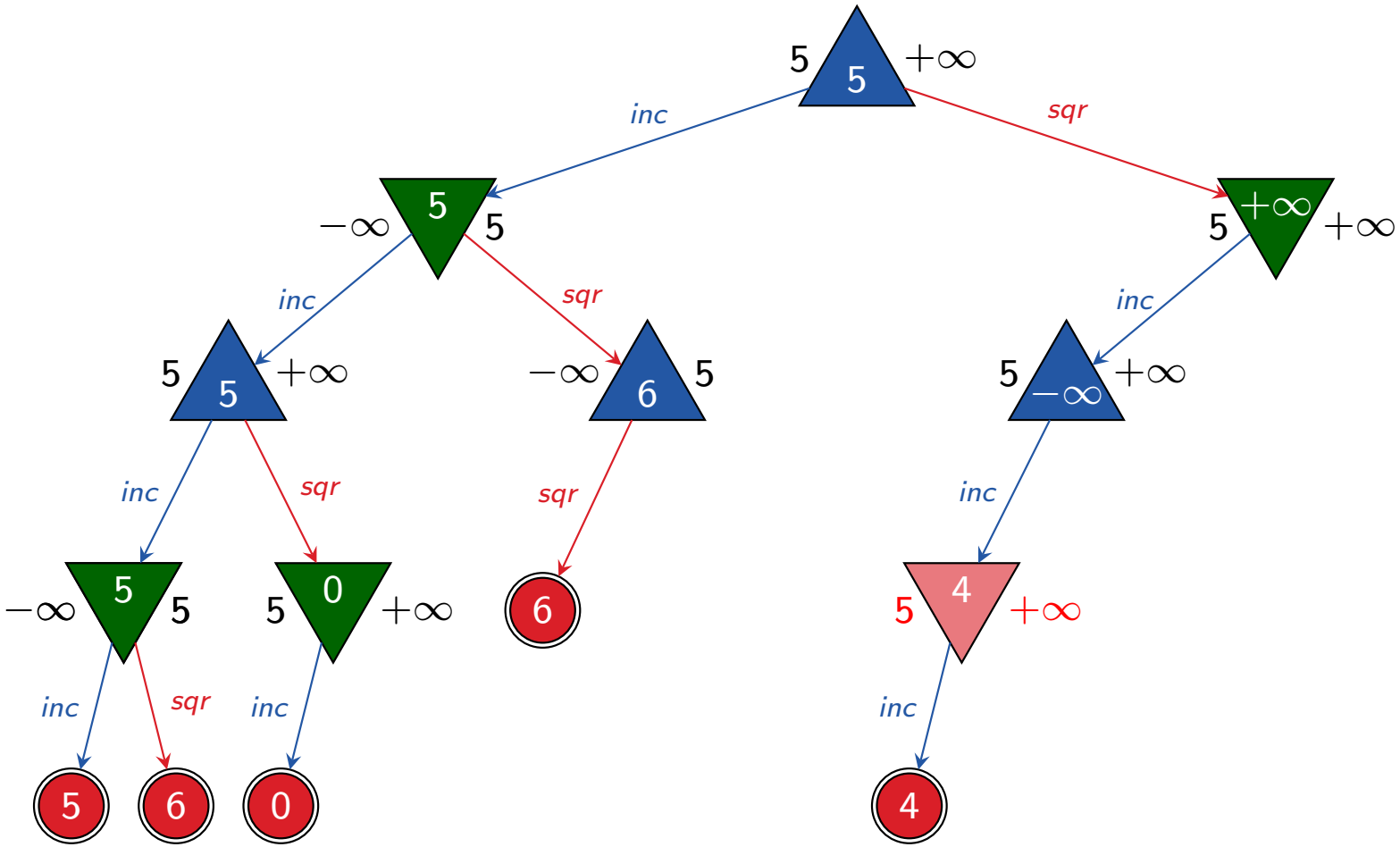
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



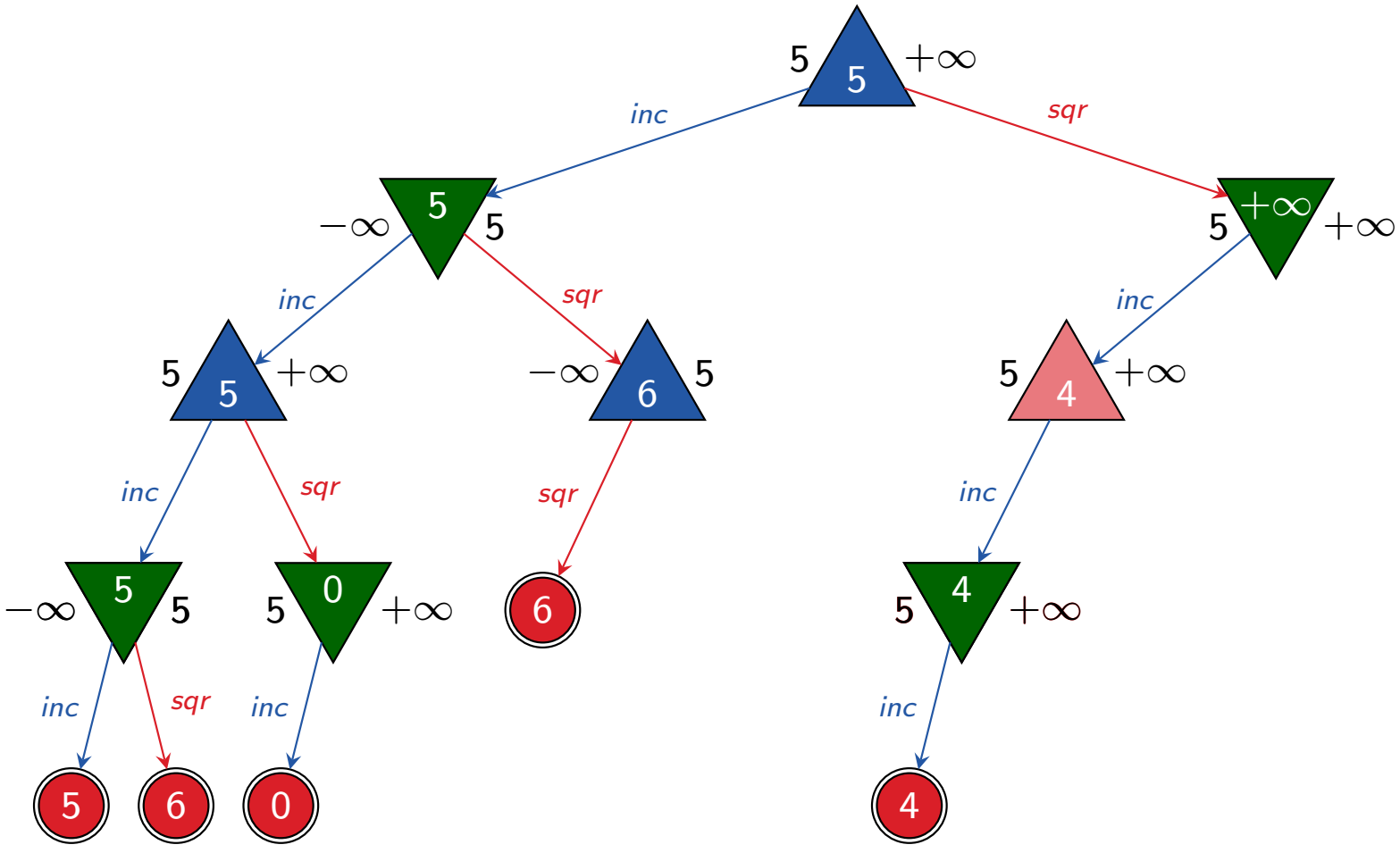
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



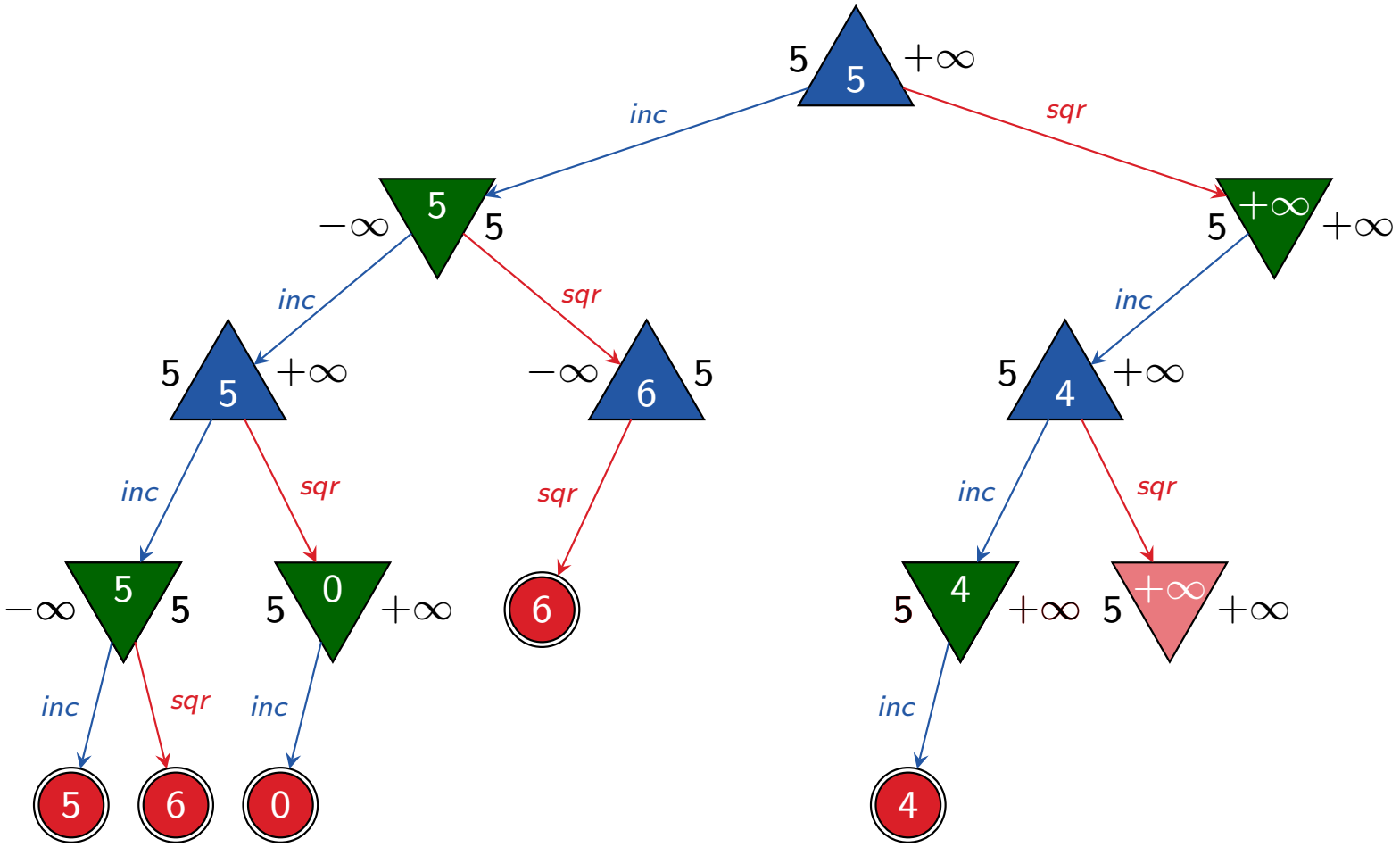
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



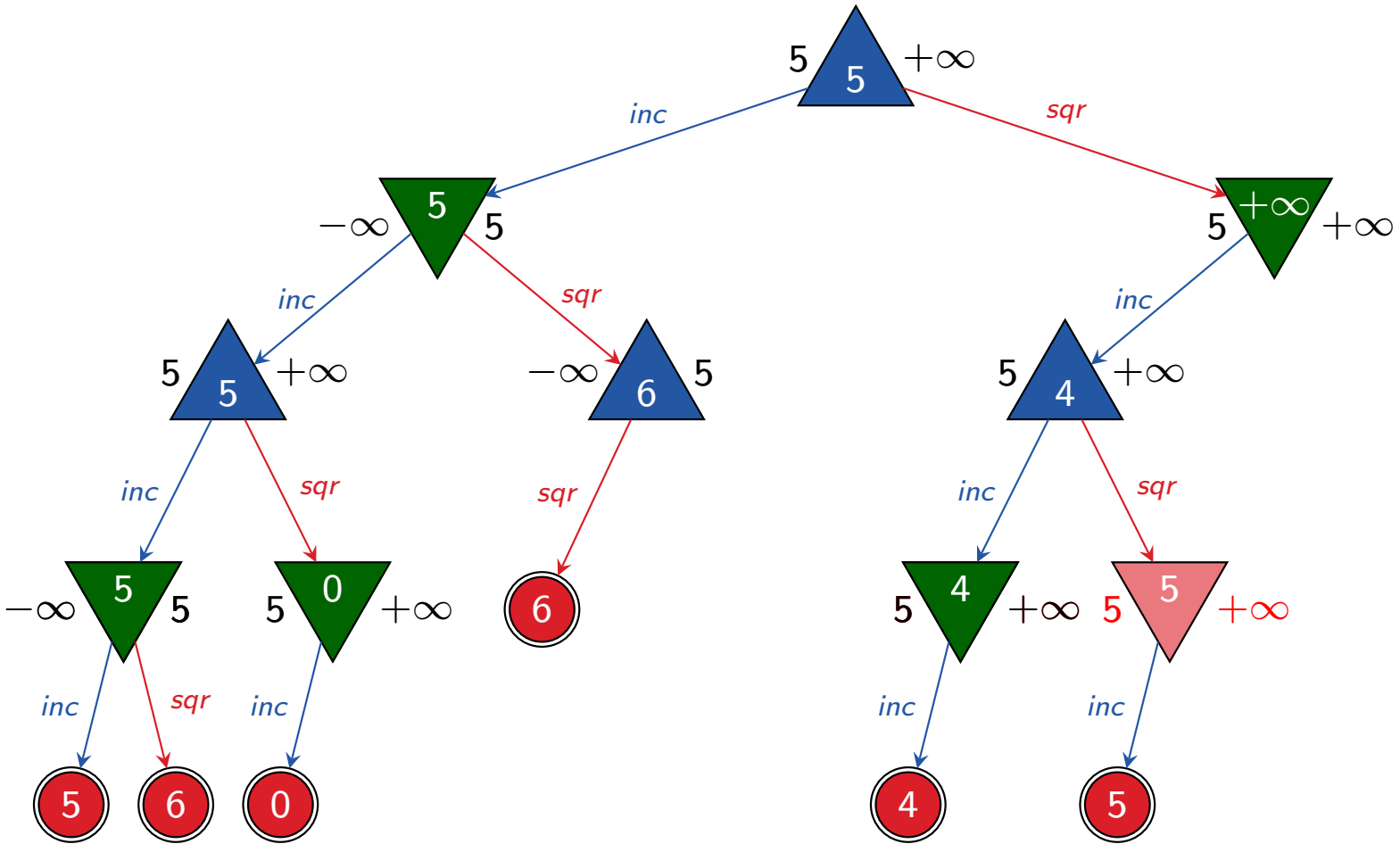
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



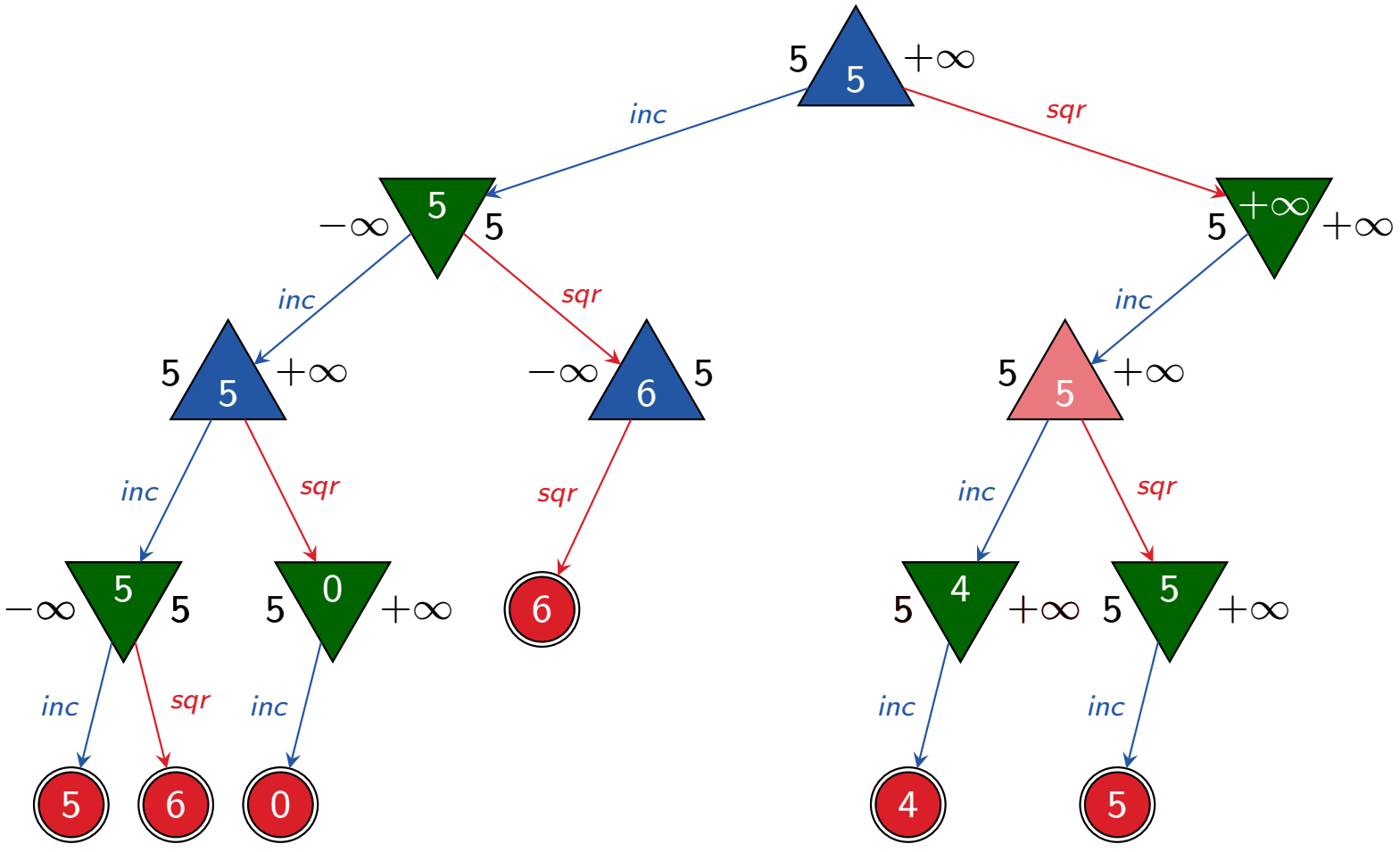
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



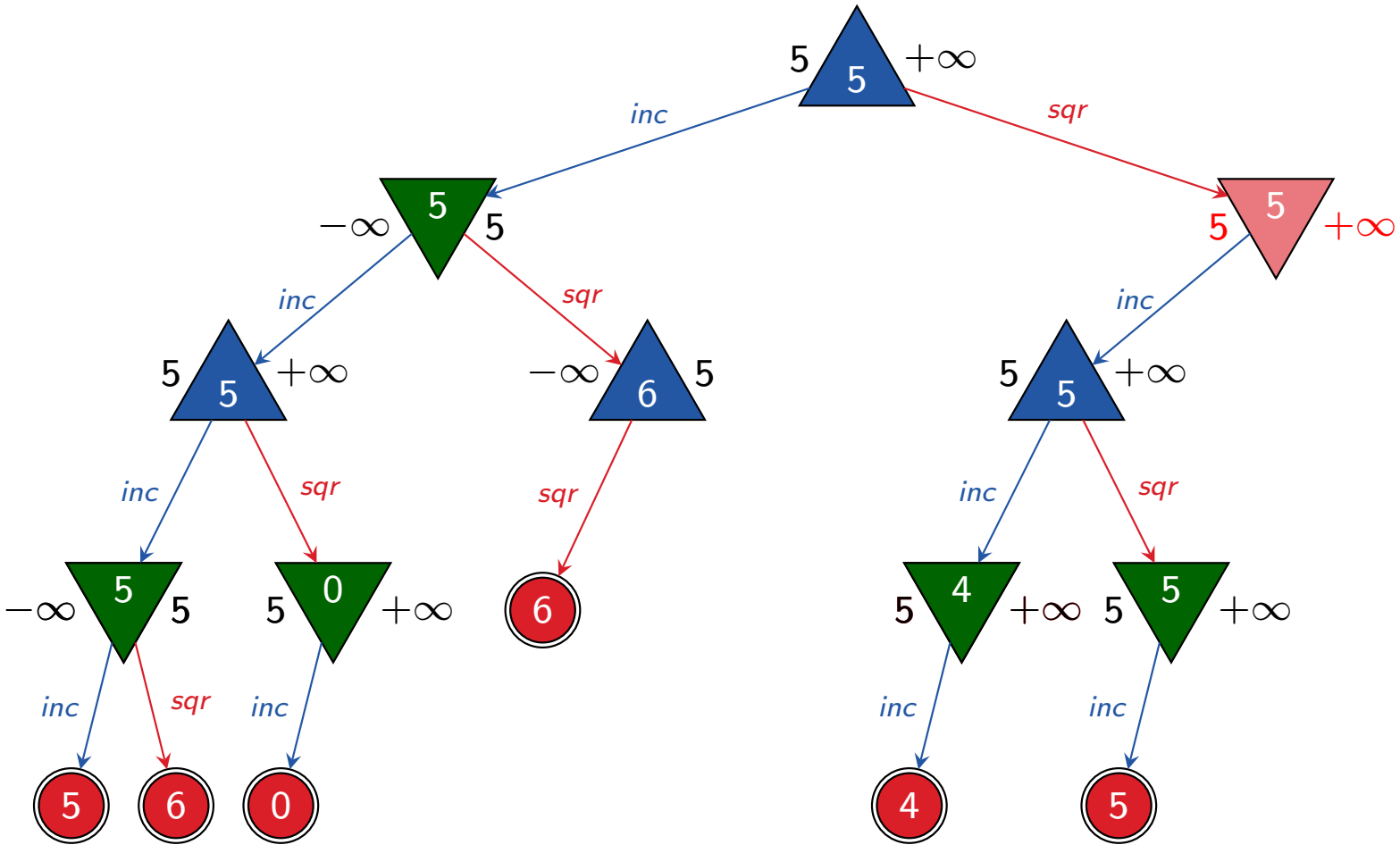
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



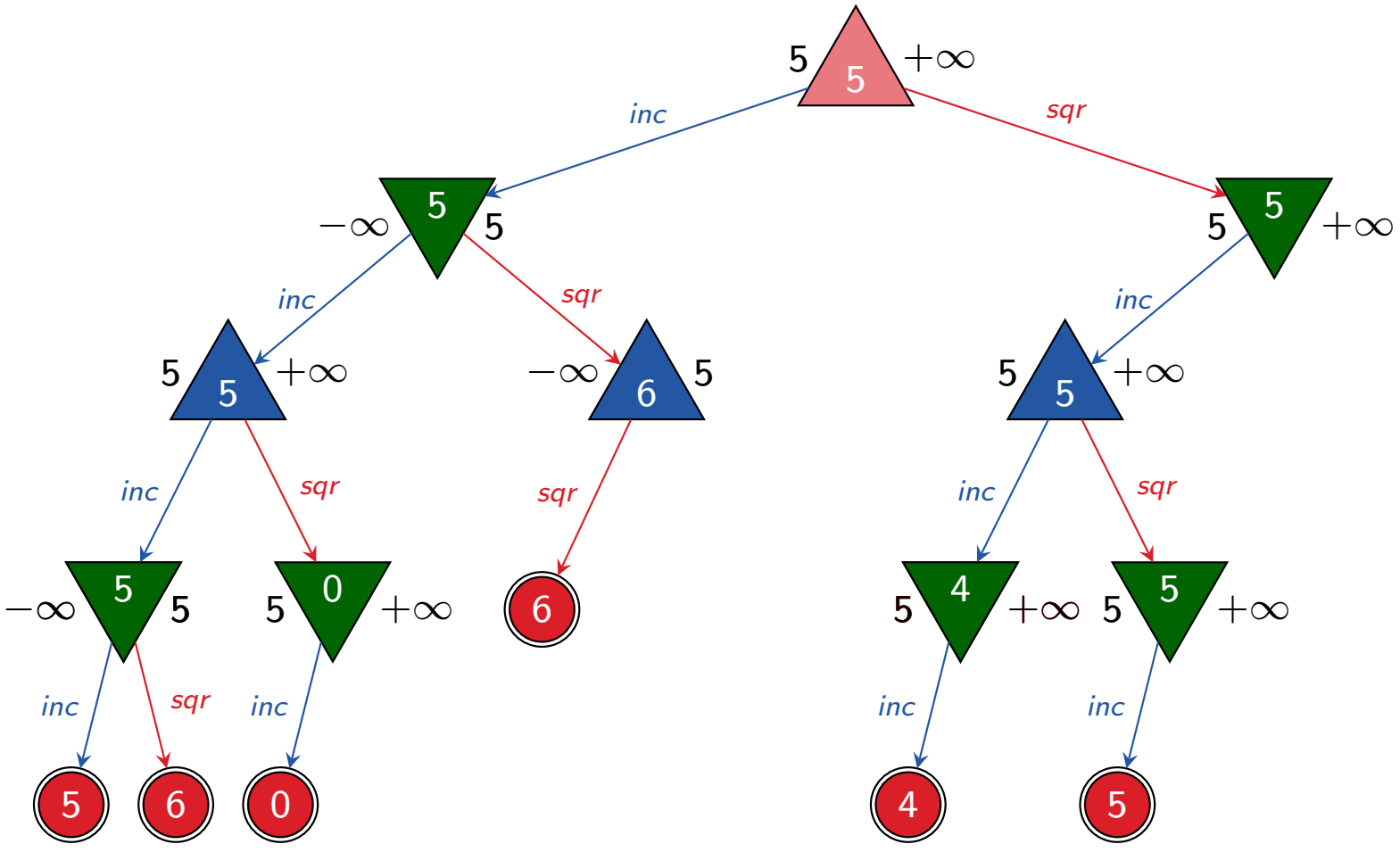
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



