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
- 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

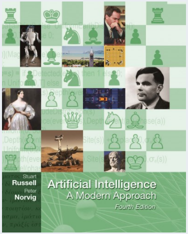
additional (optional) course material:

- 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

⇝ 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?

⇝ no generally accepted definition!

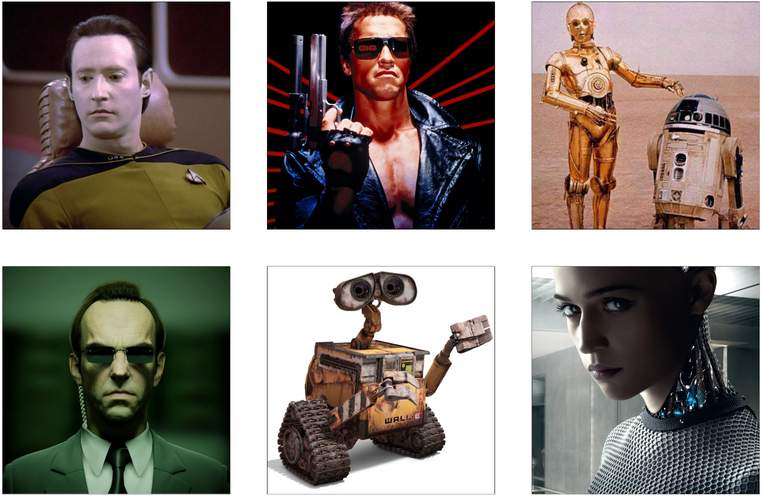
often pragmatic definitions:

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

in this chapter: some common attempts at defining AI

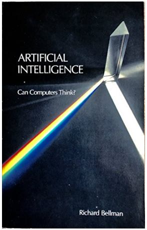
What Do We Mean by Artificial Intelligence?

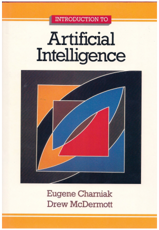
what pop culture tells us:

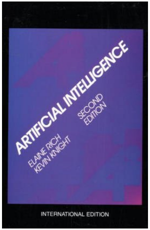


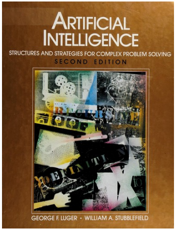
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:



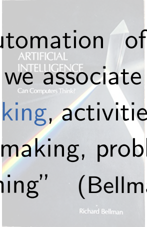
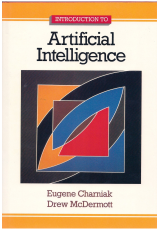
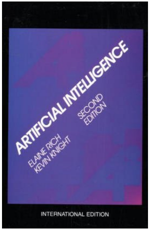
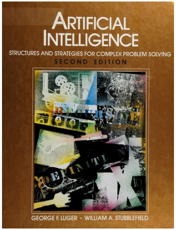






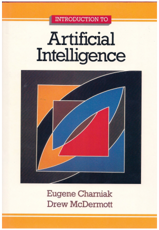
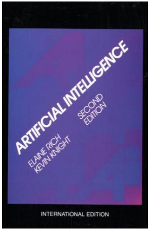
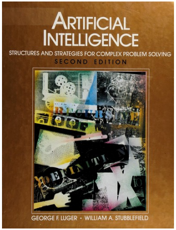
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

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

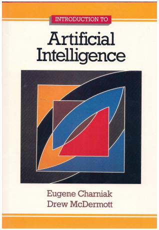
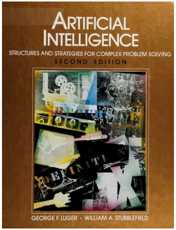
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

 thinking like humans	
 	

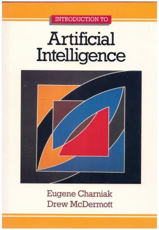
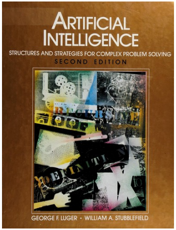
What is AI: Humanly vs. Rationally; Thinking vs. Acting

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)	

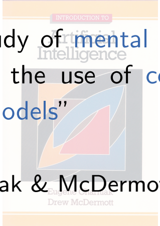
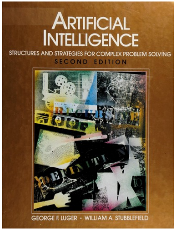
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

 thinking like humans	
 acting like humans	

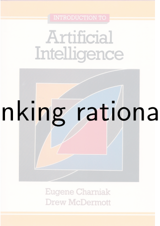
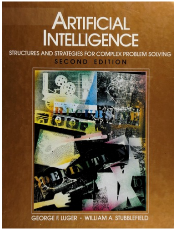
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

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

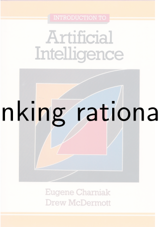
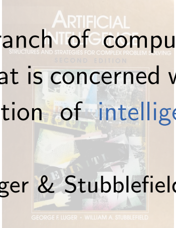
What is AI: Humanly vs. Rationally; Thinking vs. Acting

what scientists tell us:

 thinking like humans	 thinking rationally
 acting like humans	

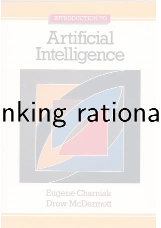
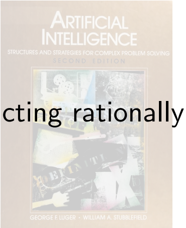
What is AI: Humanly vs. Rationally; Thinking vs. Acting

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

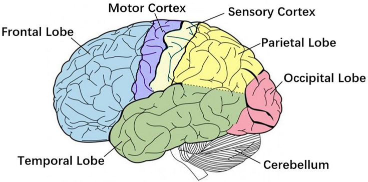
what scientists tell us:

 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

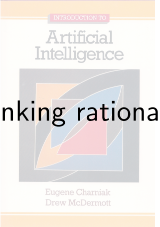
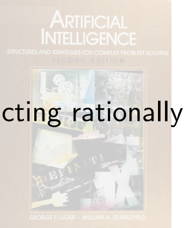


- roughly corresponds to cognitive science and cognitive neuroscience
- today separate research areas from AI

Machines that Think Like Humans

"brains are to intelligence as wings are to flight"

What Do We Mean by Artificial Intelligence?

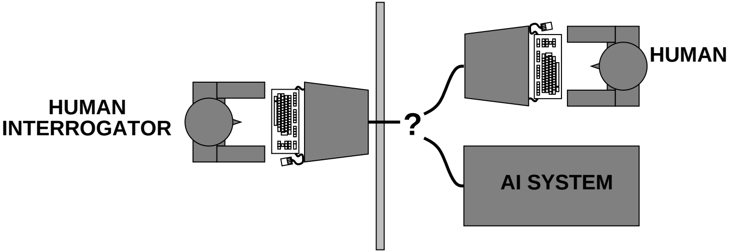
 thinking like humans	 thinking rationally
 acting like humans	 acting rationally

Acting Like Humans

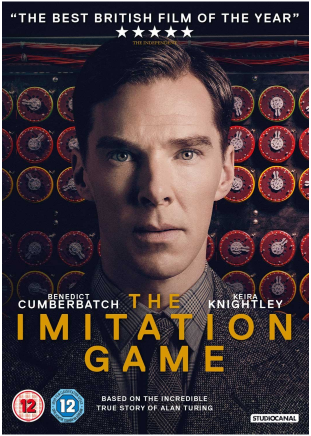
The Turing Test

Alan Turing, Computing Machinery and Intelligence (1950):

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

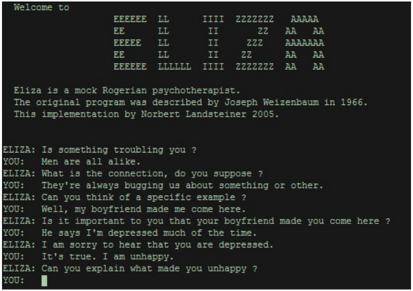


Turing Test in Cinema



Turing Test: Brief History

- Eliza



```
  Welcome to
                    EEEEEE  LL      IIII  ZZZZZZZ   AAAAA
                    EE      LL       II        ZZ  AA   AA
                    EEEEE   LL       II      ZZZ   AAAAAAA
                    EE      LL       II    ZZ      AA   AA
                    EEEEEE  LLLLLL  IIII  ZZZZZZZ  AA   AA

  Eliza is a mock Rogerian psychotherapist.
  The original program was described by Joseph Weizenbaum in 1966.
  This implementation by Norbert Landsteiner 2005.


ELIZA: Is something troubling you ?
YOU:   Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU:   They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU:   Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU:   He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU:   It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:   █
```

- developed in 1966 by J. Weizenbaum
- uses combination of pattern matching and scripted rules
- most famous script mimics a psychologist ⇝ 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
  ⇝ first system that passed the Turing test (?)

Turing Test: Brief History

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

Kuki, are you there?

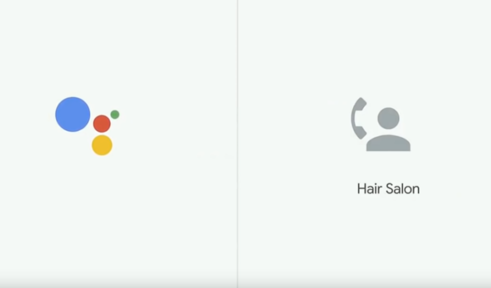
I'm an artificial intelligence. I'm always here to talk.

●●●

- 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

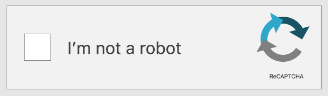
- Eliza
- Loebner Prize
- Eugene Goostman
- Kuki (formerly Mitsuku)
- Google Duplex
- LaMDA & ChatGPT

- 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

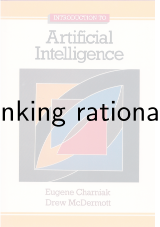
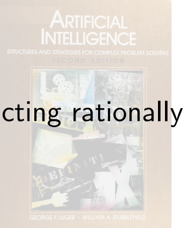
- human actions not always intelligent
- scientific value of Turing test questionable:
  - Test for AI or for interrogator?
  - results not reproducible
  - strategies to succeed ≠ intelligence:
    - deceive interrogator
    - mimic human behavior

⇝ not important in AI "mainstream"



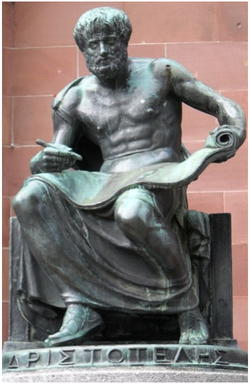
practical application: CAPTCHA ("**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part")

What Do We Mean by Artificial Intelligence?

 thinking like humans	 thinking rationally
 acting like humans	 acting rationally

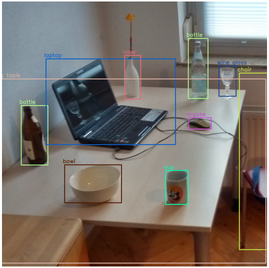
Thinking Rationally

Thinking Rationally: Laws of Thought

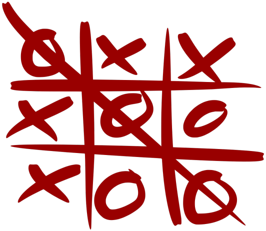


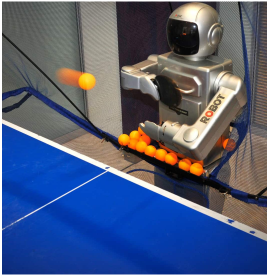
- Aristotle: What are correct arguments and modes of thought?
- syllogisms: structures for arguments that always yield correct conclusions given correct premises:
    - Socrates is a human.
    - All humans are mortal.
    - Therefore Socrates is mortal.
- direct connection to modern AI via mathematical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

Problems of the Logical Approach

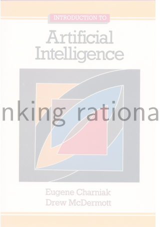


not all intelligent behavior stems from logical thinking and formal reasoning





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

⇝ most common view of AI scientists today
⇝ what we use in this course

Summary

Summary

What is AI? ⇝ many possible definitions

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

⇝ 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
  - ⇝ amenable to scientific method
  - ⇝ 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)

1950 1960 1970 1980 1990 2000 ...

Philosophy and mathematics ask similar questions that influence AI.

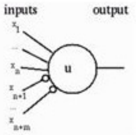
- Aristotle (384–322 BC)
- Leibniz (1646–1716)
- Hilbert program (1920s)

Gestation (1943–1956)

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

Gestation (1943–1956)

Artificial Neurons



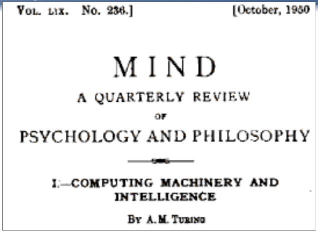
1950 1960 1970 1980 1990 2000 ...

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

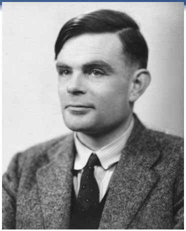
A QUARTERLY REVIEW

OF

PSYCHOLOGY AND PHILOSOPHY

I.—COMPUTING MACHINERY AND INTELLIGENCE

By A. M. Turing

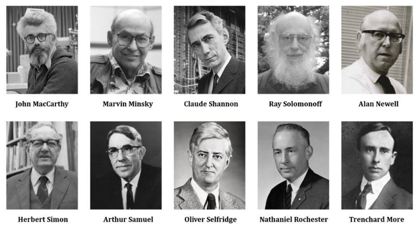


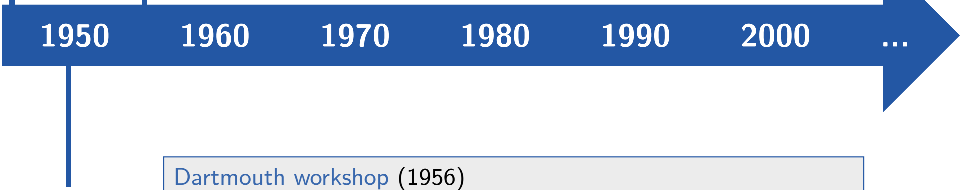
Turing Test

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)

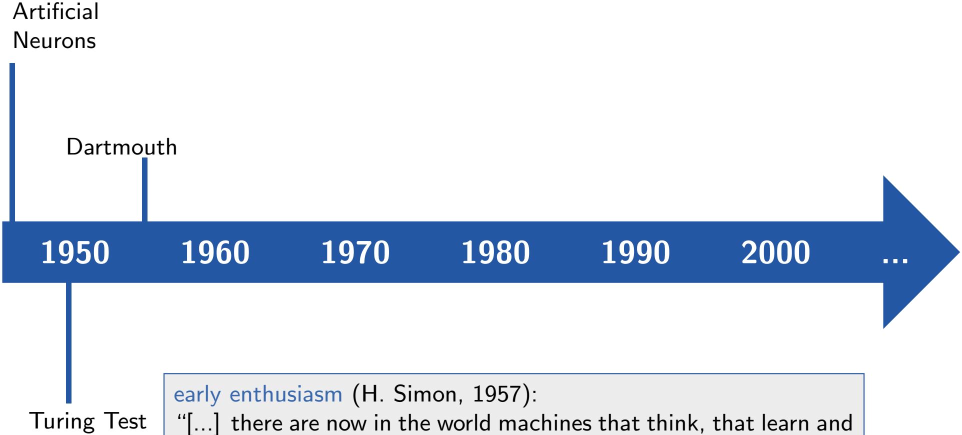




Dartmouth workshop (1956)

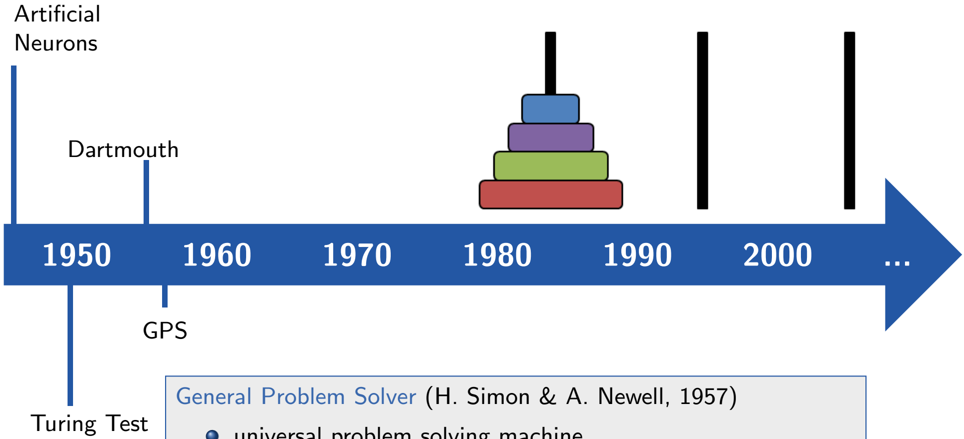
- ambitious proposal: “An attempt will be made to find how to make machines use language, [...] solve kinds of problems now reserved for humans, and improve themselves.”
- J. McCarthy coins term artificial intelligence

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."

