RUSSELL & NORVIG, CHAPTERS 1–2: INTRODUCTION TO AI

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

21 March, 2017

TABLE OF CONTENTS

WHAT IS AI? (R&N 1.1–1.2)

WHAT IS INTELLIGENCE?

STRONG AND WEAK AI

WHAT IS INTELLIGENCE?

> *”It is not my aim to surprise or shock you – but the simplest way I can summarize is to say that there are now in the world machines that can think, that learn, and that create.*
>
> *Moreover, their ability to do these things is going to increase rapidly until — in a visible future — the range of problems they can handle will be coextensive with the range to which human mind has been applied.”*
>
> *by Herbert A Simon* ***(1957)***

STRONG AND WEAK AI

Weak AI — acting intelligently

- the belief that machines can be made to act as if they are intelligent

Strong AI — being intelligent

- the belief that those machines are actually thinking

Most AI researchers don’t care

- *“the question of whether machines can think…*
  …is about as relevant as whether submarines can swim.”
  (Edsger W Dijkstra, 1984)

WEAK AI

Weak AI is a category that is flexible

- as soon as we understand how an AI-program works, it appears less “intelligent”.

And as soon as a part of AI is successful, it becomes an own research area!

- E.g., large parts of advanced search, parts of language understanding, parts of machine learning and probabilistic learning etc.

And AI is left with the remaining hard-to-solve problems!

WHAT IS AN AI SYSTEM?

Do we want a system that…

- thinks like a human?
  - cognitive neuroscience / cognitive modelling
  - AGI = artificial general intelligence
- acts like a human?
  - the Turing test
- thinks rationally?
  - “laws of thought”
  - from Aristotle’s syllogism to modern day theorem provers
- acts rationally?
  - “rational agents”
  - maximise goal achievement, given available information

A BRIEF HISTORY OF AI (R&N 1.3)

NOTABLE AI MOMENTS, 1940–2016

NOTABLE AI MOMENTS (1940–1975)

1943	McCulloch & Pitts: Boolean circuit model of brain
1950	Alan Turing's "Computing Machinery and Intelligence"
1951	Marvin Minsky develops a neural network machine
1950s	Early AI programs: e.g., Samuel's checkers program, Gelernter's Geometry Engine, Newell & Simon's Logic Theorist and General Problem Solver
1956	Dartmouth meeting: "Artificial Intelligence" adopted
1965	Robinson's complete algorithm for logical reasoning
1966	Joseph Weizenbaum creates Eliza
1969	Minsky & Papert show limitations of the perceptron Neural network research almost disappears
1971	Terry Winograd's Shrdlu dialogue system
1972	Alain Colmerauer invents Prolog programming language

NOTABLE AI MOMENTS (1975–2016)

1976	MYCIN, an expert system for disease diagnosis
1980s	Era of expert systems
1990s	Neural networks, probability theory, AI agents
1993	RoboCup initiative to build soccer-playing robots
1997	IBM Deep Blue beats the World Chess Champion
2003	Very large datasets: genomic sequences
2007	Very large datasets: WAC (web as corpus)
2011	IBM Watson wins Jeopardy
2012	US state of Nevada permits driverless cars
2014	“Deep learning”: recommendation systems, image tagging, board games, speech translation, pattern recognition
2016	Google AlphaGo beats the world’s 2nd best Go player, Lee Se-dol

INTERLUDE: WHAT IS THIS COURSE, ANYWAY?

PEOPLE, CONTENTS AND DEADLINES

PEOPLE AND LITERATURE

Course website	http://chalmersgu-ai-course.github.io/
Teachers	Peter Ljunglöf, John J. Camilleri, Jonatan Kilhamn, Inari Listenmaa, Claes Strannegård
Student representatives	Caterina Curta (N2COS), Claudia Castillo (MPALG), Ibrahim Fayaz (MPALG), Johan Ek (MPCAS), Tarun Nandakumar (MPCAS), Yan Wang (MPALG) *(updated 22nd March)*
Course book	Russell & Norvig (2002/10/14) Read it online at Chalmers library: http://goo.gl/6EMRZr

Note for GU students: ***Don't forget to register, today!***

COURSE CONTENTS

This is what you (hopefully) will learn during this course:

- Introduction to AI history, philosophy and ethics.
- Basic algorithms for searching and solving AI problems:
  - heuristic search,
  - local search,
  - nondeterministic search,
  - games and adversarial search,
  - constraint satisfaction problems.
- Group collaboration:
  - write an essay,
  - complete a programming project.

WHAT IS *NOT* IN THIS COURSE?

This course is an introduction to AI, giving a broad overview of the area and some basic algorithms.

- We do not have the time to dig into the most recent algorithms and techniques that are so hyped in current media.
- Therefore, you will not learn how these things work:
    - machine learning,
    - deep neural networks,
    - self-driving cars,
    - beating the world champion in Go,
    - etc.

DEADLINES FOR COURSE MOMENTS

Group work: Form a group

- Form a group (24 March), and sign a group contract (29 March)

Group work: Write an essay

- Write a 6-page essay about AI (12 May) + review two essays (19 May)
- Revise your essay according to the reviews you got (2 June)

Group work: Shrdlite programming project

- Intermediate labs: A* planner (5–6 April) + interpreter (26–27 April)
- Complete the final project (26 May)

Written and oral examination

- *Peer-corrected* exam (2 May) + normal re-exams (8 June, 21 August)
- Oral review of the project (29–31 May)
- Individual self- and peer evaluation (28 May)

RECURRING COURSE MOMENTS

Lectures

- Tuesday and Friday, 10:00–11:45, during weeks 12–14, 16–17

Obligatory group supervision

- Wednesdays and Thursdays (mostly) during weeks 13–14, 16–21
- Supervision is compulsory for all group members!

Drop-in supervision

- Mondays during weeks 13–14, 17–21

Practice sessions

- Tuesday and Friday, 8:00–9:45, weeks 16–17

GRADING

Higher grade than pass/3/G only depends on the group work!

- For higher grades you can collect up to 10 bonus points:
    - The essay can give 0–3 points
    - Your reviews can give 0–1 points
    - Shrdlite can give 0–6 points (every extension gives 1–3 points)
    - Your individual bonus points can be more or less than your group's

	Grade	Bonus points
Chalmers	3	0–3
	4	4–6
	5	7–10
GU	G	0–5
	VG	6–10

THE WRITTEN EXAMINATION

The exam is 2nd May (in the middle of the course)

- *Why?* So that you can focus on Shrdlite and the essay in the end

The exam is only pass/fail

- *Why?* This course is mainly a project course
  (5.0 hec group work, 2.5 hec written exam)

The exam is peer-corrected

- *Why?* It’s not only an exam, it’s also a learning experience.
- *How?* First you write your exam. We collect all theses, shuffle and hand them out again, so that you will get someone else’s exam to correct. We go through the answers on the blackboard and you correct the exam in front of you. Finally, we check all corrections.
- And don’t worry – everything will be anonymous!

THE ESSAY

Your project group will write a 6-page essay about the historical, ethical and/or philosophical aspects of an AI topic.

After submitting your essay, you will get two other essays to read and review.

You will also get reviews on your essay, which you update and submit a final version.

Claes Strannegård is responsible for the essay. He will organise supervision sessions for all of you, regarding the essay.

SHRDLITE, THE PROGRAMMING PROJECT

Your group will implement a dialogue system for controlling a robot that lives in a virtual block world and whose purpose in life is to move around objects of different forms, colors and sizes.

You will program in TypeScript

- *Why?* It’s a type-safe version of Javascript (runs in the browser), and it’s a new language for almost all of you!

Every group will get a personal supervisor, which you meet once every week.

There are two intermediate labs, which you submit by showing them to your supervisor.

Note: the Shrdlite webpage is quite long, and not everything makes sense when you start the project. Make sure to visit the webpage regularly when you are developing your project — there is a lot of important information there.

LET’S HAVE A LOOK AT THE WEB PAGES!

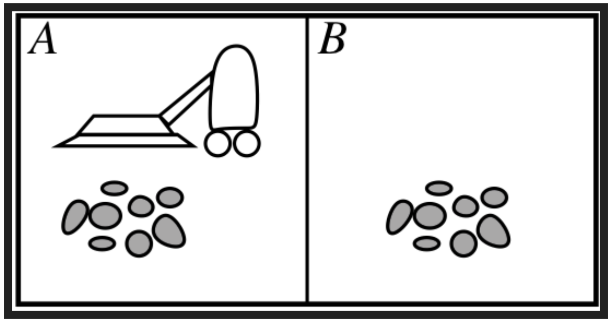
http://chalmersgu-ai-course.github.io/

AGENTS (R&N CHAPTER 2)

RATIONALITY

ENVIROMENT TYPES

EXAMPLE: A VACUUM-CLEANER AGENT



Percepts: location and contents, e.g. $(A, Dirty)$
Actions: *Left*, *Right*, *Suck*, *NoOp*

A simple agent function is:

- If the current square is dirty, then suck; otherwise, move to the other square.

How do we know if this is a good agent function?

- What is the best function? — Is there one?
- Who decides this?

RATIONALITY

Fixed performance measure evaluates the environment sequence

- one point per square cleaned up in time T?
- one point per clean square per time step, minus one per move?
- penalize for $> k$ dirty squares?

A rational agent chooses any action that
- maximizes the expected value of the performance measure
- given the percept sequence to date

Rationality and success
- Rational ≠ omniscient — percepts may not supply all relevant information
- Rational ≠ clairvoyant — action outcomes may not be as expected
- Hence, rational ≠ successful

PEAS

To design a rational agent,
we must specify the task environment,
which consists of the following four things:

***P*erformance measure**

***E*nvironment**

***A*ctuators**

***S*ensors**

EXAMPLE PEAS: AUTONOMOUS CAR

The task environment for an autonomous car:

***P*erformance measure**

getting to the right place, following traffic laws, minimising fuel consumption/time, maximising safety, …

***E*nvironment**

roads, other traffic, pedestrians, road signs, passengers, …

***A*ctuators**

steering, accelerator, brake, signals, loudspeaker, …

***S*ensors**

cameras, sonar, speedometer, GPS, odometer, microphone, …

ENVIROMENT TYPES: DIMENSIONS OF COMPLEXITY

Dimension	**Possible values**
Observable?	*full vs. partial*
Deterministic?	*deterministic vs. stochastic*
Episodic?	*episodic vs. sequential*
Static?	*static vs. dynamic (semidynamic)*
Discrete?	*discrete vs. continuous*
Number of agents	*single vs. multiple (competetive/cooperative)*

The environment type largely determines the agent design

ENVIRONMENT TYPES, EXAMPLES

	Chess (w. clock)	**Poker**	**Driving**	**Image recognition**
Observable?	*fully*	*partially*	*partially*	*fully*
Deterministic?	*determ.*	*stochastic*	*stochastic*	*determ.*
Episodic?	*sequential*	*sequential*	*sequential*	*episodic*
Static?	*semi*	*static*	*dynamic*	*static*
Discrete?	*discrete*	*discrete*	*continuous*	*disc./cont.*
N:o agents	*multiple (compet.)*	*multiple (compet.)*	*multiple (cooper.)*	*single*

The real world is (of course):

partially observable, stochastic, sequential, dynamic, continuous, multi-agent

DEFINING A SOLUTION

Given an informal description of a problem, what is a solution?

- Typically, much is left unspecified, but the unspecified parts cannot be filled in arbitrarily.
- Much work in AI is motivated by *common-sense reasoning*. The computer needs to make common-sense conclusions about the unstated assumptions.

QUALITY OF SOLUTIONS

Does it matter if the answer is wrong or answers are missing?

Classes of solutions:

- An *optimal solution* is a best solution according to some measure of solution quality.
- A *satisficing solution* is one that is good enough, according to some description of which solutions are adequate.
- An *approximately optimal solution* is one whose measure of quality is close to the best theoretically possible.
- A *probable solution* is one that is likely to be a solution.

TYPES OF AGENTS

Simple reflex agent	selects actions based on *current percept* — ignores history
Model-based reflex agent	maintains an *internal state* that depends on the percept history
Goal-based agent	has a *goal* that describes situations that are desirable
Utility-based agent	has a *utility function* that measures the performance
Learning agent	any of the above agents can be a learning agent — learning can be *online* or *offline*

PHILOSOPHY OF AI

IS AI POSSIBLE?

TURING’S OBJECTIONS TO AI

IS AI POSSIBLE?

There are different opinions…

- …some are slightly positive:
  - “every […] feature of intelligence can be so precisely described that a machine can be made to simulate it” (McCarthy et al, 1955)
- …and some lean towards the negative:
  - “AI […] stands not even a ghost of a chance of producing durable results” (Sayre, 1993)

It’s all in the definitions:

- what do we mean by “thinking” and “intelligence”?

“COMPUTING MACHINERY AND INTELLIGENCE”

The most important paper in AI, of all times:

- (and I’m not the only one who thinks that…)
- “Computing Machinery and Intelligence” (Turing, 1950)
    - introduced the “imitation game” (Turing test)
    - discussed objections against intelligent machines, including almost every objection that has been raised since then
    - it’s also easy to read… so you really have to read it!

TURING’S OBJECTIONS TO AI [1–3]

(1) The Theological Objection

- “Thinking is a function of man’s immortal soul. God has given an immortal soul to every man and woman, but not to any other animal or to machines. Hence no animal or machine can think.”

(2) The “Heads in the Sand” Objection

- “The consequences of machines thinking would be too dreadful. Let us hope and believe that they cannot do so.”

(3) The Mathematical Objection

- Based on Gödel’s incompleteness theorem.

TURING’S OBJECTIONS TO AI [4–5]

(4) The Argument from Consciousness

- “No mechanism could feel […] pleasure at its successes, grief when its valves fuse, […], be angry or depressed when it cannot get what it wants.”

(5) Arguments from Various Disabilities

- “you can make machines do all the things you have mentioned but you will never be able to make one to do X.”
- where X can… “be kind, resourceful, beautiful, friendly, […], have a sense of humour, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, […], use words properly, be the subject of its own thought, […], do something really new.”

TURING’S OBJECTIONS TO AI [6–8]

(6) Lady Lovelace’s Objection

- “The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform.”

(7) Argument from Continuity in the Nervous System

- “one cannot expect to be able to mimic the behaviour of the nervous system with a discrete-state system.”

(8) The Argument from Informality of Behaviour

- “if each man had a definite set of rules of conduct by which he regulated his life he would be no better than a machine. But there are no such rules, so men cannot be machines.”

THE FINAL OBJECTION [9]

(9) The Argument from Extrasensory Perception

- this was the strongest argument according to Turing…
- “the statistical evidence […] is overwhelming”
- “Let us play the imitation game, using as witnesses a man who is good as a telepathic receiver, and a digital computer. The interrogator can ask such questions as ‘What suit does the card in my right hand belong to?’ The man by telepathy or clairvoyance gives the right answer 130 times out of 400 cards. The machine can only guess at random, and perhaps gets 104 right, so the interrogator makes the right identification.”

STRONG AI: BRAIN REPLACEMENT

The brain replacement experiment

- by Searle (1980) and Moravec (1988)
- suppose we gradually replace each neuron in your head with an electronic copy…
    - …what will happen to your mind, your consciousness?
    - Searle argues that you will gradually feel dislocated from your body
    - Moravec argues you won’t notice anything

STRONG AI: THE CHINESE ROOM

The Chinese room experiment (Searle, 1980)

- an English-speaking person takes input and generates answers in Chinese
    - he/she has a rule book, and stacks of paper
    - the person gets input, follows the rules and produces output
- i.e., the person is the CPU, the rule book is the program and the papers is the storage device

Does the system understand Chinese?