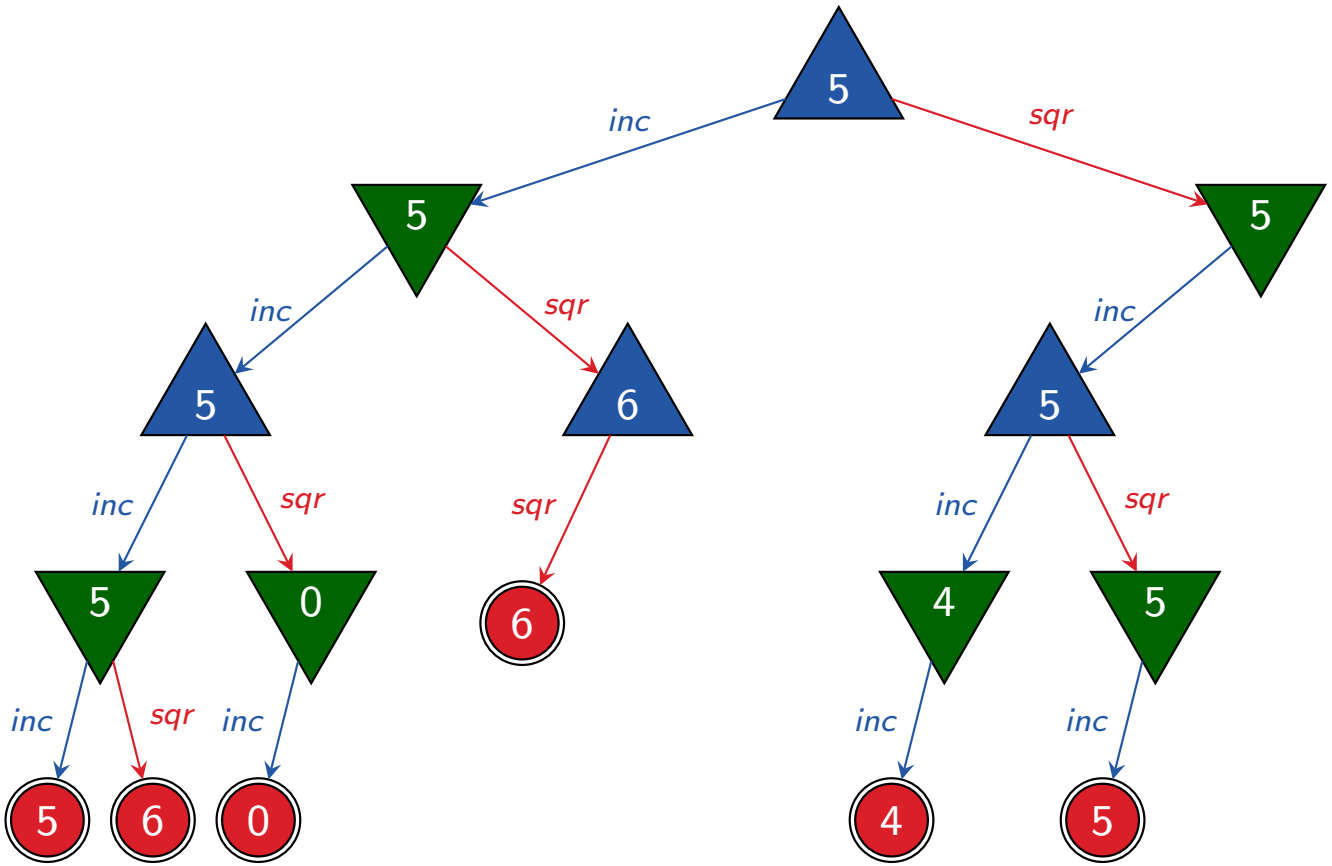
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Example



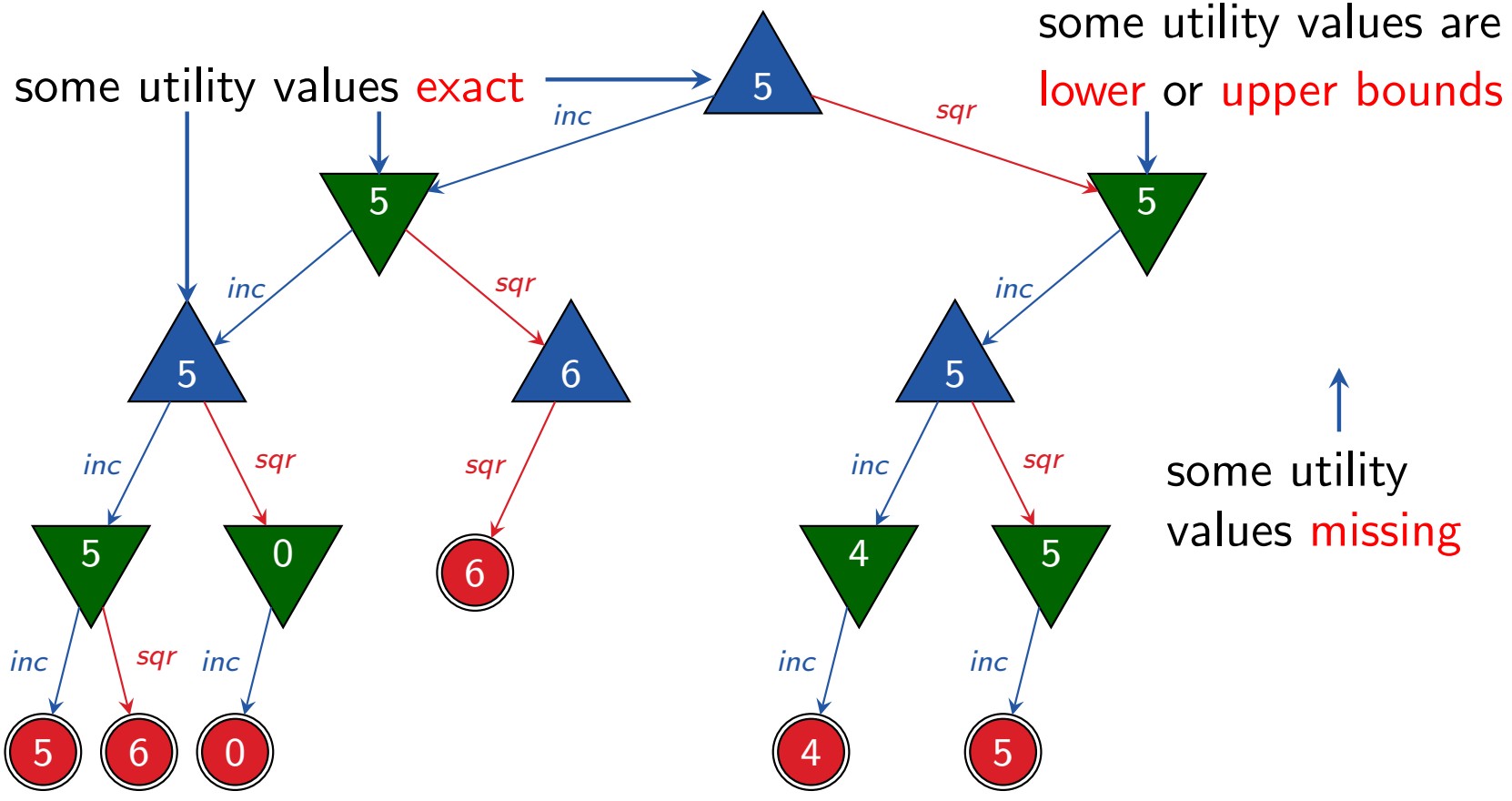
- α: lower bound of relevant utility
- a MAX subtree is pruned if utility $\geq \beta$
- β: upper bound of relevant utility
- a MIN subtree is pruned if utility $\leq \alpha$

Discussion



What do the utility values express?

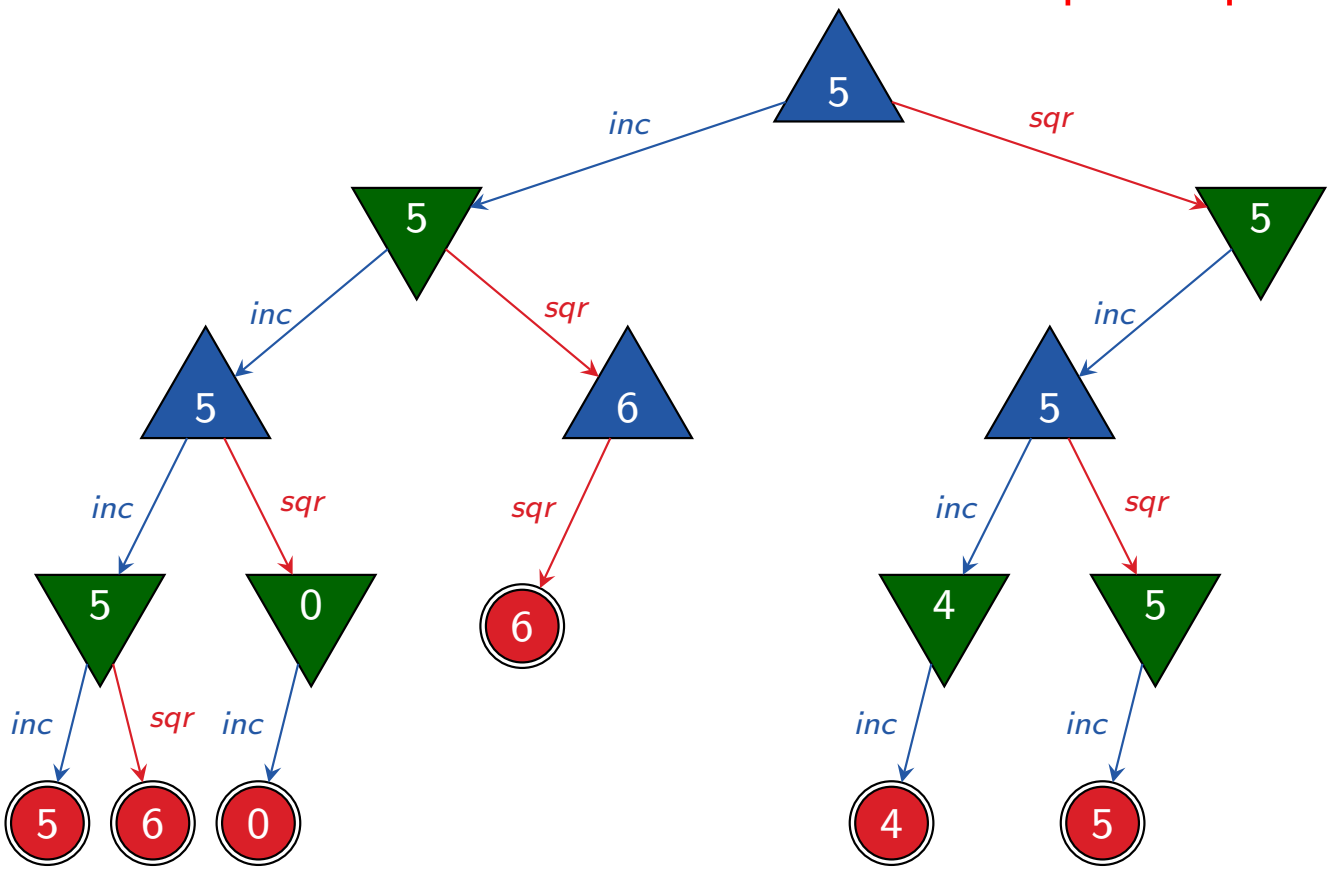
Discussion



What do the utility values express?

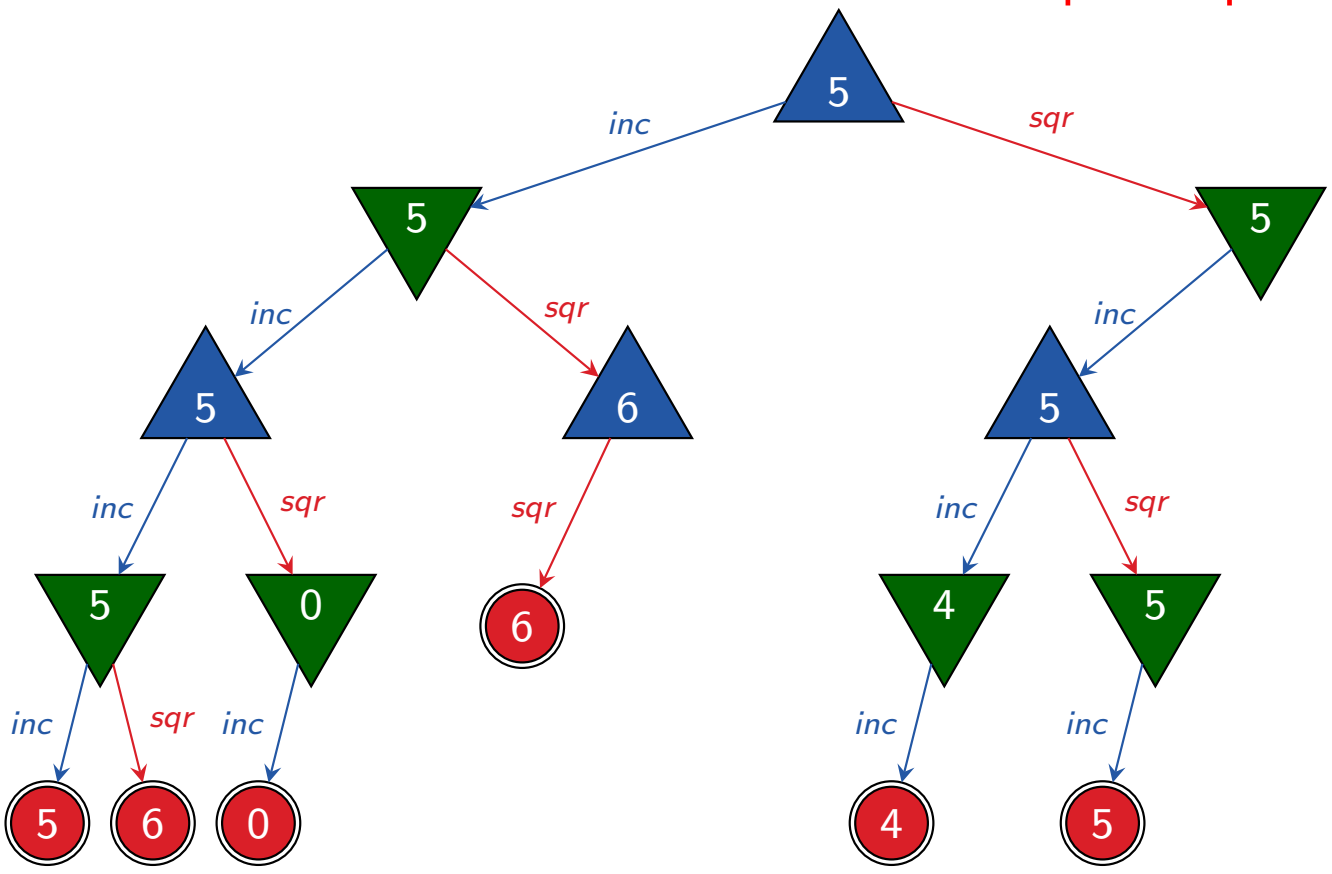
Discussion

What does this mean for the computed policy?



Discussion

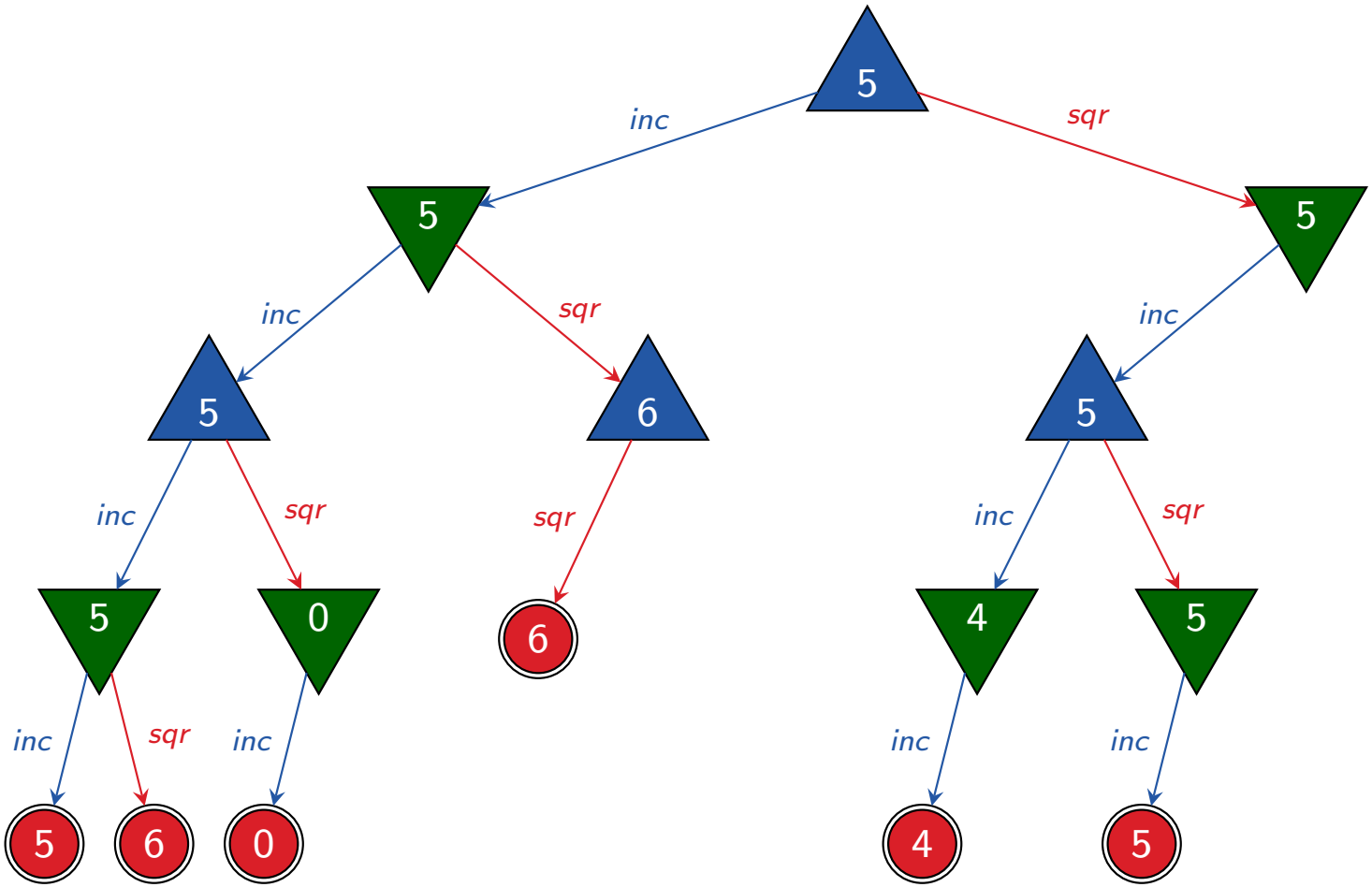
What does this mean for the computed policy?



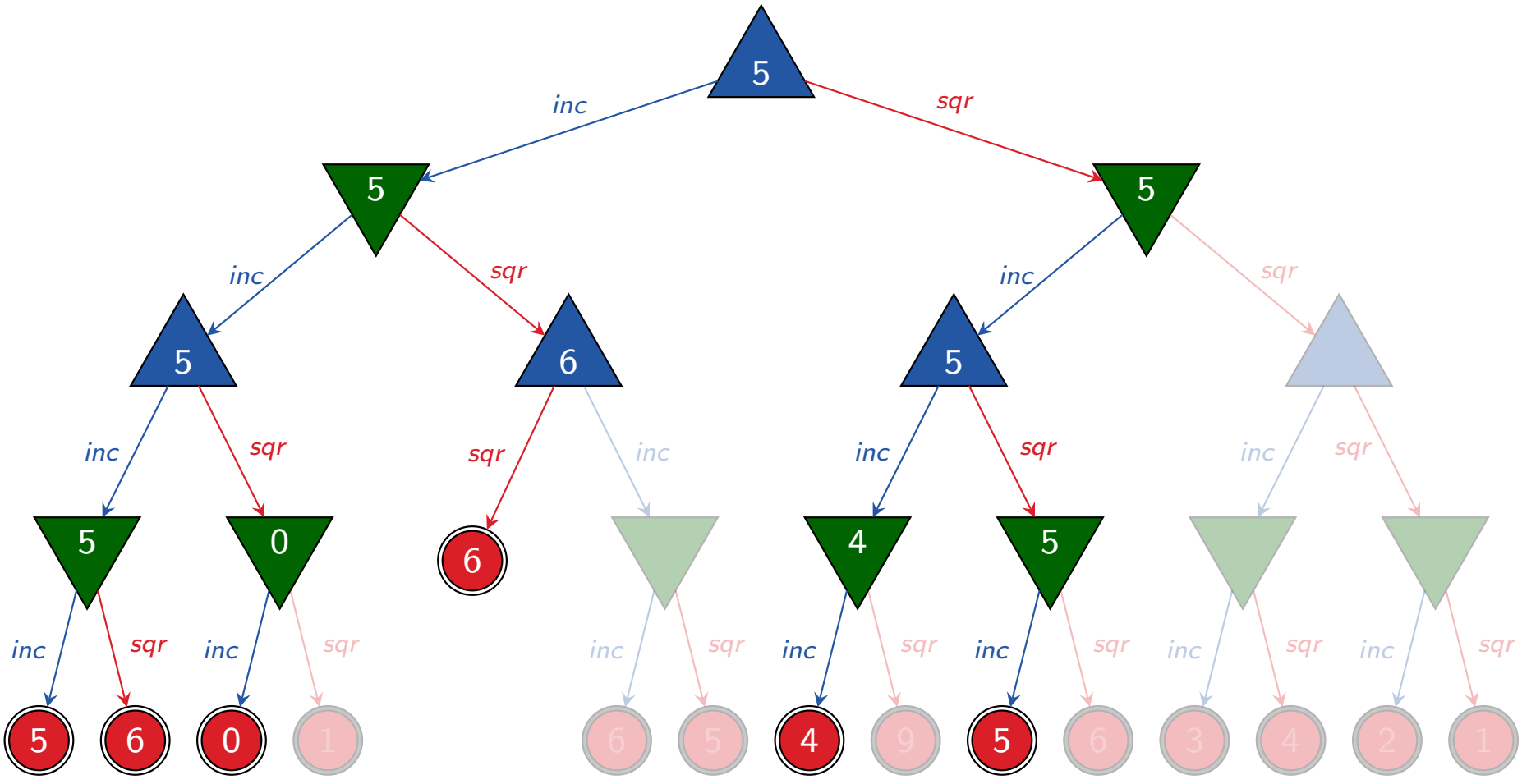
- only partial
- optimal in positions reachable under optimal play
- need to take earliest move in case of ties

Move Ordering

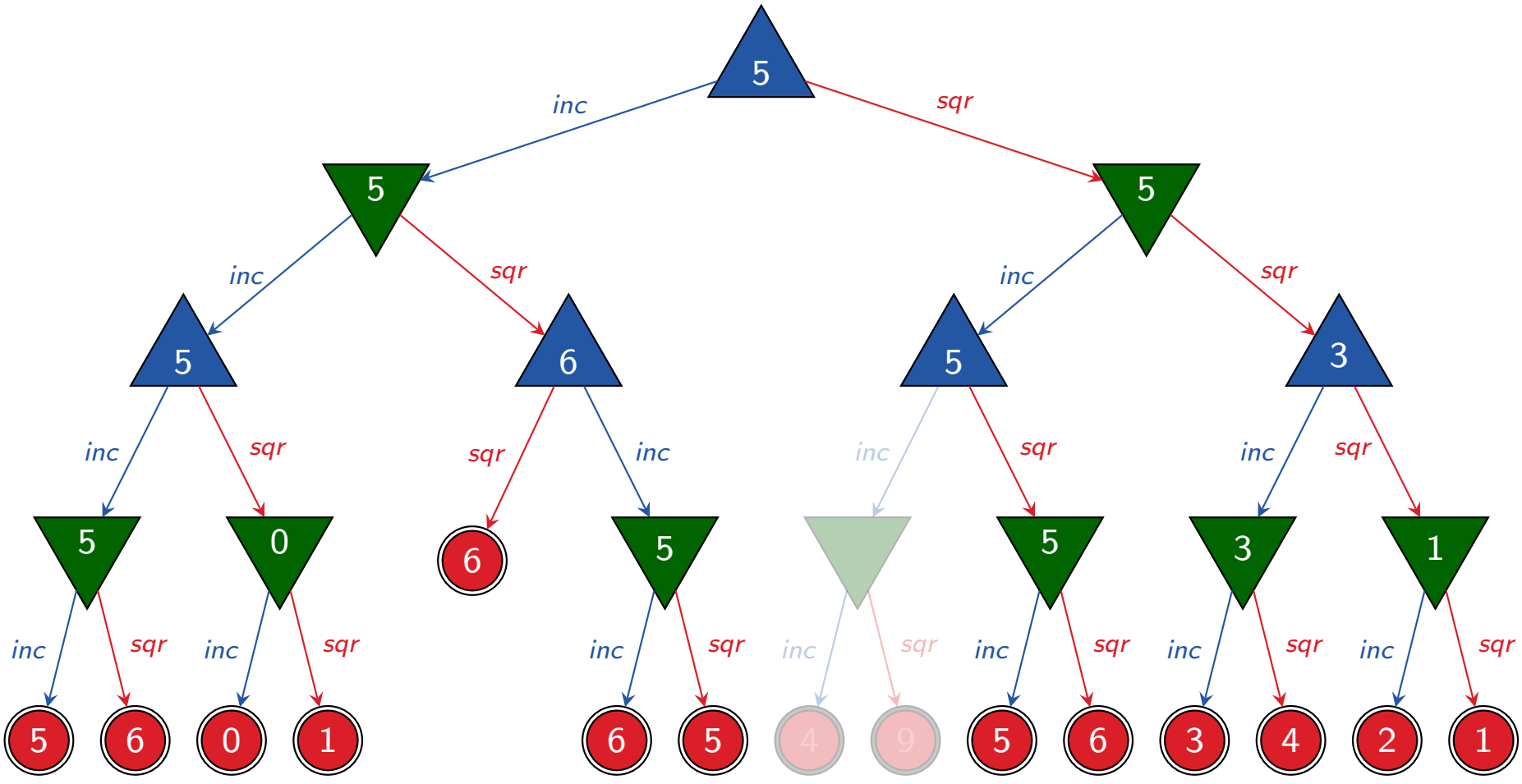
How Much Effort Do We Save?