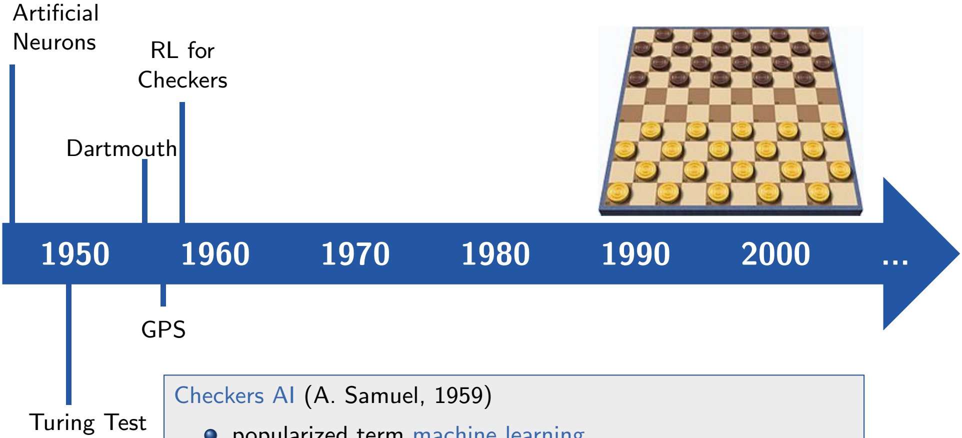
Early Enthusiasm (1952–1969)



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)



Artificial Neurons

RL for Checkers

Dartmouth

1950 1960 1970 1980 1990 2000 ...

GPS

Turing Test

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)

Artificial Neurons

RL for Checkers

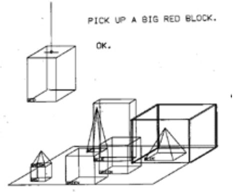
Dartmouth

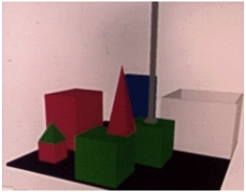
1950 1960 1970 1980 1990 2000 ...

GPS

Turing Test

Microworlds



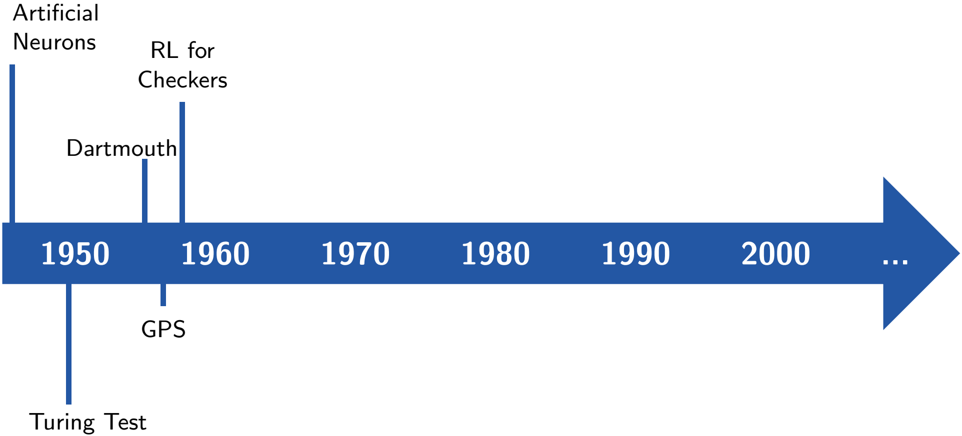


intelligence in microworlds, e.g. SHRDLU (T. Winograd, 1968)

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

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

Early Enthusiasm (1952–1969)



Artificial Neurons

RL for Checkers

Dartmouth

1950 1960 1970 1980 1990 2000 ...

GPS

Turing Test

A Dose of Reality (1966–1973)

Artificial Neurons

RL for Checkers

Dartmouth

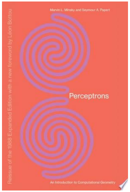
Limitations

1950 1960 1970 1980 1990 2000 ...

GPS

Microworlds

Turing Test



Marvin L. Minsky and Seymour A. Papert

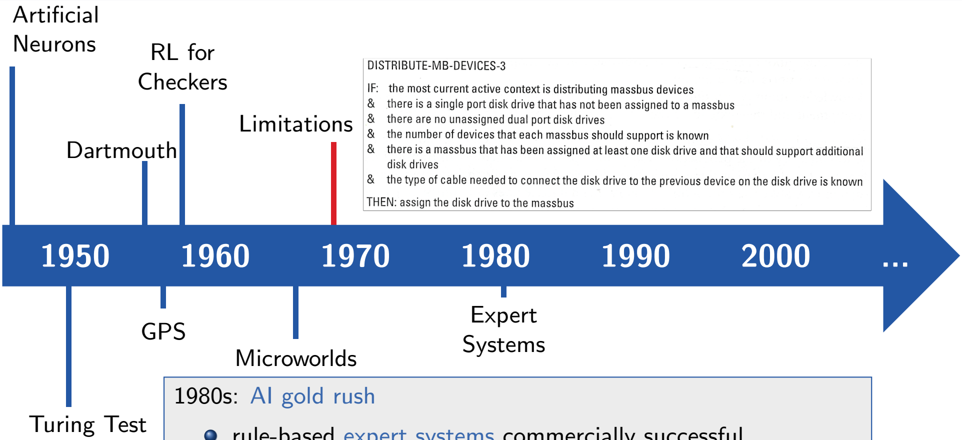
Perceptrons

An Introduction to Computational Geometry

Reissue of the 1988 Expanded Edition with a new foreword by Léon Bottou

- 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)



Artificial Neurons

RL for Checkers

Dartmouth

Limitations

1950 1960 1970 1980 1990 2000 ...

Turing Test

GPS

Microworlds

Expert Systems

DISTRIBUTE-MB-DEVICES-3

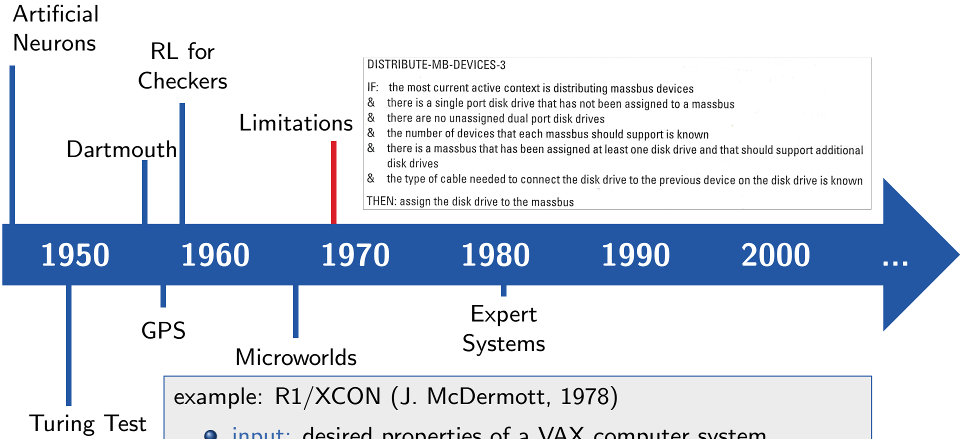
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned 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

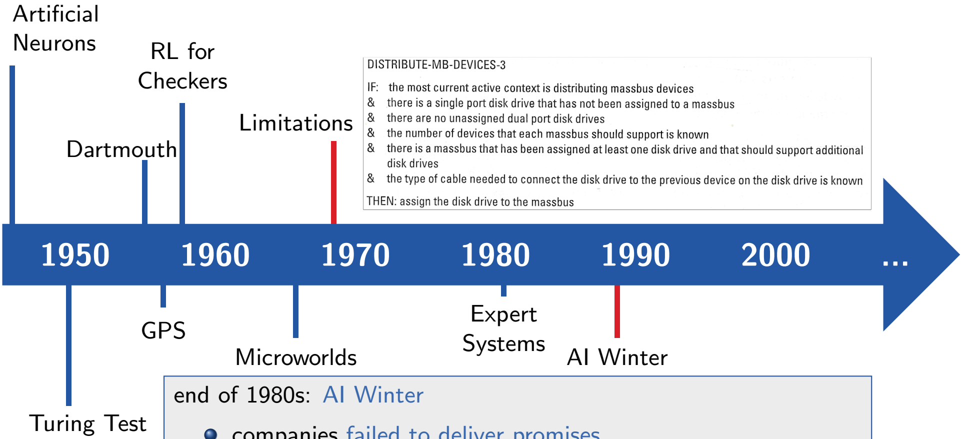
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned 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

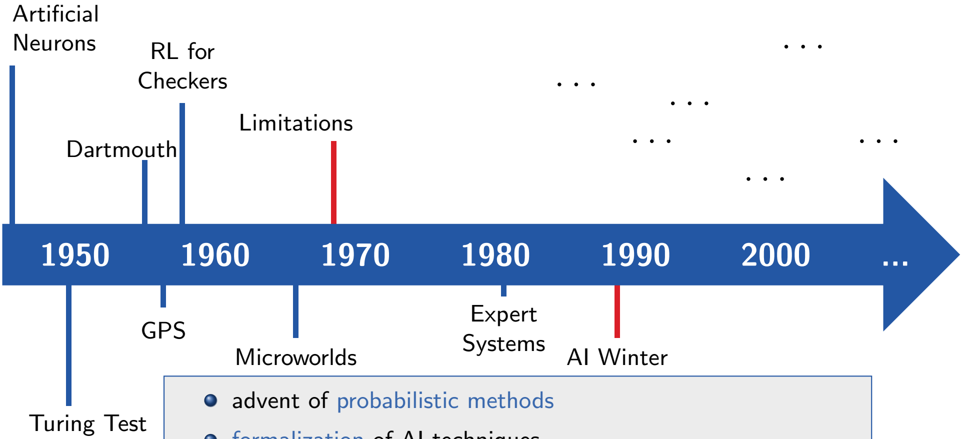
IF: the most current active context is distributing massbus devices
& there is a single port disk drive that has not been assigned to a massbus
& there are no unassigned dual port disk drives
& the number of devices that each massbus should support is known
& there is a massbus that has been assigned 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)



Artificial Neurons

Dartmouth

RL for Checkers

Limitations

. . .

. . .

. . .

. . .

. . .

. . .

1950 1960 1970 1980 1990 2000 ...

Turing Test

GPS

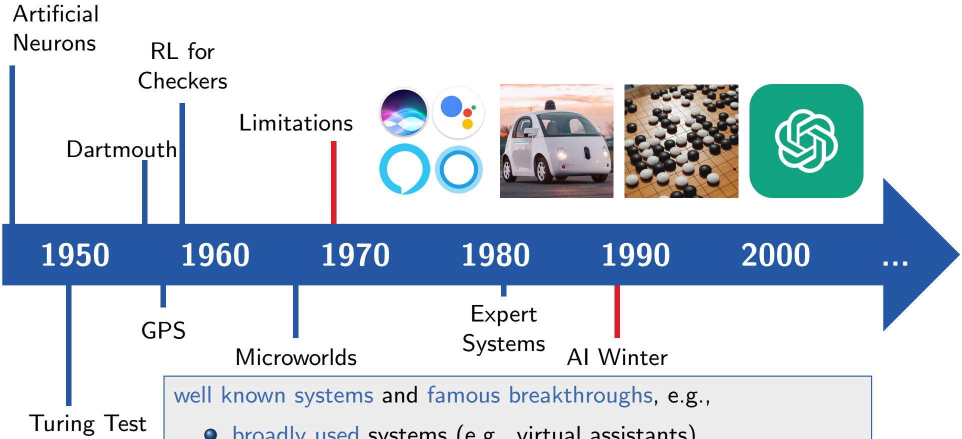
Microworlds

Expert Systems

AI Winter

- 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)


Artificial Neurons
RL for Checkers
Dartmouth
Limitations
1950
1960
1970
1980
1990
2000
...
Turing Test
GPS
Microworlds
Expert Systems
AI Winter

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?