THE TECHNOLOGICAL SINGULARITY

Will AI lead to superintelligence?

- “…ever accelerating progress of technology and changes in the mode of human life, which gives the appearance of approaching some essential singularity in the history of the race beyond which human affairs, as we know them, could not continue” (von Neumann, mid-1950s)
- “We will successfully reverse-engineer the human brain by the mid-2020s. By the end of that decade, computers will be capable of human-level intelligence.” (Kurzweil, 2011)
- “There is not the slightest reason to believe in a coming singularity.” (Pinker, 2008)

ETHICAL ISSUES OF AI

What are the possible risks of using AI technology?

- AI might be used towards undesirable ends
  - e.g., surveillance by speech recognition, detection of “terrorist phrases”
- AI might result in a loss of accountability
  - what’s the legal status of a self-driving car?
  - or a medical expert system?
- AI might mean the end of the human race
  - what if the new superintelligent race won’t obey Asimov’s robot laws?

CHAPTER 3: CLASSICAL SEARCH ALGORITHMS

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

24 March, 2017

TABLE OF CONTENTS

INTRODUCTION (R&N 3.1–3.3)

GRAPHS AND SEARCHING

EXAMPLE PROBLEMS

A GENERIC SEARCHING ALGORITHM

GRAPHS AND SEARCHING

Often we are not given an algorithm to solve a problem, but only a specification of a solution — we have to search for it.

A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.

Many AI problems can be abstracted into the problem of finding a path in a directed graph.

Often there is more than one way to represent a problem as a graph.

STATE-SPACE SEARCH: COMPLEXITY DIMENSIONS

Observable?	fully
Deterministic?	deterministic
Episodic?	episodic
Static?	static
Discrete?	discrete
N:o of agents	single

Most complex problems (partly observable, stochastic, sequential) usualy have components using state-space search.

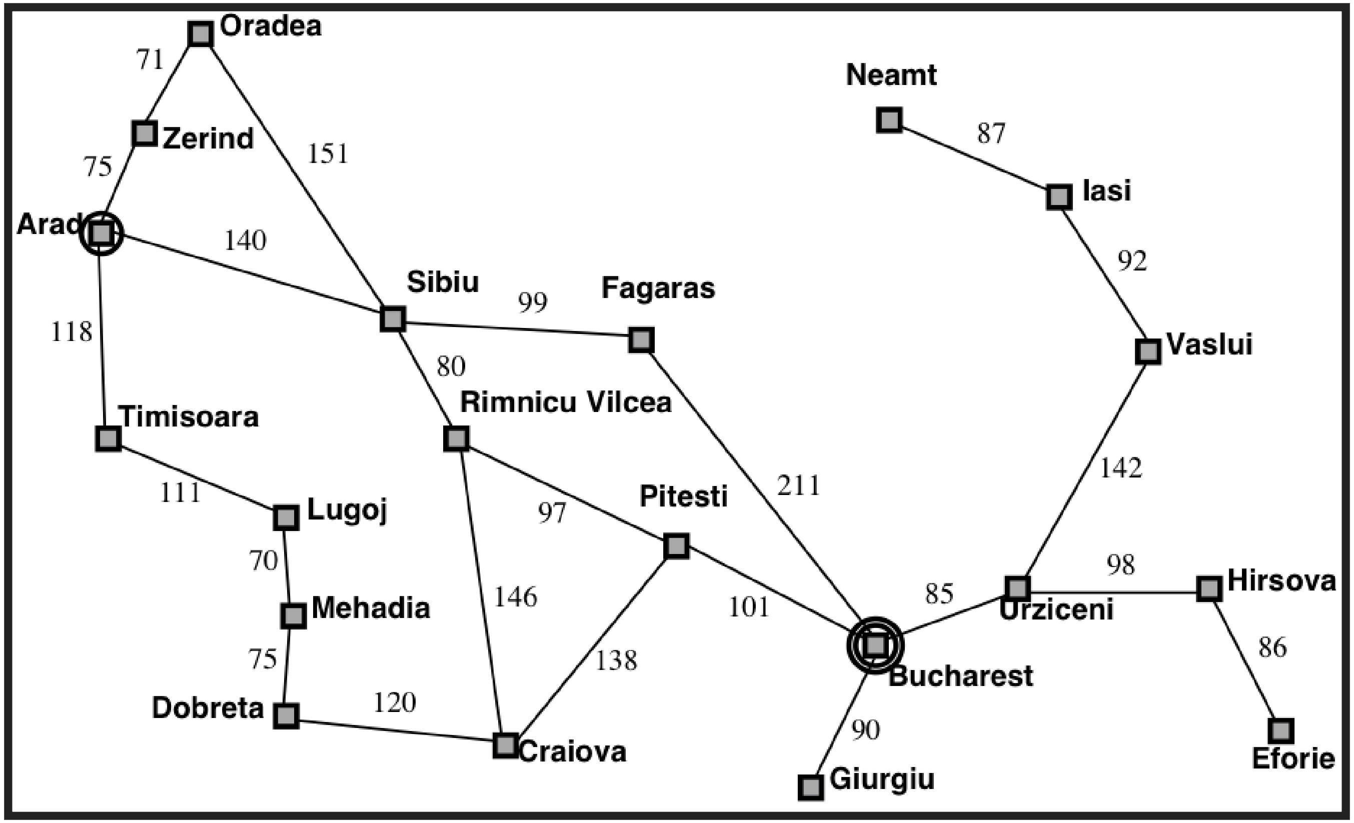
DIRECTED GRAPHS

A *graph* consists of a set N of *nodes* and a set A of ordered pairs of nodes, called *arcs*.

- Node n_2 is a *neighbor* of n_1 if there is an arc from n_1 to n_2.
  That is, if $(n_1, n_2) \in A$.
- A *path* is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ such that $(n_{i-1}, n_i) \in A$.
- The *length* of path $(n_0, n_1, \ldots, n_k)$ is k.
- A *solution* is a path from a start node to a goal node,
  given a set of *start nodes* and *goal nodes*.
- (Russel & Norvig sometimes call the graph nodes *states*).

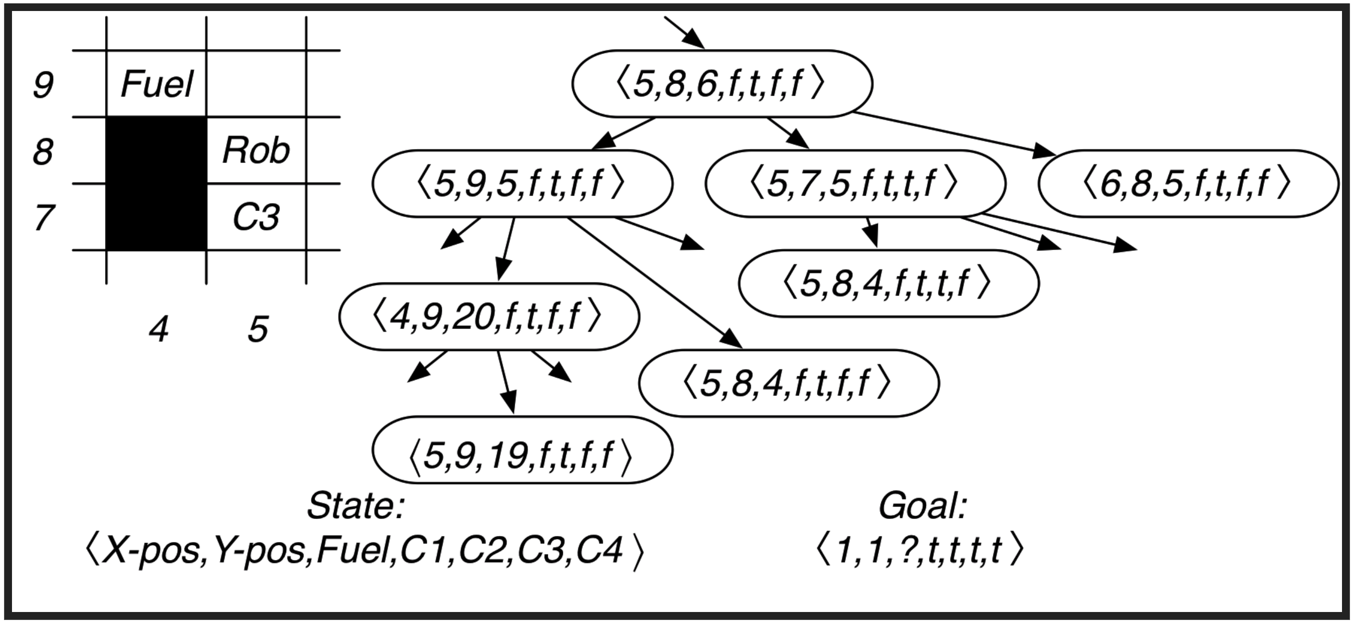
EXAMPLE: TRAVEL IN ROMANIA

We want to drive from Arad to Bucharest in Romania



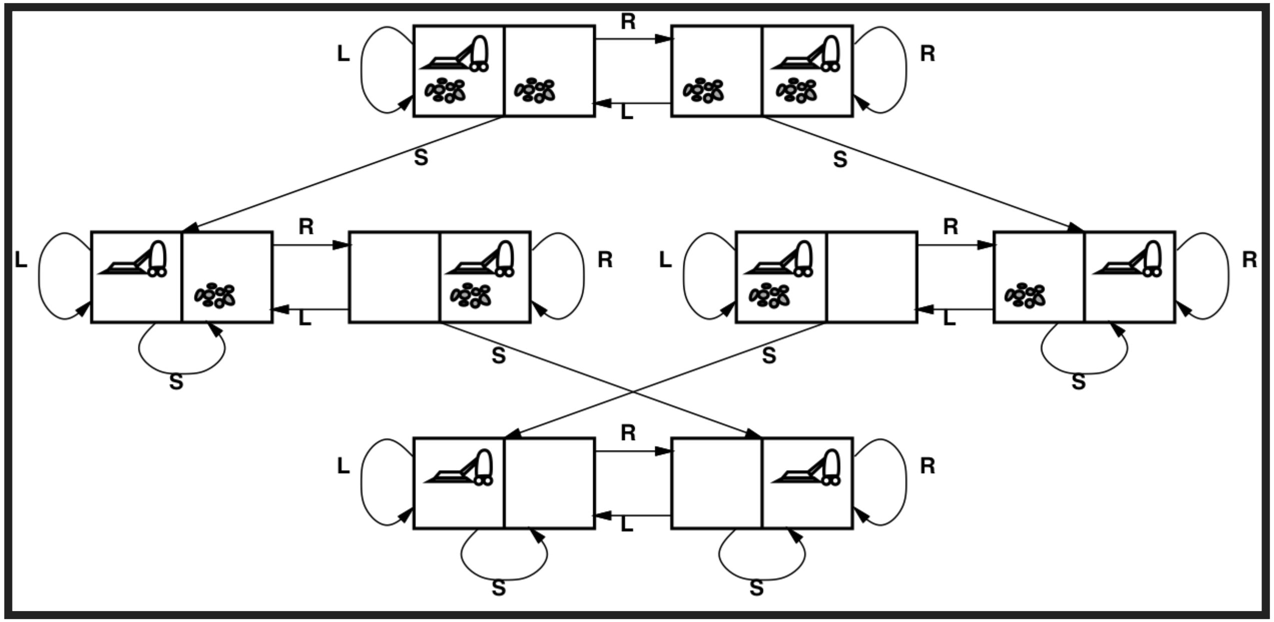
EXAMPLE: GRID GAME

Grid game: Rob needs to collect coins C_1, C_2, C_3, C_4, without running out of fuel, and end up at location (1,1):



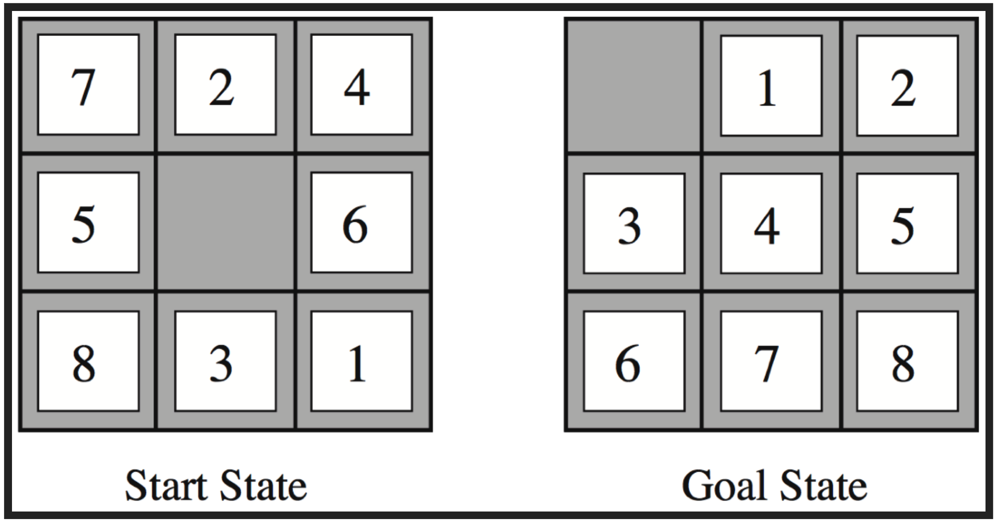
What is a good representation of the *search states* and the *goal*?

EXAMPLE: VACUUM-CLEANING AGENT



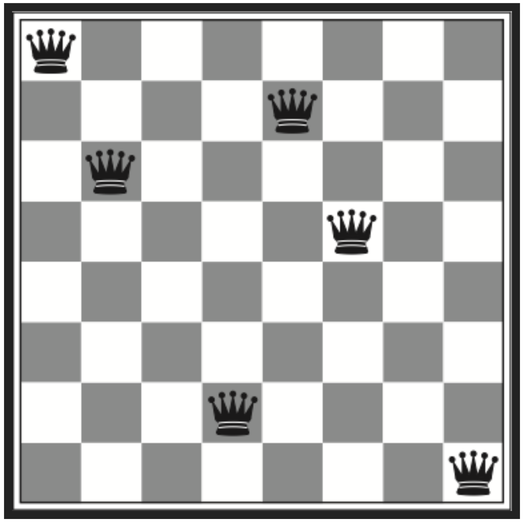
States	*[room A dirty?, room B dirty?, robot location]*
Initial state	*any state*
Actions	*left, right, suck, do-nothing*
Goal test	*[false, false, –]*
Path cost	*1 per action (0 for do-nothing)*

EXAMPLE: THE 8-PUZZLE



States	*a 3 x 3 matrix of integers*
Initial state	*any state*
Actions	*move the blank space: left, right, up, down*
Goal test	*equal to the goal state*
Path cost	*1 action (0 for do-nothing)*

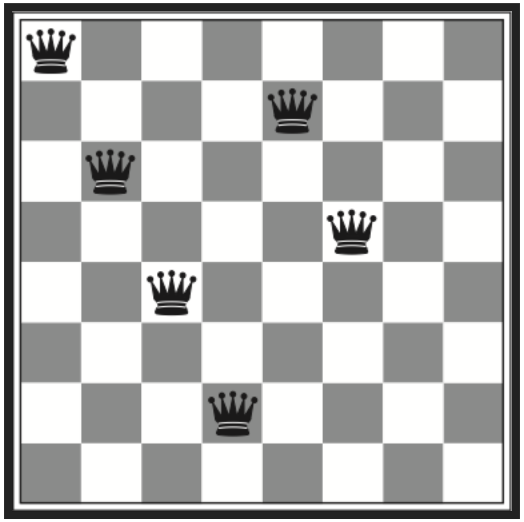
EXAMPLE: THE 8-QUEENS PROBLEM



States	*any arrangement of 0 to 8 queens on the board*
Initial state	*no queens on the board*
Actions	*add a queen to any empty square*
Goal test	*8 queens on the board, none attacked*
Path cost	*1 per move*

This gives us $64 \times 63 \times \cdots \times 57 \approx 1.8 \times 10^{14}$ possible paths to explore!