How Much Effort Do We Save?



Were We Lucky?



if successors are considered in reverse order, we prune only a few positions

Move Ordering

idea: first consider the successors that are likely to be best

- domain-specific ordering function
  e.g., chess: captures $<$ threats $<$ forward moves $<$ backward moves
- dynamic move-ordering
  - first try moves that were good in the past
  - e.g., in iterative deepening search:
    best moves from previous iteration

How Much Do We Gain with Alpha-Beta Pruning?

assumption: uniform game tree, depth d, branching factor $b \geq 2$; MAX and MIN positions alternate

- perfect move ordering
  - best move at every position is considered first
  - maximizing move for MAX, minimizing move for MIN
  - effort reduced from $O(b^d)$ (minimax) to $O(b^{d/2})$
  - doubles the search depth that can be achieved in same time
- random move ordering
  - effort still reduced to $O(b^{3d/4})$

In practice, we can often get close to the perfect move ordering.

Heuristic Alpha-Beta Search

- combines evaluation function and alpha-beta search
- often uses additional techniques, e.g.
  - quiescence search
  - transposition tables
  - forward pruning
  - specialized subprocedures for certain parts of the game (e.g., opening libraries and endgame databases)
  - ...

Summary

Summary

alpha-beta search

- stores which utility both players can force somewhere else in the game tree
- exploits this information to avoid unnecessary computations
- can have significantly lower search effort than minimax
- best case: search twice as deep in the same time

Foundations of Artificial Intelligence

G4. Board Games: Stochastic Games

Malte Helmert

University of Basel

May 19, 2025

Board Games: Overview

chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Expected Value

Discrete Random Variable

- a random event (like the result of a die roll)
  - is described in terms of a random variable X
  - with associated domain $\mathrm{dom}(X)$
  - and a probability distribution over the domain
- if the number of outcomes of a random event is finite (like here), the random variable is a discrete random variable
- and the probability distribution is given as a probability $P(X = x)$ that the outcome is $x \in \mathrm{dom}(X)$

Discrete Random Variable: Example

informal description:

- you plan to invest in stocks and can afford one share
- your analyst expects these stock price changes:

Bellman Inc.	Markov Tec.
$+2$ with 30%	$+4$ with 20%
$+1$ with 60%	$+2$ with 30%
± 0 with 10%	-1 with 50%

Discrete Random Variable: Example

informal description:

- you plan to invest in stocks and can afford one share
- your analyst expects these stock price changes:

Bellman Inc.	Markov Tec.
$+2$ with 30%	$+4$ with 20%
$+1$ with 60%	$+2$ with 30%
± 0 with 10%	-1 with 50%

formal model:

- discrete random variables B and M
- $\mathrm{dom}(B) = \{2, 1, 0\}$
  $\mathrm{dom}(M) = \{4, 2, -1\}$
- $P(B = 2) = 0.3$ $P(M = 4) = 0.2$
  $P(B = 1) = 0.6$ $P(M = 2) = 0.3$
  $P(B = 0) = 0.1$ $P(M = -1) = 0.5$

Expected Value

- the expected value $\mathbb{E}[X]$ of a random variable X is a weighted average of its outcomes
- it is computed as the probability-weighted sum of all outcomes $x \in \mathrm{dom}(X)$, i.e.,

$$\mathbb{E}[X] = \sum_{x \in \mathrm{dom}(X)} P(X = x) \cdot x$$

- in stochastic environments, it is rational to deal with uncertainty by optimizing expected values

Expected Value: Example

formal model:

- discrete random variables
  B and M
- $\text{dom}(B) = \{2, 1, 0\}$
  $\text{dom}(M) = \{4, 2, -1\}$
- $P(B = 2) = 0.3$ $P(M = 4) = 0.2$
  $P(B = 1) = 0.6$ $P(M = 2) = 0.3$
  $P(B = 0) = 0.1$ $P(M = -1) = 0.5$

Expected Value: Example

formal model:

- discrete random variables B and M
- $\text{dom}(B) = \{2, 1, 0\}$
  $\text{dom}(M) = \{4, 2, -1\}$
- $P(B = 2) = 0.3$ $P(M = 4) = 0.2$
  $P(B = 1) = 0.6$ $P(M = 2) = 0.3$
  $P(B = 0) = 0.1$ $P(M = -1) = 0.5$

expected gain:

$$\begin{aligned}\mathbb{E}[B] &= P(B = 2) \cdot 2 + P(B = 1) \cdot 1 + P(B = 0) \cdot 0 \\ &= 0.3 \cdot 2 + 0.6 \cdot 1 + 0.1 \cdot 0 = 1.2\end{aligned}$$

$$\begin{aligned}\mathbb{E}[M] &= P(M = 4) \cdot 4 + P(M = 2) \cdot 2 + P(M = -1) \cdot -1 \\ &= 0.2 \cdot 4 + 0.3 \cdot 2 + 0.5 \cdot -1 = 0.9\end{aligned}$$

Expected Value: Example

formal model:

- discrete random variables B and M
- $\text{dom}(B) = \{2, 1, 0\}$
  $\text{dom}(M) = \{4, 2, -1\}$
- $P(B = 2) = 0.3$ $P(M = 4) = 0.2$
  $P(B = 1) = 0.6$ $P(M = 2) = 0.3$
  $P(B = 0) = 0.1$ $P(M = -1) = 0.5$

expected gain:

$$\mathbb{E}[B] = P(B = 2) \cdot 2 + P(B = 1) \cdot 1 + P(B = 0) \cdot 0 = 0.3 \cdot 2 + 0.6 \cdot 1 + 0.1 \cdot 0 = 1.2$$

$$\mathbb{E}[M] = P(M = 4) \cdot 4 + P(M = 2) \cdot 2 + P(M = -1) \cdot -1 = 0.2 \cdot 4 + 0.3 \cdot 2 + 0.5 \cdot -1 = 0.9$$

rational decision: buy Bellman Inc.

Stochastic Games

Definition

Definition (stochastic game)

A stochastic game is a 7-tuple $\mathcal{S} = \langle S, A, T, s_I, S_G, \mathit{utility}, \mathit{player} \rangle$ with

- finite set of positions S
- finite set of moves A
- transition function $T : S \times A \times S \mapsto [0, 1]$ that is well-defined for $\langle s, a \rangle$ (see below)
- initial position $s_I \in S$
- set of terminal positions $S_G \subseteq S$
- utility function $\mathit{utility} : S_G \to \mathbb{R}$
- player function $\mathit{player} : S \setminus S_G \to \{\mathrm{MAX}, \mathrm{MIN}\}$

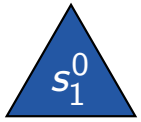
A transition function is well-defined for $\langle s, a \rangle$ if $\sum_{s' \in S} T(s, a, s') = 1$ (then a is applicable in s) or $\sum_{s' \in S} T(s, a, s') = 0$.

Example: Stochastic Inc-and-Square Game

- As an example, we consider a variant of the bounded inc-and-square game from Chapter G1.
- The sqr move now acts stochastically:
  - It squares the current value v (mod 10) with probability $\frac{v}{10}$.
  - Otherwise it doubles the current value v (mod 10) (with prob. $1 - \frac{v}{10}$).
- We also reduce the maximum game length to 3 moves (counting both players) to make the example smaller.
- Everything else stays the same.

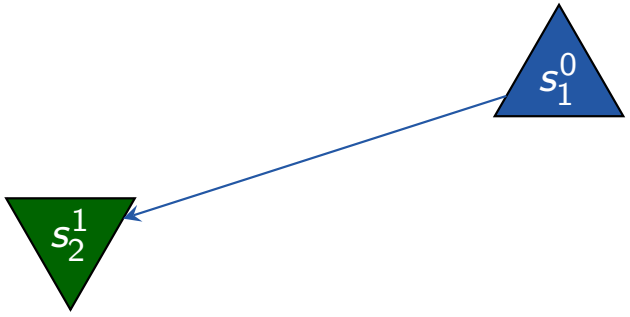
Expectiminimax

Idea and Example



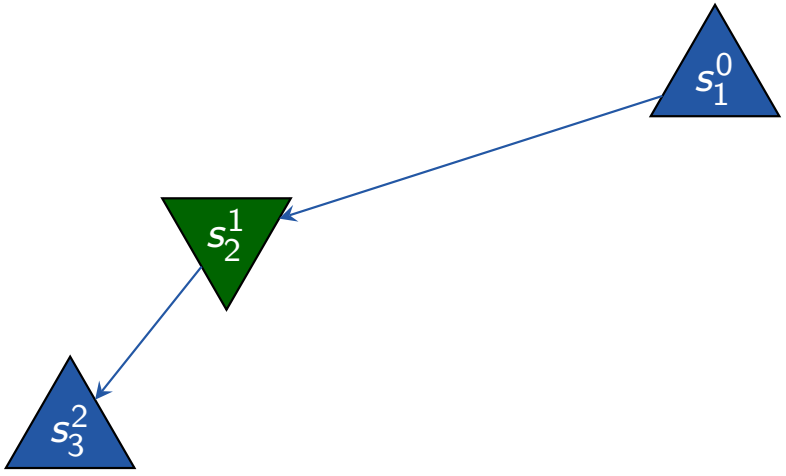
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



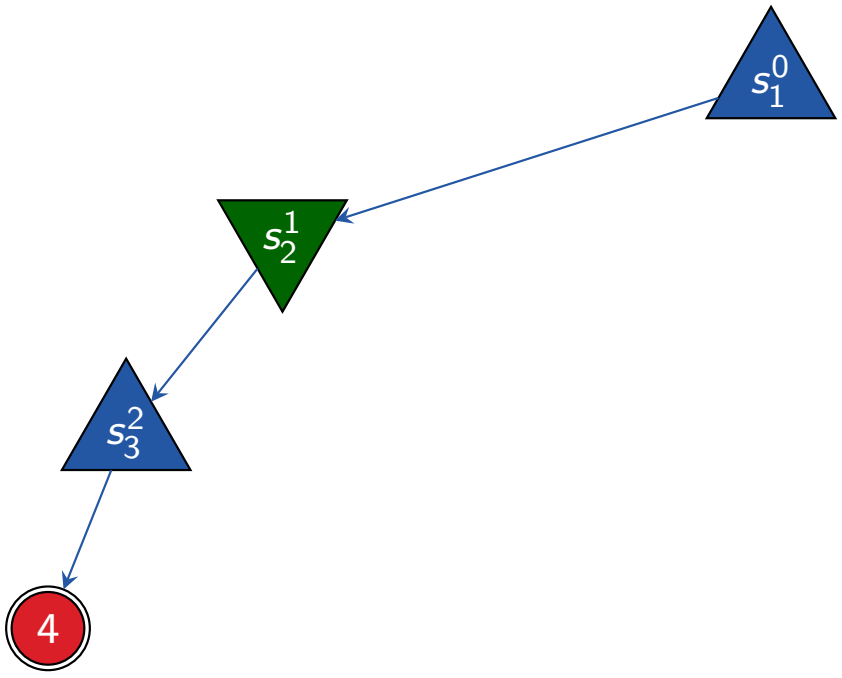
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



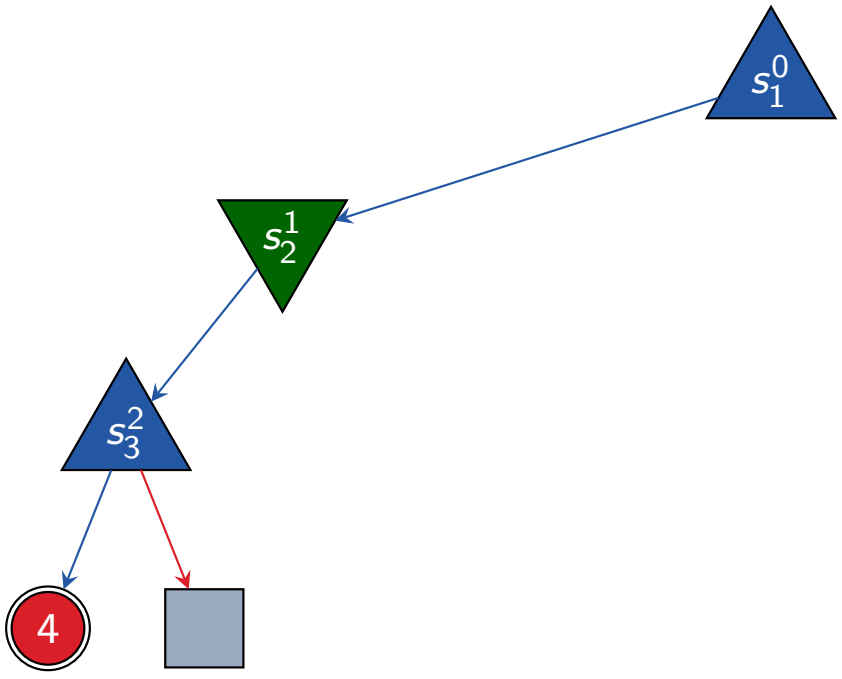
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



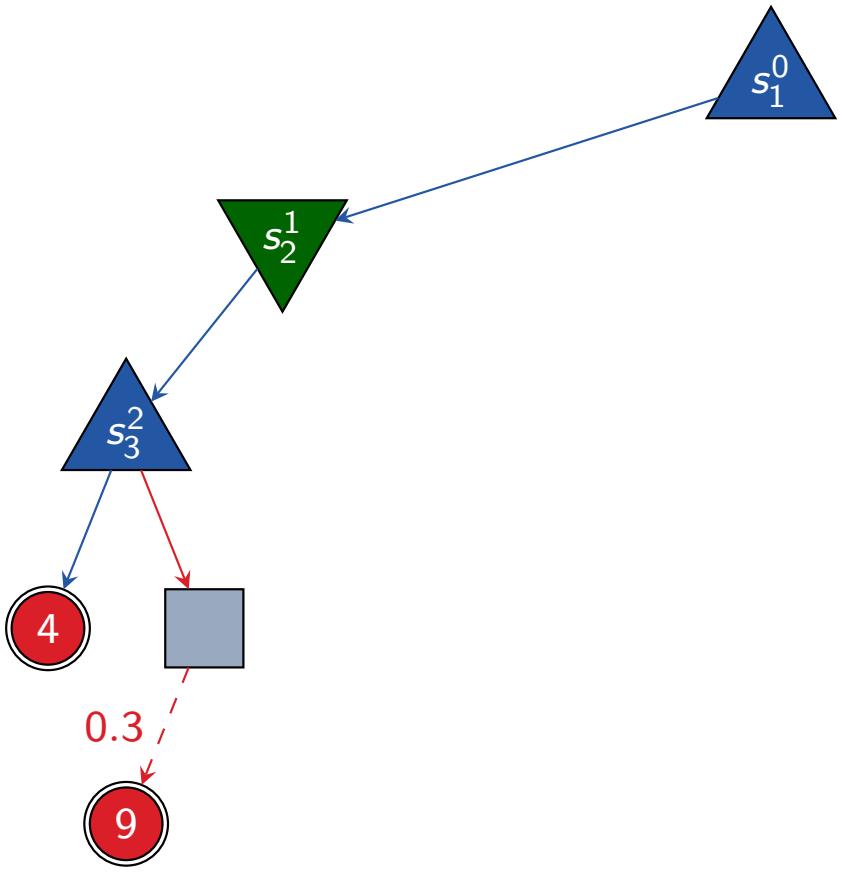
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



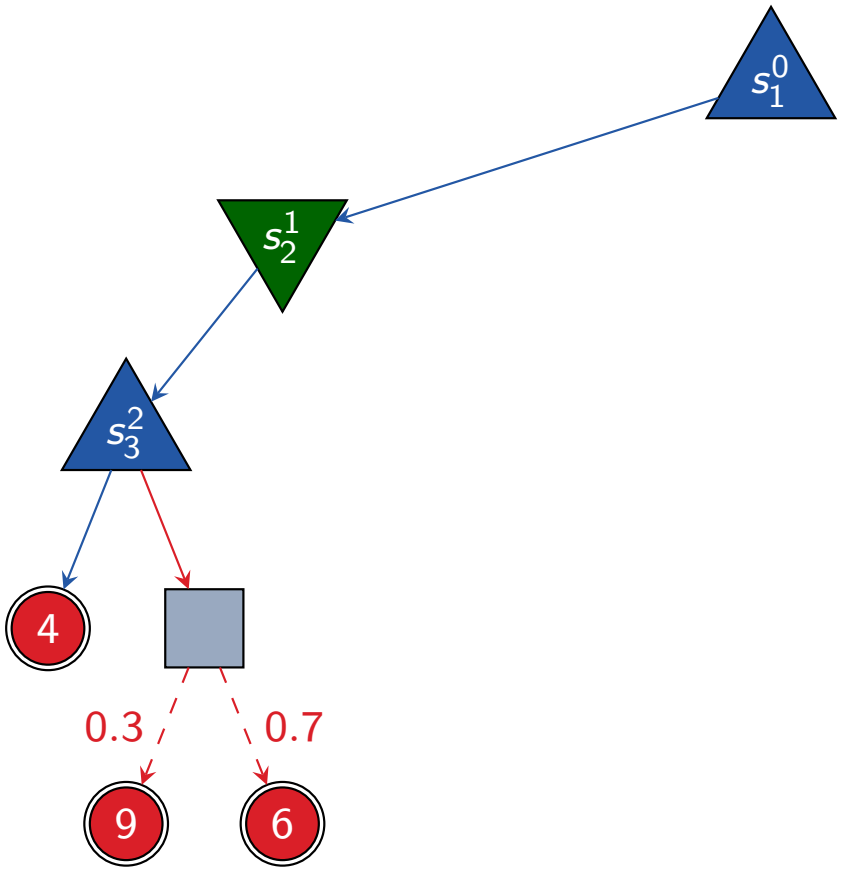
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



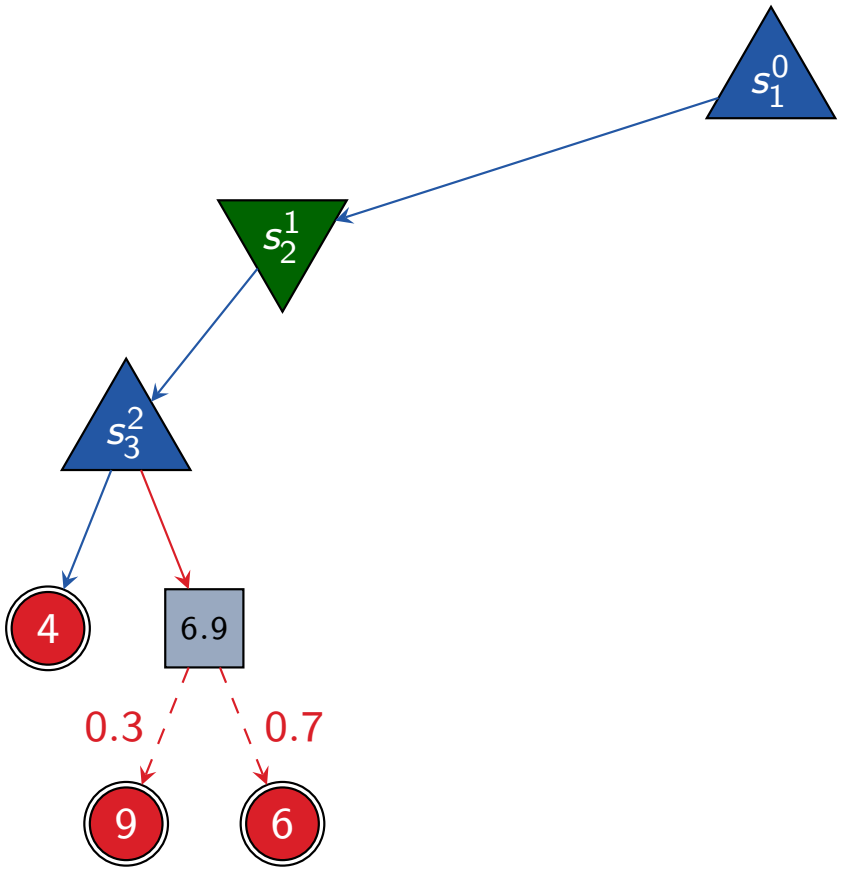
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



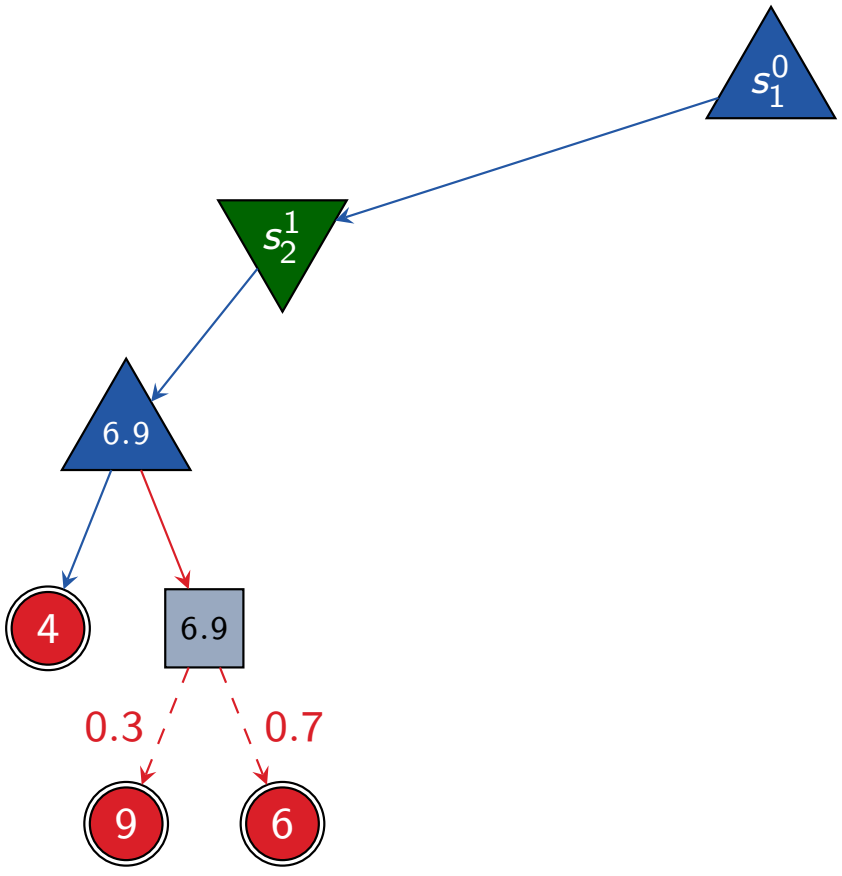
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



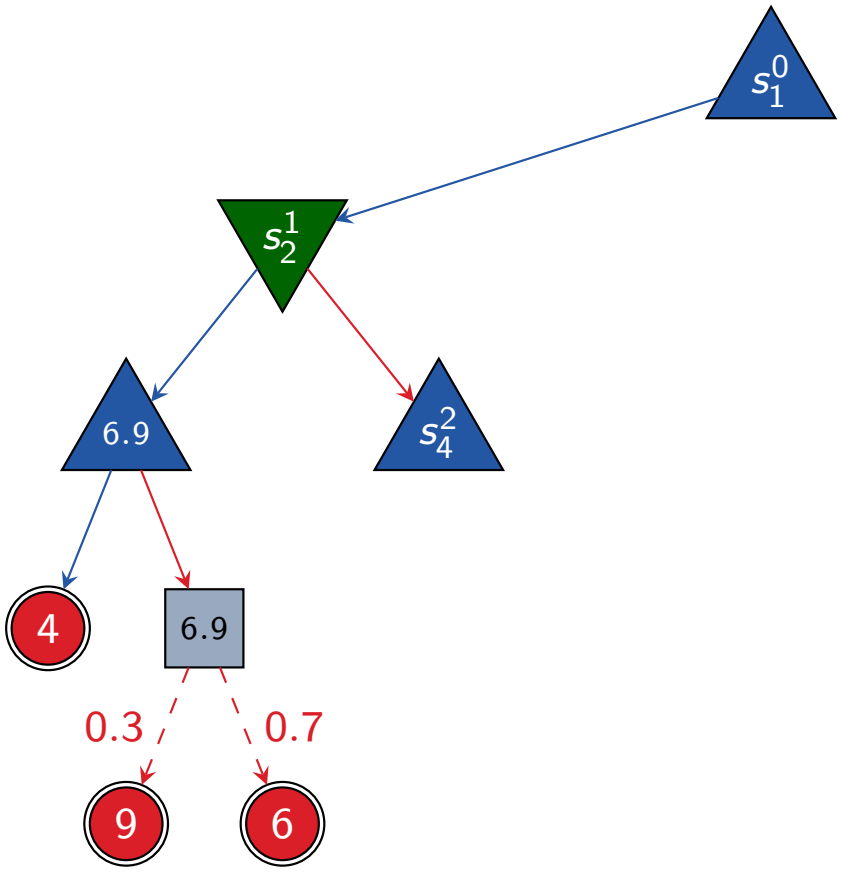
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