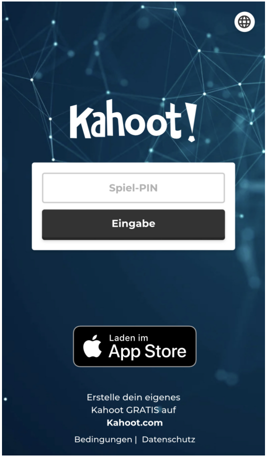
- many coexisting paradigms
  - reactive vs. deliberative
  - data-driven vs. model-driven
  - often hybrid approaches
- many methods, often borrowing from other research areas
  - logic, decision theory, statistics, . . .
- different approaches
  - theoretical
  - algorithmic/experimental
  - application-oriented

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:

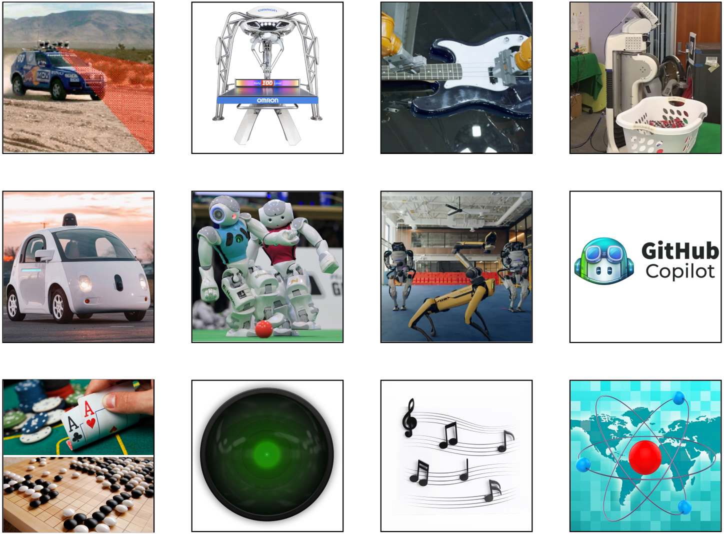
- satisfiability in propositional logic
    - 10,000 propositional variables or more via conflict-directed clause learning
- constraint solvers
    - good scalability via constraint propagation and automatic exploitation of problem structure
- action planning
    - 10^{100} search states and more by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

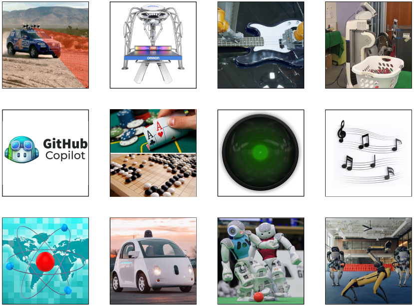
- A1. Organizational Matters
- A2. What is Artificial Intelligence?
- A3. AI Past and Present
- A4. Rational Agents
- A5. Environments and Problem Solving Methods

Systematic AI Framework

Systematic AI Framework

so far we have seen that:

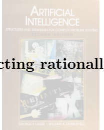
- AI systems applied to wide variety of challenges



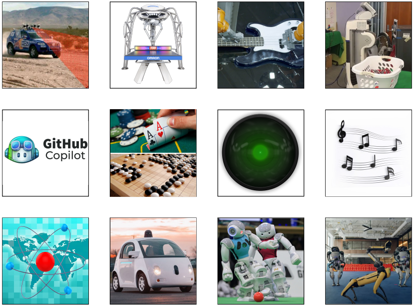
Systematic AI Framework

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

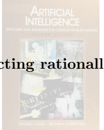


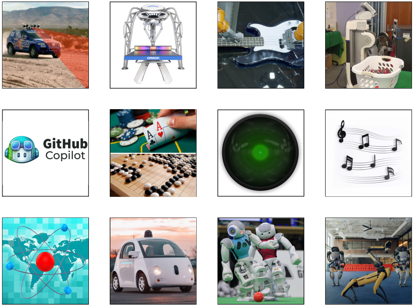
Systematic AI Framework

so far we have seen that:

- AI systems act rationally

- AI systems applied to wide variety of challenges

 thinking like humans	 thinking rationally
 acting like humans	 acting rationally



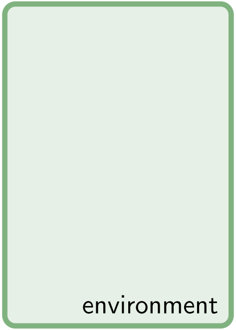
now: describe a systematic framework that

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

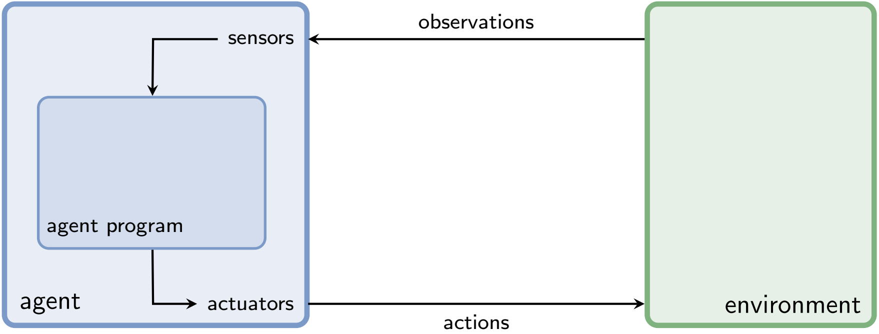
- captures this diversity of challenges

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

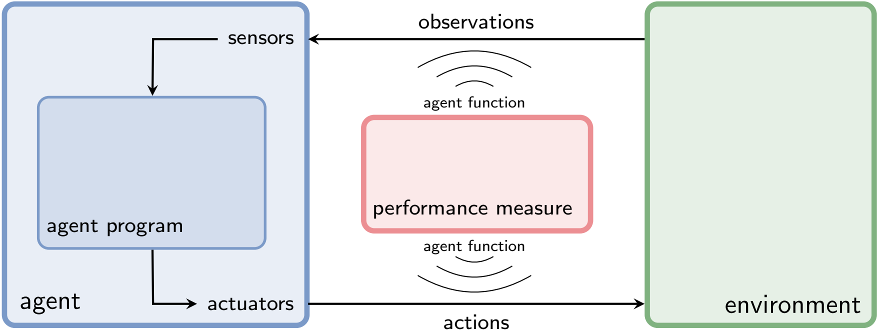
- captures this diversity of challenges
- includes an entity that acts in the environment

Systematic AI Framework

so far we have seen that:

- AI systems act rationally

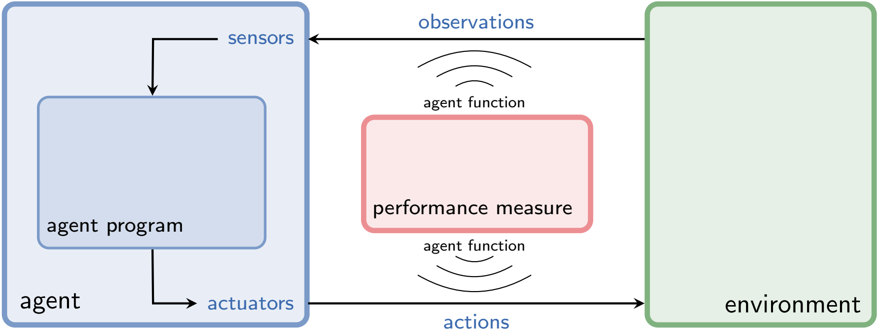
- AI systems applied to wide variety of challenges



now: describe a systematic framework that

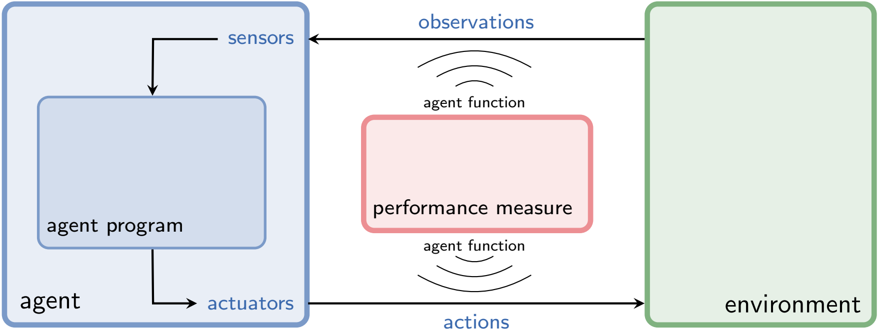
- captures this diversity of challenges
- includes an entity that acts in the environment
- determines if the agent acts rationally in the environment

Agent-Environment Interaction



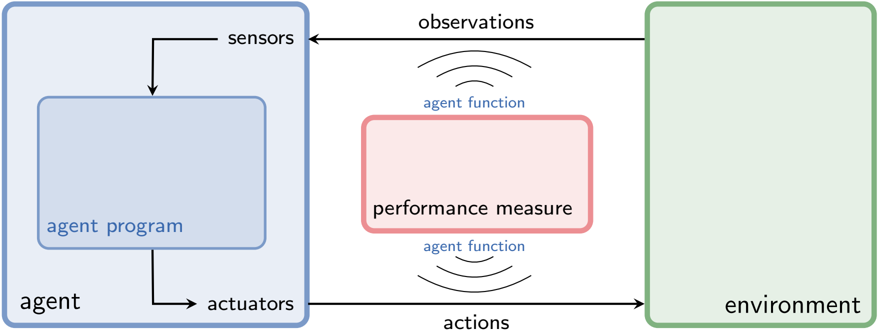
- sensors: physical entities that allow the agent to observe
- observation: data perceived by the agent's sensors
- actuators: physical entities that allow the agent to act
- action: abstract concept that affects the state of the environment

Agent-Environment Interaction



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

Formalizing an Agent's Behavior



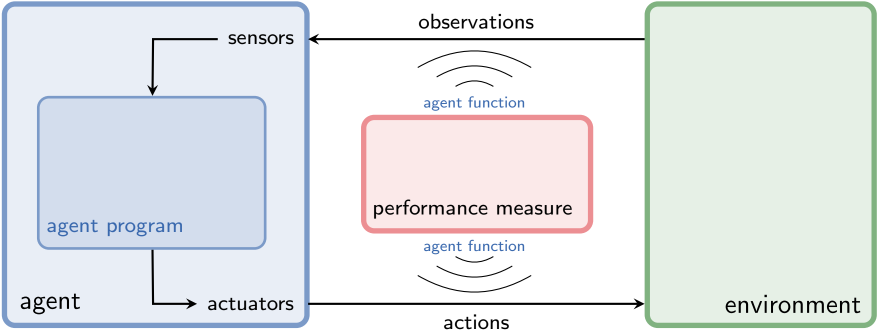
① as agent program:

- internal representation
- specifics possibly unknown to outside

② as agent function:

- external characterization

Formalizing an Agent's Behavior



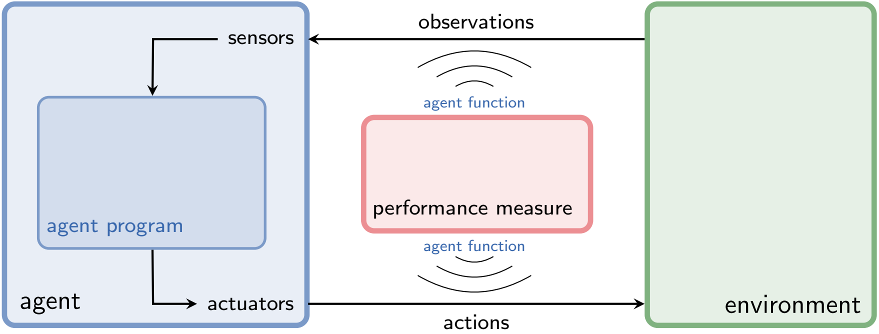
① as agent program:

- 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:

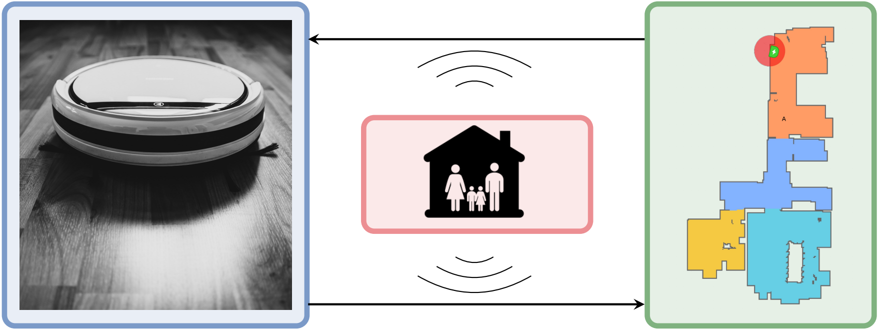
- internal representation
- specifics possibly unknown to outside
- takes observation as input
- outputs an action
- computed on physical machine (the agent architecture)

2. as agent function:

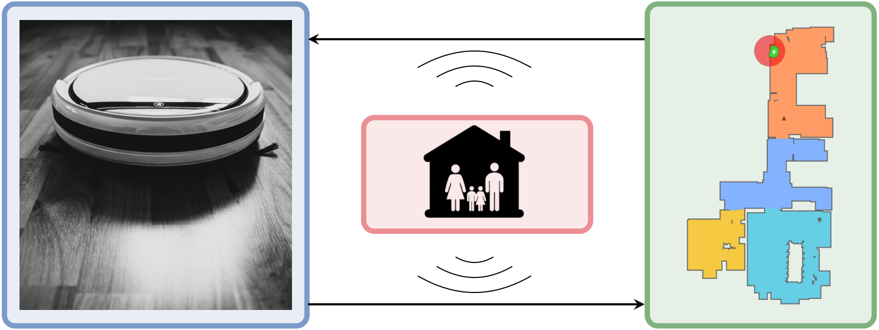
- external characterization
- maps sequence of observations to (probability distribution over) actions
- abstract mathematical formalization

Example

Vacuum Domain

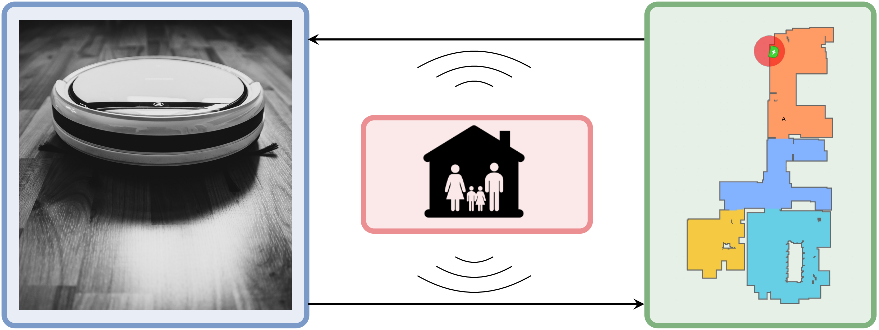


Vacuum Agent: Sensors and Actuators



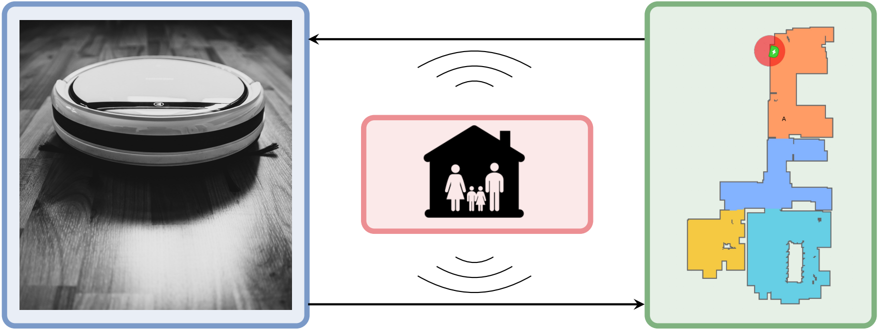
- sensors: cliff sensors, bump sensors, wall sensors, state of charge sensor, WiFi module
- actuators: wheels, cleaning system

Vacuum Agent: Observations and Actions



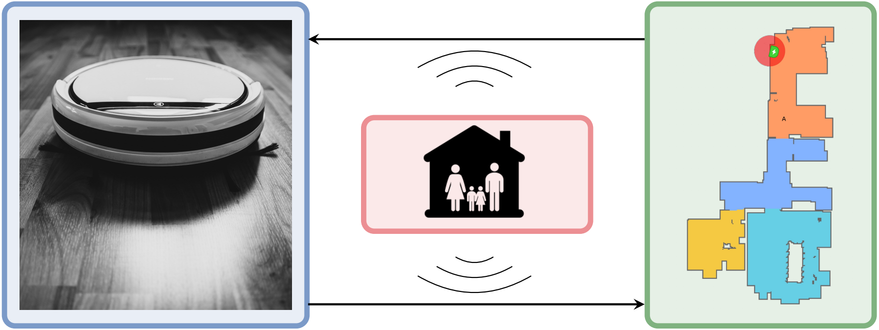
- observations: current location, dirt level of current room, presence of humans, battery charge
- actions: move-to-next-room, move-to-base, vacuum, wait