EXAMPLE: THE 8-QUEENS PROBLEM (ALTERNATIVE)



States	*one queen per column in leftmost columns, none attacked*
Initial state	*no queens on the board*
Actions	*add a queen to a square in the leftmost empty column, make sure that no queen is attacked*
Goal test	*8 queens on the board, none attacked*
Path cost	*1 per move*

Using this formulation, we have only 2,057 paths!

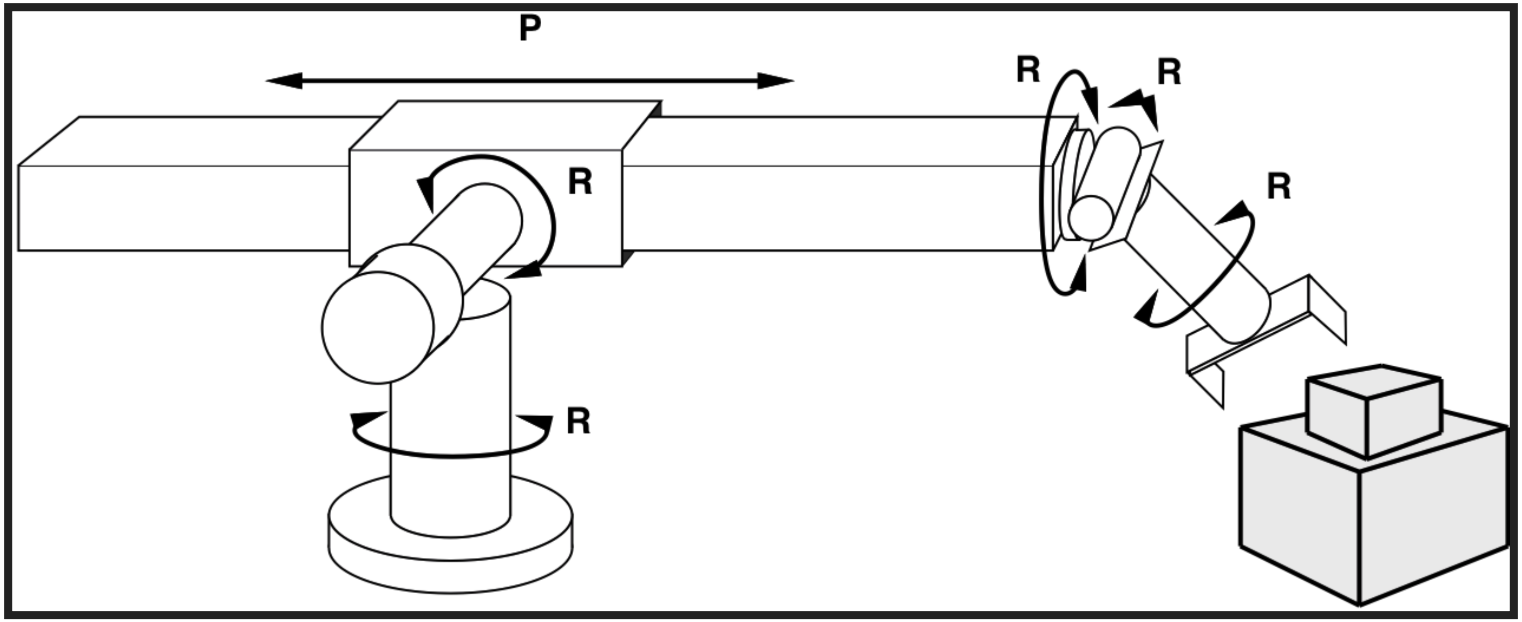
EXAMPLE: KNUTH’S CONJECTURE

Donald Knuth conjectured that all positive integers can be obtained by starting with the number 4 and applying some combination of the factorial, square root, and floor.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

States	*positive numbers* $(1, 2, 2.5, 3, \sqrt{2}, 1.23 \cdot 10^{456}, \sqrt{\sqrt{2}}, \ldots)$
Initial state	*4*
Actions	*apply factorial, square root, or floor operation*
Goal test	*any positive integer (e.g., 5)*
Path cost	*1 per move*

EXAMPLE: ROBOTIC ASSEMBLY



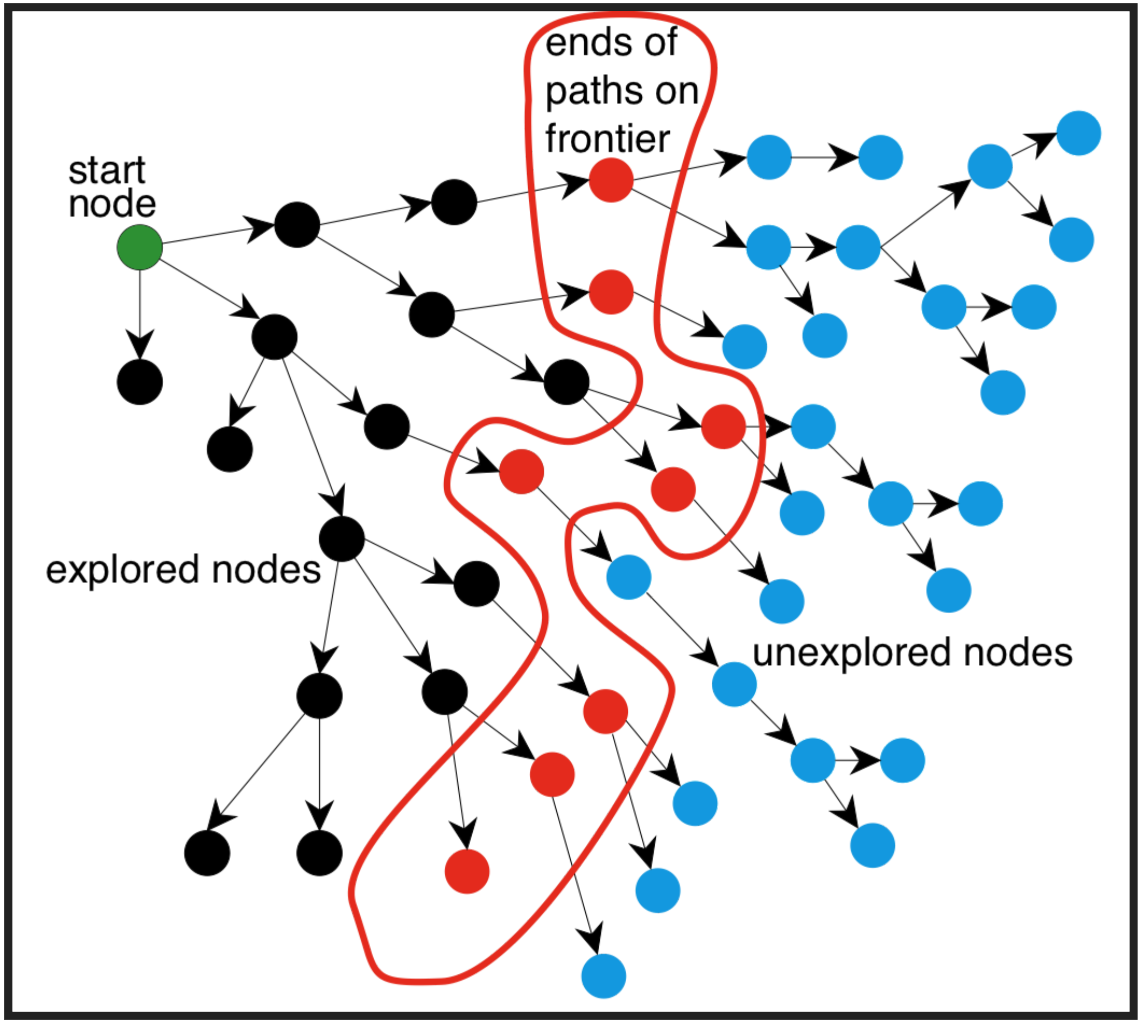
States	*real-valued coordinates of robot joint angles parts of the object to be assembled*
Actions	*continuous motions of robot joints*
Goal test	*complete assembly of the object*
Path cost	*time to execute*

HOW DO WE SEARCH IN A GRAPH?

A generic search algorithm:

- Given a graph, start nodes, and a goal description, incrementally explore paths from the start nodes.
- Maintain a *frontier* of nodes that are to be explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

ILLUSTRATION OF SEARCHING IN A GRAPH



TURNING TREE SEARCH INTO GRAPH SEARCH

Tree search: Don’t check if nodes are visited multiple times
Graph search: Keep track of visited nodes

```
function Search(graph, initialState, goalState):
    initialise frontier using the initialState
    initialise exploredSet to the empty set
    while frontier is not empty:
        select and remove node from frontier
        if node.state is a goalState then return node
        add node to exploredSet
        for each child in ExpandChildNodes(node, graph):
            add child to frontier … if child is not in frontier or exploredSet
    return failure
```

GRAPH NODES VS. SEARCH NODES

The nodes used while searching are not the same as the graph nodes:

Search nodes should contain more information:

- the corresponding graph node (called state in R&N)
- the total path cost from the start node
- the estimated (heuristic) cost to the goal
- enough information to be able to calculate the final path

```
procedure ExpandChildNodes(parent, graph):
    for each (action, child, edgecost) in graph.successors(parent.state):
        yield new SearchNode(child,
                             …total cost so far…,
                             …estimated cost to goal…,
                             …information for calculating final path…)
```

UNINFORMED SEARCH (R&N 3.4)

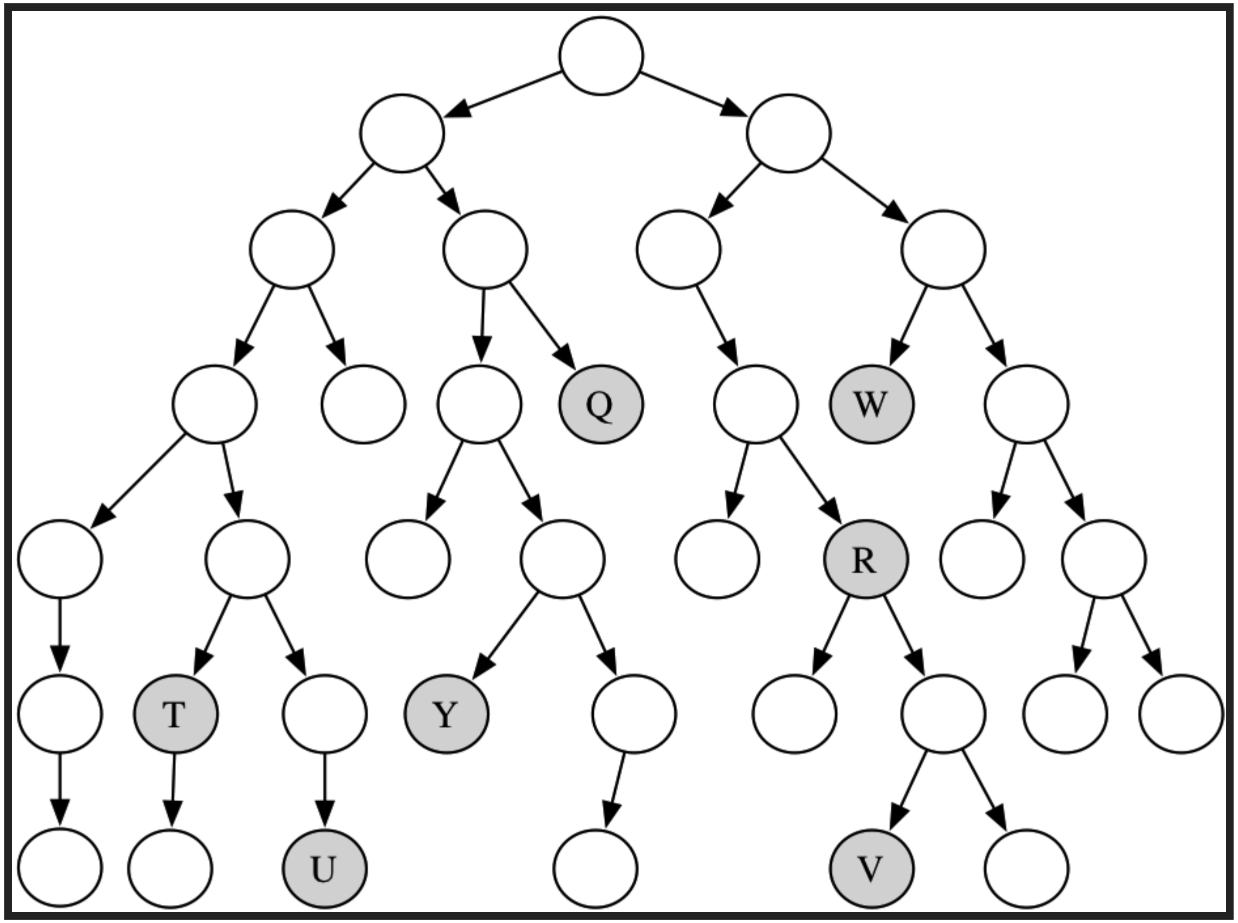
DEPTH-FIRST SEARCH

BREADTH-FIRST SEARCH

UNIFORM-COST SEARCH

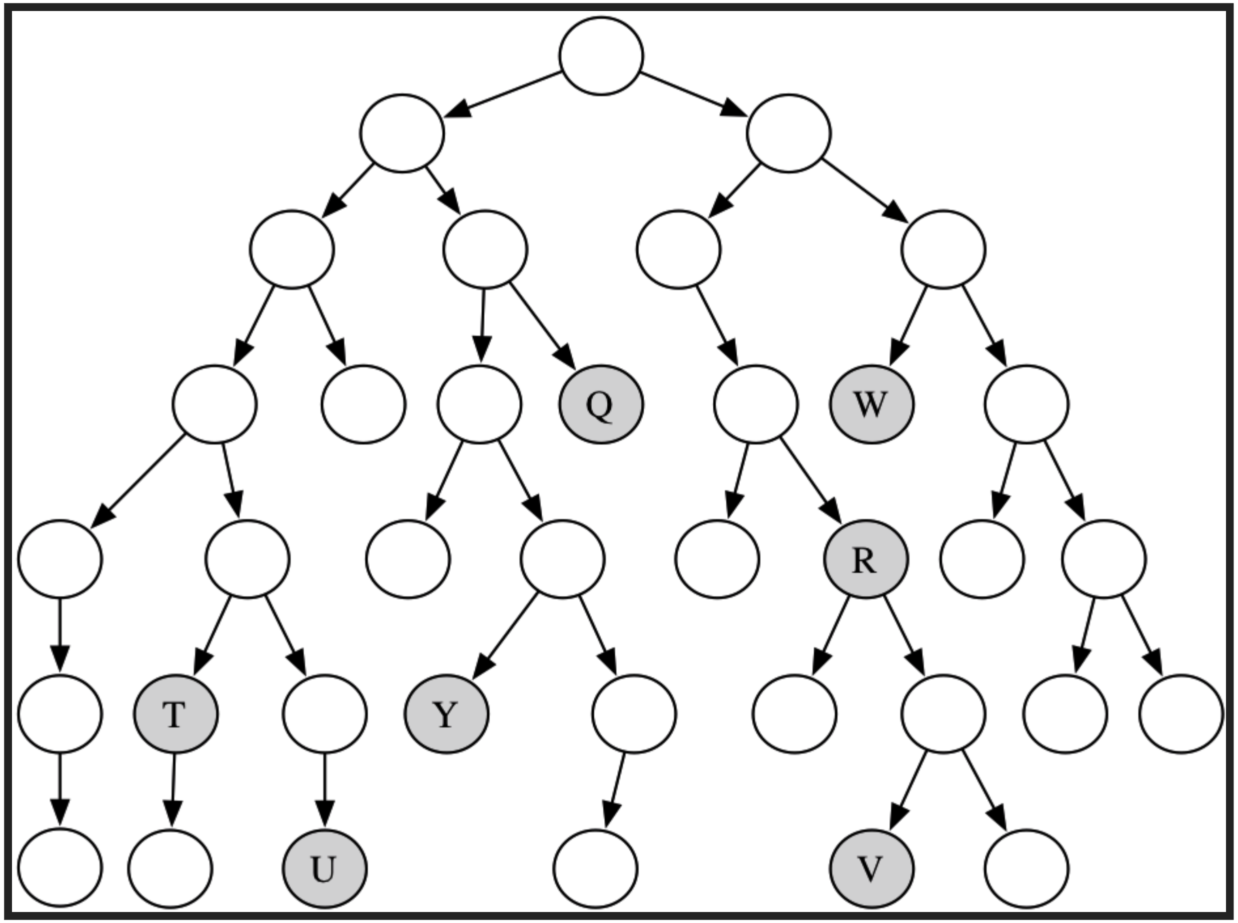
QUESTION TIME: DEPTH-FIRST SEARCH

Which shaded goal will a depth-first search find first?