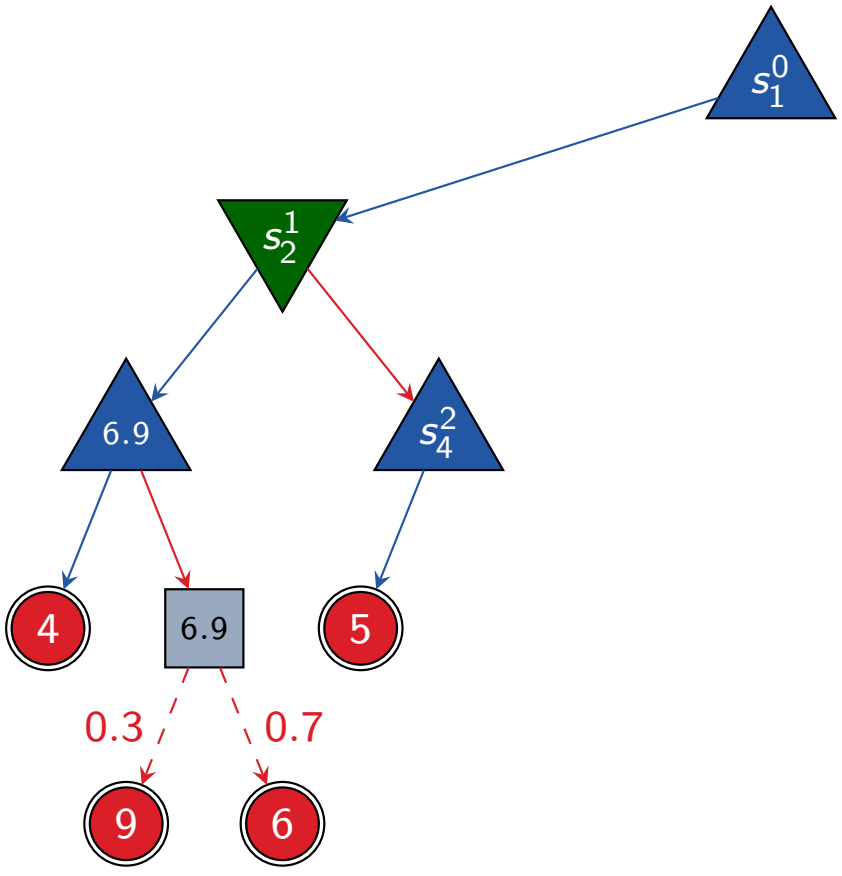
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



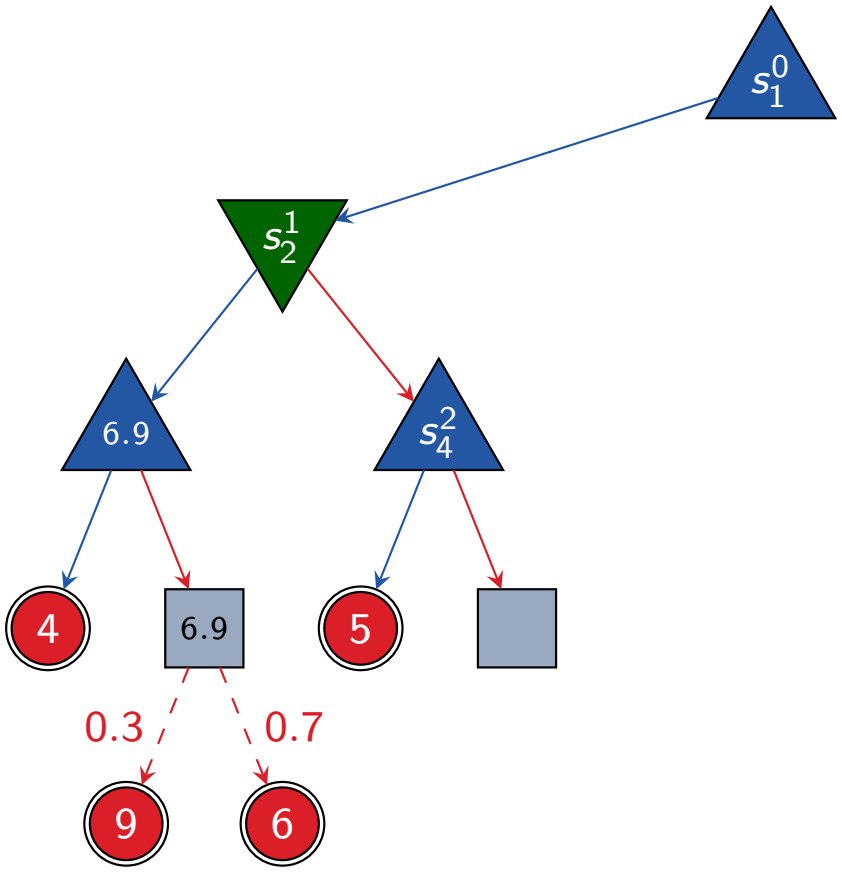
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



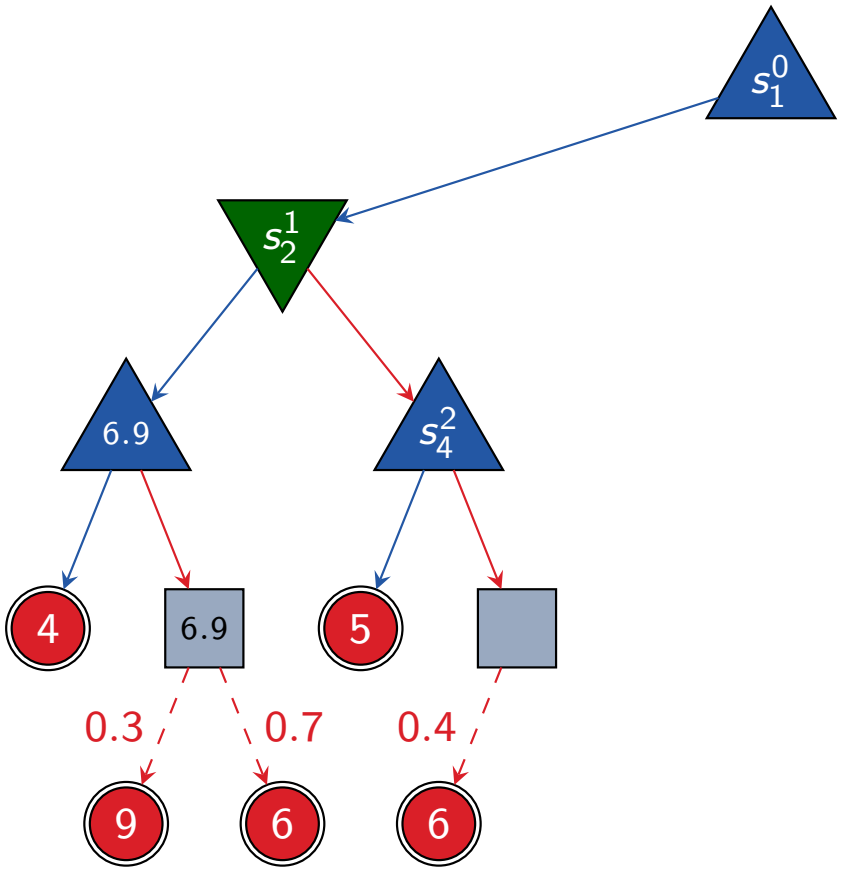
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



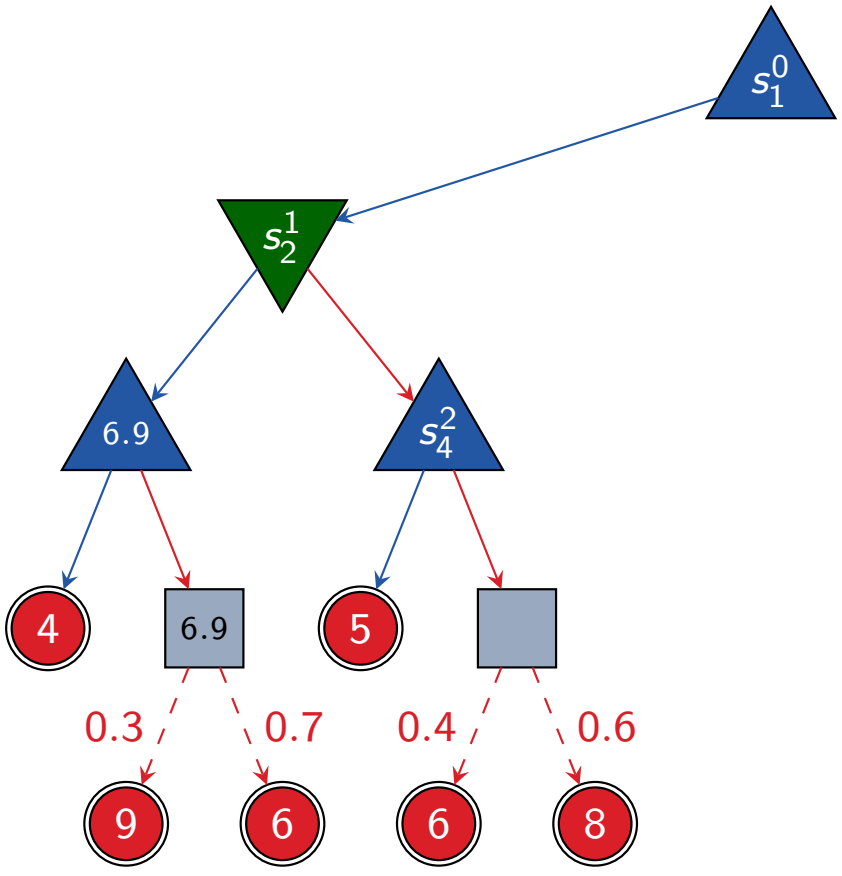
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



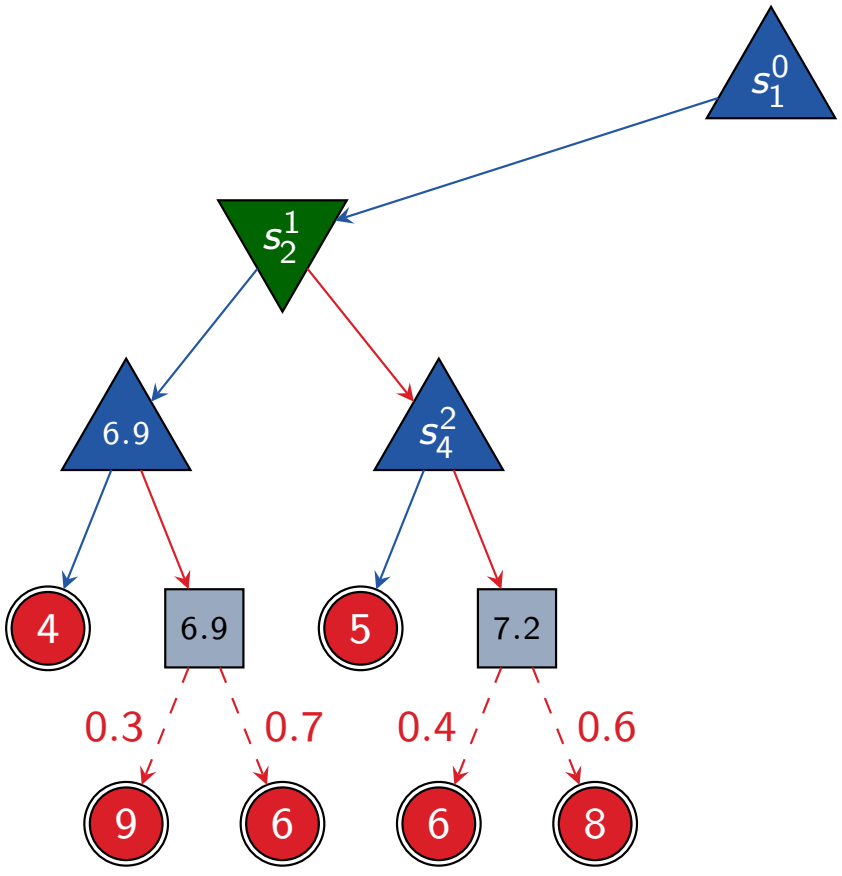
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



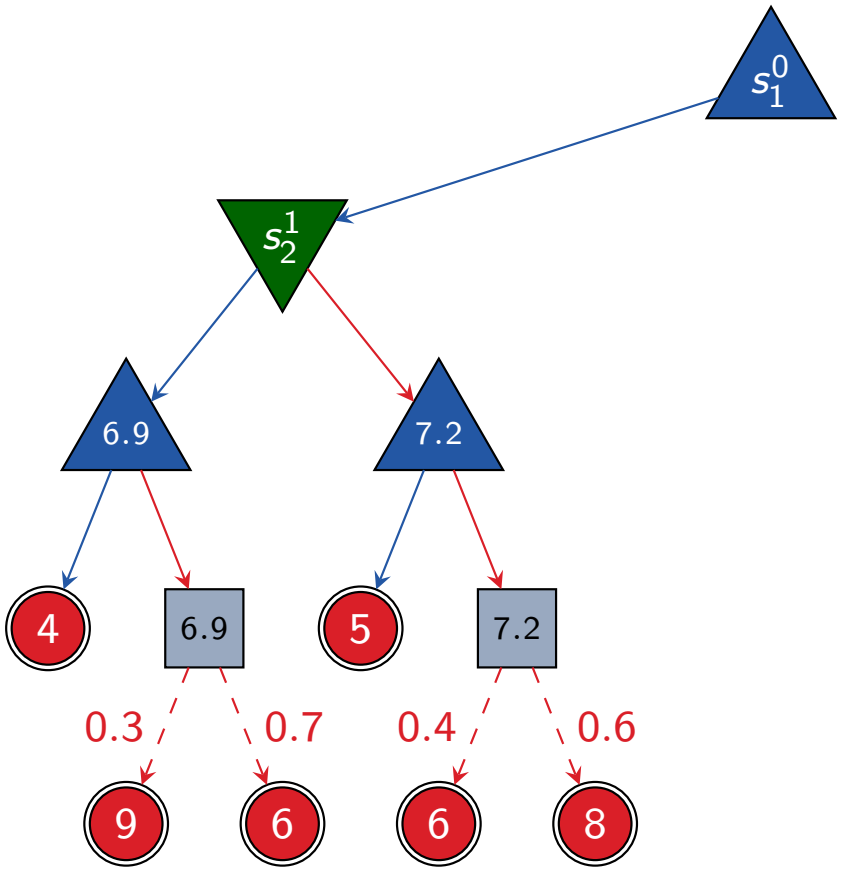
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



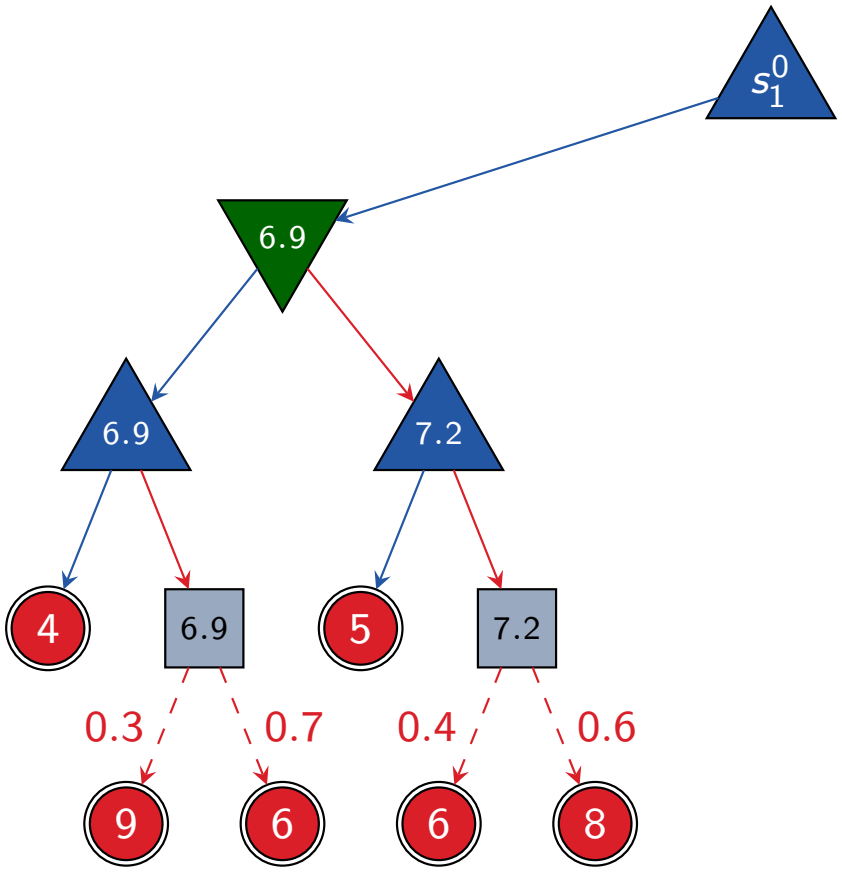
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



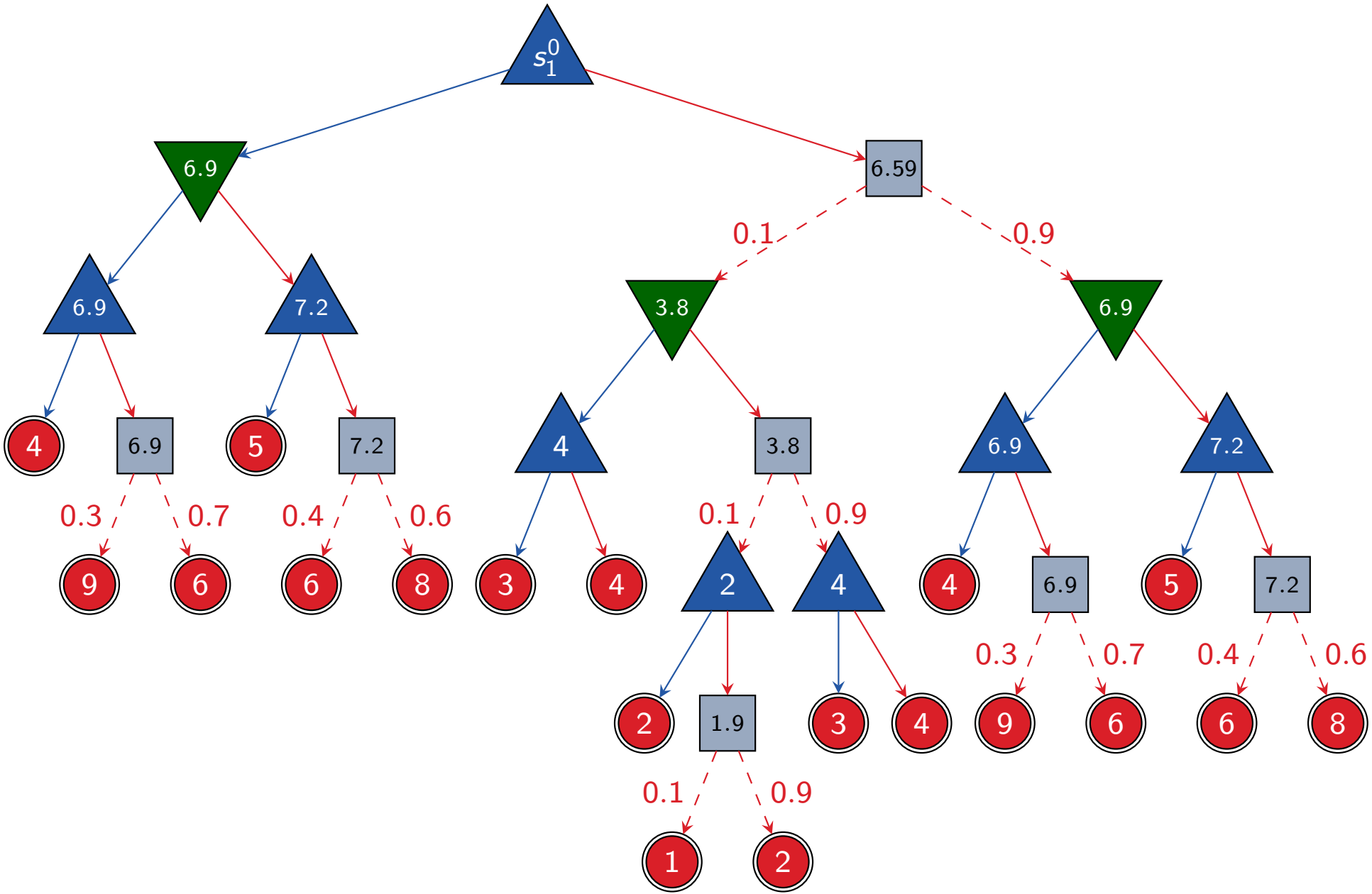
- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example

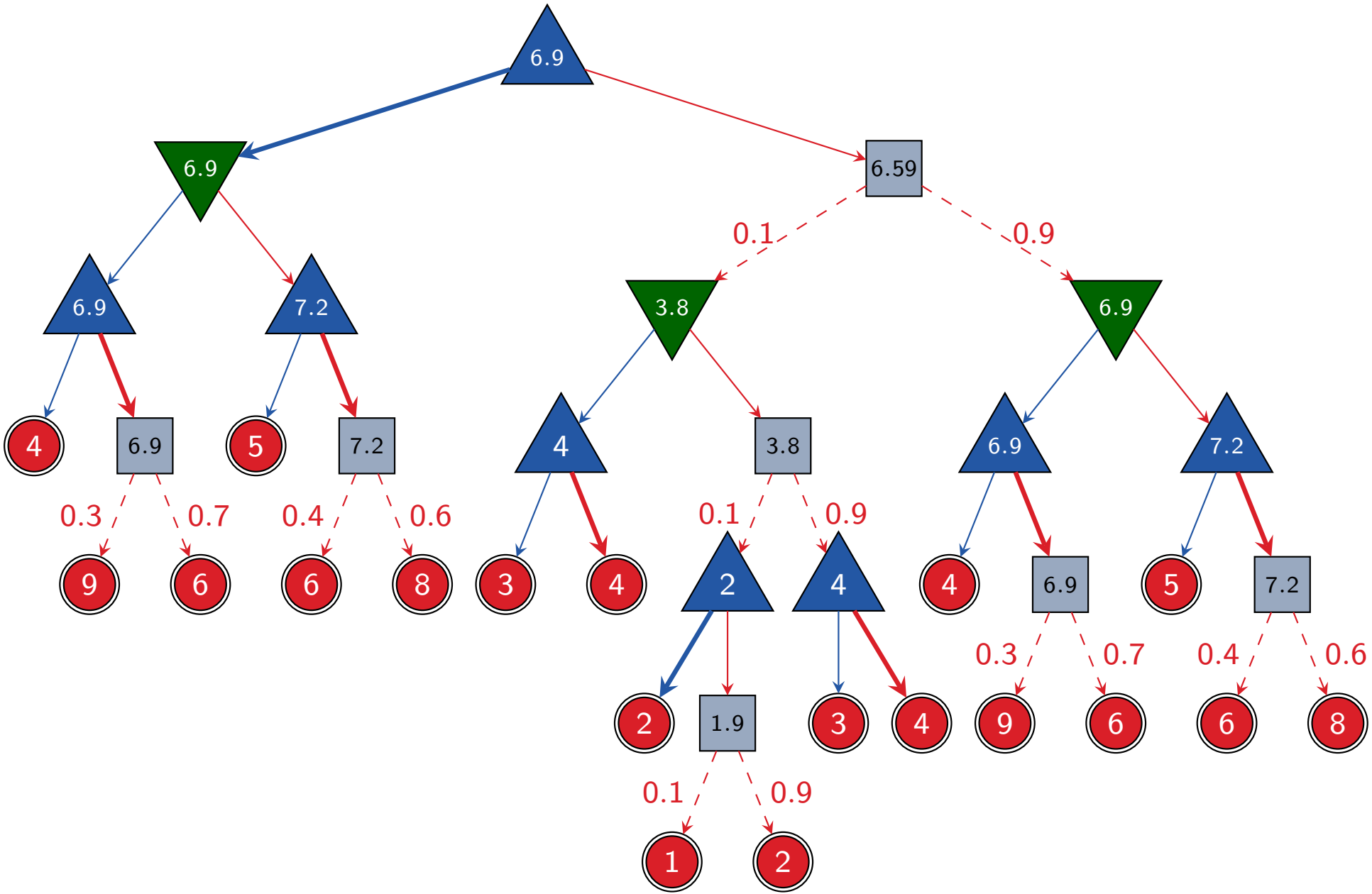


- depth-first search in game tree
- determine utility value of terminal positions with utility function
- compute utility value of inner nodes bottom-up through the tree:
  - MIN's turn: utility value is minimum of utility values of children
  - MAX's turn: utility value is maximum of utility values of children
  - chance: utility value is expected value of utility values of children
- policy for MAX: select action that leads to maximum utility value of children

Idea and Example



Idea and Example



Discussion

- expectiminimax is the simplest (decent) search algorithm for stochastic games
- yields optimal policy (in the game-theoretic sense, i.e., under the assumption that the opponent plays perfectly)
- MAX obtains at least the utility value computed for the root in expectation, no matter how MIN plays
- if MIN plays perfectly, MAX obtains exactly the computed value in expectation

The same improvements as for minimax are possible (evaluation functions, alpha-beta search).

Summary

Summary

- Stochastic games are board games with an additional element of chance.
- Expectiminimax is a minimax variant for stochastic games with identical behavior in MAX and MIN nodes.
- In chance nodes, it propagates the expected value (probability-weighted sum) of all successors.
- Expectiminimax has same guarantees as minimax, but in expectation.