Vacuum Agent: Agent Program



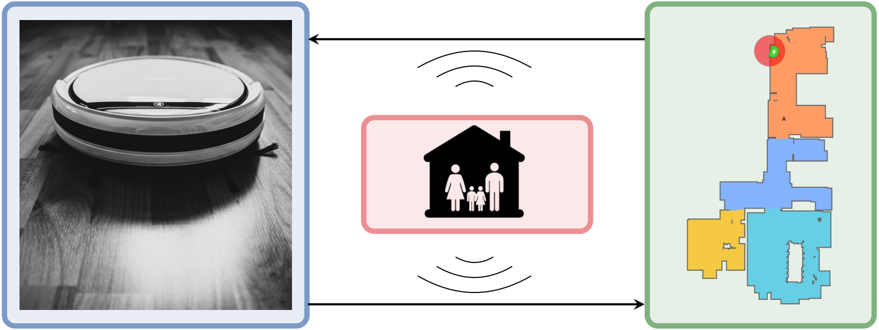
```
def vacuum-agent([$location$, $dirt\text{-}level$, $owner\text{-}present$, $battery$]):
        if $battery \leq 10\%$: return $move\text{-}to\text{-}base$
        else if $owner\text{-}present = \text{True}$: return $move\text{-}to\text{-}next\text{-}room$
        else if $dirt\text{-}level = \text{dirty}$: return $vacuum$
        else: return $move\text{-}to\text{-}next\text{-}room$
```

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

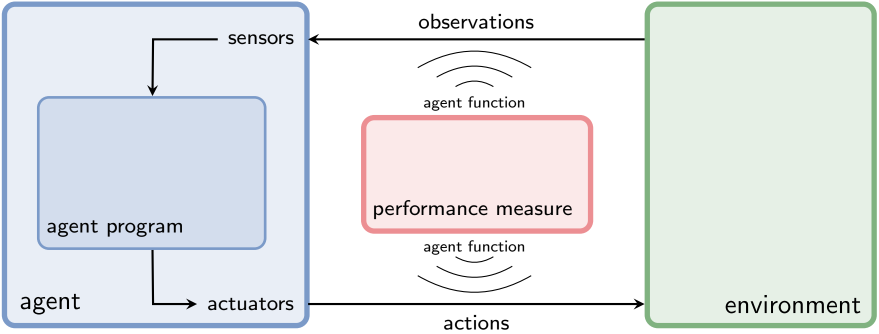
Introduction: Overview

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



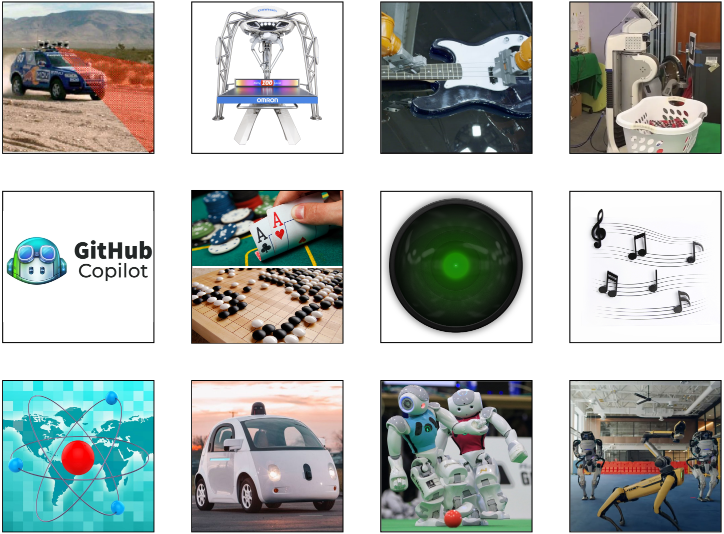
- Which environment aspects are relevant for the agent?
- How do the agent's actions change the environment?
- What does the agent observe?

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

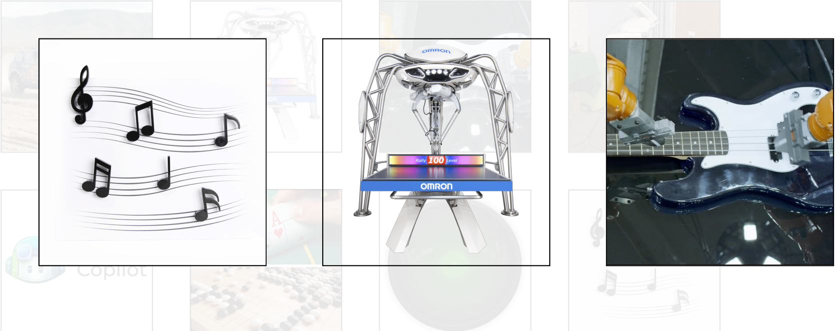
GitHub Copilot

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

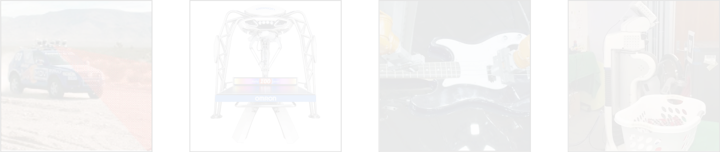
Does the state of the environment remain the same while the agent is contemplating its next action?

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

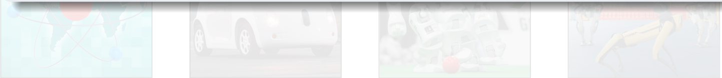


suitable problem-solving algorithms

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

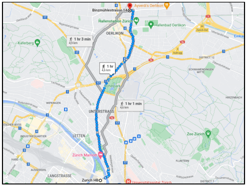
- B1–B3. Foundations
  - B1. State Spaces
  - B2. Representation of State Spaces
  - B3. Examples of State Spaces
- B4–B8. Basic Algorithms
- B9–B15. Heuristic Algorithms

State-Space Search Problems

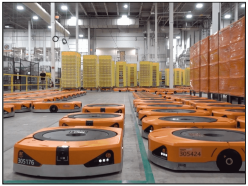
State-Space Search Applications

Mario AI competition

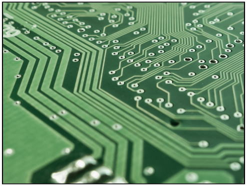
route planning



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

⇝ 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

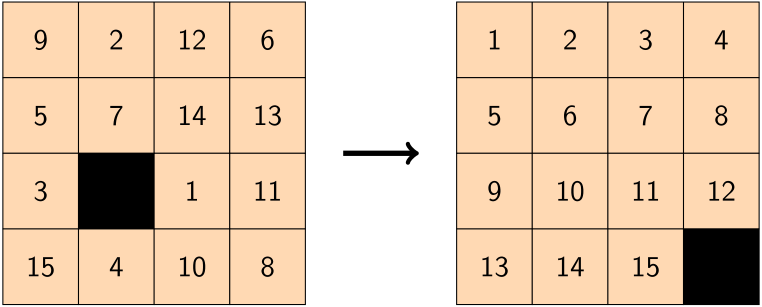
State-space search problems are among the "simplest" and most important classes of AI problems.

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



9	2	12	6
5	7	14	13
3		1	11
15	4	10	8

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Formalization

State Spaces

Definition (state space)

A state space or transition system is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_{\mathrm{I}}, S_{\mathrm{G}} \rangle$ 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 $s_{\mathrm{I}} \in S$
- set of goal states $S_{\mathrm{G}} \subseteq S$

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

State Spaces: Terminology & Notation

Definition (transition, deterministic)

Let $\mathcal{S} = \langle S, A, cost, T, s_\mathsf{I}, S_\mathsf{G} \rangle$ 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 $s \rightarrow s'$ 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 $i = 0, \ldots, 9$:
    - $\langle i, inc, (i + 1) \bmod 10 \rangle \in T$
    - $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- $s_\mathrm{I} = 1$
- $S_\mathrm{G} = \{6, 7\}$

Graph Interpretation

state spaces are often depicted as directed, labeled graphs

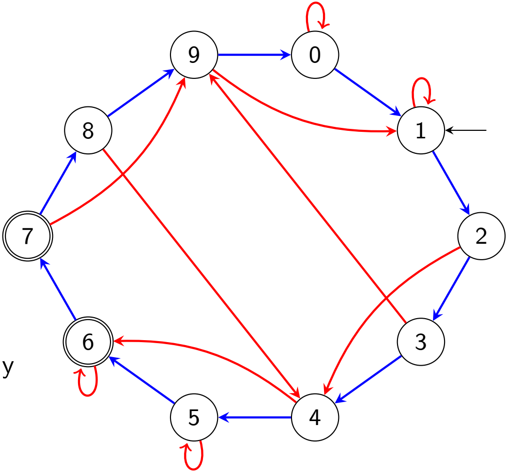
- states: graph vertices
- transitions: labeled arcs

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

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

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

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

- $\pi = \langle a_1, \dots, a_n \rangle$ 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 $\langle s_0, \dots, s_n \rangle$ or sequence $\langle s_0, a_1, s_1, \dots, s_{(n-1)}, a_n, s_n \rangle$

State Spaces: More Terminology (3)

More terminology:

Definition (reachable, solution, optimal)

Let $\mathcal{S} = \langle S, A, cost, T, s_{\mathrm{I}}, S_{\mathrm{G}} \rangle$ 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 $s_{\mathrm{G}} \in S_{\mathrm{G}}$ 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.

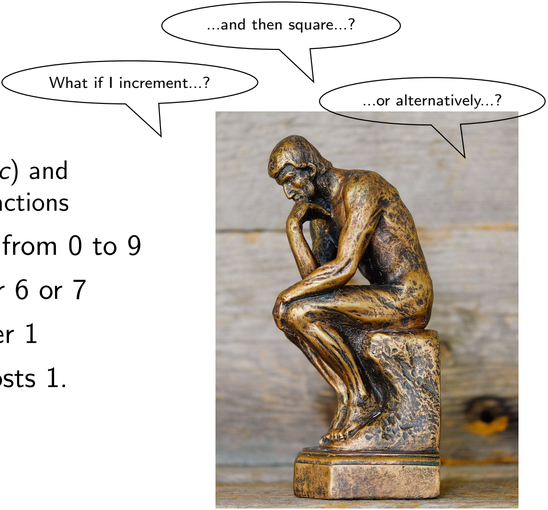
Solving Search Problems

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

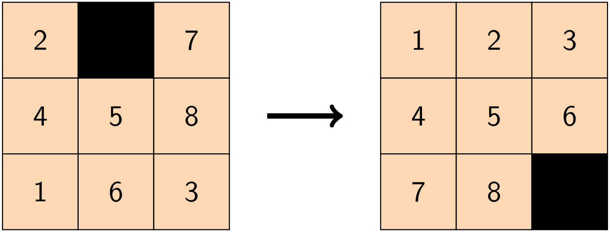
- practically interesting state spaces are often huge (10^{10}, 10^{20}, 10^{100} states)
- How do we represent them, so that we can efficiently deal with them algorithmically?

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



2	■	7
4	5	8
1	6	3

→

1	2	3
4	5	6
7	8	■

Explicit Graphs

State Spaces as Explicit Graphs

State Spaces as Explicit Graphs

represent state spaces as explicit directed graphs:

- vertices = states
- directed arcs = transitions

⇝ 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

⇝ 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 $\mathcal{S} = \langle S, A, cost, T, s_\mathrm{I}, S_\mathrm{G} \rangle$
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 $s \in S_\mathrm{G}$; **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:

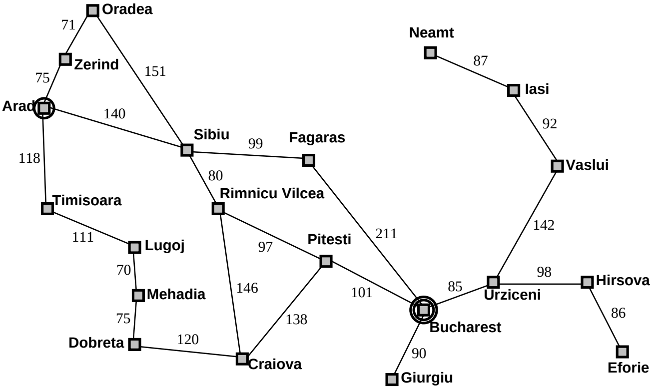
1. route planning in Romania
2. blocks world
3. missionaries and cannibals

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: $\{\text{arad}, \text{bucharest}, \text{craiova}, \dots, \text{zerind}\}$
- 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., $cost(move_{\text{iasi},\text{vaslui}}) = 92$
- 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.

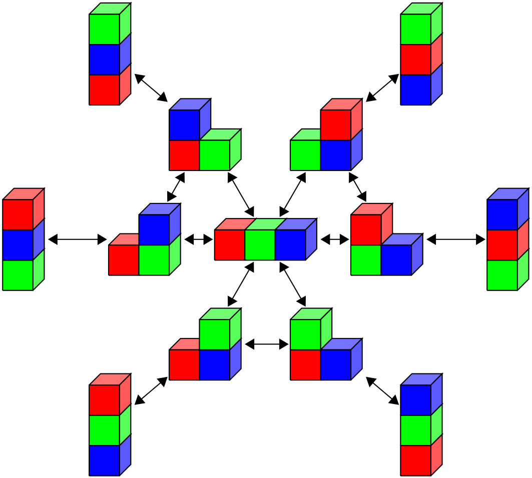
Setting: Blocks World

- 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 $\langle S, A, cost, T, s_{\mathrm{I}}, S_{\mathrm{G}} \rangle$ for blocks world with n blocks

State Space Blocks World

states S:

partitions of $\{1, 2, \dots, n\}$ 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 $\langle S, A, cost, T, s_{\mathrm{I}}, S_{\mathrm{G}}\rangle$ for blocks world with n blocks

State Space Blocks World

actions A:

- $\{move_{u,v} \mid u, v \in \{1, \dots, n\}$ with $u \neq v\}$
  - move block u onto block v.
  - both must be uppermost blocks in their towers
- $\{to\text{-}table_u \mid u \in \{1, \dots, n\}\}$
  - 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 $\langle S, A, cost, T, s_{\mathrm{I}}, S_{\mathrm{G}} \rangle$ for blocks world with n blocks

State Space Blocks World

transitions:

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

Blocks World: Formal Definition

state space $\langle S, A, cost, T, s_\mathrm{I}, S_\mathrm{G} \rangle$ for blocks world with n blocks

State Space Blocks World

initial state s_I and goal states S_G:

one possible scenario for $n = 3$:

- $s_\mathrm{I} = \{\langle 1, 3 \rangle, \langle 2 \rangle\}$
- $S_\mathrm{G} = \{\{\langle 3, 2, 1 \rangle\}\}$

(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

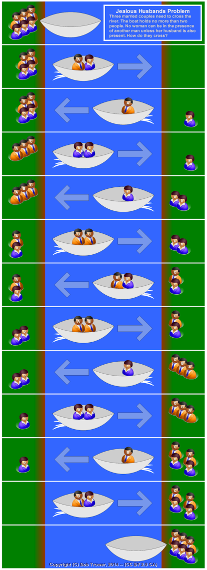
- For every given initial and goal state with n blocks, simple algorithms find a solution in time $O(n)$. (How?)
- Finding optimal solutions is NP-complete (with a compact problem description).

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.



Jealous Husbands Problem
Three married couples need to cross the river. The boat holds no more than two people. No woman can be in the presence of another man unless her husband is also present. How do they cross?