QUESTION TIME: BREADTH-FIRST SEARCH

Which shaded goal will a breadth-first search find first?



DEPTH-FIRST SEARCH

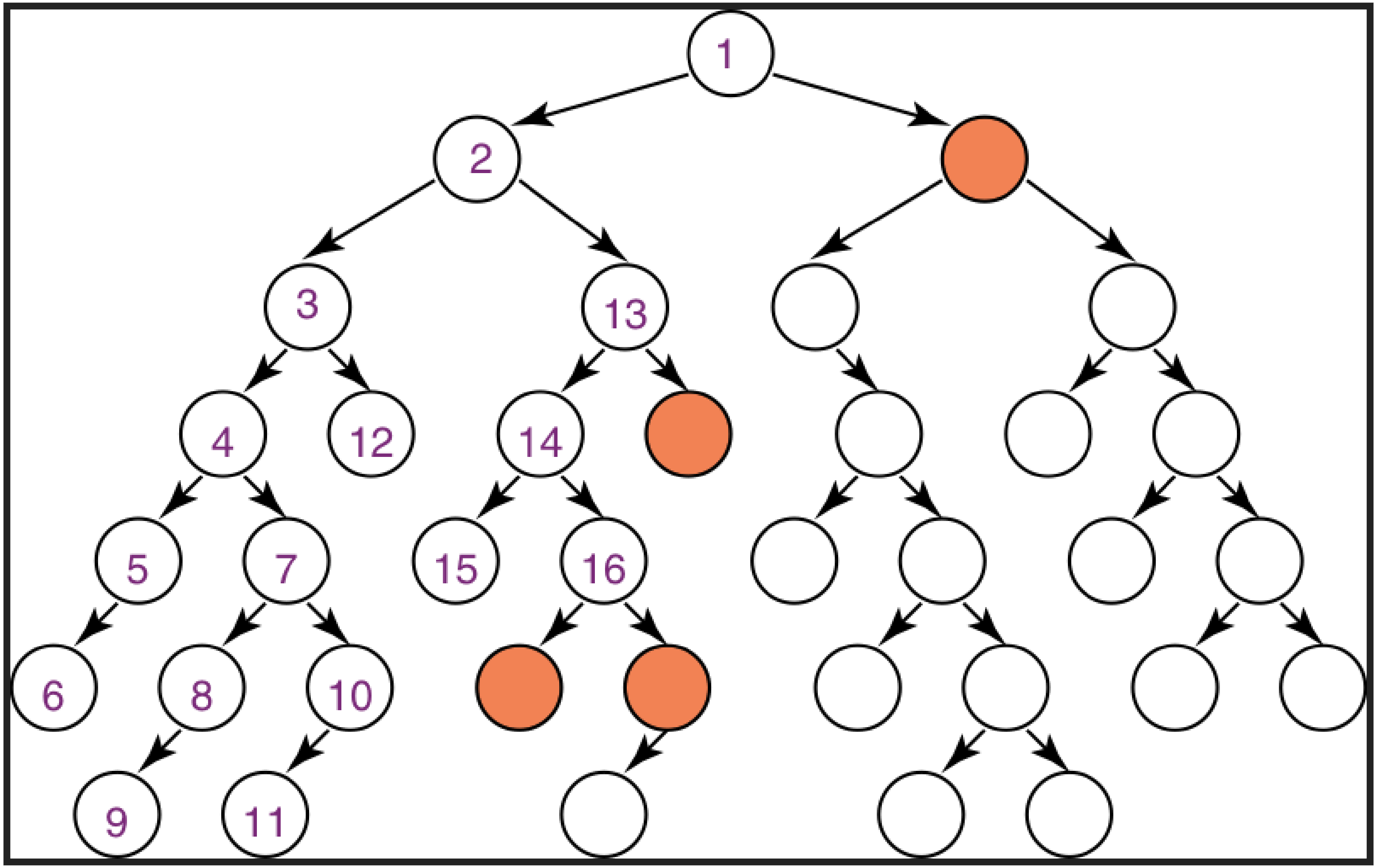
Depth-first search treats the frontier as a stack.

It always selects one of the last elements added to the frontier.

If the list of nodes on the frontier is $[p_1, p_2, p_3, \ldots]$, then:

- p_1 is selected (and removed).
- Nodes that extend p_1 are added to the front of the stack (in front of p_2).
- p_2 is only selected when all nodes from p_1 have been explored.

ILLUSTRATIVE GRAPH: DEPTH-FIRST SEARCH



COMPLEXITY OF DEPTH-FIRST SEARCH

Does DFS guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

BREADTH-FIRST SEARCH

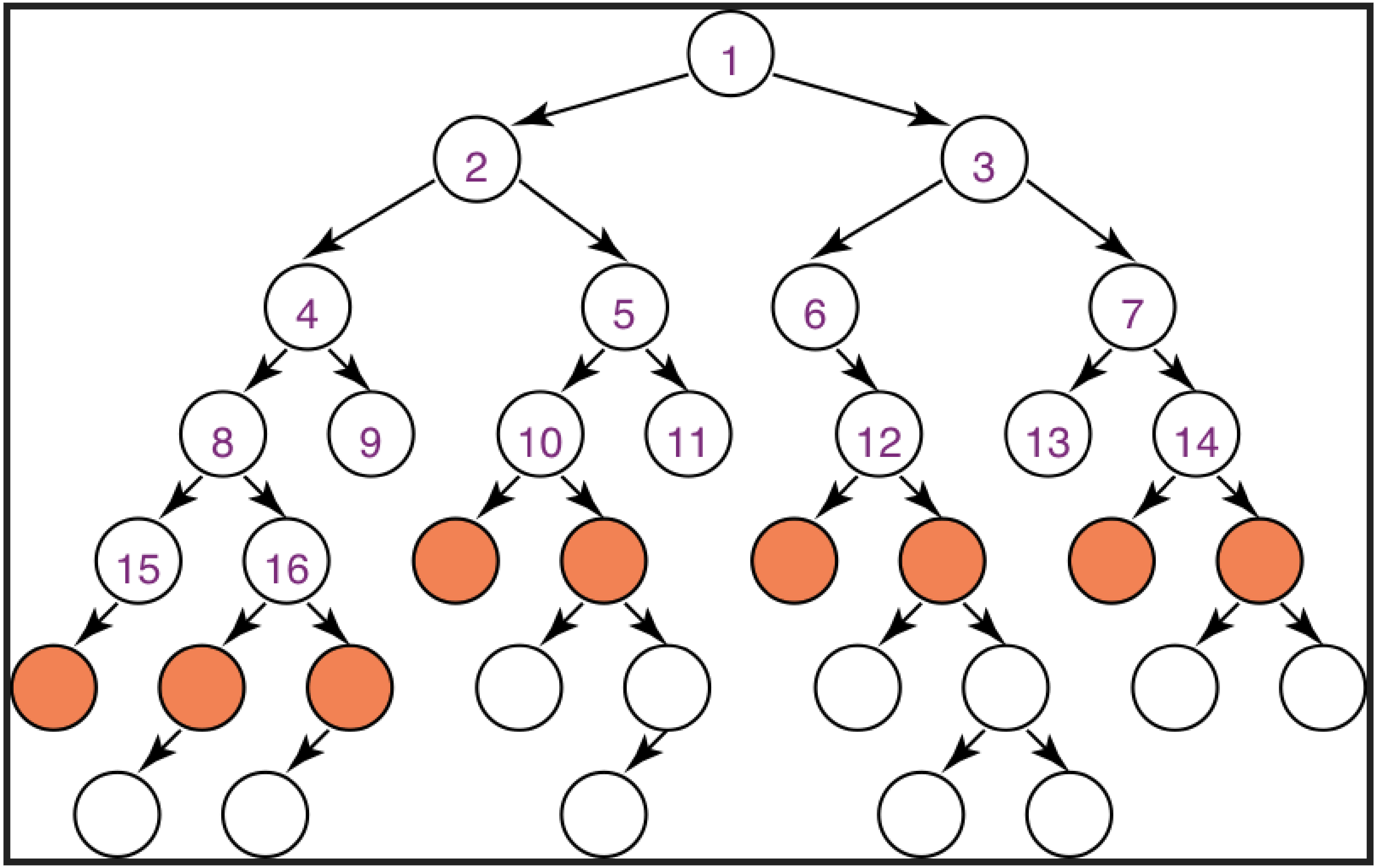
Breadth-first search treats the frontier as a queue.

It always selects one of the earliest elements added to the frontier.

If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$, then:

- p_1 is selected (and removed).
- Its neighbors are added to the end of the queue, after p_r.
- p_2 is selected next.

ILLUSTRATIVE GRAPH: BREADTH-FIRST SEARCH



COMPLEXITY OF BREADTH-FIRST SEARCH

Does BFS guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

UNIFORM-COST SEARCH

Weighted graphs:

- Sometimes there are *costs* associated with arcs.
  The cost of a path is the sum of the costs of its arcs.

$$cost(n_0, \ldots, n_k) = \sum_{i=1}^{k} |(n_{i-1}, n_i)|$$

  An *optimal solution* is one with minimum cost.

Uniform-cost search:

- Uniform-cost search selects a path on the frontier with the lowest cost.
- The frontier is a *priority queue* ordered by path cost.
- It finds a least-cost path to a goal node — i.e., uniform-cost search is optimal
- When arc costs are equal $\Rightarrow$ breadth-first search.

HEURISTIC SEARCH (R&N 3.5–3.6)

GREEDY BEST-FIRST SEARCH

A* SEARCH

ADMISSIBLE AND CONSISTENT HEURISTICS

HEURISTIC SEARCH

Previous methods don’t use the goal to select a path to explore.

Main idea: don’t ignore the goal when selecting paths.

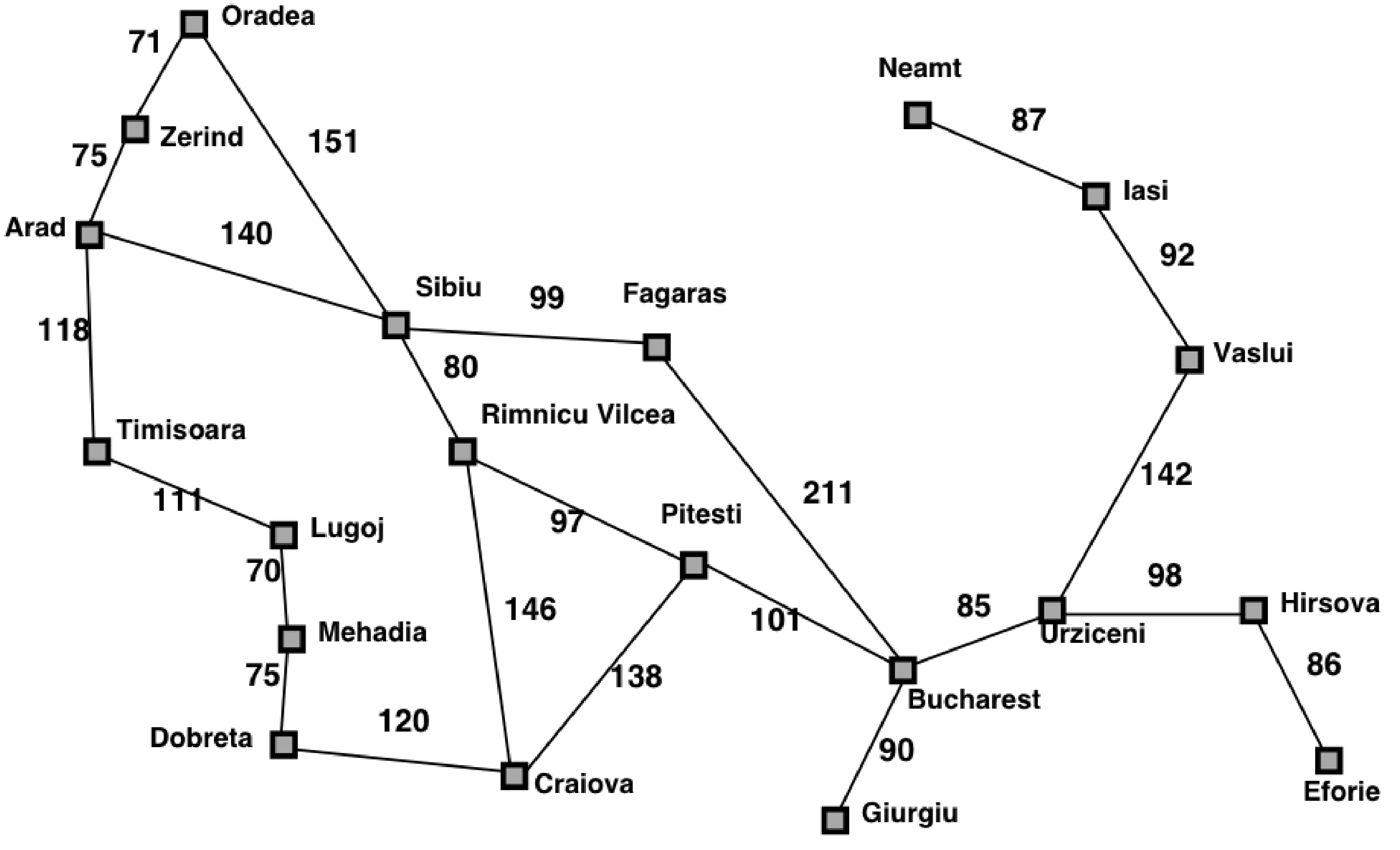
- Often there is extra knowledge that can guide the search: ***heuristics***.
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- $h(n)$ is an *underestimate* if there is no path from n to a goal with cost less than $h(n)$.
- An *admissible heuristic* is a nonnegative heuristic function that is an underestimate of the actual cost of a path to a goal.