Foundations of Artificial Intelligence

G5. Board Games: Monte-Carlo Tree Search Framework

Malte Helmert

University of Basel

May 21, 2025

Board Games: Overview

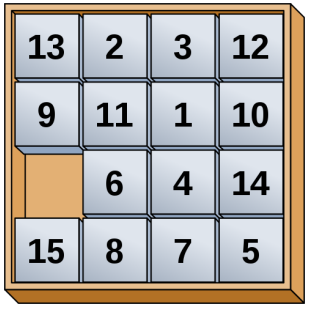
chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Introduction

Monte-Carlo Tree Search

algorithms considered previously:

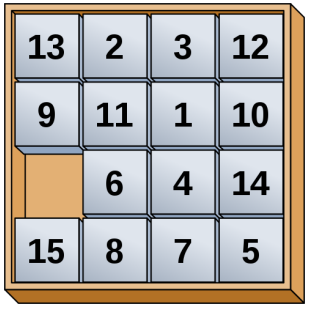


systematic search:

- systematic exploration of search space
- computation of (state) quality follows performance metric

Monte-Carlo Tree Search

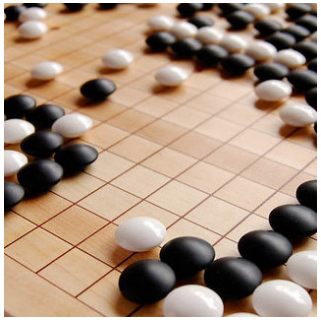
algorithms considered previously:



systematic search:

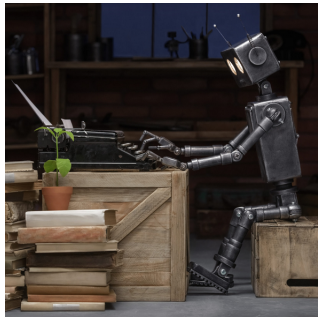
- systematic exploration of search space
- computation of (state) quality follows performance metric

algorithms considered today:



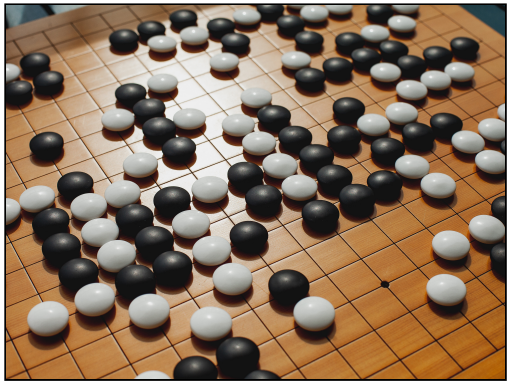
search based on Monte-Carlo methods:

- sampling of game simulations
- estimation of (state) quality by averaging over simulation results



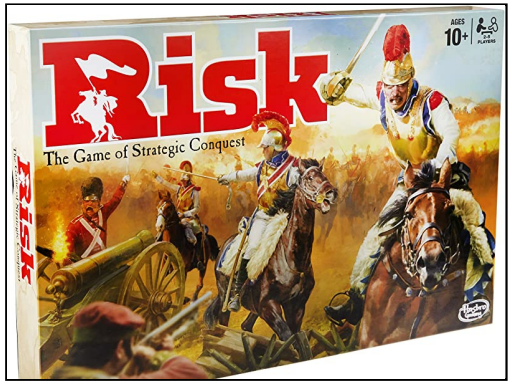
Game Applications

board games



hidden information games

stochastic games



general game playing

real-time strategy games

dynamic difficulty adjustment

Świechowski et al., Monte Carlo Tree Search: a review of recent modifications and applications (2023)

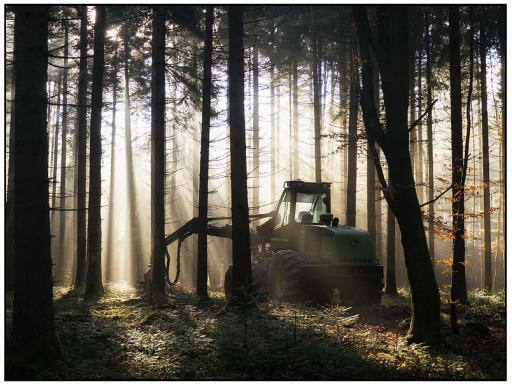
Applications Beyond Games

story generation

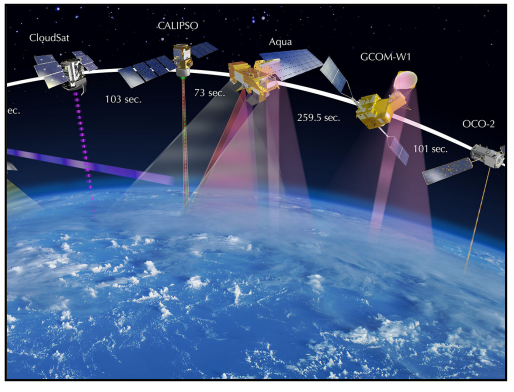
chemical synthesis

UAV routing

coast security



forest harvesting



Earth observation

Świechowski et al., Monte Carlo Tree Search: a review of recent modifications and applications (2023)

MCTS Environments

MCTS environments cover entire spectrum of properties.

We study MCTS under the same restrictions as before, i.e.,

- environment classification,
- problem solving method,
- objective of the agent and
- performance measure

are identical to Chapters G1–G3.

MCTS extensions exist that allow us to drop most restrictions.

Monte-Carlo Tree Search

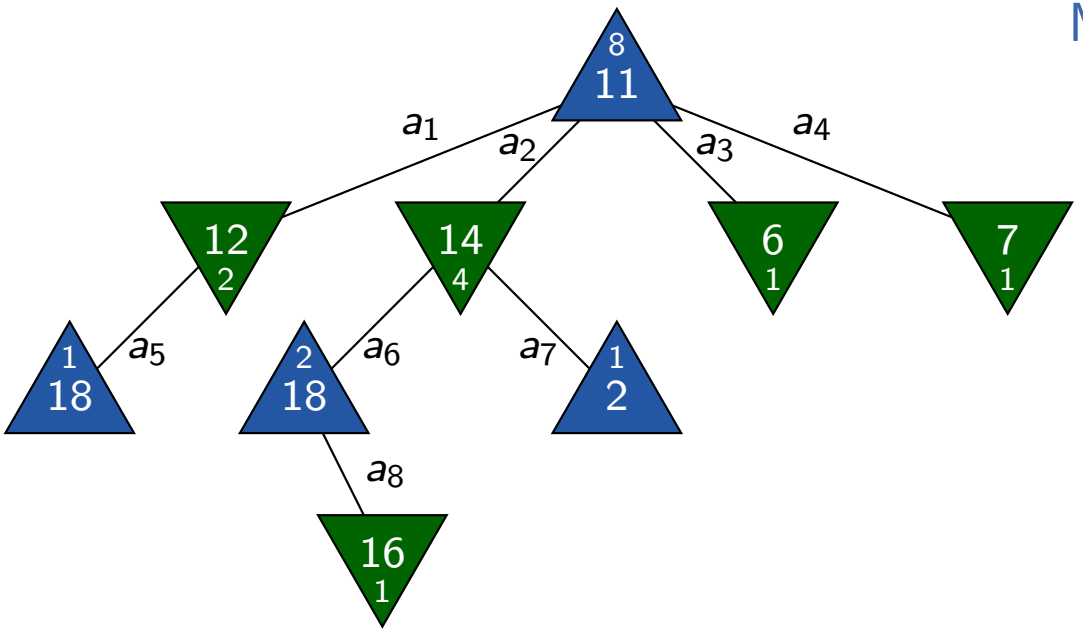
Data Structures

Monte-Carlo tree search

- is a tree search variant
  $\rightsquigarrow$ no closed list
- iteratively performs game simulations from the initial position (called trial or rollout)
  $\rightsquigarrow$ no (explicit) open list

$\rightsquigarrow$ MCTS nodes are the only used data structure

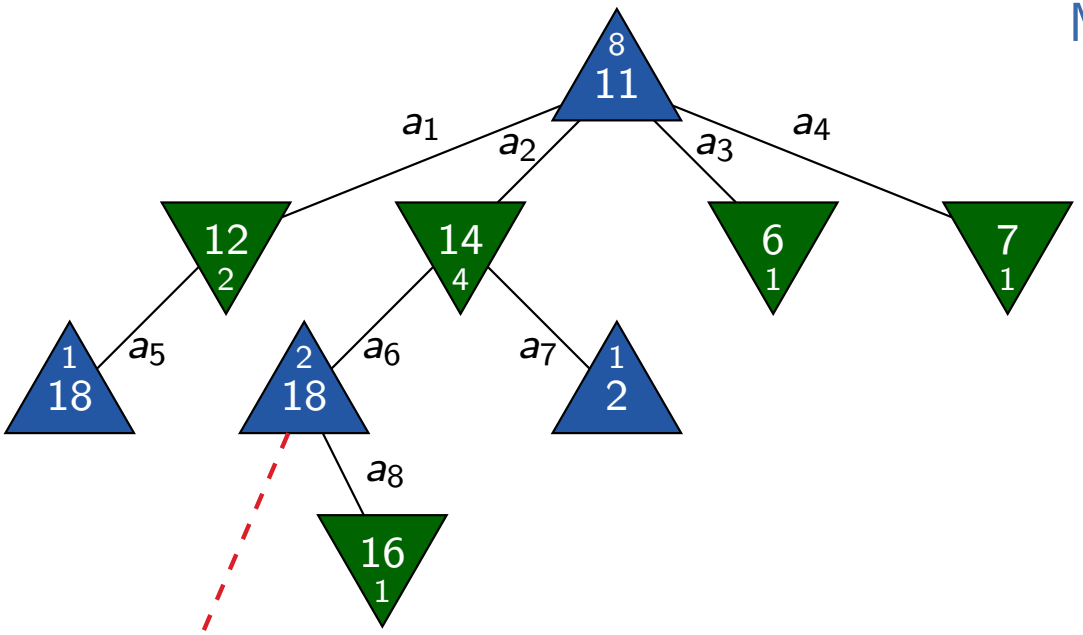
Data Structure: MCTS Nodes



MCTS nodes store

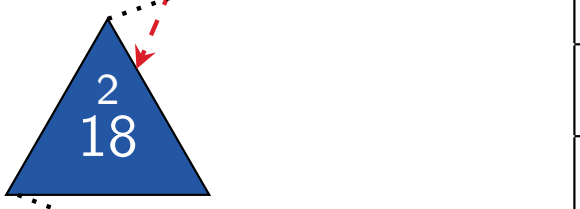
- a reached position
- how it was reached
- its successors
- a utility estimate ($\hat{v}$)
- a visit counter (N)
- possibly additional information

Data Structure: MCTS Nodes



MCTS nodes store

- a reached position
- how it was reached
- its successors
- a utility estimate ($\hat{v}$)
- a visit counter (N)
- possibly additional information



position:	not displayed
move:	a_6
successors:	[**none**, 16 1]
$\hat{v}$:	18
N:	2
...:	...

Monte-Carlo Tree Search: Idea

Monte-Carlo Tree Search (MCTS) ideas:

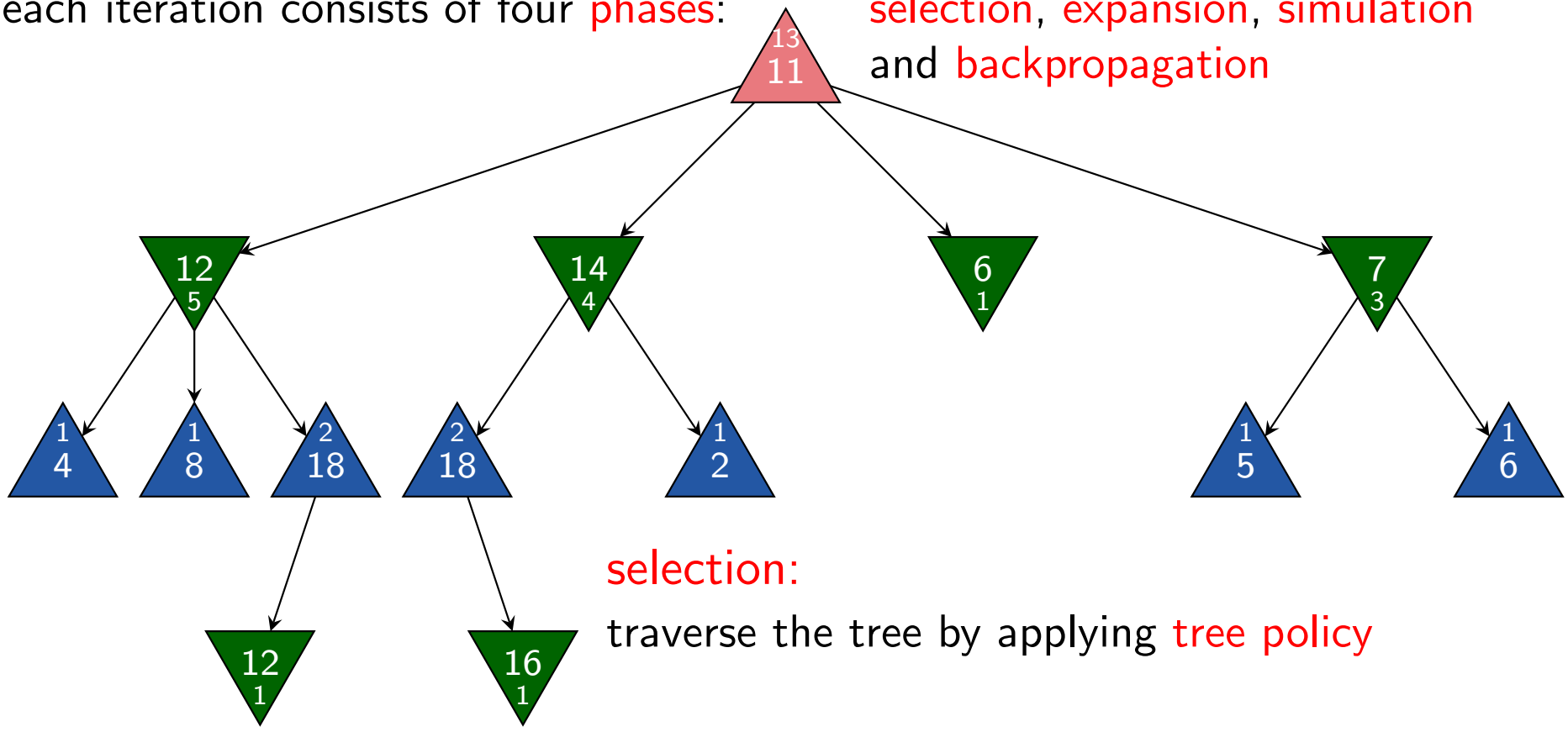
- build a partial game tree
- by performing trials as long as resources (deliberation time, memory) allow
- initially, the tree contains only the root node
- each trial adds (at most) one node to the tree

after termination, play the associated move of a successor of the root node with highest utility estimate

Idea and Example

each iteration consists of four phases:

selection, expansion, simulation and backpropagation



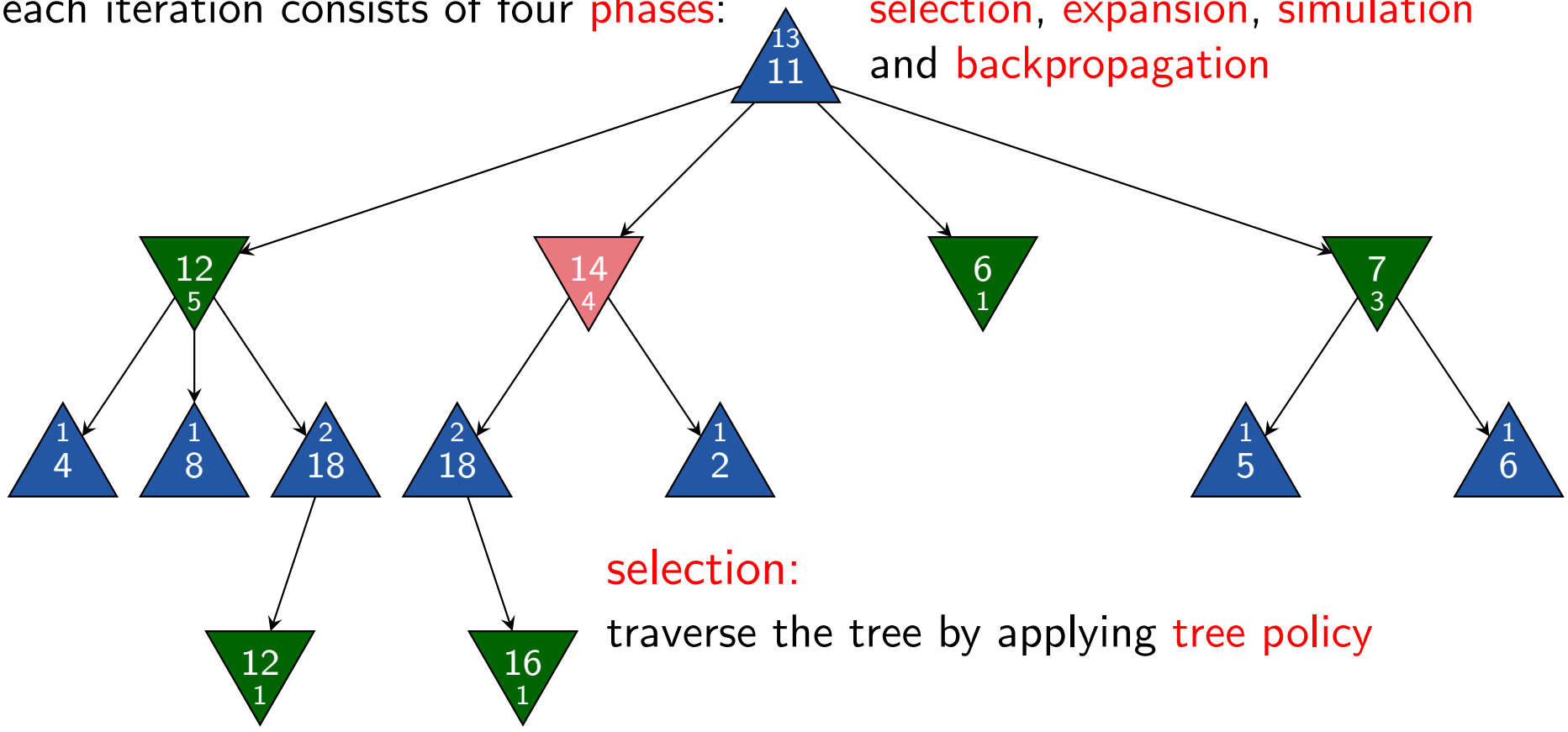
selection:

traverse the tree by applying tree policy

Idea and Example

each iteration consists of four phases:

selection, expansion, simulation and backpropagation

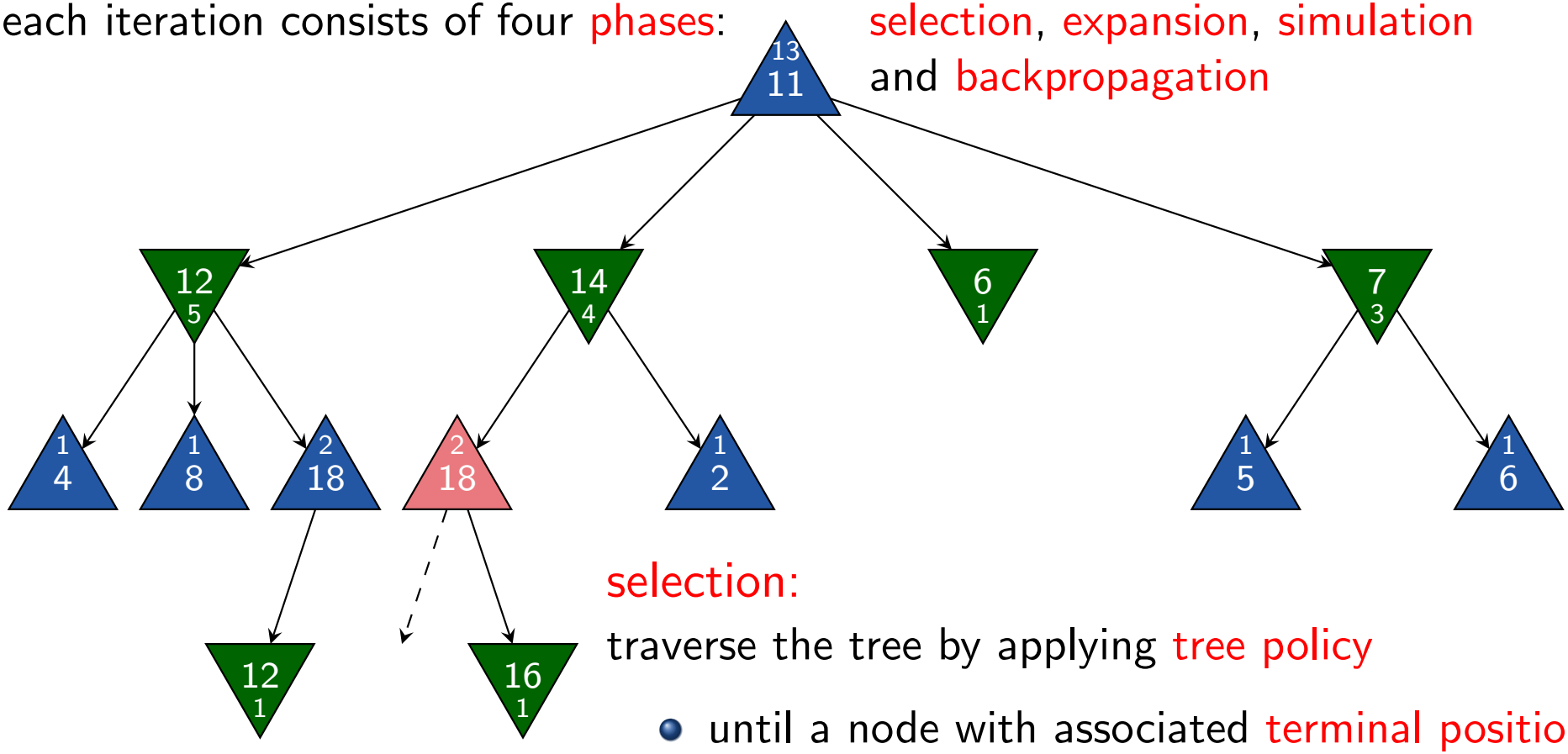


selection:

traverse the tree by applying tree policy

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



selection:

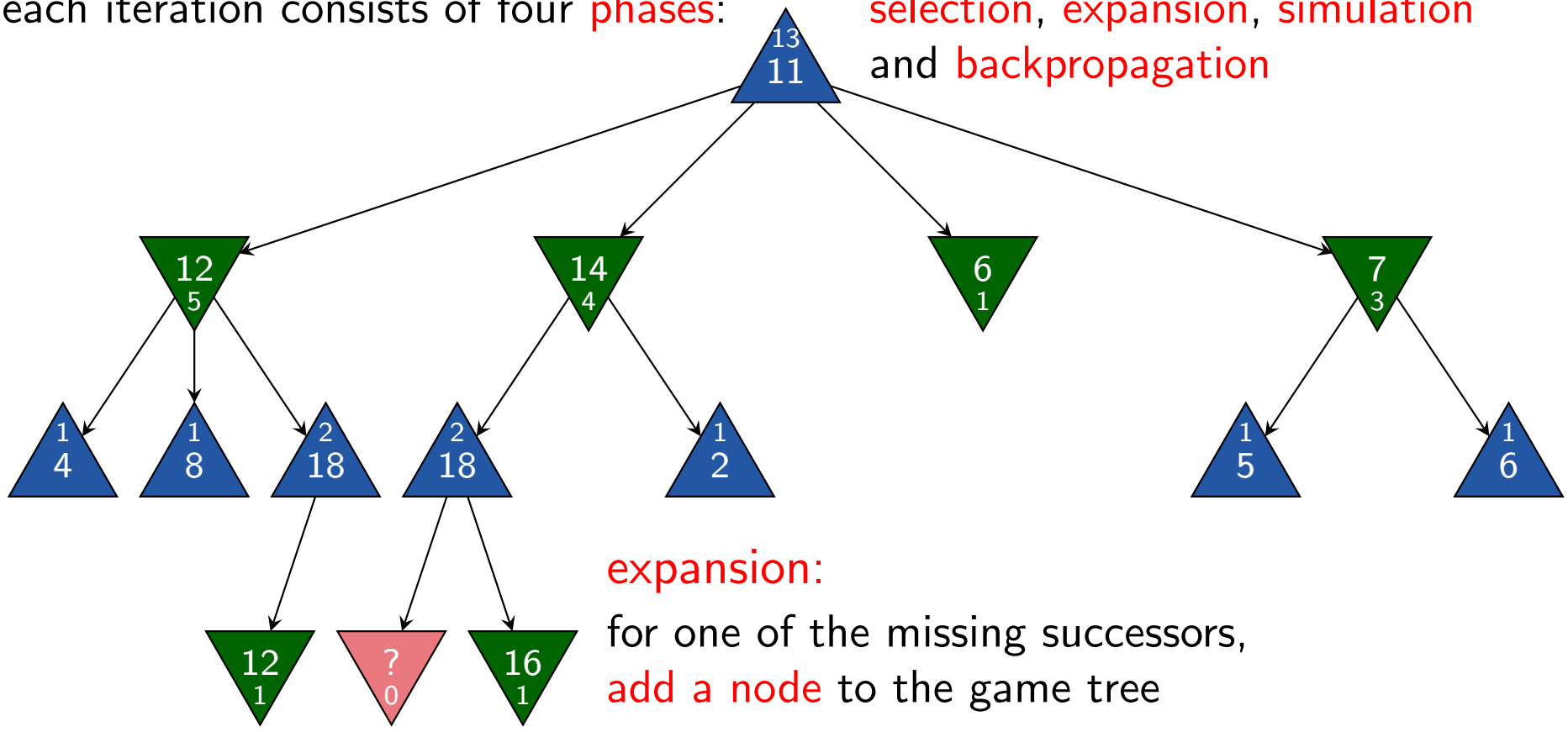
traverse the tree by applying tree policy