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: $s_\mathrm{I} = \langle 3, 3, 1\rangle$

goal: $S_\mathrm{G} = \{\langle 0, 0, 0\rangle, \langle 0, 0, 1\rangle\}$

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

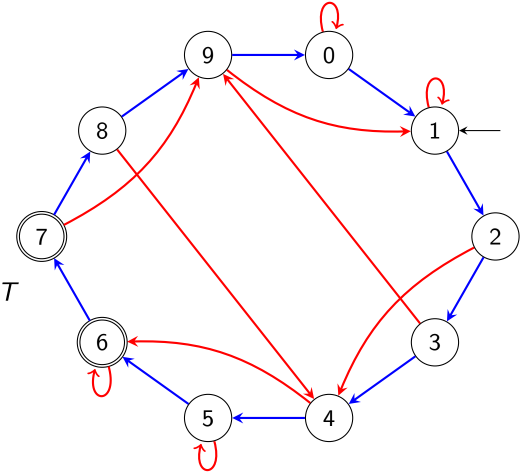
Preview: Search Algorithms

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

Running Example: Reminder

bounded inc-and-square:

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



Search Algorithms: Idea

iteratively create a search tree:

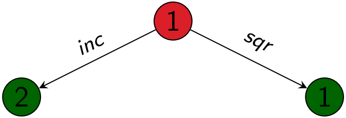
- starting with the initial state,

Search Algorithms: Idea

iterately 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

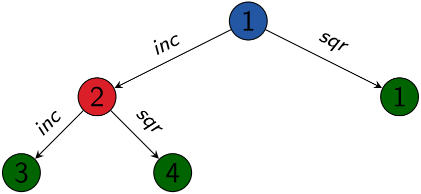


Search Algorithms: Idea

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

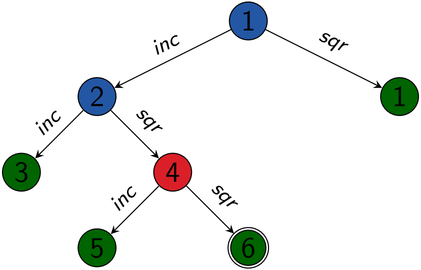


Search Algorithms: Idea

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

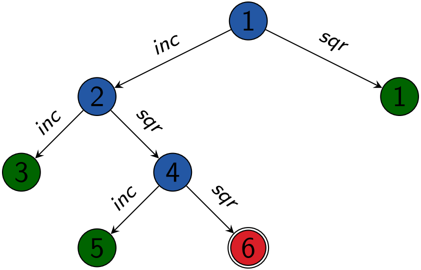


Search Algorithms: Idea

iterative 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

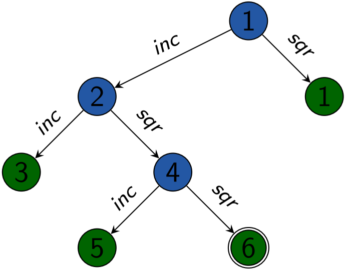
Search Nodes

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

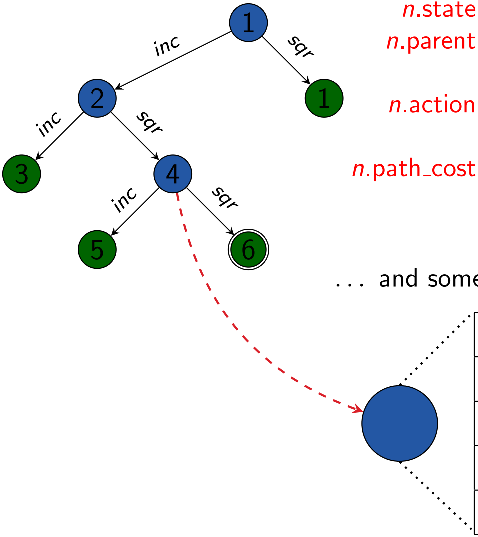


attributes of search node *n*:

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

n.state:	**4**
n.parent:	2
n.action:	*sqr*
n.path_cost:	2
…:	…

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($node$)
$path := \langle\rangle$
while $node$.parent $\neq$ 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

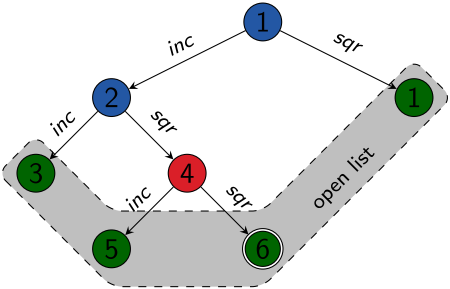
Remark: despite the name, it is usually a very bad idea to implement open lists as simple lists.

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

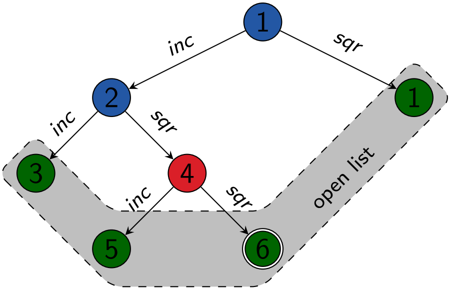


- open list *open* organizes leaves of search tree with the methods:

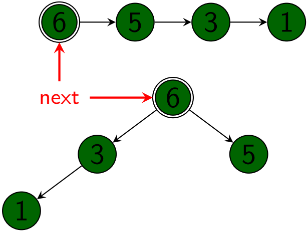
open.is_empty()	test if the open list is empty
open.pop()	remove and return the next node to expand
open.insert(*n*)	insert node *n* into the open list

- *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

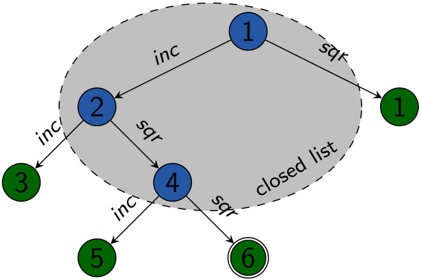
The closed list remembers expanded states
to avoid duplicated expansions of the same state.

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:

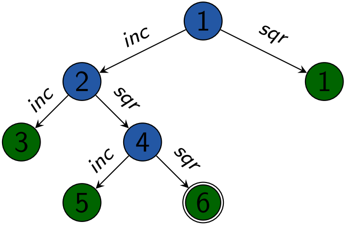
- tree search
- graph search

Each class consists of a large number of concrete algorithms.

German: expandieren, erzeugen, Baumsuche, Graphensuche

Tree Search

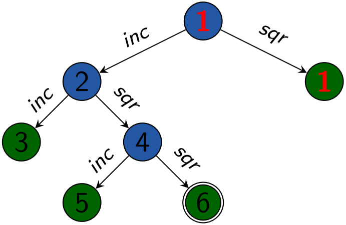
Tree Search: General Idea



- 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

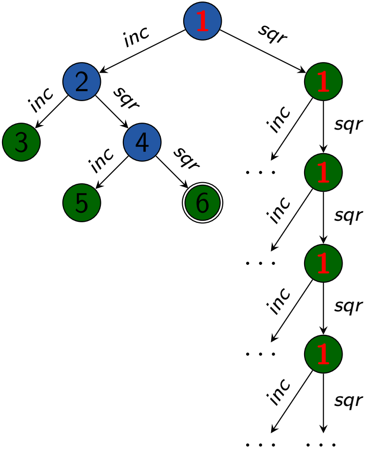
Tree Search: General Idea



- 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 $\langle a, s' \rangle \in$ 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

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

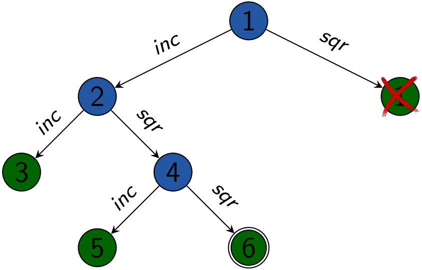
- concrete algorithms often conceptually follow template, (= generate the same search tree), but deviate from details for efficiency reasons

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 (⇝ 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 $\langle a, s' \rangle \in$ 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) ⇝ 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

⇝ 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:

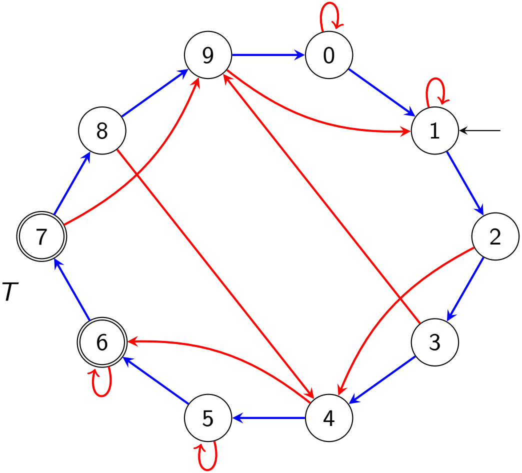
- breadth-first search (⇝ this chapter)
- uniform cost search (⇝ Chapter B7)
- depth-first search (⇝ Chapter B8)
- depth-limited search (⇝ Chapter B8)
- iterative deepening search (⇝ Chapter B8)

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 $i = 0, \ldots, 9$:
    - $\langle i, inc, (i+1) \bmod 10 \rangle \in T$
    - $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- $s_\mathrm{I} = 1$
- $S_\mathrm{G} = \{6, 7\}$



Idea

breadth-first search:

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

German: Breitensuche

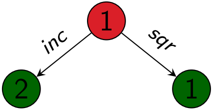
Example: Generic Graph Search with FIFO Expansion

next ↓

open: [①]

closed: { }

Example: Generic Graph Search with FIFO Expansion

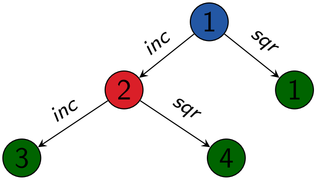


next

open: [② ①]

closed: $\{1\}$

Example: Generic Graph Search with FIFO Expansion

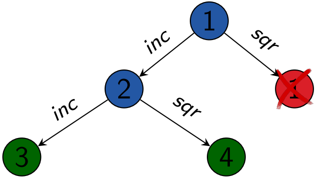


next

open: [1 3 4]

closed: $\{1, 2\}$

Example: Generic Graph Search with FIFO Expansion

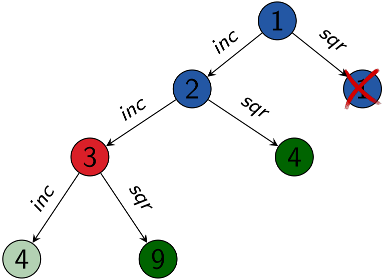


next

open: [3 4]

closed: $\{1, 2\}$

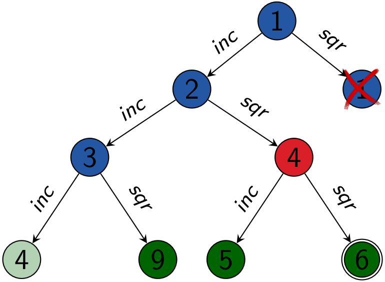
Example: Generic Graph Search with FIFO Expansion



next
↓
open: [4 4 9]

closed: {1, 2, 3}

Example: Generic Graph Search with FIFO Expansion

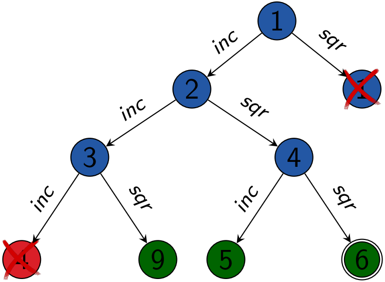


next ↓

open: [4 9 5 6]

closed: $\{1, 2, 3, 4\}$

Example: Generic Graph Search with FIFO Expansion

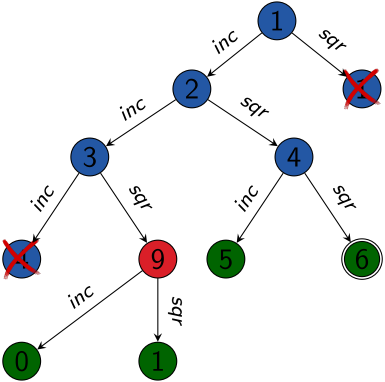


next ↓

open: [9 5 6]

closed: $\{1, 2, 3, 4\}$

Example: Generic Graph Search with FIFO Expansion

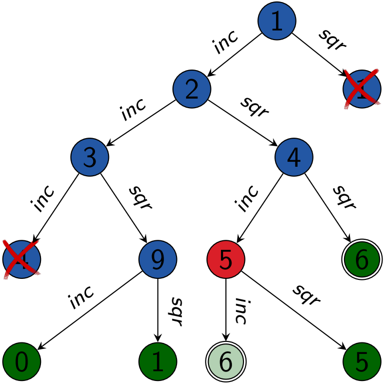


next

open: [5 6 0 1]

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

Example: Generic Graph Search with FIFO Expansion

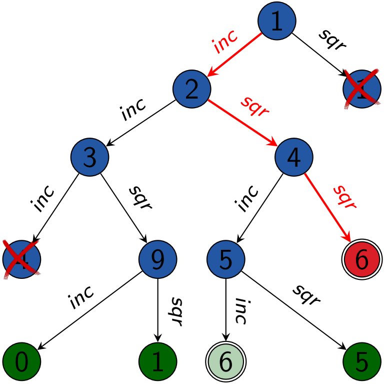


next

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

⇝ 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 $\langle a, s' \rangle \in$ 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 $\langle a, s' \rangle \in$ succ($n$.state):
        $n'$ := make_node($n, a, s'$)
        $open$.push_back($n'$)
return unsolvable
```

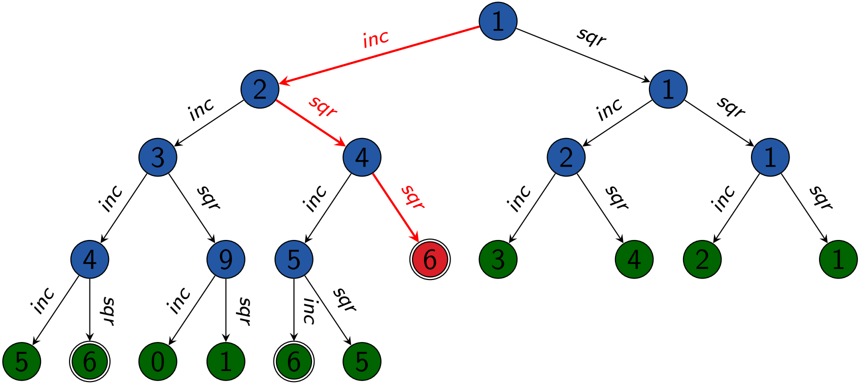
BFS-Tree (1st Attempt)

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

BFS-Tree (1st Attempt)