EXAMPLE HEURISTIC FUNCTIONS

Here are some example heuristic functions:

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance (SLD) from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed, $h(n) = d(n)/v_{\max}$.
- If the goal is to collect all of the coins and not run out of fuel, we can use an estimate of how many steps it will take to collect the coins and return to goal position, without caring about the fuel consumption.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

EXAMPLE HEURISTIC: ROMANIA DISTANCES



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

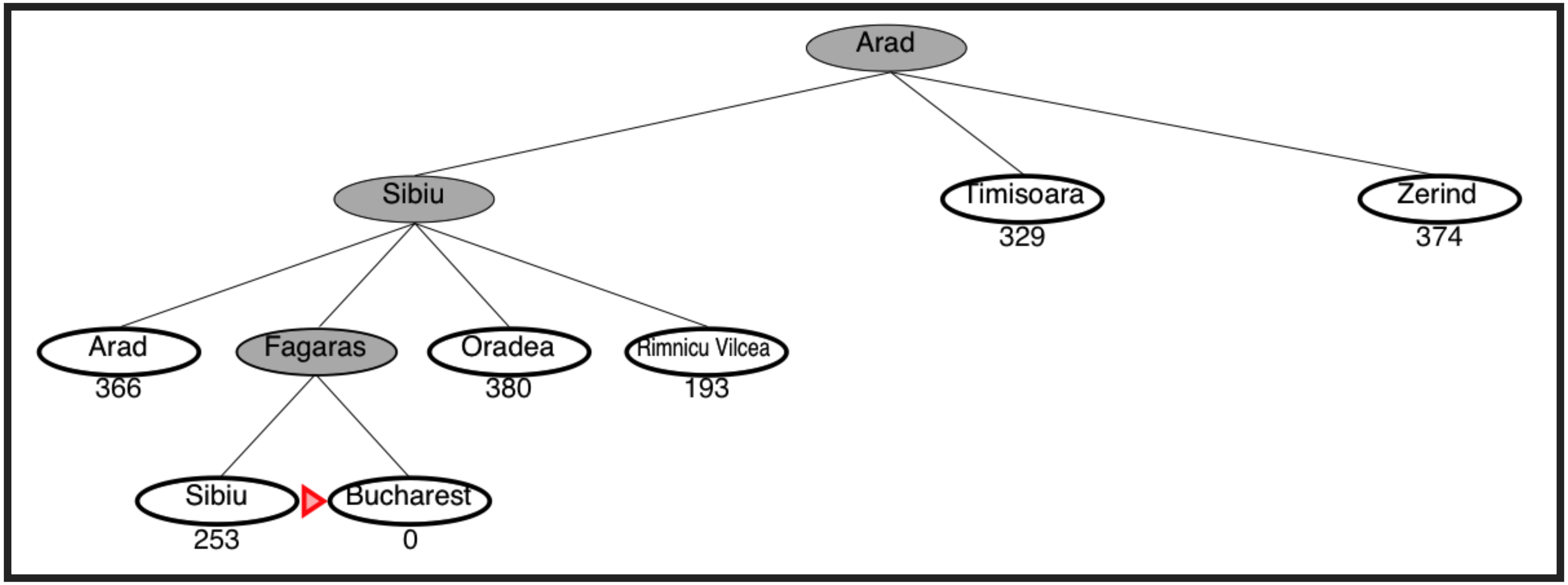
GREEDY BEST-FIRST SEARCH

Main idea: select the path whose end is closest to a goal according to the heuristic function.

Best-first search selects a path on the frontier with minimal h-value.

It treats the frontier as a priority queue ordered by h.

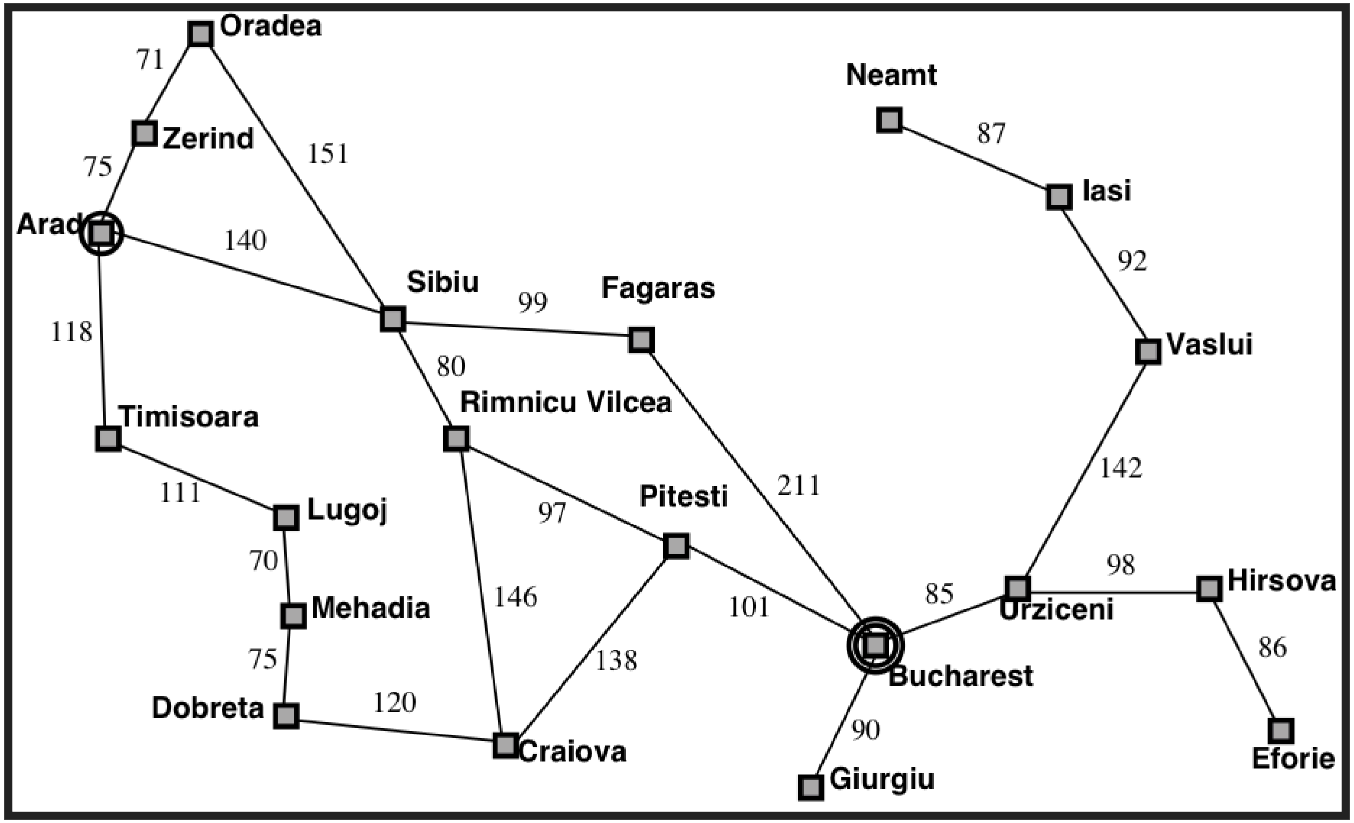
GREEDY SEARCH EXAMPLE: ROMANIA



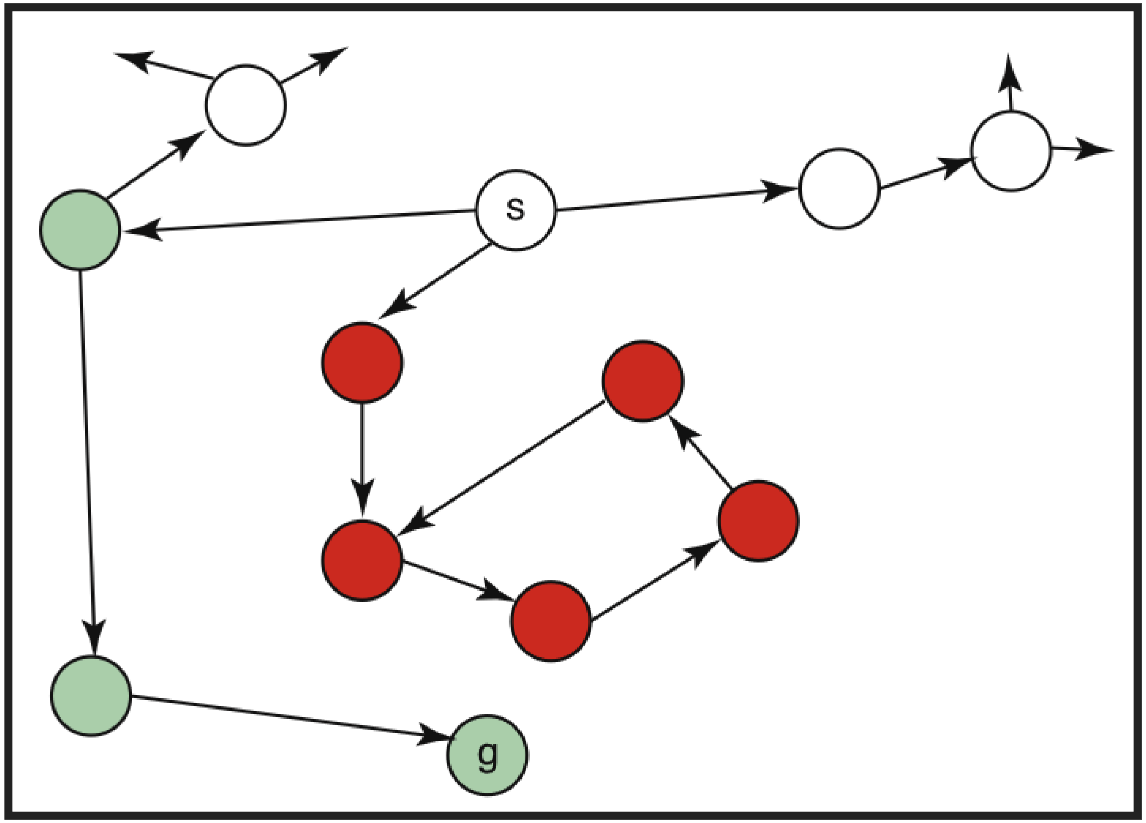
This is not the shortest path!

GREEDY SEARCH IS NOT OPTIMAL

Greedy search returns the path: *Arad–Sibiu–Fagaras–Bucharest* (450km)
The optimal path is: *Arad–Sibiu–Rimnicu–Pitesti–Bucharest* (418km)



BEST-FIRST SEARCH AND INFINITE LOOPS



Best-first search might fall into an infinite loop!

COMPLEXITY OF BEST-FIRST SEARCH

Does best-first search guarantee to find the path with fewest arcs?

What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

A* SEARCH

A* search uses both path cost and heuristic values.

$cost(p)$ is the cost of path p.

$h(p)$ estimates the cost from the end node of p to a goal.

$f(p) = cost(p) + h(p)$, estimates the total path cost of going from the start node, via path p to a goal:

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p}}_{cost(p)} \; \underbrace{n \xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

A* SEARCH

A* is a mix of lowest-cost-first and best-first search.

It treats the frontier as a priority queue ordered by $f(p)$.

It always selects the node on the frontier with
the lowest estimated distance from the start
to a goal node constrained to go via that node.

COMPLEXITY OF A* SEARCH

Does A* search guarantee to find the path with fewest arcs?

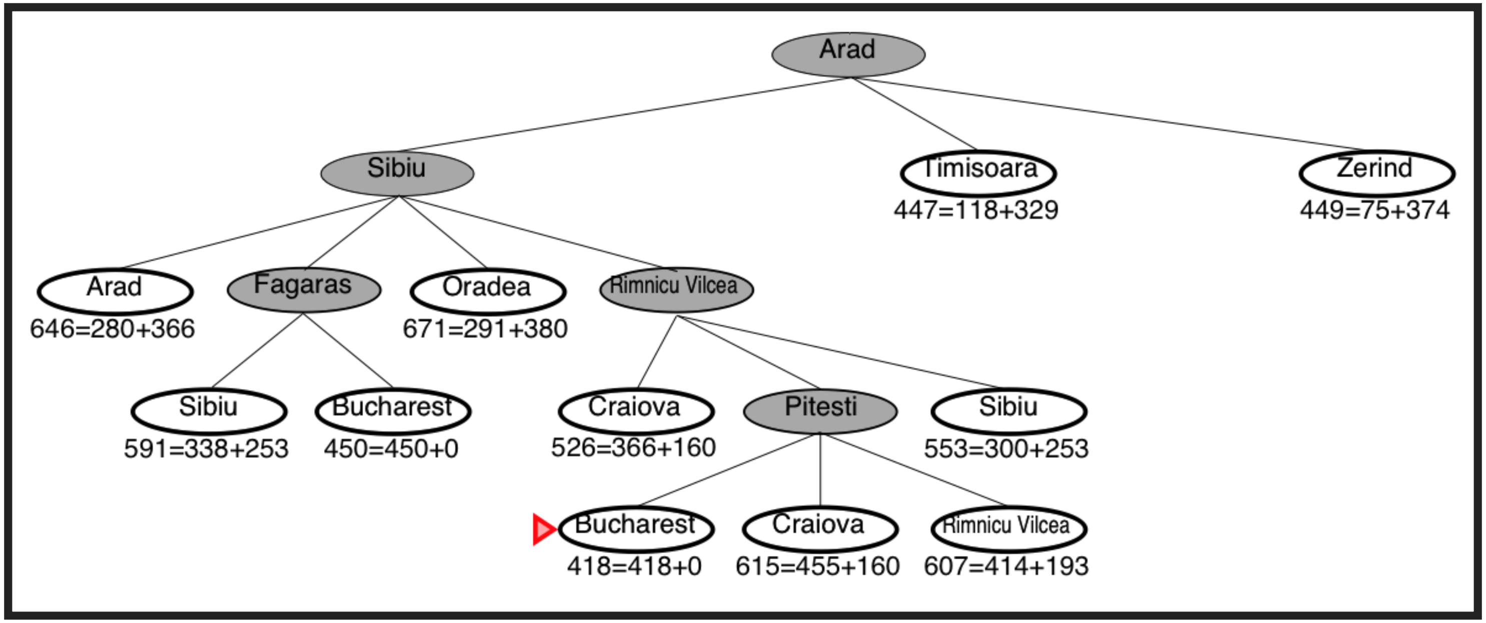
What happens on infinite graphs or on graphs with cycles if there is a solution?

What is the time complexity as a function of the path length?

What is the space complexity as a function of the path length?

How does the goal affect the search?