- until a node with associated terminal position
- or a node with missing successor is reached

Idea and Example

each iteration consists of four phases:

selection, expansion, simulation and backpropagation

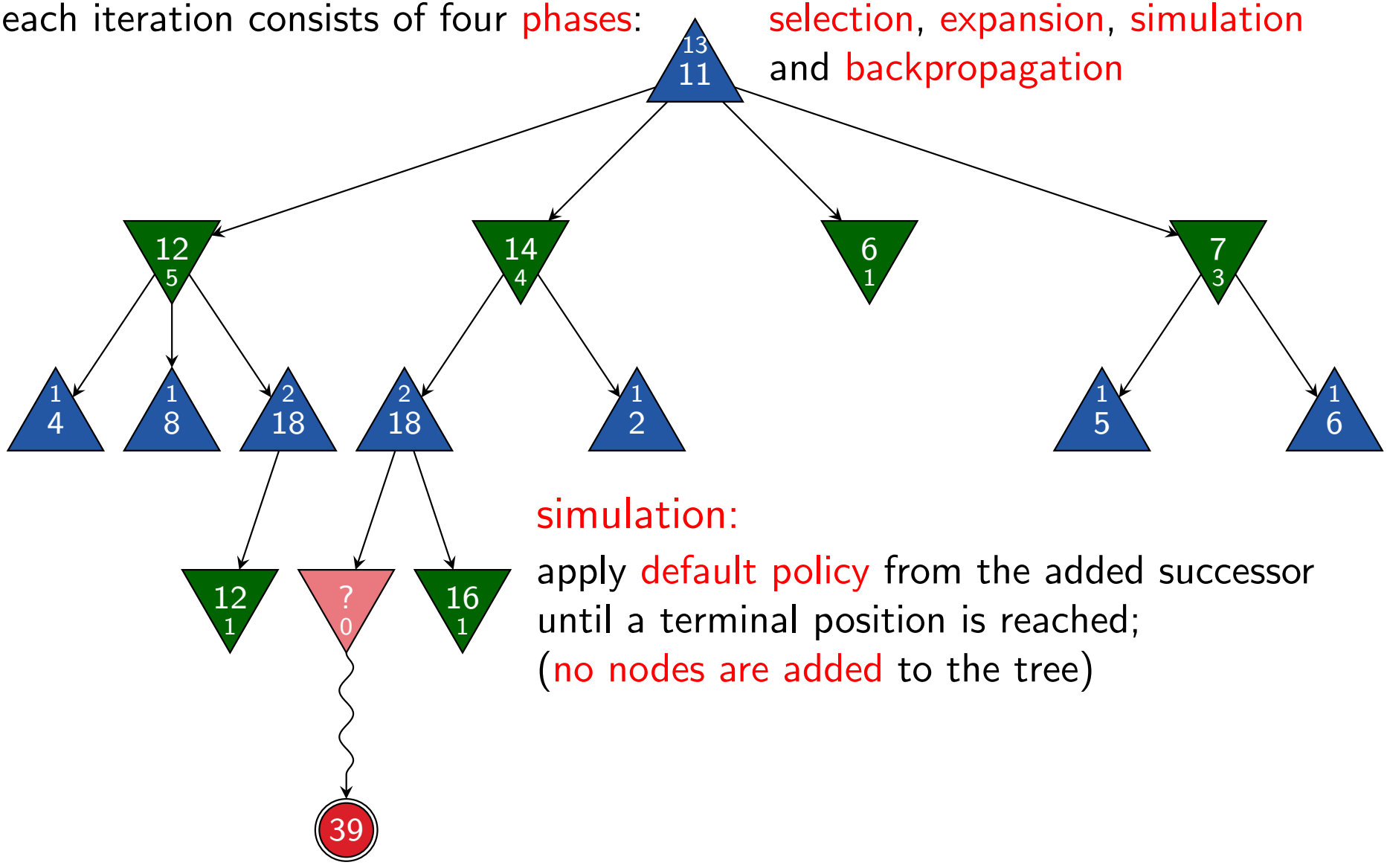


expansion:

for one of the missing successors, add a node to the game tree

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation

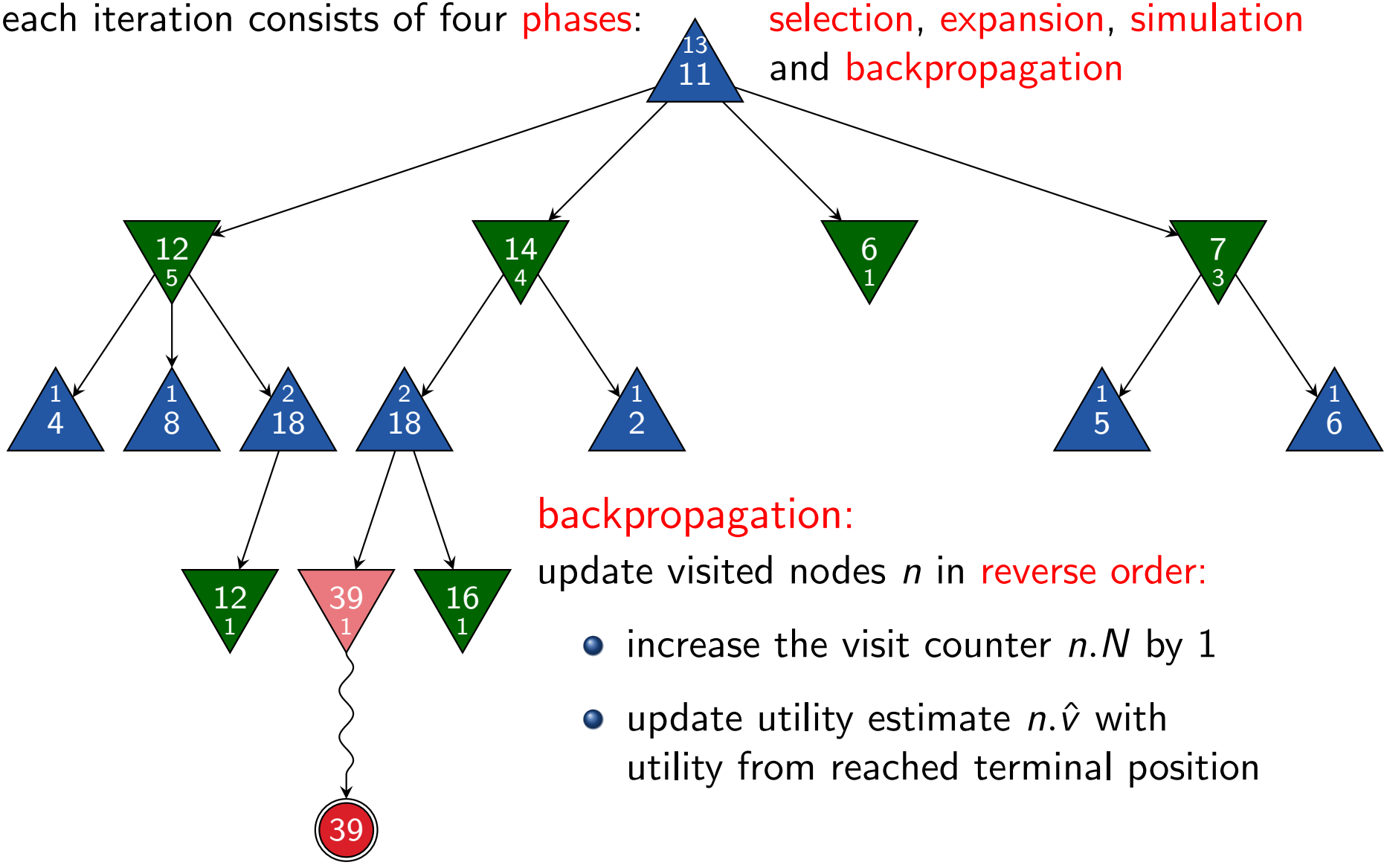


simulation:

apply default policy from the added successor until a terminal position is reached;
(no nodes are added to the tree)

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



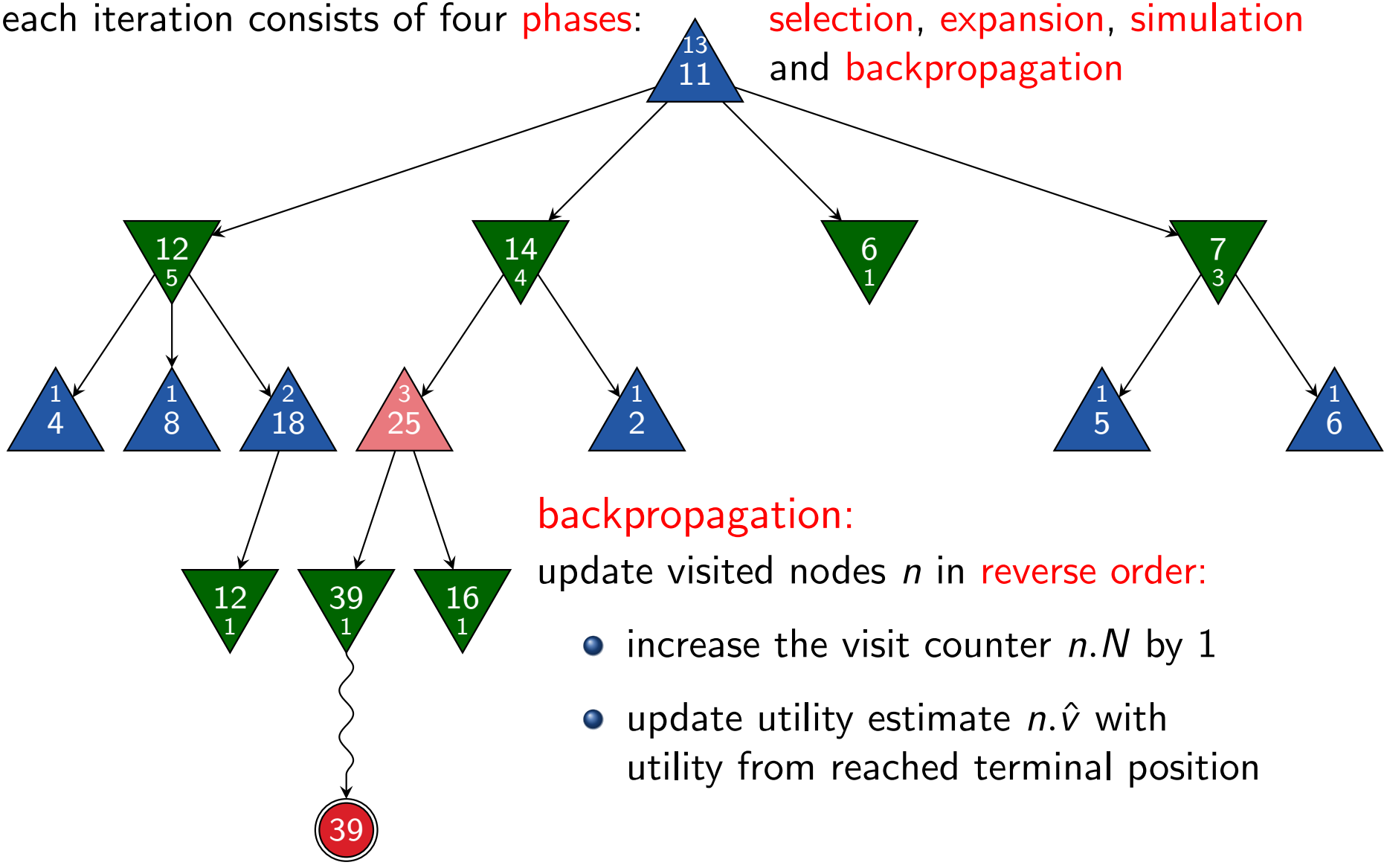
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



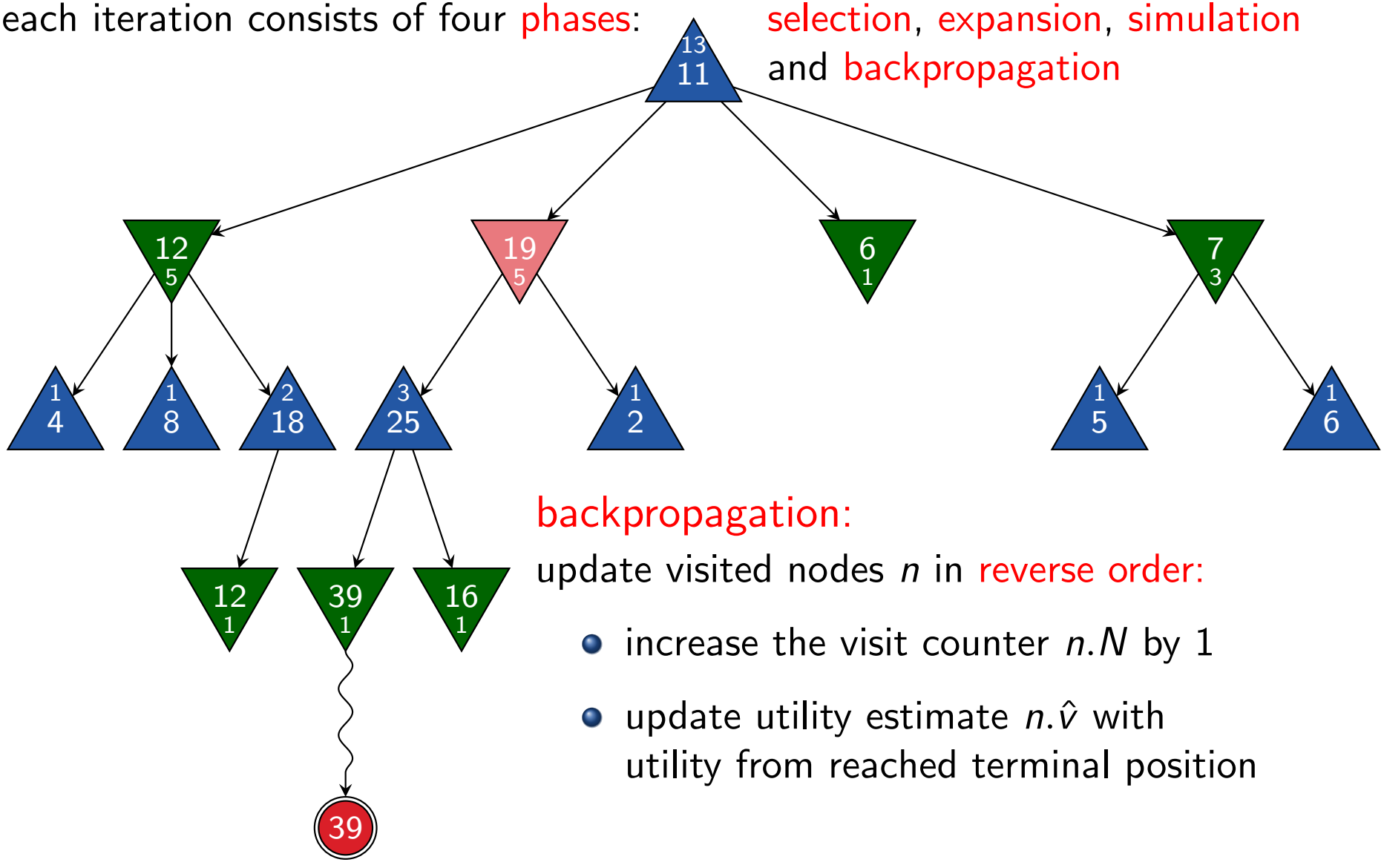
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



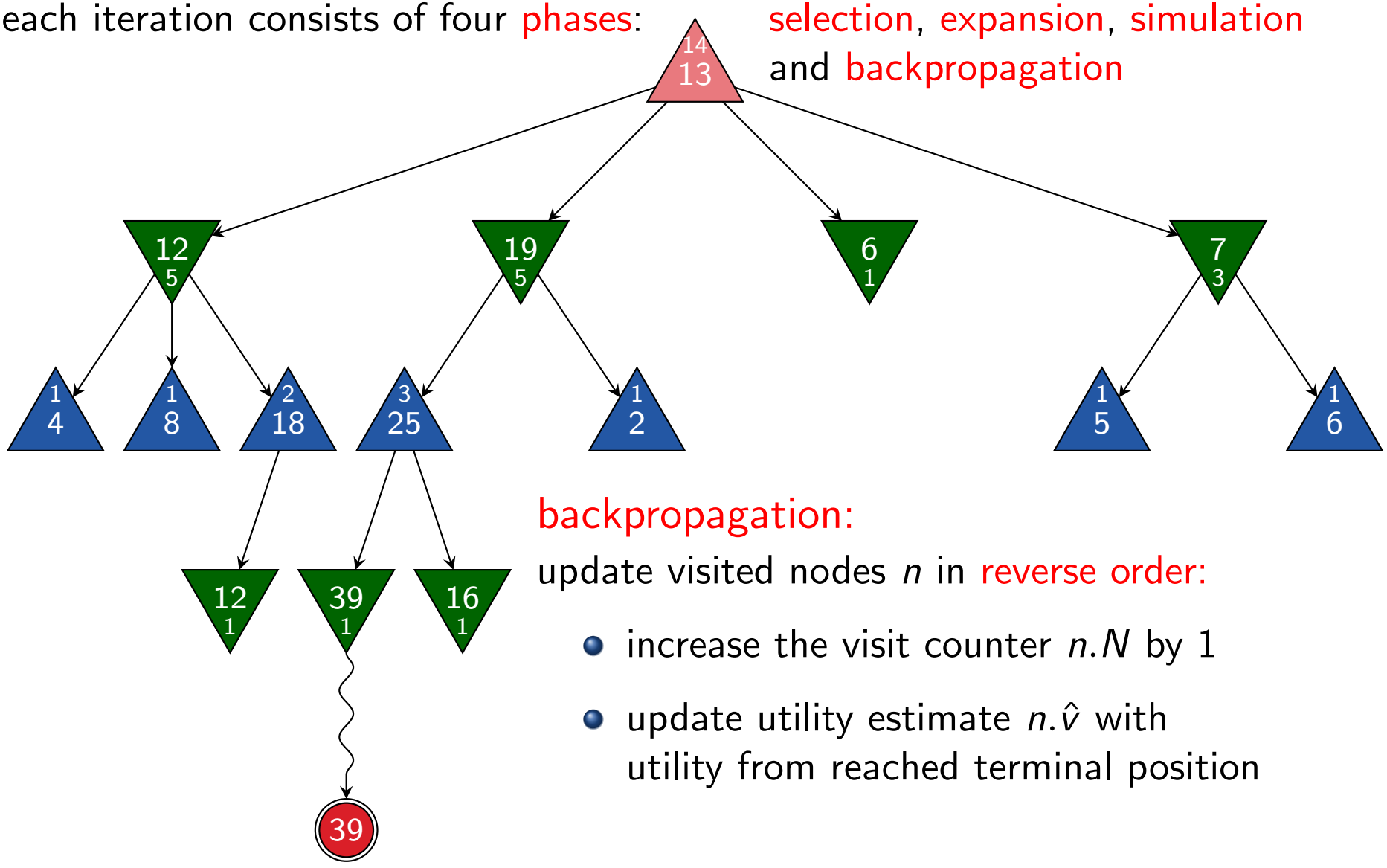
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



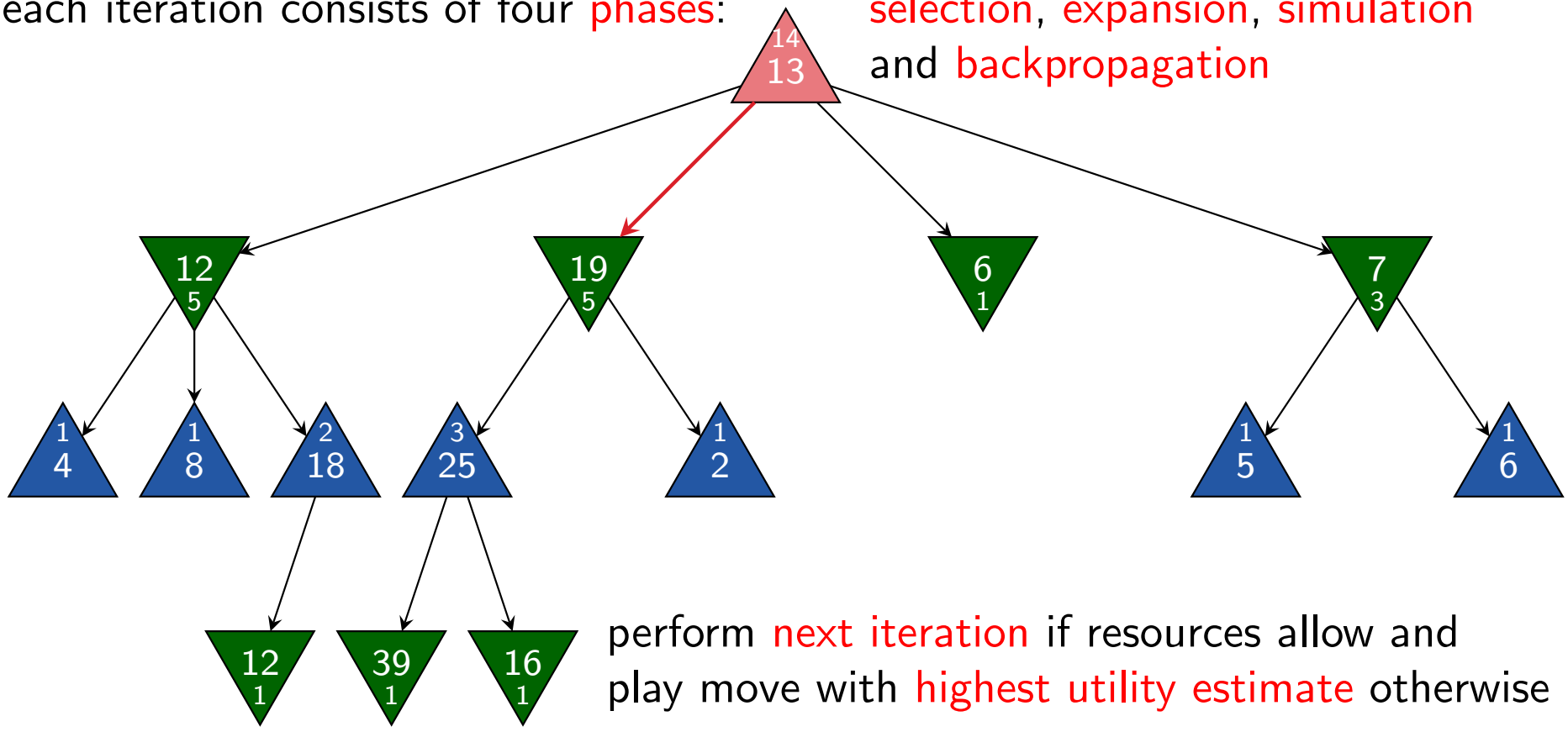
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



perform next iteration if resources allow and play move with highest utility estimate otherwise

Monte-Carlo Tree Search: Pseudo-Code

Monte-Carlo Tree Search

n_0 := create_root_node()
while time_allows():
    visit_node(n_0)
n_{best} := $\arg\max_{n \in \text{succ}(n_0)} n.\hat{v}$
return n_{best}.move

Monte-Carlo Tree Search: Pseudo-Code

function visit_node(n)

if is_terminal(n.position):
    $utility := utility(n.\text{position})$
else:
    $s := n.\text{get\_unvisited\_successor}()$
    if s **is none**:
        $n' := \text{apply\_tree\_policy}(n)$
        $utility := \text{visit\_node}(n')$
    else:
        $utility := \text{simulate\_game}(s)$
        $n.\text{add\_and\_initialize\_child\_node}(s, utility)$
$n.N := n.N + 1$
$n.\hat{v} := n.\hat{v} + \frac{utility - n.\hat{v}}{n.N}$
return *utility*

Summary

Summary

- Monte-Carlo methods compute averages over a number of random samples.
- Monte-Carlo Tree Search (MCTS) algorithms simulate a playout of the game
- and iteratively build a search tree, adding (at most) one node in each iteration.
- MCTS is parameterized by a tree policy and a default policy.

Foundations of Artificial Intelligence

G6. Board Games: Monte-Carlo Tree Search Variants

Malte Helmert

University of Basel

May 21, 2025

Board Games: Overview

chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Monte-Carlo Tree Search: Pseudo-Code

function visit_node(n)

if is_terminal(n.position):
    $utility := utility(n.\text{position})$
else:
    $s := n.\text{get\_unvisited\_successor}()$
    if s **is none**:
        $n' := \text{apply\_tree\_policy}(n)$
        $utility := \text{visit\_node}(n')$
    else:
        $utility := \text{simulate\_game}(s)$
        $n.\text{add\_and\_initialize\_child\_node}(s, utility)$
$n.N := n.N + 1$
$n.\hat{v} := n.\hat{v} + \frac{utility - n.\hat{v}}{n.N}$
return *utility*

Simulation Phase

Simulation Phase

idea: determine initial utility estimate by simulating game following a default policy

Definition (default policy)

Let $\mathcal{S} = \langle S, A, T, s_I, S_G, \mathit{utility}, \mathit{player} \rangle$ be a game.
A default policy for $\mathcal{S}$ is a mapping $\pi_{\mathrm{def}} : S \times A \mapsto [0, 1]$ s.t.

1. $\pi_{\mathrm{def}}(s, a) > 0$ implies that move a is applicable in position s
2. $\sum_{a \in A} \pi_{\mathrm{def}}(s, a) = 1$ for all $s \in S$

In the call to simulate_game(s),

- the default policy is applied starting from position s (determining decisions for both players)
- until a terminal position s_G is reached
- and $\mathit{utility}(s_G)$ is returned.