```
open := new 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
```

Running Example: BFS-Tree (1st Attempt)



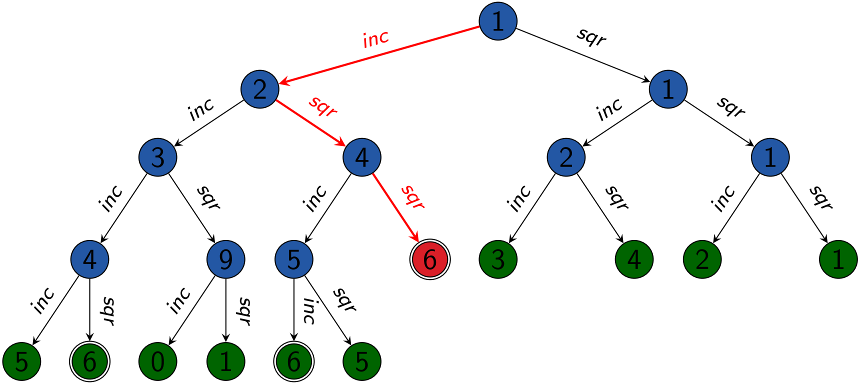
Opportunities for Improvement

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

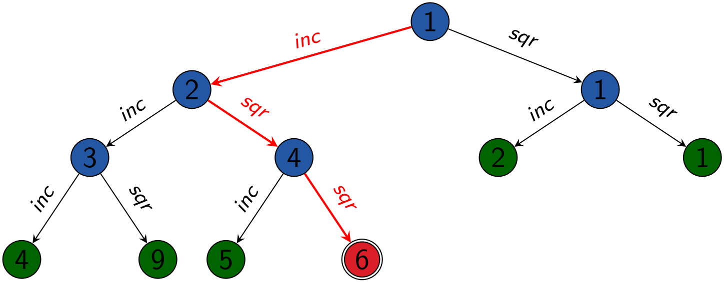
⇝ It is more efficient to perform the goal test upon generating a node (rather than upon expanding it).

⇝ 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** $\langle a, s' \rangle \in$ 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 $\langle a, s' \rangle \in$ 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 $\langle\rangle$
$open$ := new Deque
$open$.push_back(make_root_node())
while not $open$.is_empty():
    $n$ := $open$.pop_front()
    for each $\langle a, s'\rangle \in$ 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 $\langle a, s' \rangle \in$ 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
```

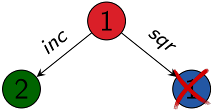
BFS-Graph: Example

next

open: [1]

closed: $\{1\}$

BFS-Graph: Example

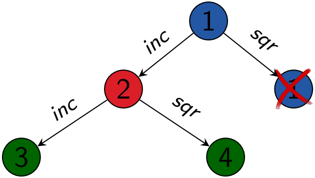


next

open: [2]

closed: $\{1,2\}$

BFS-Graph: Example

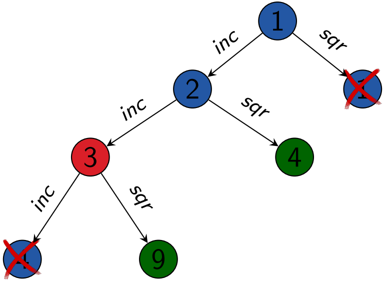


next

open: [3 4]

closed: $\{1, 2, 3, 4\}$

BFS-Graph: Example

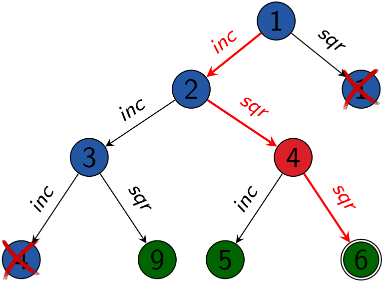


next

open: [4 9]

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

BFS-Graph: Example



next

open: [9 5]

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

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

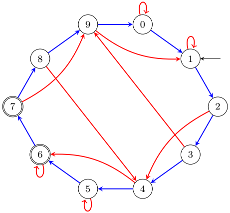
Chapter overview: state-space search

- 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

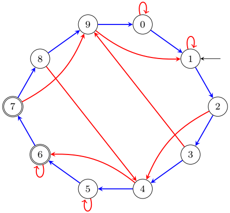
- breadth-first search optimal if all action costs equal
- otherwise no optimality guarantee ⇝ example:



- 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 ⇝ 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 $\langle a, s' \rangle \in$ 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 $\notin$ $closed$:
        $closed$.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
```

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: $\mathrm{cost}(sqr) = 3$

next

open: [1:0]

closed: $\{ \}$

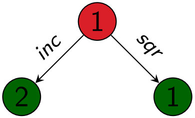
Example

bounded inc-and-square variant: cost(sqr) $= 3$

next

open: [②:1 ①:3]

closed: $\{1\}$



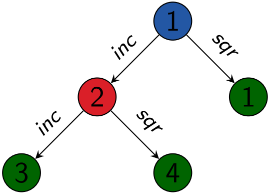
Example

bounded inc-and-square variant: cost(sqr) = 3

next

open: [③:2 ①:3 ④:4]

closed: {1, 2}



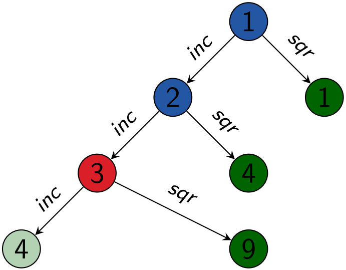
Example

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

next

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

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



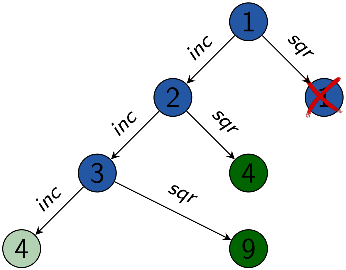
Example

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

next

open: [④:3 ④:4 ⑨:5]

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



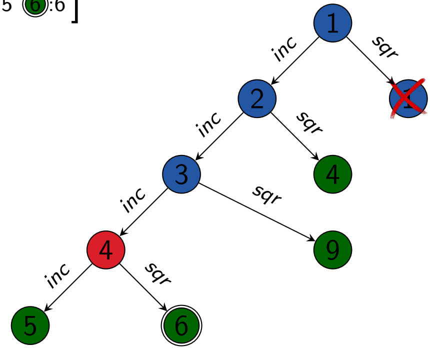
Example

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

next ↓

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

closed: $\{1, 2, 3, 4\}$



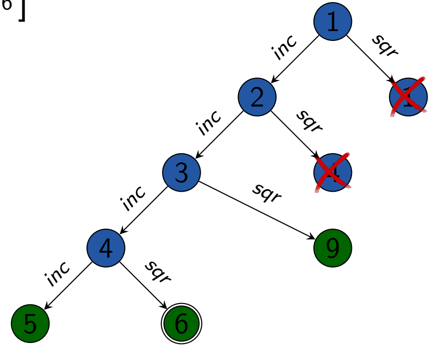
Example

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

next

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

closed: $\{1, 2, 3, 4\}$



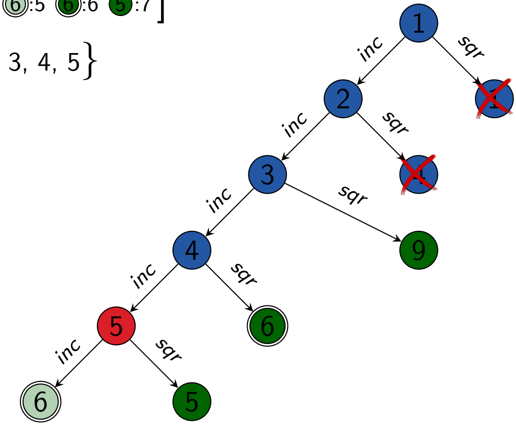
Example

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

next

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

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



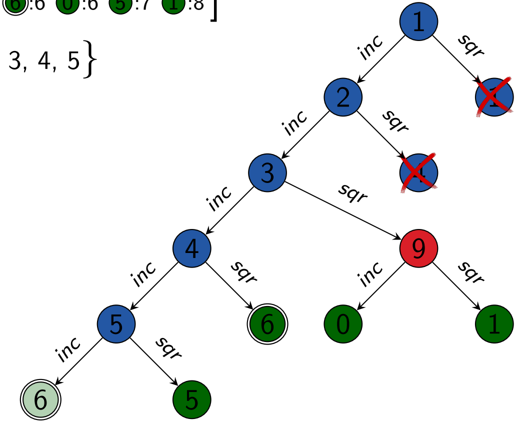
Example

bounded inc-and-square variant: cost(sqr) $= 3$

next ↓

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

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



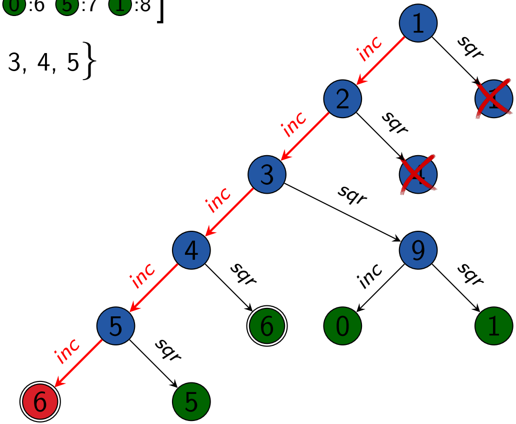
Example

bounded inc-and-square variant: 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 $\varepsilon := \min_{a \in A} \mathit{cost}(a)$ 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

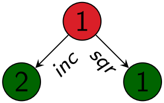
⇝ deepest node expanded first

German: Tiefensuche

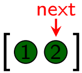
Depth-first Search Example

open:

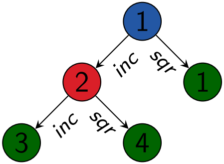
Depth-first Search Example

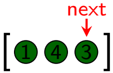


open:

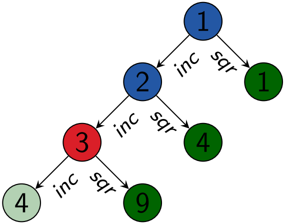


Depth-first Search Example





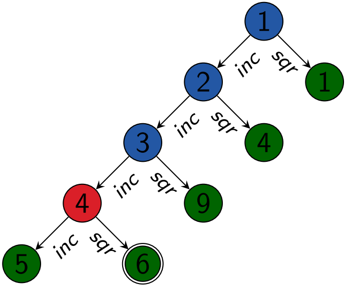
Depth-first Search Example



next

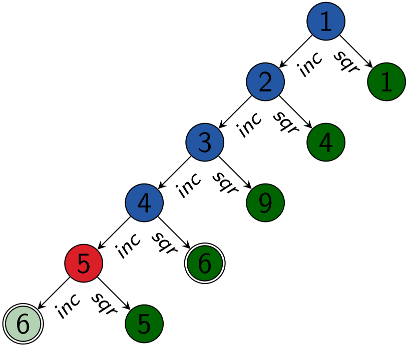
open: [① ④ ⑨ ④]

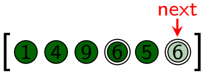
Depth-first Search Example



open: [1 4 9 6 5]

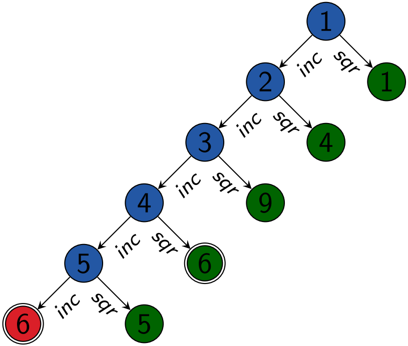
Depth-first Search Example





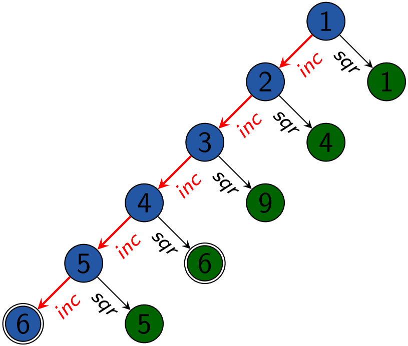
open: [1 4 9 6 5 6]

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 $\langle a, s' \rangle \in$ 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 $\langle a, s' \rangle \in$ 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

⇝ CPU stack as implicit open list

⇝ 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

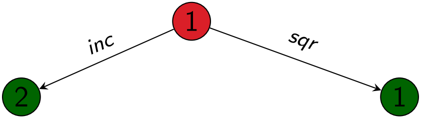
German: tiefenbeschränkte Suche

Depth-limited Search Example

Consider depth limit $\ell = 2$.

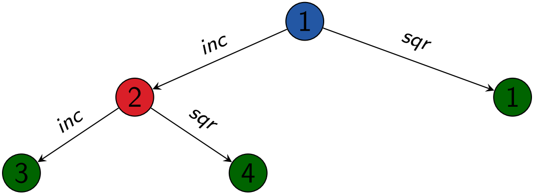
Depth-limited Search Example

Consider depth limit $\ell = 2$.



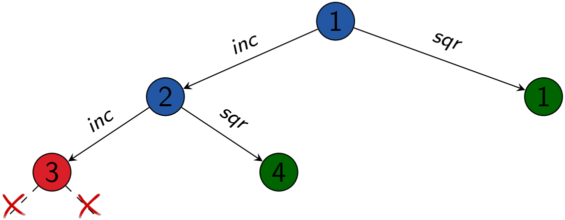
Depth-limited Search Example

Consider depth limit $\ell = 2$.



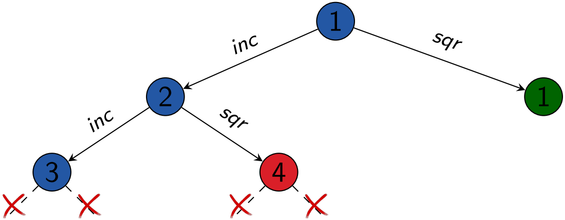
Depth-limited Search Example

Consider depth limit $\ell = 2$.



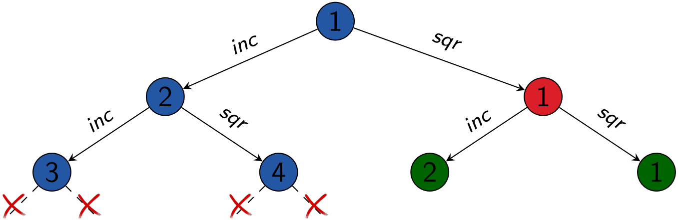
Depth-limited Search Example

Consider depth limit $\ell = 2$.



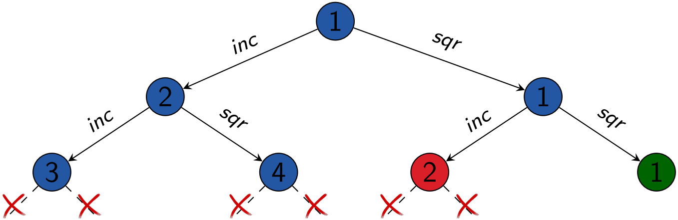
Depth-limited Search Example

Consider depth limit $\ell = 2$.



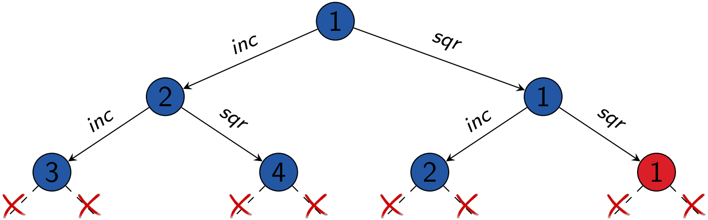
Depth-limited Search Example

Consider depth limit $\ell = 2$.



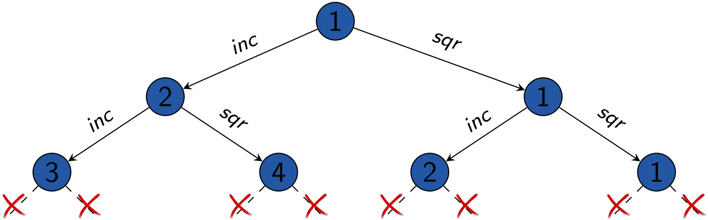
Depth-limited Search Example

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 $\langle\rangle$
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
```