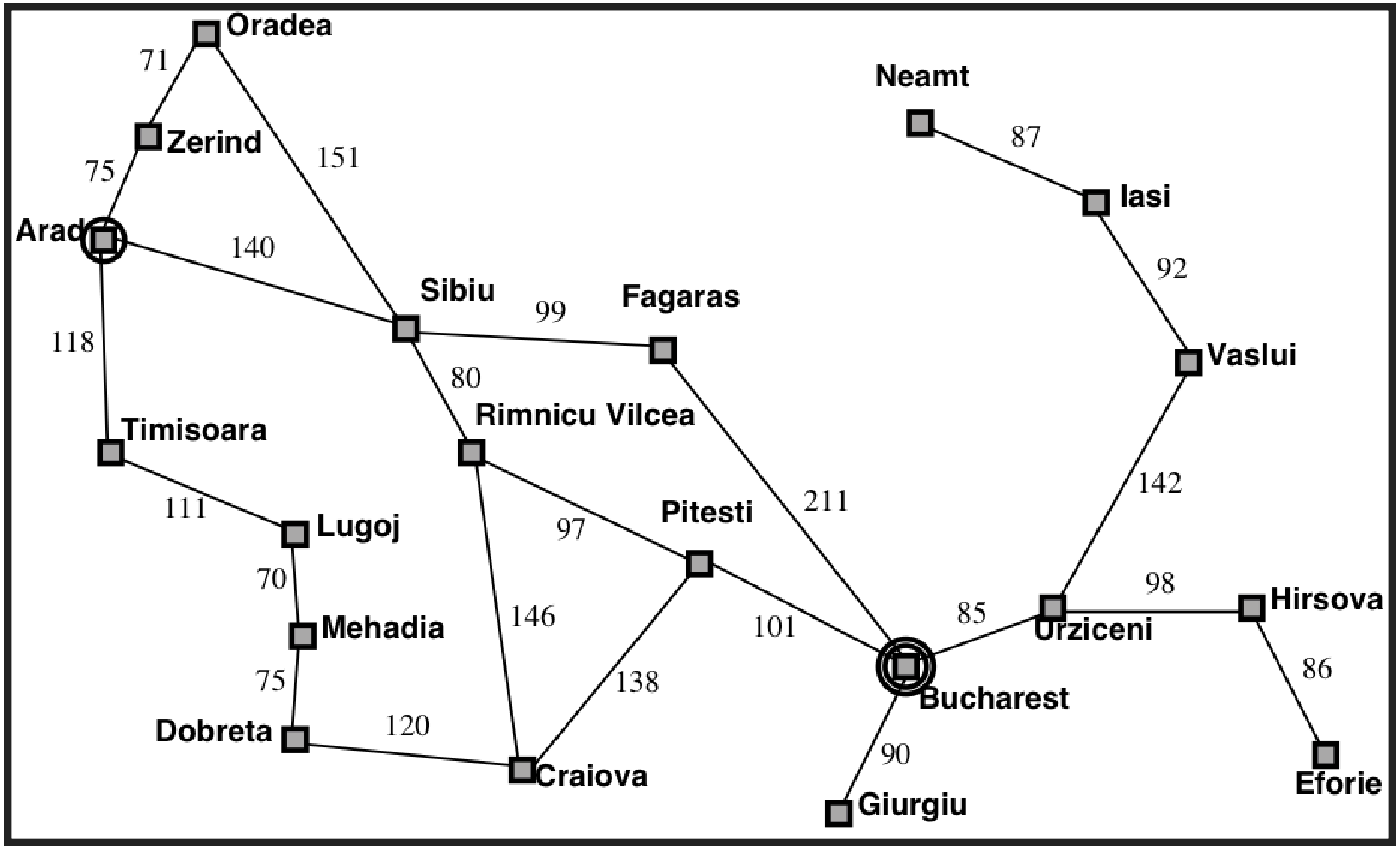
A* SEARCH EXAMPLE: ROMANIA



A guarantees that this is the shortest path!*

A* SEARCH IS OPTIMAL

The optimal path is: *Arad–Sibiu–Rimnicu–Pitesti–Bucharest* (418km)



A* ALWAYS FINDS A SOLUTION

A* will always find a solution if there is one, because:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A* halts, because the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

ADMISSIBILITY (OPTIMALITY) OF A*

If there is a solution, A* always finds an optimal one first, provided that:

- the branching factor is finite,
- arc costs are bounded above zero (i.e., there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is nonnegative and an underestimate of the cost of the shortest path from n to a goal node.

A* FINDS AN OPTIMAL SOLUTION FIRST

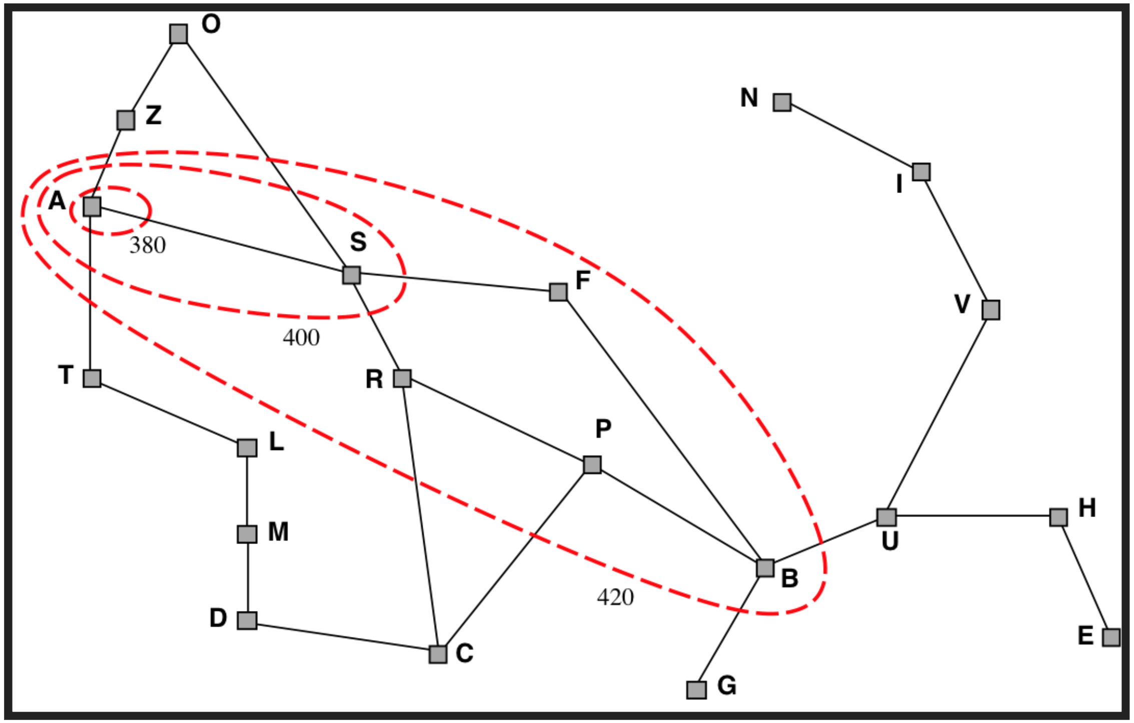
The first path that A* finds to a goal is an optimal path, because:

- The f-value for any node on an optimal solution path
  is less than or equal to the f-value of an optimal solution.
  This is because h is an underestimate of the actual cost
- Thus, the f-value of a node on an optimal solution path
  is less than the f-value for any non-optimal solution.
- Thus, a non-optimal solution can never be chosen while
  a node exists on the frontier that leads to an optimal solution.
  Because an element with minimum f-value is chosen at each step
- So, before it can select a non-optimal solution, it will have to pick
  all of the nodes on an optimal path, including each of the optimal solutions.

ILLUSTRATION: WHY IS A* ADMISSIBLE?

A* gradually adds “f-contours” of nodes (cf. BFS adds layers).

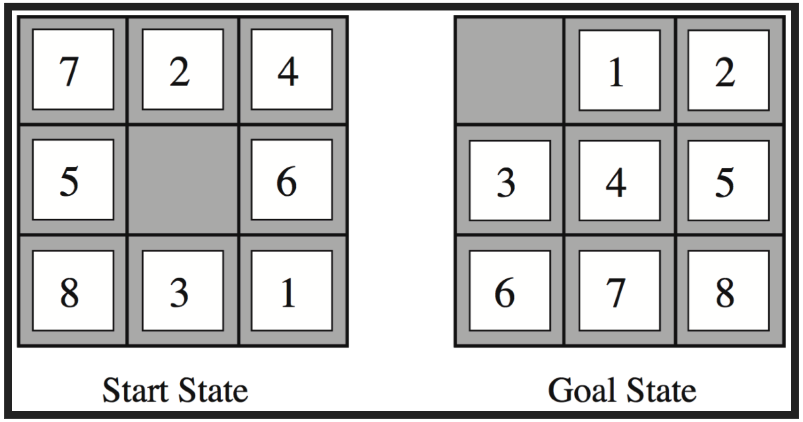
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$.



QUESTION TIME: HEURISTICS FOR THE 8 PUZZLE

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



$h_1(StartState)$ = *8*

$h_2(StartState)$ = *3+1+2+2+2+3+3+2 = 18*

DOMINATING HEURISTICS

If (admissible) $h_2(n) \geq h_1(n)$ for all n,
then h_2 **dominates** h_1 and is better for search.

Typical search costs (for 8-puzzle):

depth = 14	DFS ≈ 3,000,000 nodes
	A*(h_1) = 539 nodes
	A*(h_2) = 113 nodes
depth = 24	DFS ≈ 54,000,000,000 nodes
	A*(h_1) = 39,135 nodes
	A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b, the **maximum** heuristics $h(n)$ is also admissible and dominates both:

$$h(n) = \max(h_a(n), h_b(n))$$

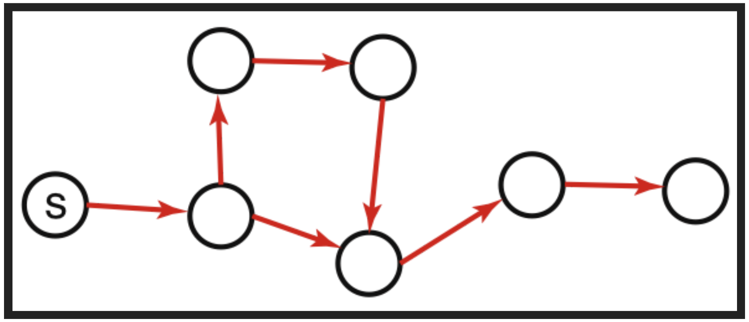
HEURISTICS FROM A RELAXED PROBLEM

Admissible heuristics can be derived from the exact solution cost of a relaxed problem:

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is never greater than the optimal solution cost of the real problem

GRAPH-SEARCH = MULTIPLE-PATH PRUNING



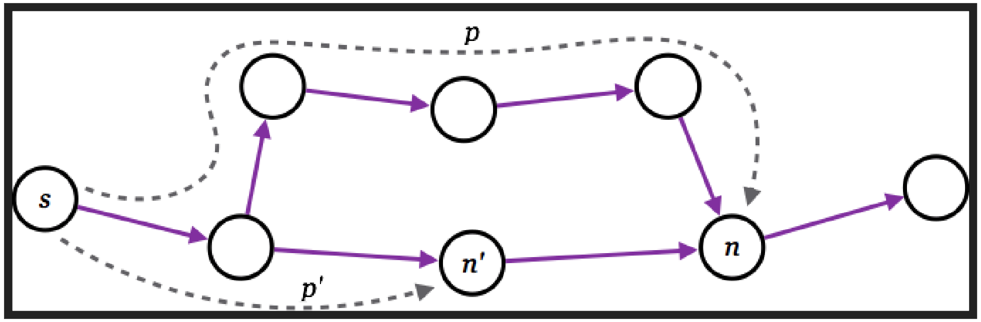
Graph search keeps track of visited nodes, so we don't visit the same node twice.

- Suppose that the first time we visit a node is not via the most optimal path

  ⇒ then graph search will return a suboptimal path

- Under which circumstances can we guarantee that A* graph search is optimal?

WHEN IS A* GRAPH SEARCH OPTIMAL?



Suppose path p to n was selected, but there is a shorter path p' to n.

Suppose path p' ends at node n'.

p was selected before p', which means that: $cost(p) + h(n) \leq cost(p') + h(n')$.

Suppose $cost(n', n)$ is the actual cost of a path from n' to n.
The path to n via p' is shorter than p, i.e.: $cost(p') + cost(n', n) < cost(p)$.

Combining the two: $cost(n', n) < cost(p) - cost(p') \leq h(n') - h(n)$

So, the problem won't occur if $|h(n') - h(n)| \leq cost(n', n)$.

CONSISTENCY, OR MONOTONICITY

A heuristic function h is **consistent** (or monotone) if $|h(m) - h(n)| \leq cost(m, n)$ for every arc (m, n).

- (This is a form of triangle inequality)
- If h is consistent, then A* graph search will always finds the shortest path to a goal.
- This is a strengthening of admissibility.

SUMMARY OF OPTIMALITY OF A*

A* *tree search* is optimal if:

- the heuristic function $h(n)$ is **admissible**
- i.e., $h(n)$ is nonnegative and an underestimate of the actual cost
- i.e., $h(n) \leq cost(n, goal)$, for all nodes n

A* *graph search* is optimal if:

- the heuristic function $h(n)$ is **consistent**
- i.e., $|h(m) - h(n)| \leq cost(m, n)$, for all arcs (m, n)

SUMMARY OF TREE SEARCH STRATEGIES

Search strategy	**Frontier selection**	**Halts if solution?**	**Halts if no solution?**	**Space usage**
Depth first	Last node added	*No*	*No*	*Linear*
Breadth first	First node added	*Yes*	*No*	*Exp*
Best first	Global min $h(p)$	*No*	*No*	*Exp*
Lowest cost first	Minimal $cost(p)$	*Yes*	*No*	*Exp*
A*	Minimal $f(p)$	*Yes*	*No*	*Exp*

Halts if: If there is a path to a goal, it can find one, even on infinite graphs.
Halts if no: Even if there is no solution, it will halt on a finite graph (with cycles).
Space: Space complexity as a function of the length of the current path.

EXAMPLE DEMO

Here is an example demo of several different search algorithms, including A*. Furthermore you can play with different heuristics:

http://qiao.github.io/PathFinding.js/visual/

Note that this demo is tailor-made for planar grids, which is a special case of all possible search graphs.

CHAPTERS 3–4: MORE SEARCH ALGORITHMS

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

28 March, 2017

TABLE OF CONTENTS

HEURISTIC SEARCH (R&N 3.5–3.6)

GREEDY BEST-FIRST SEARCH (3.5.1)

A* SEARCH (3.5.2)

ADMISSIBLE AND CONSISTENT HEURISTICS (3.6–3.6.2)

THE GENERIC TREE SEARCH ALGORITHM

Tree search: Don’t check if nodes are visited multiple times

```
function Search(graph, initialState, goalState):
    initialise frontier using the initialState
    while frontier is not empty:
        select and remove node from frontier
        if node.state is a goalState then return node
        for each child in ExpandChildNodes(node, graph):
            add child to frontier
    return failure
```

DEPTH-FIRST AND BREADTH-FIRST SEARCH

THESE ARE THE TWO BASIC SEARCH ALGORITHMS

Depth-first search (DFS)

- implement the frontier as a Stack
- space complexity: $O(bm)$
- incomplete: might fall into an infinite loop, doesn’t return optimal solution

Breadth-first search (BFS)

- implement the frontier as a Queue
- space complexity: $O(b^m)$
- complete: always finds a solution, if there is one
- (when edge costs are constant, BFS is also optimal)

COST-BASED SEARCH

IMPLEMENT THE FRONTIER AS A PRIORITY QUEUE, ORDERED BY $\mathbf{f(n)}$

Uniform-cost search (this is not a heuristic algorithm)

- expand the node with the lowest path cost
- $f(n) = g(n)$
- complete and optimal

Greedy best-first search

- expand the node which is closest to the goal (according to some heuristics)
- $f(n) = h(n)$
- incomplete: might fall into an infinite loop, doesn’t return optimal solution

A* search

- expand the node which has the lowest estimated cost from start to goal
- $f(n) = g(n) + h(n)$ = estimated cost of the cheapest solution through n
- complete and optimal (if $h(n)$ is admissible/consistent)

A* TREE SEARCH IS OPTIMAL!