Implementations

"standard" implementation: Monte-Carlo random walk

- in each position, select a move uniformly at random
- until a terminal position is reached
- policy very cheap to compute
- uninformed $\rightsquigarrow$ often not sufficient for good results
- not always cheap to simulate

alternative: game-specific default policy

- hand-crafted or
- learned offline

Gelly and Silver, Combining Online and Offline Knowledge in UCT (ICML, 2007)

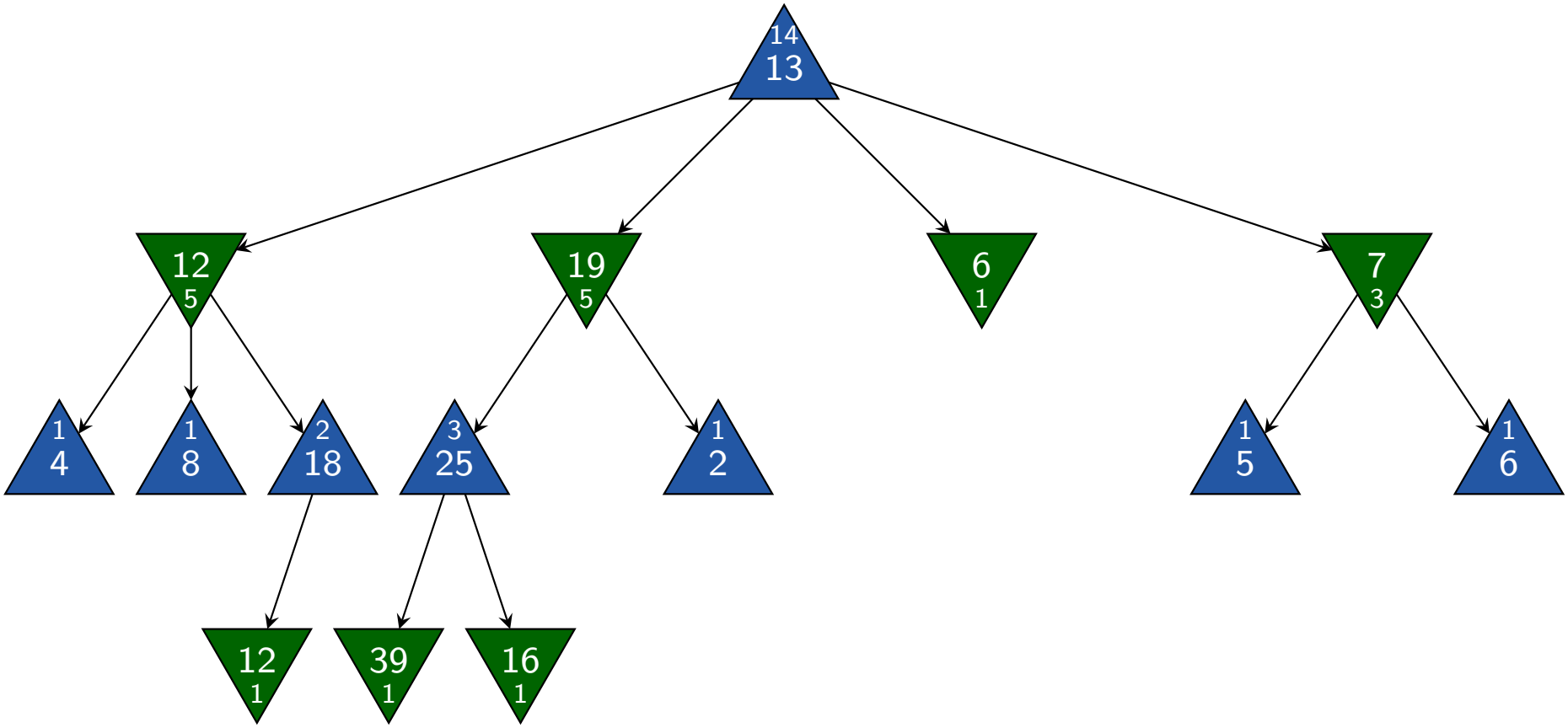
Default Policy vs. Evaluation Function

- default policy simulates a game to obtain utility estimate
  $\rightsquigarrow$ default policy must be evaluated in many positions
- if default policy is expensive to compute or poorly informed, simulations are expensive
- observe: simulating a game to the end is just a specific implementation of an evaluation function
- many modern implementations replace default policy with evaluation function that directly computes a utility estimate

$\rightsquigarrow$ MCTS becomes a heuristic search algorithm

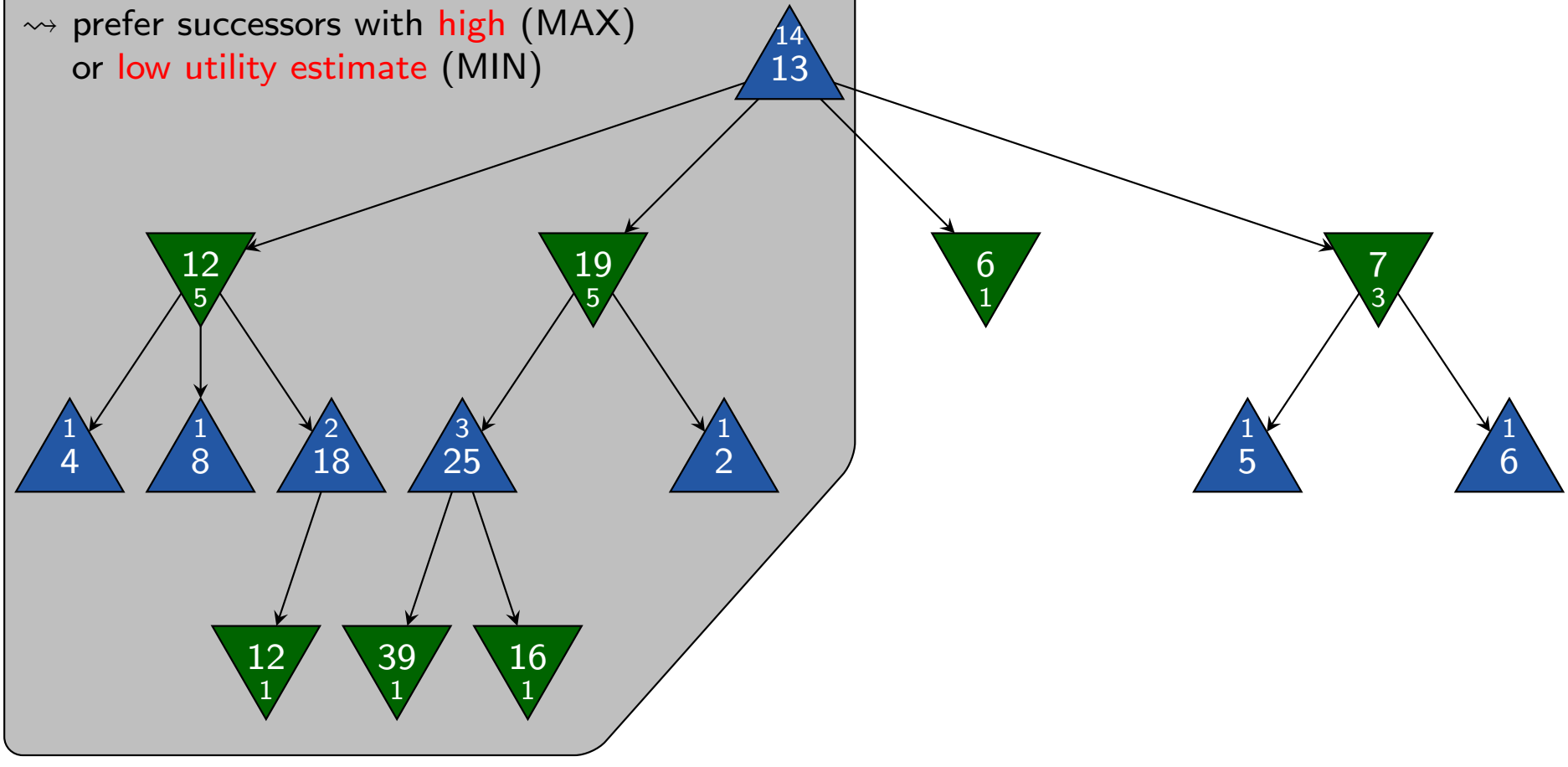
Tree Policy

Objective of Tree Policy (1)

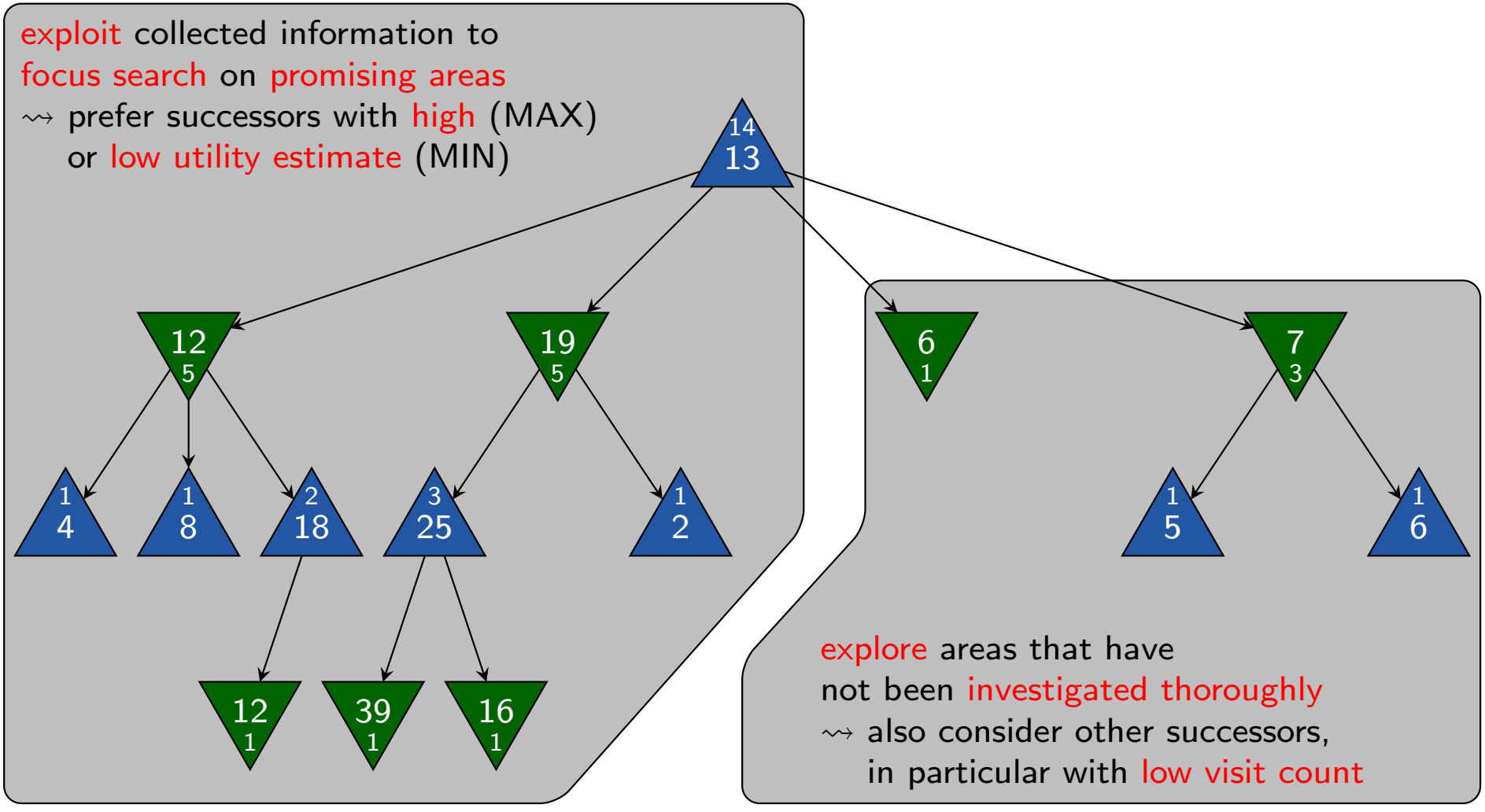


Objective of Tree Policy (1)

exploit collected information to
focus search on promising areas
$\rightsquigarrow$ prefer successors with high (MAX)
or low utility estimate (MIN)



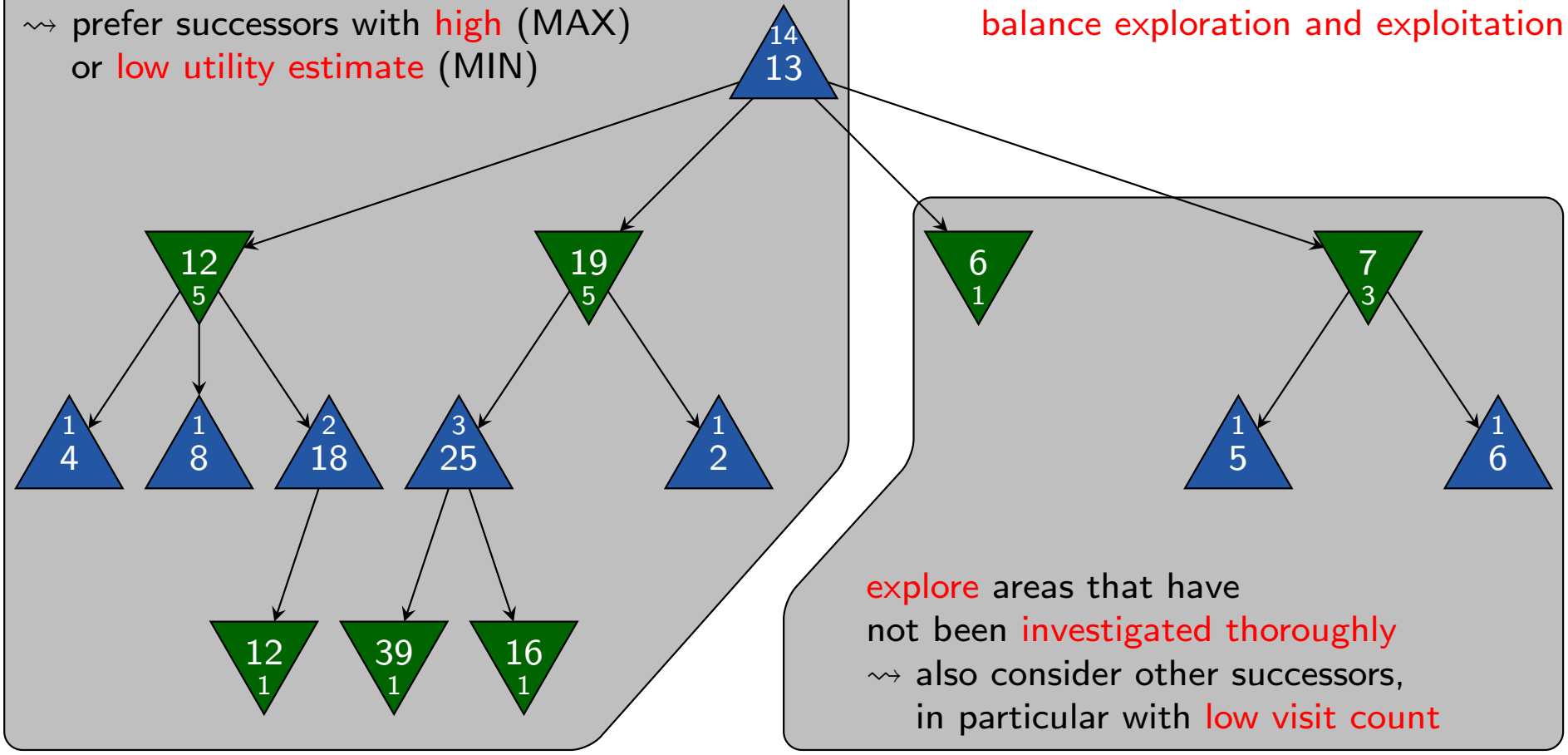
Objective of Tree Policy (1)



Objective of Tree Policy (1)

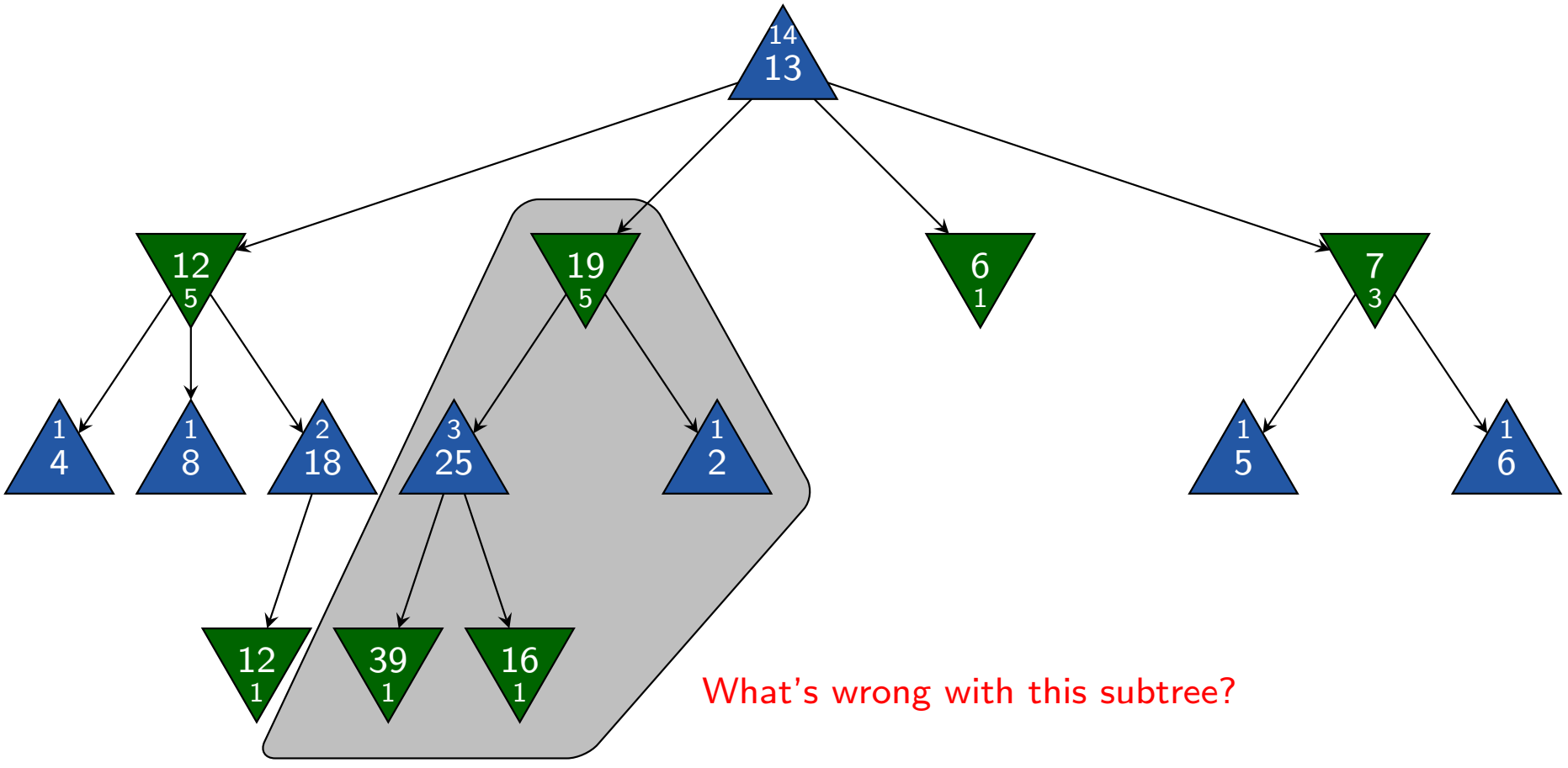
exploit collected information to
focus search on promising areas
⇝ prefer successors with high (MAX)
or low utility estimate (MIN)

These are contradictory objectives!
⇝ 1st central challenge for tree policy:
balance exploration and exploitation

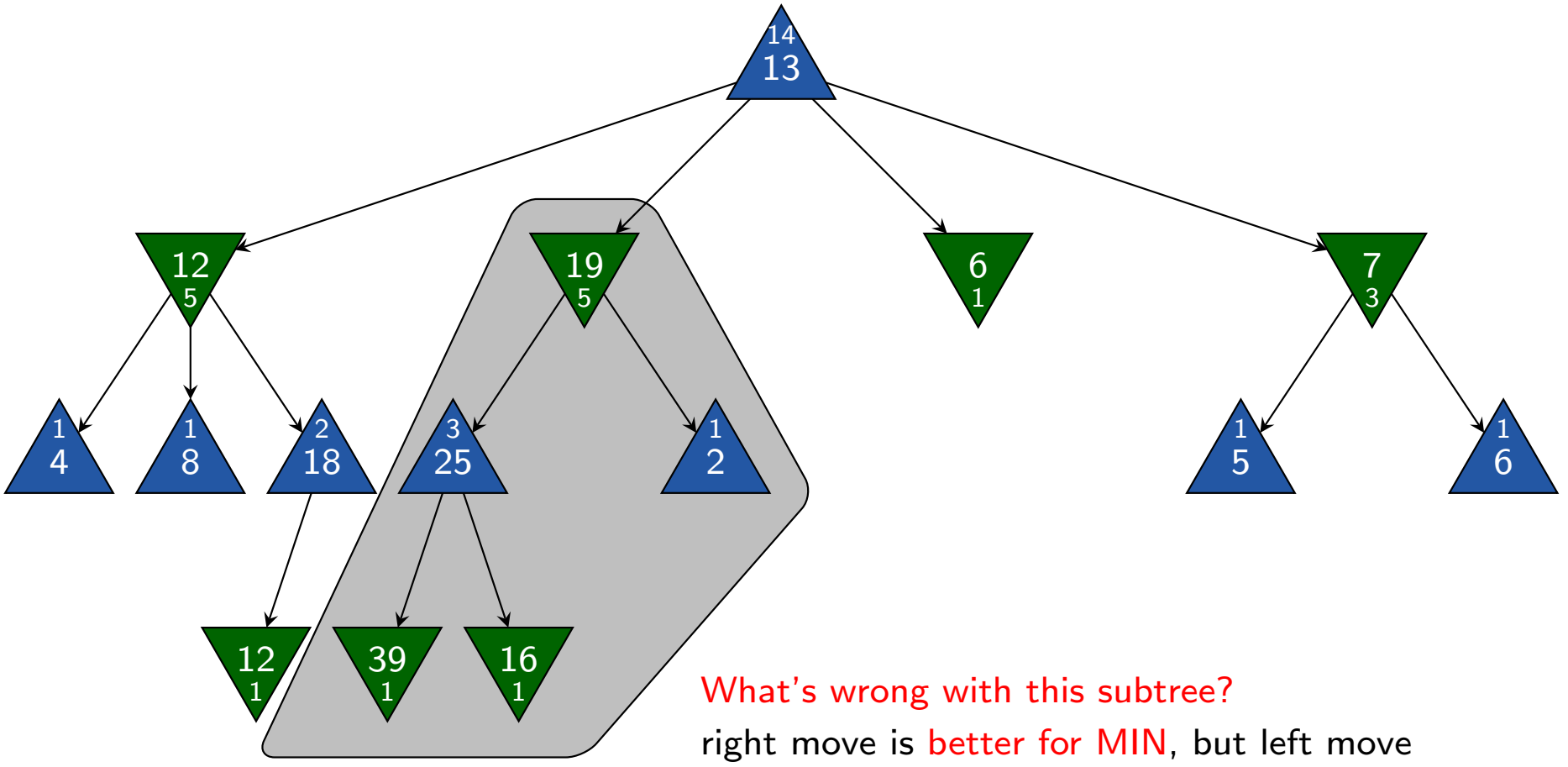


explore areas that have
not been investigated thoroughly
⇝ also consider other successors,
in particular with low visit count

Objective of Tree Policy (2)



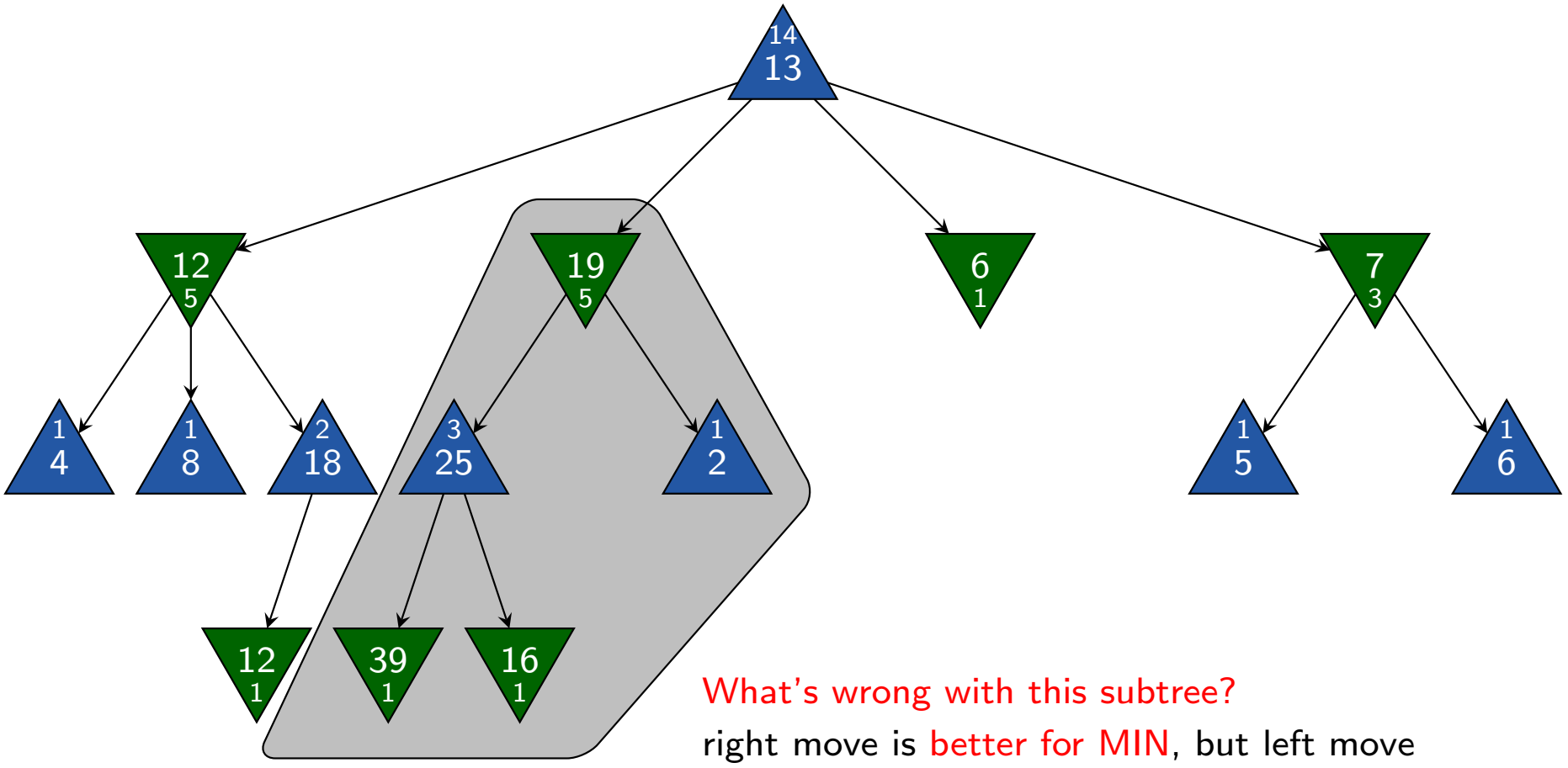
Objective of Tree Policy (2)



What's wrong with this subtree?

right move is better for MIN, but left move has higher influence on utility estimate

Objective of Tree Policy (2)



What's wrong with this subtree?

right move is better for MIN, but left move has higher influence on utility estimate

$\rightsquigarrow$ 2nd central challenge for tree policy: exploit much more often than explore (in the limit)

Asymptotic Optimality

Definition (asymptotic optimality)

Let $\mathcal{S}$ be a game with set of positions S.
Let $v^*(s)$ denote the (true) utility of position $s \in S$.