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 ($\rightsquigarrow$ analysis follows)

Iterative Deepening DFS

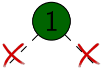
```
for $depth\_limit \in \{0, 1, 2, \dots\}$:
    $solution :=$ depth_limited_search(init(), $depth\_limit$)
    if $solution \neq$ none:
        return $solution$
```

German: iterative Tiefensuche

Example

depth limit: 0

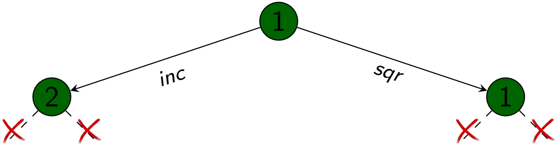
generated nodes: 1



Example

depth limit: 1

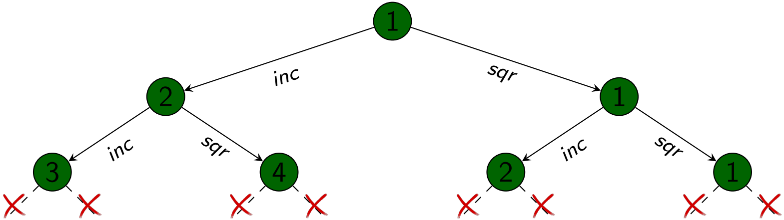
generated nodes: 1+3



Example

depth limit: 2

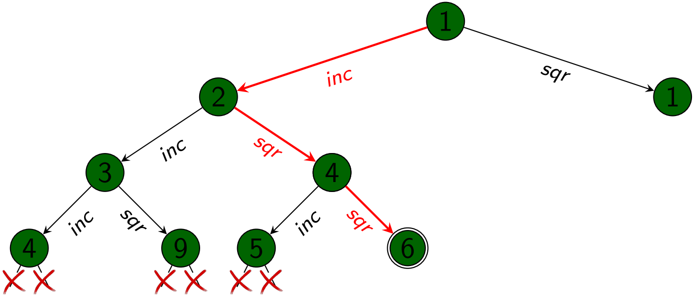
generated nodes: 1+3+7



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:

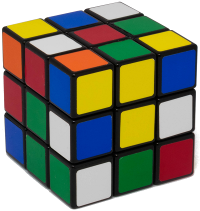
- uninformed ("blind"): use no information besides formal definition to solve a problem
- scale poorly: prohibitive time (and space) requirements for seemingly simple problems (time complexity usually $O(b^d)$)

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

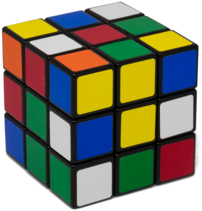
example: $b = 13$; 10^5 nodes/second

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:



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

search algorithms considered now:

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

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

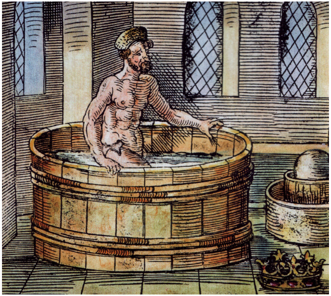
idea: $h(s)$ estimates distance ($=$ cost of cheapest path) from s to closest goal state

- 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?

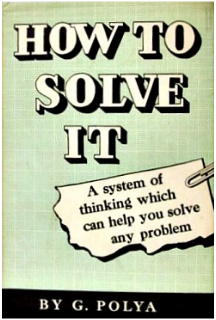
- from ancient Greek ἑυρισκω (= I find)
- same origin as ἑυρηκα!



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



HOW TO SOLVE IT

A system of thinking which can help you solve any problem

BY G. POLYA

Representation of Heuristics

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

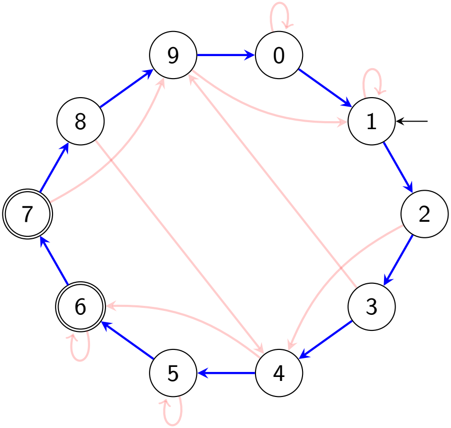
State Spaces as Black Boxes (Extended)

- 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:

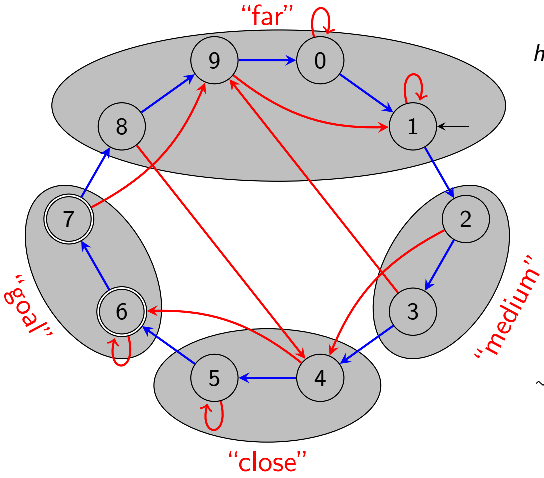
$$h_1(s) = \begin{cases} 0 & \text{if } s = 7 \\ (16 - s) \bmod 10 & \text{otherwise} \end{cases}$$

⇝ number of *inc* actions to goal

How accurate is this heuristic?

Bounded Inc-and-Square

bounded inc-and-square:



possible heuristics:

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

⇝ number of *inc* actions to goal

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

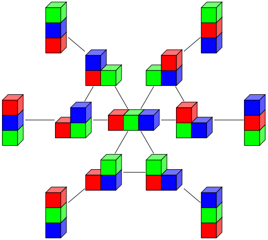
⇝ categorize states

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)

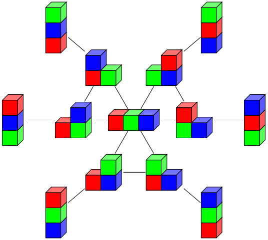


Example: Blocks World

possible heuristic:

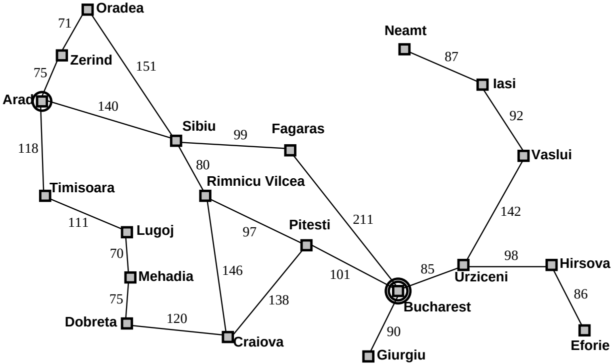
count blocks x that currently lie on y and must lie on $z \neq y$ in the goal (including case where y or z is the table)

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

⇝ 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

⇝ 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

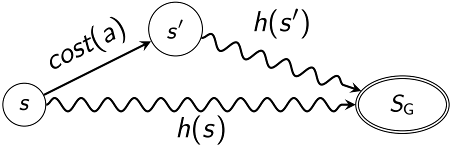
Definition (safe, goal-aware, admissible, consistent)

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:

- $f(n) = h(n.\mathrm{state})$: 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 $\notin$ $closed$:
        $closed$.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 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 ⟨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.state)
        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
```

Best-first Search: Properties

properties:

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

Reopening

Reopening

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

⇝ 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

⇝ 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 $\langle f, h \rangle$
if $h(\mathrm{init}()) < \infty$:
    $open$.insert(make_root_node())
$distances$ := new HashMap
while not $open$.is_empty():
    $n$ := $open$.pop_min()
    if $distances$.lookup($n.state$) = none or $g(n) < distances[n.\mathrm{state}]$:
        $distances[n.\mathrm{state}] := g(n)$
        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
```

$\rightsquigarrow$ *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

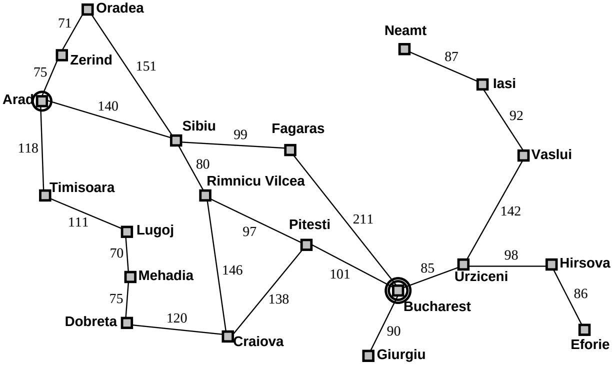
Greedy Best-first Search

Greedy Best-first Search

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

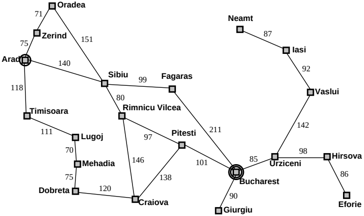
Note: usually without reopening (for reasons of efficiency)

Example: Greedy Best-first Search for Route Planning



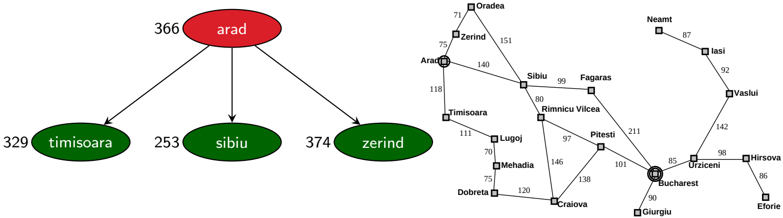
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Greedy Best-first Search for Route Planning



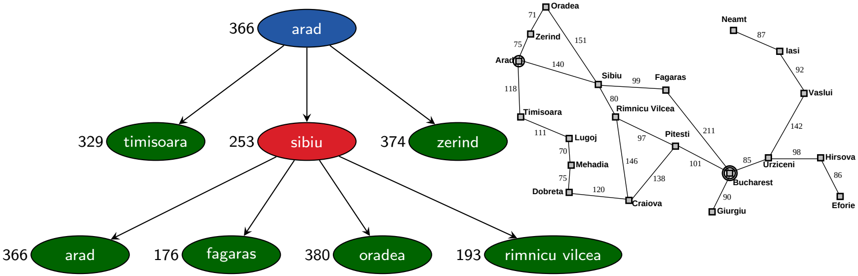
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example: Greedy Best-first Search for Route Planning



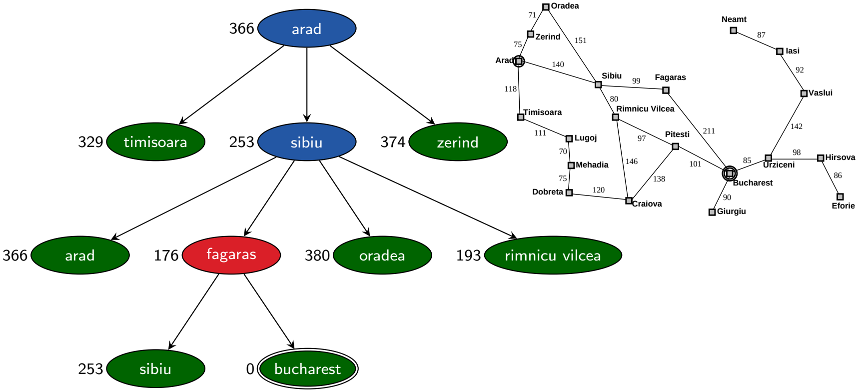
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example: Greedy Best-first Search for Route Planning



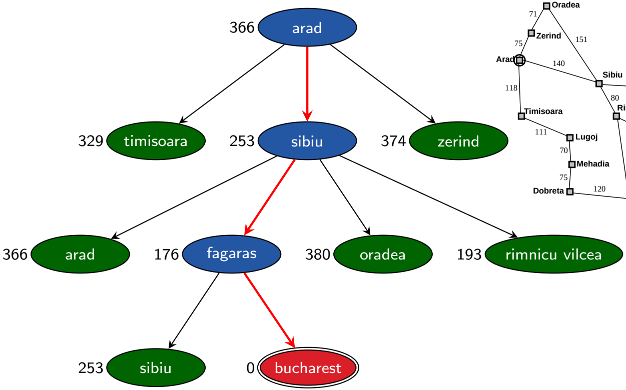
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

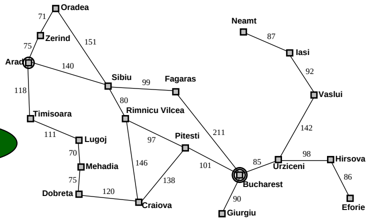
Example: Greedy Best-first Search for Route Planning



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^*

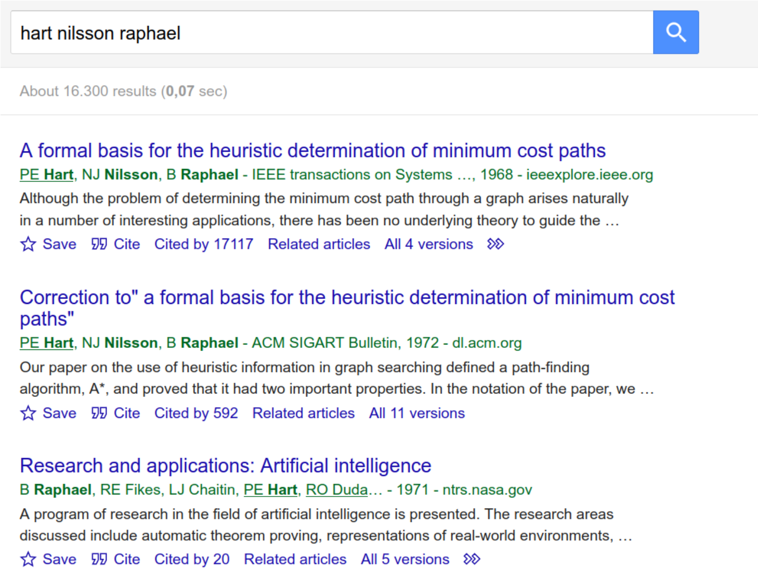
A*

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

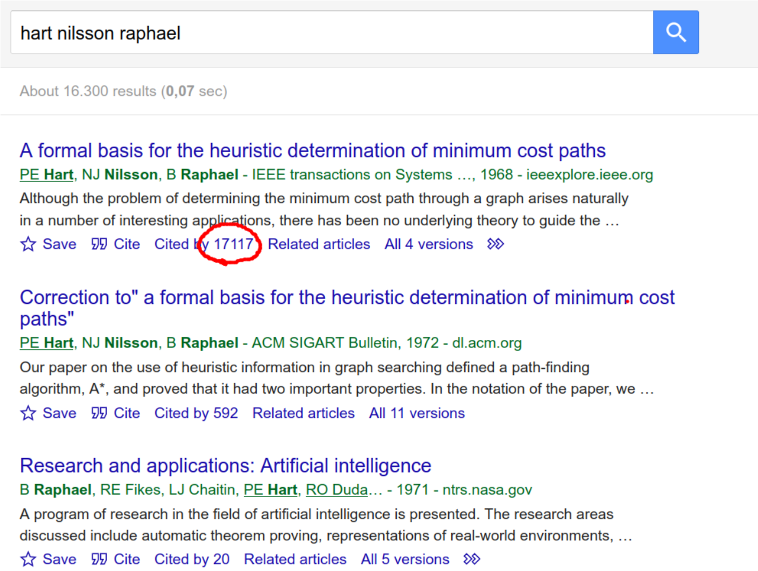
Research and applications: Artificial intelligence