A* always finds an optimal solution first, provided that:

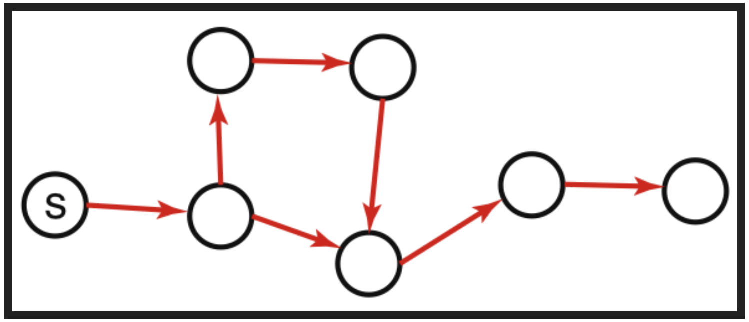
- the branching factor is finite,
- arc costs are *bounded above zero*
(i.e., there is some $\epsilon > 0$ such that all
of the arc costs are greater than ϵ), and
- $h(n)$ is ***admissible***
- i.e., $h(n)$ is *nonnegative* and an *underestimate* of
the cost of the shortest path from n to a goal node.

THE GENERIC GRAPH SEARCH ALGORITHM

Tree search: Don’t check if nodes are visited multiple times
Graph search: Keep track of visited nodes

```
function Search(graph, initialState, goalState):
    initialise frontier using the initialState
    initialise exploredSet to the empty set
    while frontier is not empty:
        select and remove node from frontier
        if node.state is a goalState then return node
        add node to exploredSet
        for each child in ExpandChildNodes(node, graph):
            if child is not in frontier or exploredSet:
                add child to frontier
    return failure
```

GRAPH-SEARCH = MULTIPLE-PATH PRUNING



Graph search keeps track of visited nodes, so we don’t visit the same node twice.

- Suppose that the first time we visit a node is not via the most optimal path

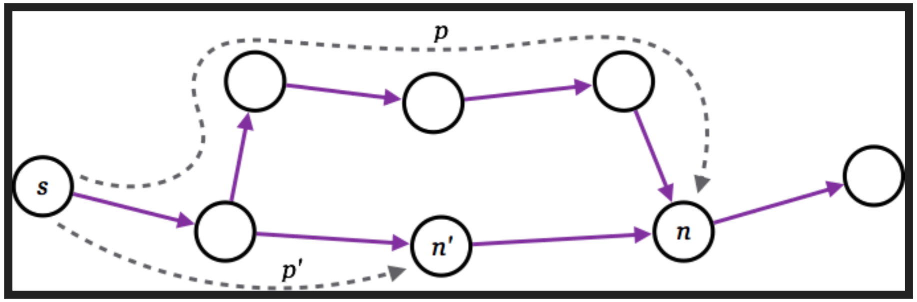
  ⇒ then graph search will return a suboptimal path

- Under which circumstances can we guarantee that A* graph search is optimal?

WHEN IS A* GRAPH SEARCH OPTIMAL?

If h is *consistent*, then A* graph search is optimal:

- Consistency is defined as: $h(n') \leq cost(n', n) + h(n)$ for all arcs (n', n)
- **Lemma**: the f values along any path $[\ldots, n', n, \ldots]$ are nondecreasing:
  - **Proof**: $g(n) = g(n') + cost(n', n)$, therefore:
  - $f(n) = g(n) + h(n) = g(n') + cost(n', n) + h(n) \geq g(n') + h(n')$;
  - therefore: $f(n) \geq f(n')$, i.e., f is nondecreasing
- **Theorem**: whenever A* expands a node n, the optimal path to n has been found
  - **Proof**: Assume this is not true;
  - then there must be some n' still on the frontier, which is on the optimal path to n;
  - but $f(n') \leq f(n)$;
  - and then n' must already have been expanded $\Longrightarrow$ *contradiction*!



STATE-SPACE CONTOURS

The f values in A* are nondecreasing, therefore:

first	A* expands all nodes with $f(n) < C$
then	A* expands all nodes with $f(n) = C$
finally	A* expands all nodes with $f(n) > C$

A* will not expand any nodes with $f(n) > C*$,
where $C*$ is the cost of an optimal solution.

SUMMARY OF OPTIMALITY OF A*

A* *tree search* is optimal if:

- the heuristic function $h(n)$ is **admissible**
- i.e., $h(n)$ is nonnegative and an underestimate of the actual cost
- i.e., $h(n) \leq cost(n, goal)$, for all nodes n

A* *graph search* is optimal if:

- the heuristic function $h(n)$ is **consistent** (or monotone)
- i.e., $|h(m) - h(n)| \leq cost(m, n)$, for all arcs (m, n)

SUMMARY OF TREE SEARCH STRATEGIES

Search strategy	Frontier selection	Halts if solution?	Halts if no solution?	Space usage
Depth first	Last node added	*No*	*No*	*Linear*
Breadth first	First node added	*Yes*	*No*	*Exp*
Greedy best first	Minimal $h(n)$	*No*	*No*	*Exp*
Uniform cost	Minimal $g(n)$	*Optimal*	*No*	*Exp*
A*	$f(n) = g(n) + h(n)$	*Optimal**	*No*	*Exp*

**Provided that $h(n)$ is admissible.*

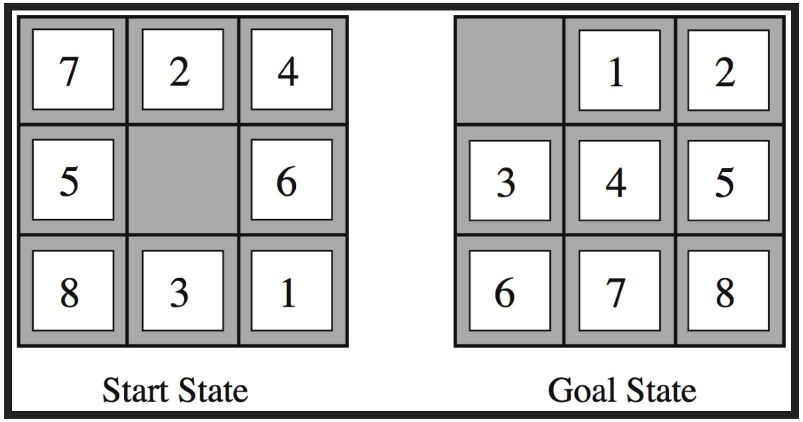
Halts if: If there is a path to a goal, it can find one, even on infinite graphs.
Halts if no: Even if there is no solution, it will halt on a finite graph (with cycles).
Space: Space complexity as a function of the length of the current path.

RECAPITULATION: HEURISTICS FOR THE 8 PUZZLE

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



$$h_1(StartState) = 8$$

$$h_2(StartState) = 3+1+2+2+2+3+3+2 = 18$$

DOMINATING HEURISTICS

If (admissible) $h_2(n) \geq h_1(n)$ for all n,
then h_2 **dominates** h_1 and is better for search.

Typical search costs (for 8-puzzle):

depth = 14	DFS ≈ 3,000,000 nodes
	A*(h_1) = 539 nodes
	A*(h_2) = 113 nodes
depth = 24	DFS ≈ 54,000,000,000 nodes
	A*(h_1) = 39,135 nodes
	A*(h_2) = 1,641 nodes

Given any admissible heuristics h_a, h_b, the **maximum** heuristics $h(n)$
is also admissible and dominates both:

$$h(n) = \max(h_a(n), h_b(n))$$

HEURISTICS FROM A RELAXED PROBLEM

Admissible heuristics can be derived from the exact solution cost of a relaxed problem:

- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is never greater than the optimal solution cost of the real problem

NON-ADMISSIBLE (NON-CONSISTENT) A* SEARCH

A* search with admissible (consistent) heuristics is optimal

But what happens if the heuristics is non-admissible?

- i.e., what if $h(n) > c(n, goal)$, for some n?
- the solution is not guaranteed to be optimal…
- …but it will find *some* solution!

Why would we want to use a non-admissible heuristics?

- sometimes it's easier to come up with a heuristics that is almost admissible
- and, often, the search terminates faster!

EXAMPLE DEMO

Here is an example demo of several different search algorithms, including A*. Furthermore you can play with different heuristics:

http://qiao.github.io/PathFinding.js/visual/

Note that this demo is tailor-made for planar grids, which is a special case of all possible search graphs.

MORE SEARCH STRATEGIES (R&N 3.4–3.5)

ITERATIVE DEEPENING (3.4.4–3.4.5)

BIDIRECTIONAL SEARCH (3.4.6)

MEMORY-BOUNDED HEURISTIC SEARCH (3.5.3)

ITERATIVE DEEPENING

BFS is guaranteed to halt but uses exponential space.
DFS uses linear space, but is not guaranteed to halt.

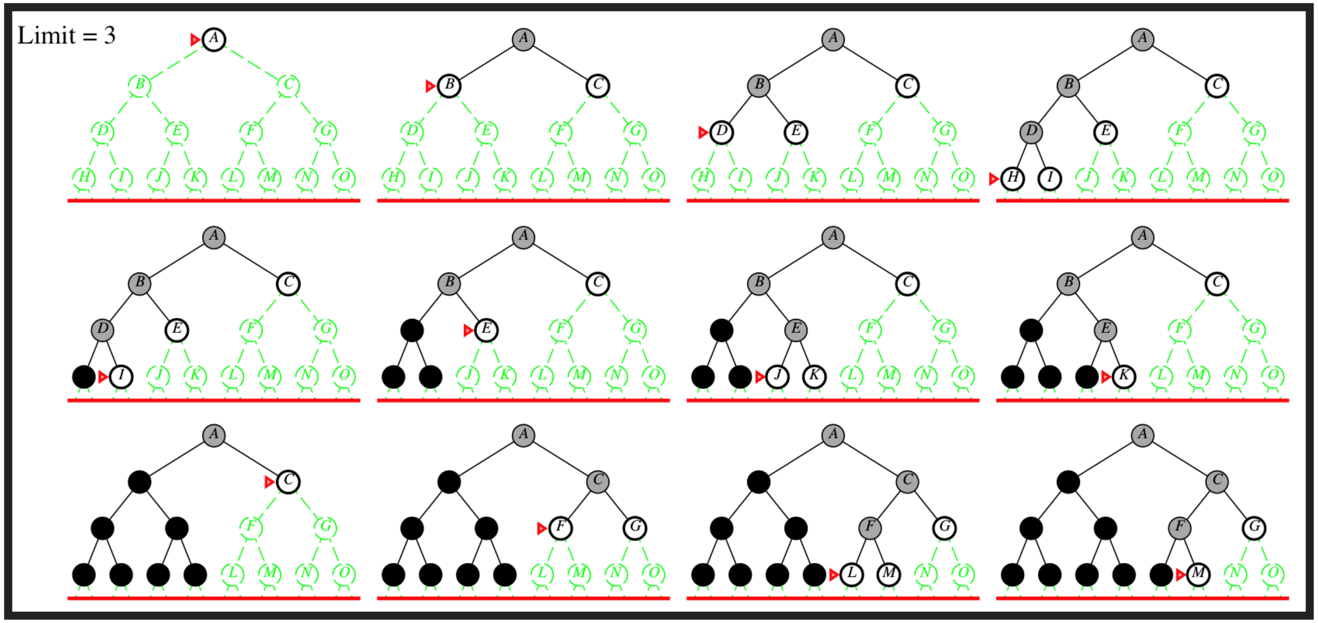
Idea: take the best from BFS and DFS — recompute elements of the frontier rather than saving them.

- Look for paths of depth 0, then 1, then 2, then 3, etc.
- Depth-bounded DFS can do this in linear space.

Iterative deepening search calls depth-bounded DFS with increasing bounds:

- If a path cannot be found at *depth-bound*, look for a path at *depth-bound* + 1.
- Increase *depth-bound* when the search fails unnaturally
  (i.e., if *depth-bound* was reached).

ITERATIVE DEEPENING EXAMPLE



Depth bound = *3*

ITERATIVE-DEEPENING SEARCH

```
function IDSearch(graph, initialState, goalState)
    for limit in 0, 1, 2, …:
        result := DepthLimitedSearch([initialState], limit)
        if result ≠ cutoff then return result

function DepthLimitedSearch([$n_0, \ldots, n_k$], limit):
    if $n_k$ is a goalState then return path [$n_0, \ldots, n_k$]
    else if limit = 0 then return cutoff
    else:
        failureType := failure
        for each neighbor n of $n_k$:
            result := DepthLimitedSearch([$n_0, \ldots, n_k, n$], limit–1)
            if result is a path then return result
            else if result = cutoff then failureType := cutoff
        return failureType
```

ITERATIVE DEEPENING COMPLEXITY

Complexity with solution at depth k and branching factor b:

level	**breadth-first**	**iterative deepening**	**# nodes**
1	1	k	b
2	1	$k-1$	b^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k\left(\frac{b}{b-1}\right)^2$	

Numerical comparison for $k = 5$ and $b = 10$:

BFS = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110
IDS = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

Note: IDS recalculates shallow nodes several times,
but this doesn’t have a big effect compared to BFS!

BIDIRECTIONAL SEARCH (3.4.6)

DIRECTION OF SEARCH

The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.

Forward branching factor: number of arcs going out from a node.

Backward branching factor: number of arcs going into a node.

Search complexity is $O(b^n)$. Therefore, we should use forward search if forward branching factor is less than backward branching factor, and vice versa.

Note: when a graph is dynamically constructed, the backwards graph may not be available.

BIDIRECTIONAL SEARCH

Idea: search backward from the goal and forward from the start simultaneously.

- This can result in an exponential saving, because $2b^{k/2} \ll b^k$.
- The main problem is making sure the frontiers meet.

One possible implementation:

- Use BFS to gradually search backwards from the goal, building a set of locations that will lead to the goal.
    - this can be done using *dynamic programming*
- Interleave this with forward heuristic search (e.g., A*) that tries to find a path to these interesting locations.

DYNAMIC PROGRAMMING

Idea: for statically stored graphs, build a table of the actual distance $dist(n)$, of the shortest path from node n to a goal.

- This can be built backwards from the goal:

$$\begin{aligned} dist(n) \quad &= \quad \textbf{if } isGoal(n) \textbf{ then } 0 \\ & \qquad \textbf{else } \min_{(n,m)\in G}(|(n,m)| + dist(m)) \end{aligned}$$

The calculation of $dist$ can be interleaved with a forward heuristic search.

MEMORY-BOUNDED A* (3.5.3)

The biggest problem with A* is the space usage.
Can we make an iterative deepening version?