Let $n.\hat{v}^k$ denote the utility estimate
of a search node n after k trials.

An MCTS algorithm is asymptotically optimal if

$$\lim_{k \to \infty} n.\hat{v}^k = v^*(n.\text{position})$$

for all search nodes n.

Asymptotic Optimality

a tree policy is asymptotically optimal if

- it explores forever:
  - every position is eventually added to the game tree and visited infinitely often (requires that the game tree is finite)
  - $\rightsquigarrow$ after a finite number of trials, all trials end in a terminal position and the default policy is no longer used
- and it is greedy in the limit:
  - the probability that an optimal move is selected converges to 1
  - $\rightsquigarrow$ in the limit, backups based on trials where only an optimal policy is followed dominate suboptimal backups

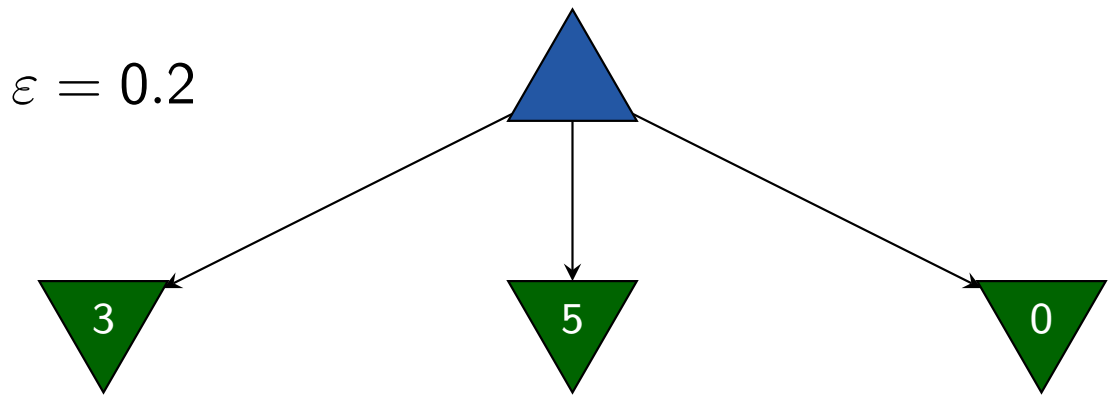
Tree Policy: Examples

ε-greedy: Idea and Example

- tree policy with constant parameter ε
- with probability $1 - \varepsilon$, pick a greedy move which leads to:
  - a successor with highest utility estimate (for MAX)
  - a successor with lowest utility estimate (for MIN)
- otherwise, pick a non-greedy successor uniformly at random

ε-greedy: Idea and Example

- tree policy with constant parameter ε
- with probability $1 - \varepsilon$, pick a greedy move which leads to:
  - a successor with highest utility estimate (for MAX)
  - a successor with lowest utility estimate (for MIN)
- otherwise, pick a non-greedy successor uniformly at random

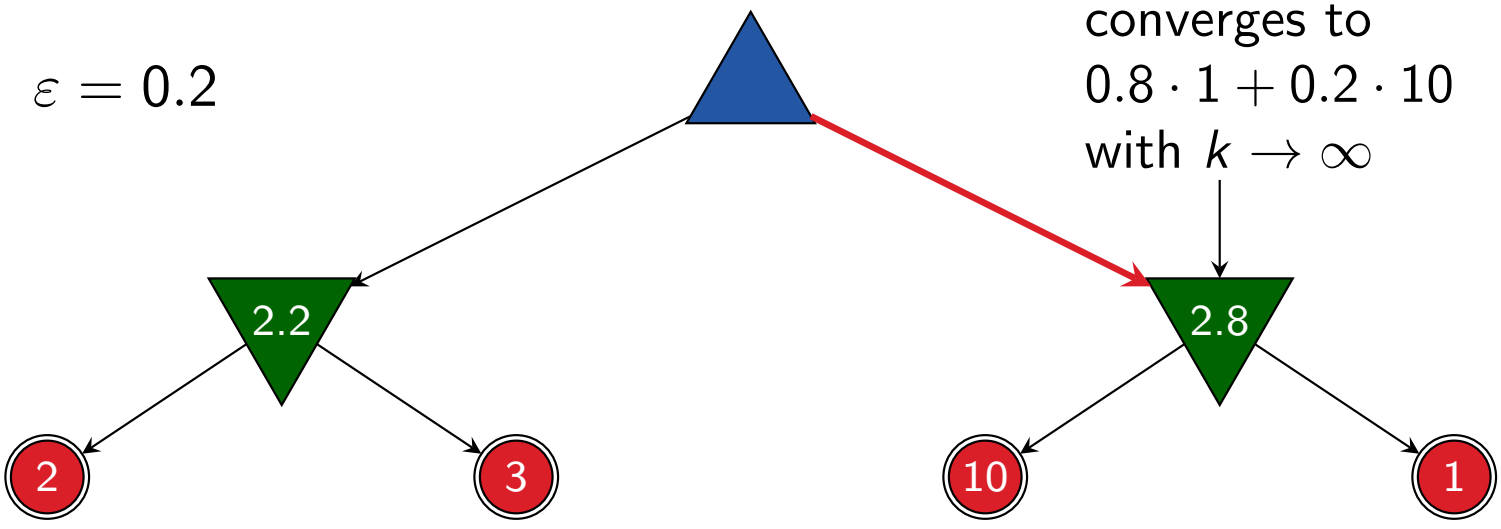


$\mathbb{P}(n_1) = 0.1$ $\mathbb{P}(n_2) = 0.8$ $\mathbb{P}(n_3) = 0.1$

($\mathbb{P}(n)$ denotes probability that successor n is selected)

ε-greedy: Optimality

ε-greedy is not asymptotically optimal:

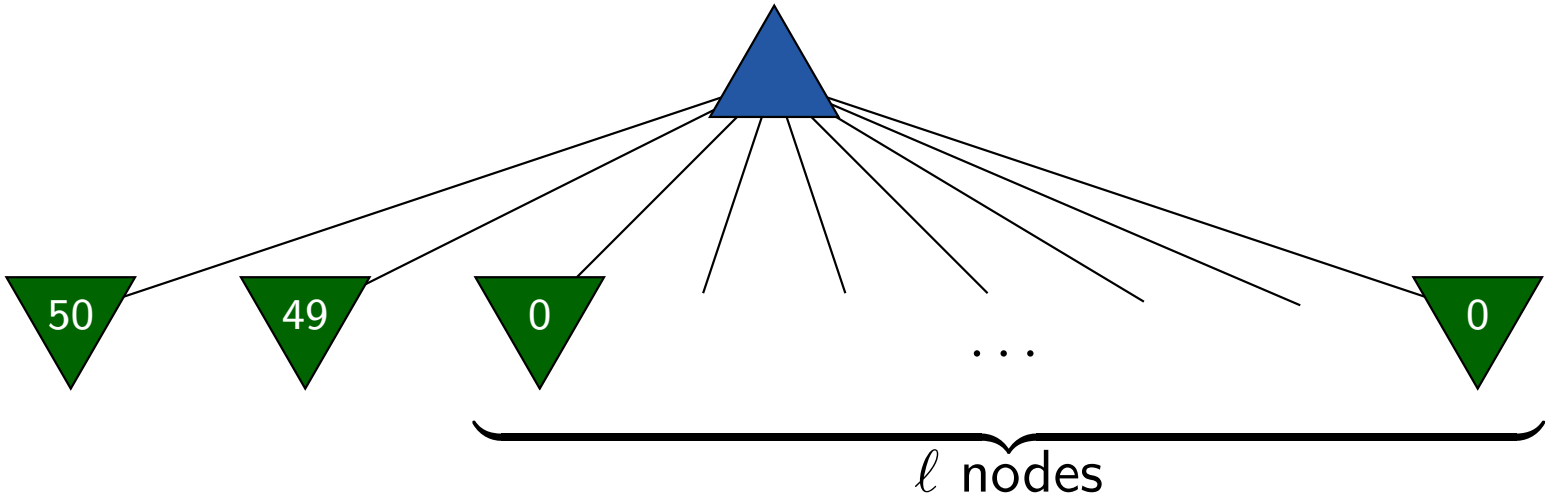


variants that are asymptotically optimal exist
(e.g., decaying ε, minimax backups)

ε-greedy: Weakness

problem:

when ε-greedy explores, all non-greedy moves are treated equally



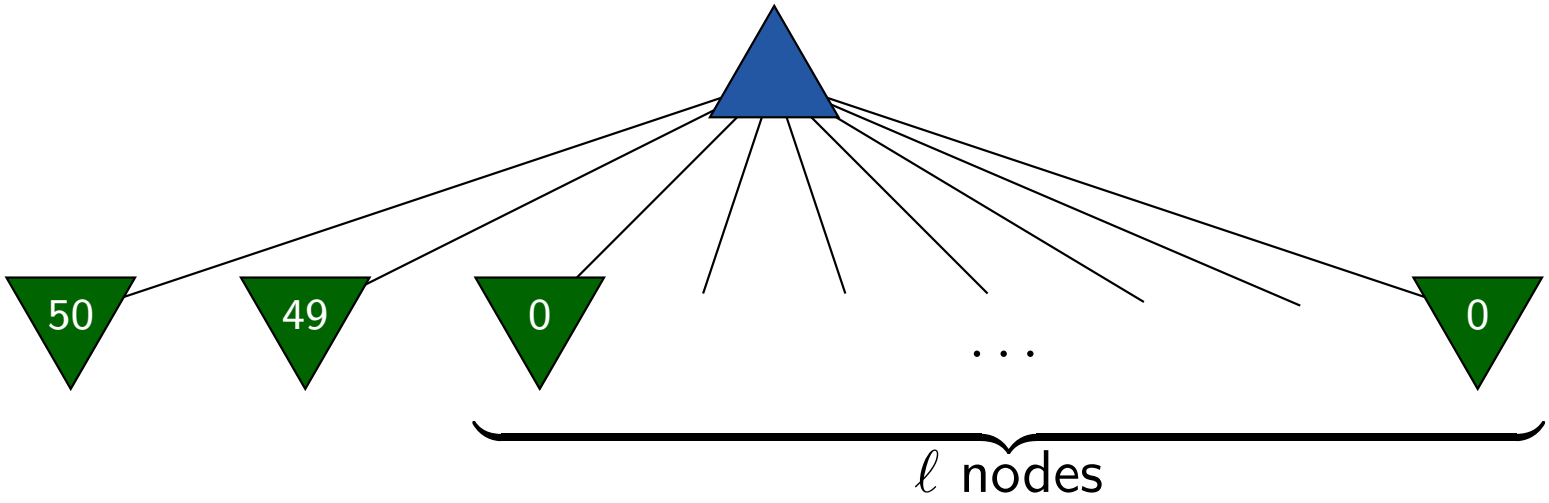
e.g., $\varepsilon = 0.2, \ell = 9$: $\mathbb{P}(n_1) = 0.8$, $\mathbb{P}(n_2) = 0.02$

Softmax: Idea and Example

- tree policy with constant parameter $\tau > 0$
- select moves with a frequency that directly relates to their utility estimate
- Boltzmann exploration selects moves proportionally to $\mathbb{P}(n) \propto e^{\frac{n.\hat{v}}{\tau}}$ for MAX and to $\mathbb{P}(n) \propto e^{\frac{-n.\hat{v}}{\tau}}$ for MIN

Softmax: Idea and Example

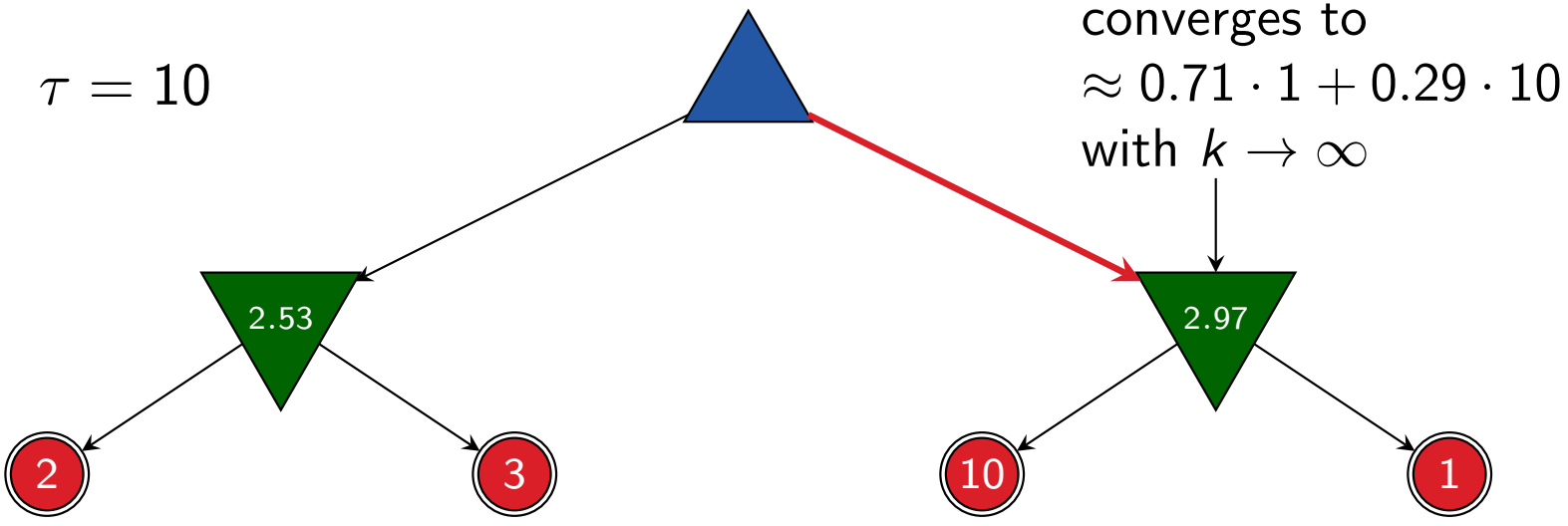
- tree policy with constant parameter $\tau > 0$
- select moves with a frequency that directly relates to their utility estimate
- Boltzmann exploration selects moves proportionally to $\mathbb{P}(n) \propto e^{\frac{n.\hat{v}}{\tau}}$ for MAX and to $\mathbb{P}(n) \propto e^{\frac{-n.\hat{v}}{\tau}}$ for MIN



e.g., $\tau = 10, \ell = 9$: $\mathbb{P}(n_1) \approx 0.51$, $\mathbb{P}(n_2) \approx 0.46$

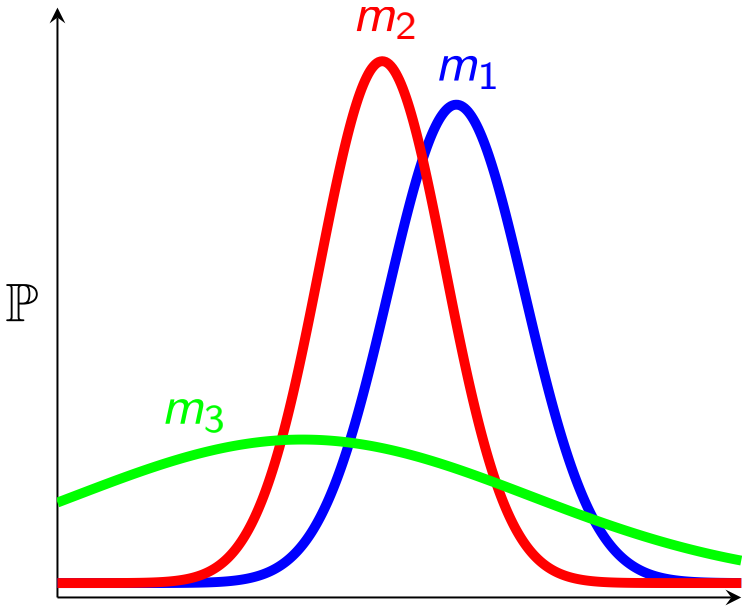
Boltzmann exploration: Optimality

Boltzmann exploration is not asymptotically optimal:

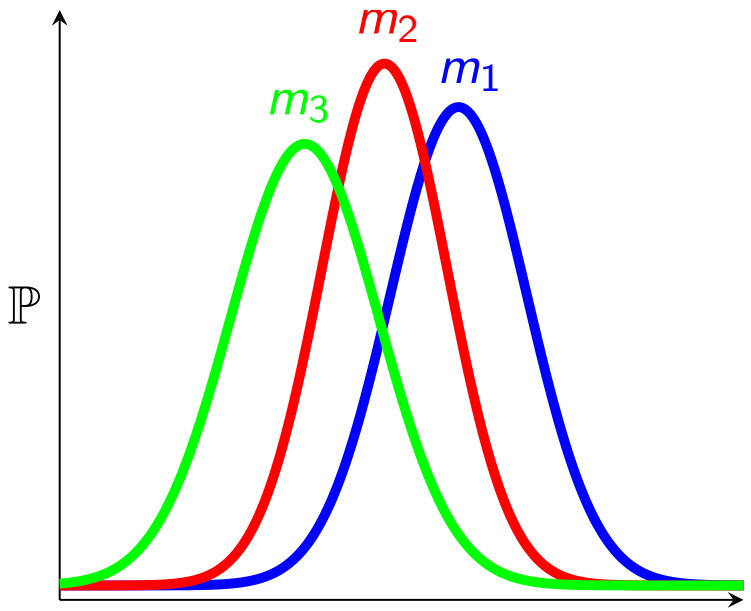


variants that are asymptotically optimal exist
(e.g., decaying τ, minimax backups)

Boltzmann Exploration: Weakness



scenario 1: high variance for m_3



scenario 2: low variance for m_3

- Boltzmann exploration only considers mean of sampled utilities for the given moves
- as we sample the same node many times, we can also gather information about variance (how reliable the information is)
- Boltzmann exploration ignores the variance, treating the two scenarios equally

Upper Confidence Bounds: Idea

balance exploration and exploitation by preferring moves that

- have been successful in earlier iterations (exploit)
- have been selected rarely (explore)

Upper Confidence Bounds: Idea

upper confidence bound for MAX:

- select successor n' of n that maximizes $n'.\hat{v} + B(n')$
- based on utility estimate $n'.\hat{v}$
- and a bonus term $B(n')$
- select $B(n')$ such that $v^*(n'.\text{position}) \leq n'.\hat{v} + B(n')$ with high probability
- idea: $n'.\hat{v} + B(n')$ is an upper confidence bound on $n'.\hat{v}$ under the collected information

(for MIN: maximize $-n'.\hat{v} + B(n')$)

Upper Confidence Bounds: UCB1

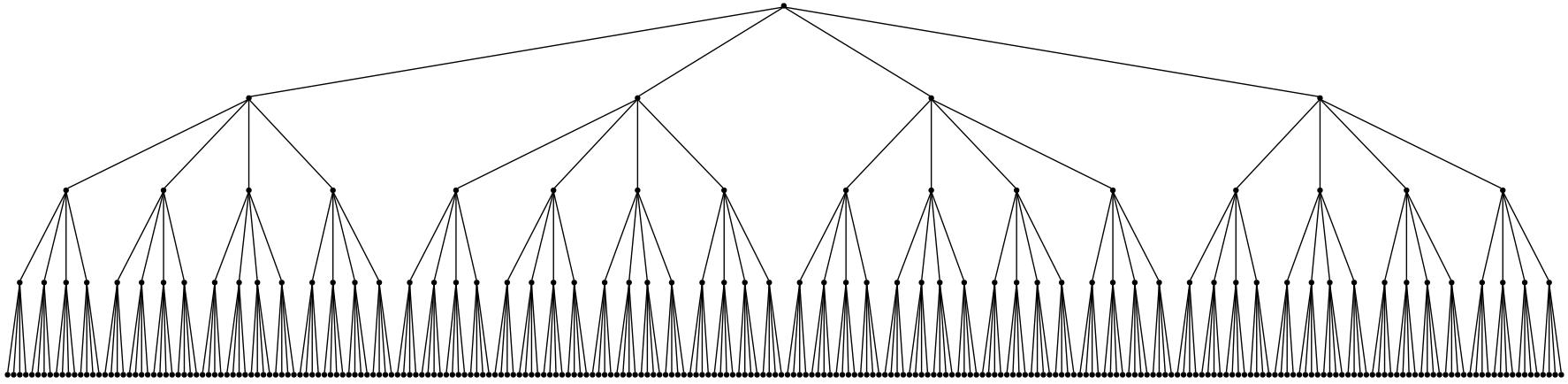
- use $B(n') = \sqrt{\frac{2 \cdot \ln n.N}{n'.N}}$ as bonus term
- bonus term is derived from Chernoff-Hoeffding bound, which
  - gives the probability that a sampled value (here: $n'.\hat{v}$)
  - is far from its true expected value (here: $v^*(n'.\text{position})$)
  - in dependence of the number of samples (here: $n'.N$)
- picks an optimal move exponentially more often in the limit

UCB1 is asymptotically optimal.

Comparison of Game Algorithms

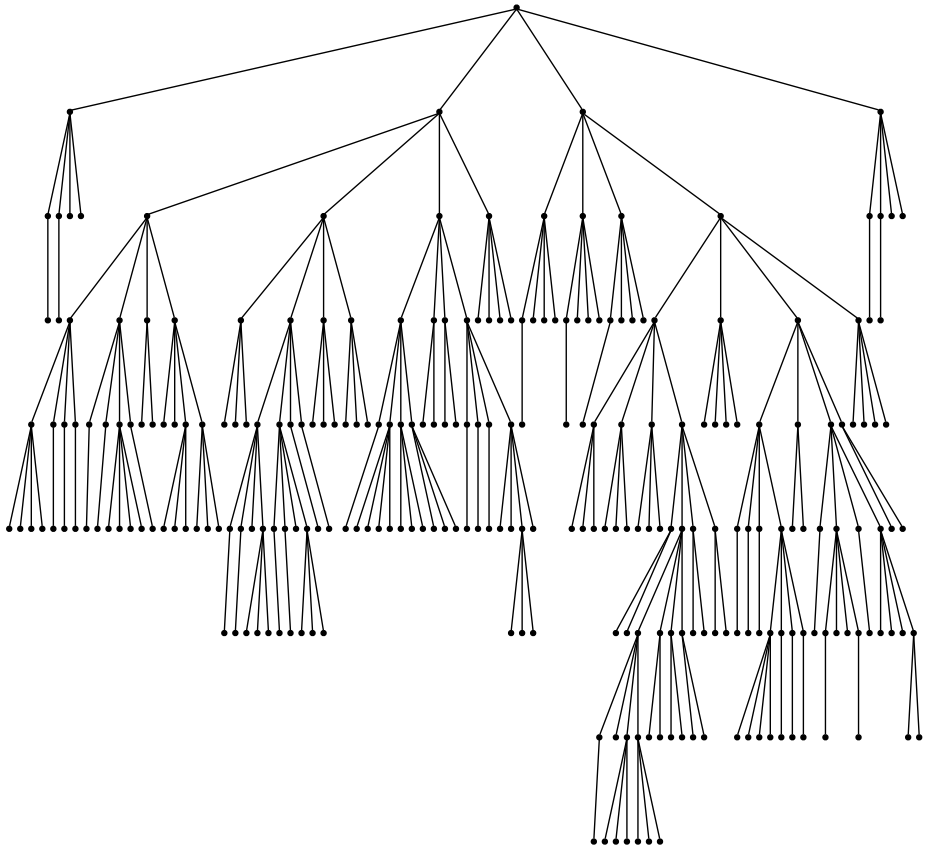
Minimax Tree

full tree up to depth 4



alpha-beta search with same effort:
$\rightsquigarrow$ depth 6–8 with good move ordering

MCTS Tree



Summary

Summary

- tree policy is crucial for MCTS
  - ϵ-greedy favors greedy moves and treats all others equally
  - Boltzmann exploration selects moves proportionally to an exponential function of their utility estimates
  - UCB1 favors moves that were successful in the past or have been explored rarely
- for each, there are applications where they perform best
- good default policies are domain-dependent and hand-crafted or learned offline
- using evaluation functions instead of a default policy often pays off