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

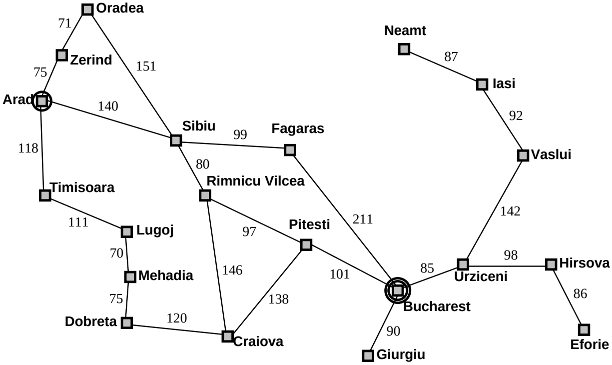
Research and applications: Artificial intelligence

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, …

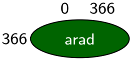
☆ Save 99 Cite Cited by 20 Related articles All 5 versions

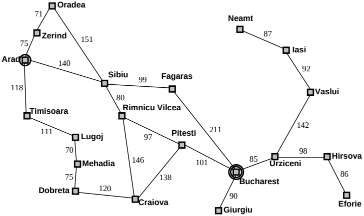
Example: A* for Route Planning



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

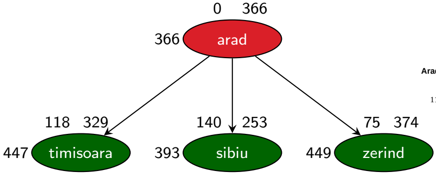
Example A* for Route Planning

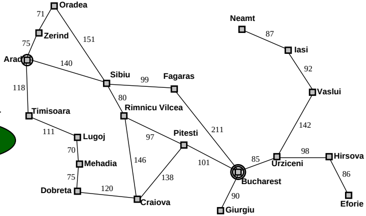




Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

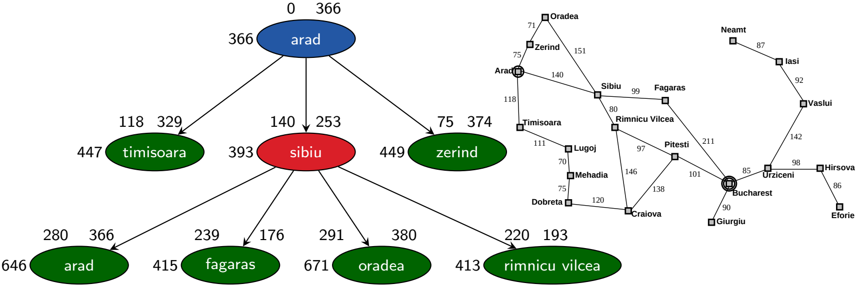
Example A* for Route Planning





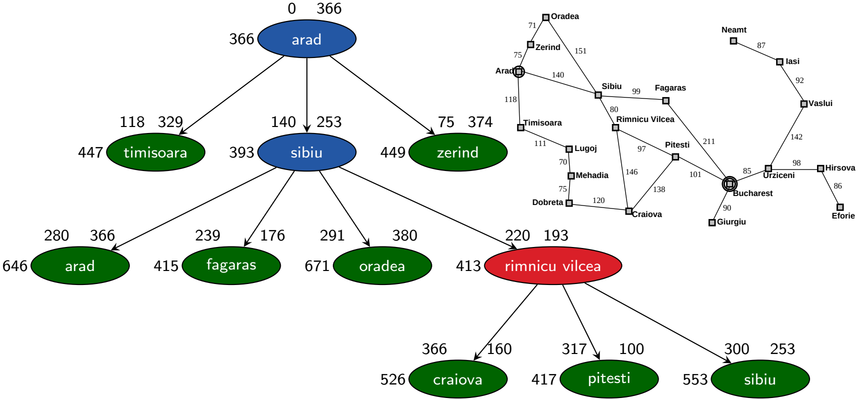
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



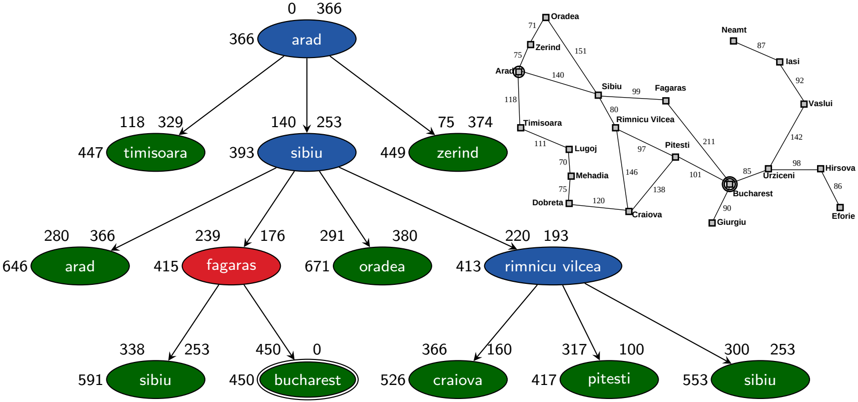
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A^* for Route Planning



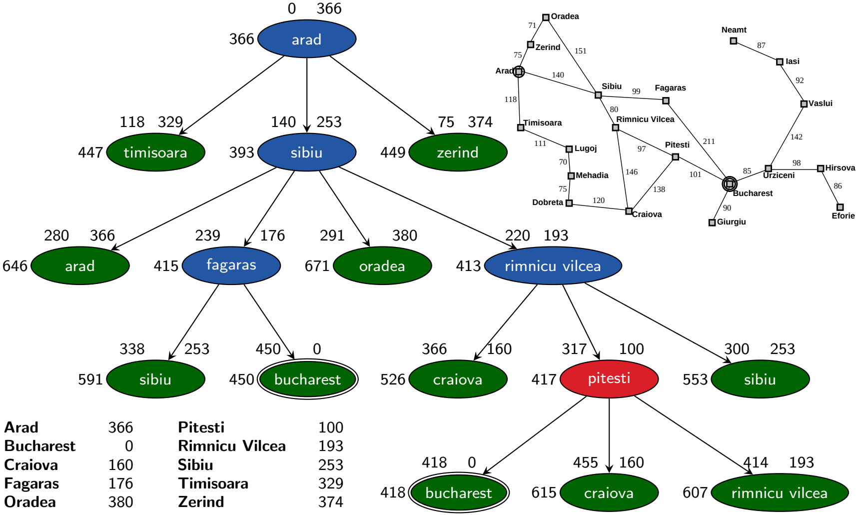
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

Example A* for Route Planning



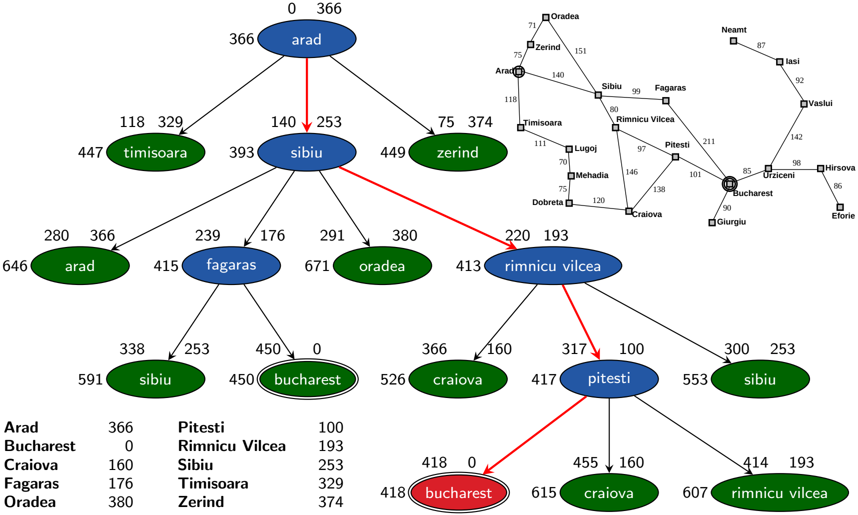
Arad	366	**Pitesti**	100
Bucharest	0	**Rimnicu Vilcea**	193
Craiova	160	**Sibiu**	253
Fagaras	176	**Timisoara**	329
Oradea	380	**Zerind**	374

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

⇝ 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:
$f(n) = g(n) + w \cdot h(n.\mathrm{state})$,
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*)

⇝ 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.

⇝ 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($s', g + \text{cost}(a), f\_limit$)
    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($s'$, $g + \text{cost}(a)$, $f\_limit$)
    if $solution \neq \textbf{none}$:
        $solution$.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
  - ⇝ exponentially slower in many state spaces
  - ⇝ 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!

⇝ next chapters: a closer look at A*

- A* with reopening ⇝ this chapter
- A* without reopening ⇝ 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 $\langle f, h \rangle$
if $h(\text{init}()) < \infty$:
    $open$.insert(make_root_node())
$distances$ := new HashMap
while not $open$.is_empty():
    $n$ := $open$.pop_min()
    if $distances$.lookup($n.state$) = none or $g(n) < distances[n.\text{state}]$:
        $distances[n.\text{state}] := g(n)$
        if is_goal($n$.state):
            return extract_path($n$)
        for each $\langle a, s' \rangle \in$ succ($n$.state):
            if $h(s') < \infty$:
                $n' := \text{make\_node}(n, a, s')$
                $open$.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?)
- $g^*(n.\text{state}) \leq g(n)$ 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 $\textit{distances}[s] = g^*(s)$.

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 $\langle s_\mathrm{I}, \dots, s, s' \rangle$ 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 $distances[s] := g^*(s)$.

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 $n.\mathrm{state} = s$ 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.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 $\langle s_0, \dots, s_n \rangle$ 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 $\langle s_0, \dots, s_n \rangle$ 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 $s_0 = s_{\text{I}}$ 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 $\langle s_0, \dots, s_{i-1}, s_i \rangle$ 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 $\langle s_0, \dots, s_{i-1}, s_i \rangle$ 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 $\langle s_0, \dots, s_{i-1}, s_i \rangle$ 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'.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 $\langle s_0, \dots, s_{i-1}, s_i \rangle$ 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'.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 $\langle f, h\rangle$
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($n$)
        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
```

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 $s = n$.state, $s' = n'$.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 $s = n.\mathrm{state}$, $s' = n'.\mathrm{state}$.

- 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 $s = n.\mathrm{state}$, $s' = n'.\mathrm{state}$.

- 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 $f(n) = f_\mathrm{b}$.

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 $f(n) = f_{\mathrm{b}}$.
- 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 $f(n) = f_{\mathrm{b}}$.
- 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:

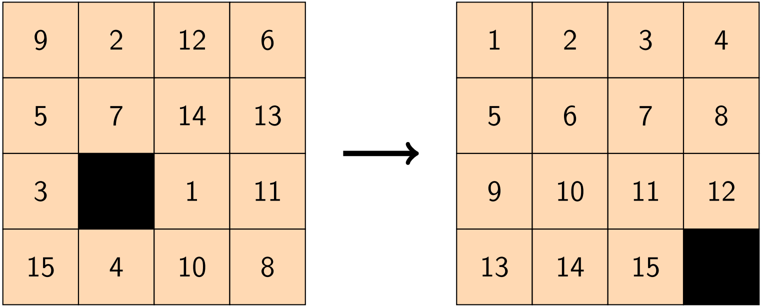
- if state space is a tree: time complexity of A^* grows linearly in solution length (Pohl 1969; Gaschnig 1977)
- general search spaces: runtime of A^* grows exponentially in solution length (Helmert & Röger 2008)

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)



9	2	12	6
5	7	14	13
3		1	11
15	4	10	8

$\longrightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

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.

⇝ 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?

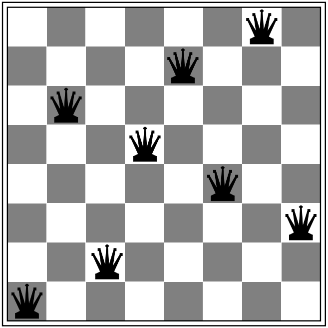
German: 8-Damen-Problem

- 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

- $C = \{1, \dots, 8\}^8$
- $S = \{\langle r_1, \dots, r_8 \rangle \mid \forall 1 \leq i < j \leq 8 : r_i \neq r_j \land |r_i - r_j| \neq |i - j|\}$
- 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
  ⇝ Part B
- formulation as constraint network ⇝ Part D
- formulation as logical satisfiability problem ⇝ Part E
- formulation as mathematical optimization problem (LP/IP)
  ⇝ not in this course
- local search ⇝ 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 ⇝ very memory-efficient (however, not complete or optimal)
- often initialization with random candidate
- iterative improvement by hill climbing

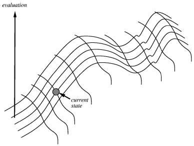
Hill Climbing

Hill Climbing (for Maximization Problems)

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

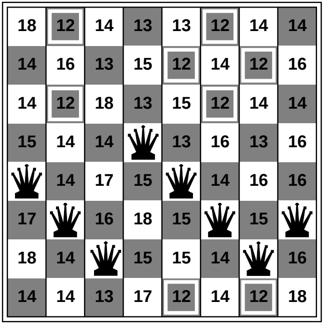
- always terminates (Why?)
- no guarantee that result is a solution
- if result is a solution, it is locally optimal w.r.t. h, but no global quality guarantees

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



18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

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

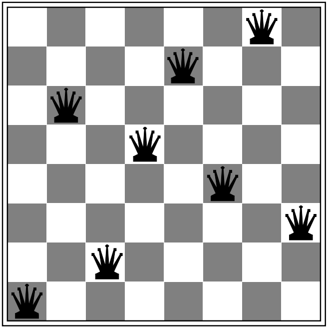
- C1. Introduction and Hill-Climbing
- C2. Advanced Techniques

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:

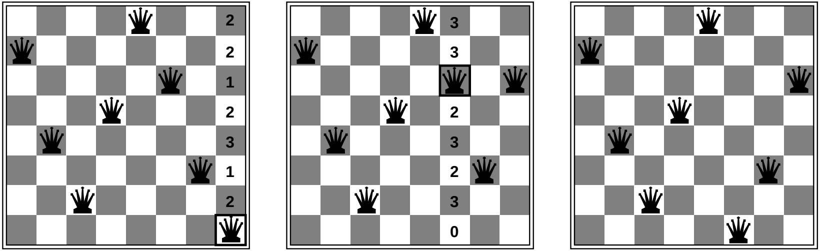
- with a bound of 100 steps solution found in 96% of the cases
- on average 22 steps until solution found

⇝ 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).



⇝ 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 $t \in \{1, \dots, N\}$:
    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)

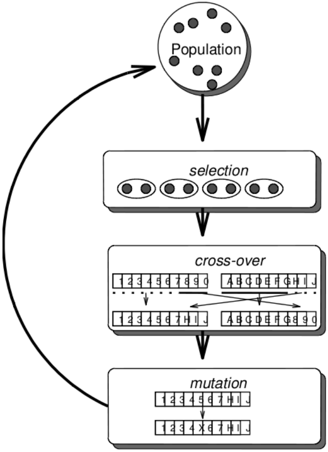
German: Evolution, Selektion, Kreuzung, Mutation, Genom, Fitnessfunktion, Population, Individuen

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)