- IDA*: use the f value as the cutoff cost
    - the cutoff is the smalles f value that exceeded the previous cutoff
    - often useful for problems with unit step costs
    - **problem**: with real-valued costs, it risks regenerating too many nodes
- RBFS: recursive best-first search
    - similar to DFS, but continues along a path until $f(n) > limit$
    - $limit$ is the f value of the best *alternative path* from an ancestor
    - if $f(n) > limit$, recursion unwinds to alternative path
    - **problem**: regenerates too many nodes
- SMA* and MA*: (simplified) memory-bounded A*
    - uses all available memory
    - when memory is full, it drops the worst leaf node from the frontier

LOCAL SEARCH (R&N 4.1)

HILL CLIMBING (4.1.1–4.1.2)

POPULATION-BASED METHODS (4.1.3–4.1.4)

ITERATIVE BEST IMPROVEMENT

In many optimization problems, the path is irrelevant

- the goal state itself is the solution

Then the state space can be the set of “complete” configurations

- e.g., for 8-queens, a configuration can be any board with 8 queens (it is irrelevant in which order the queens are added)

In such cases, we can use *iterative improvement* algorithms;
we keep a single “current” state, and try to improve it

- e.g., for 8-queens, we start with 8 queens on the board, and gradually move some queen to a better place

The goal would be to find an optimal configuration

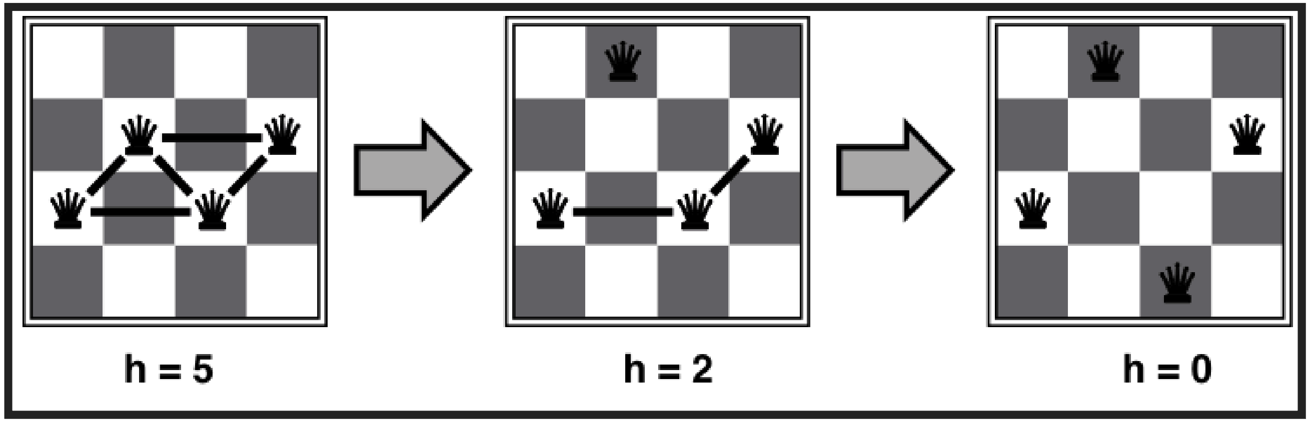
- e.g., for 8-queens, where no queen is threatened

This takes constant space, and is suitable for online and offline search

EXAMPLE: n-QUEENS

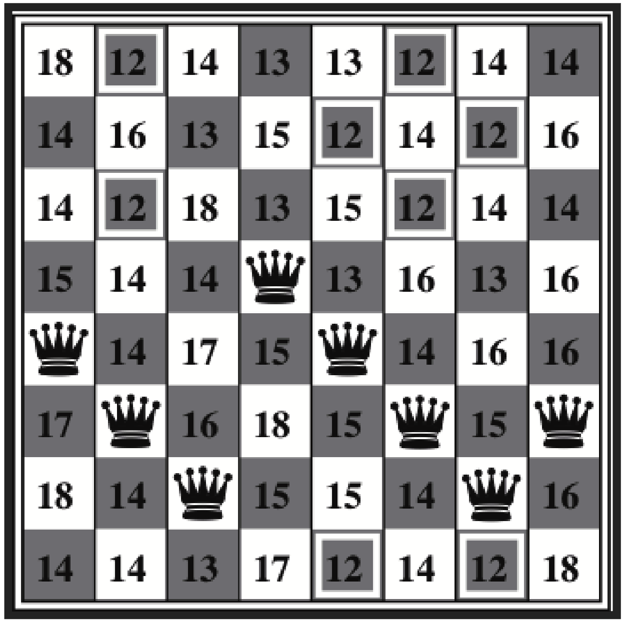
Put n queens on an $n \times n$ board, in separate columns

Move a queen to reduce the number of conflicts;
repeat until we cannot move any queen anymore
$\Rightarrow$ then we are at a local maximum, hopefully it is global too

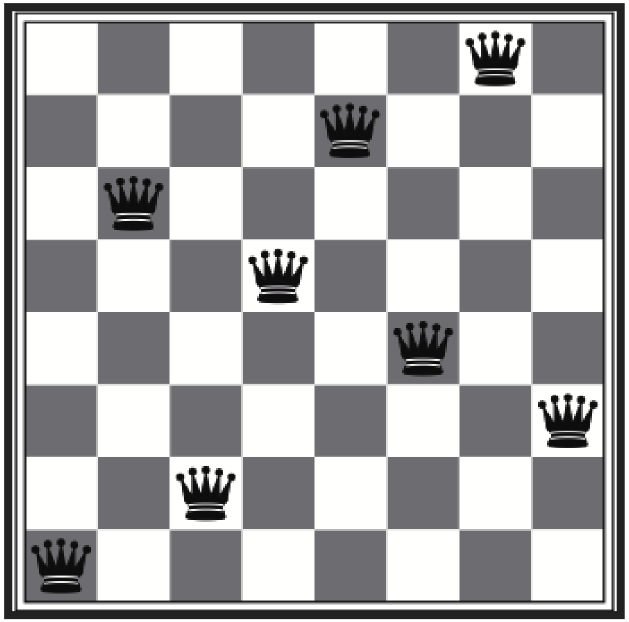


This almost always solves n-queens problems
almost instantaneously for very large n (e.g., n = 1 million)

EXAMPLE: 8-QUEENS



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

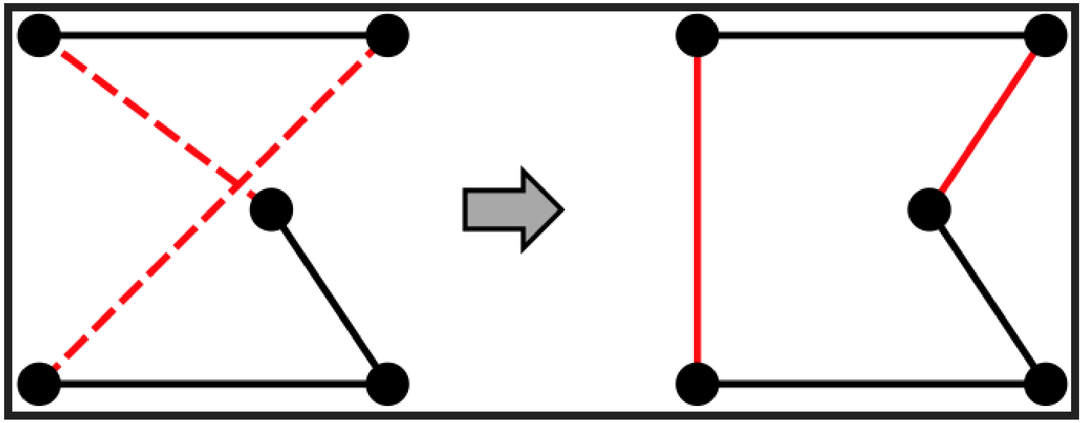


Move a queen within its column, choose the minimum n:o of conflicts

- the best moves are marked above (conflict value: 12)
- after 5 steps we reach a local minimum (conflict value: 1)

EXAMPLE: TRAVELLING SALESPERSON

Start with any complete tour, and perform pairwise exchanges



Variants of this approach get within 1% of optimal
very quickly with thousands of cities

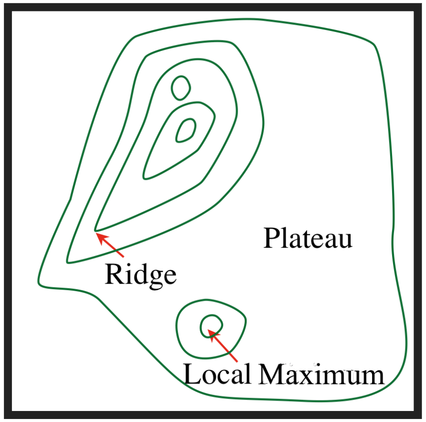
HILL CLIMBING SEARCH (4.1.1–4.1.2)

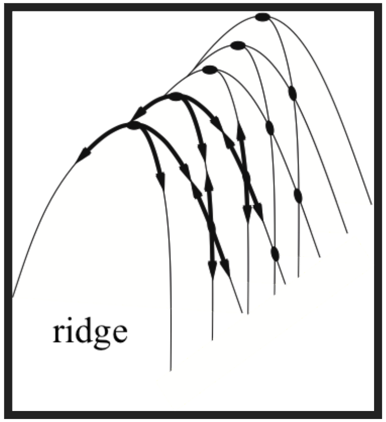
Also called gradient/steepest ascent/descent,
or greedy local search.

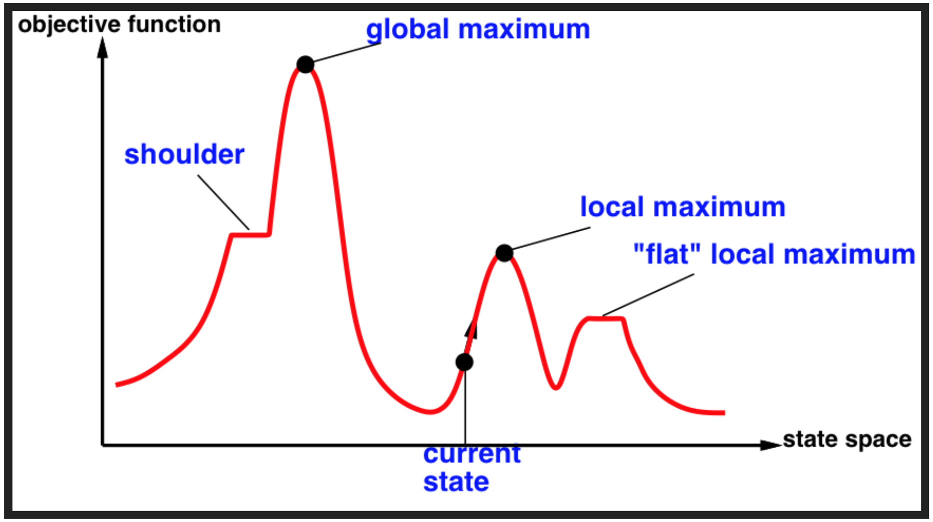
```
function HillClimbing(graph, initialState):
    current := initialState
    loop:
        neighbor := a highest-valued successor of current
        if neighbor.value ≤ current.value then return current
        current := neighbor
```

PROBLEMS WITH HILL CLIMBING

Local maxima — Ridges — Plateaux







RANDOMIZED ALGORITHMS

Consider two methods to find a minimum value:

- Greedy ascent: start from some position,
  keep moving upwards, and report maximum value found
- Pick values at random, and report maximum value found

Which do you expect to work better to find a global maximum?

Can a mix work better?

RANDOMIZED HILL CLIMBING

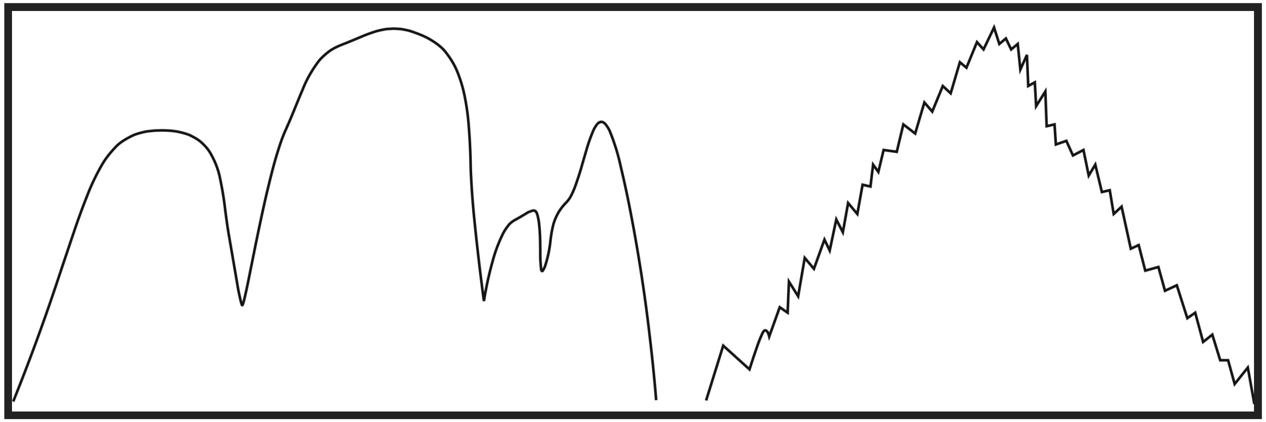
As well as upward steps we can allow for:

- *Random steps:* (sometimes) move to a random neighbor.
- *Random restart:* (sometimes) reassign random values to all variables.

Both variants can be combined!

1-DIMENSIONAL ILLUSTRATIVE EXAMPLE

Two 1-dimensional search spaces; you can step right or left:



Which method would most easily find the global maximum?

- random steps or random restarts?

What if we have hundreds or thousands of dimensions?

- …where different dimensions have different structure?

SIMULATED ANNEALING

Simulated annealing is an implementation of random steps:

```
function SimulatedAnnealing(problem, schedule):
    current := problem.initialState
    for t in 1, 2, …:
        T := schedule(t)
        if T = 0 then return current
        next := a randomly selected neighbor of current
        ΔE := next.value – current.value
        if ΔE > 0 or with probability $e^{\Delta E/T}$:
            current := next
```

T is the “cooling temperature”, which decreases slowly towards 0

The cooling speed is decided by the *schedule*

POPULATION-BASED METHODS (4.1.3–4.1.4)

LOCAL BEAM SEARCH

Idea: maintain a population of k states in parallel, instead of one.

- At every stage, choose the k best out of all of the neighbors.
    - when $k = 1$, it is normal hill climbing search
    - when $k = \infty$, it is breadth-first search
- The value of k lets us limit space and parallelism.
- *Note*: this is not the same as k searches run in parallel!
- *Problem*: quite often, all k states end up on the same local hill.

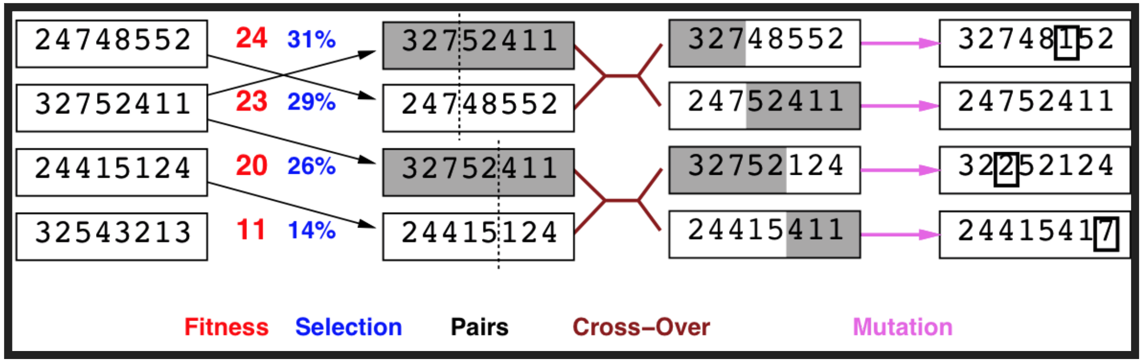
STOCHASTIC BEAM SEARCH

Similar to beam search, but it chooses the next k individuals *probabilistically*.

- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Similar to natural selection:
  each individual mutates and the fittest ones survive.

GENETIC ALGORITHMS

Similar to stochastic beam search,
but *pairs* of individuals are combined to create the offspring.



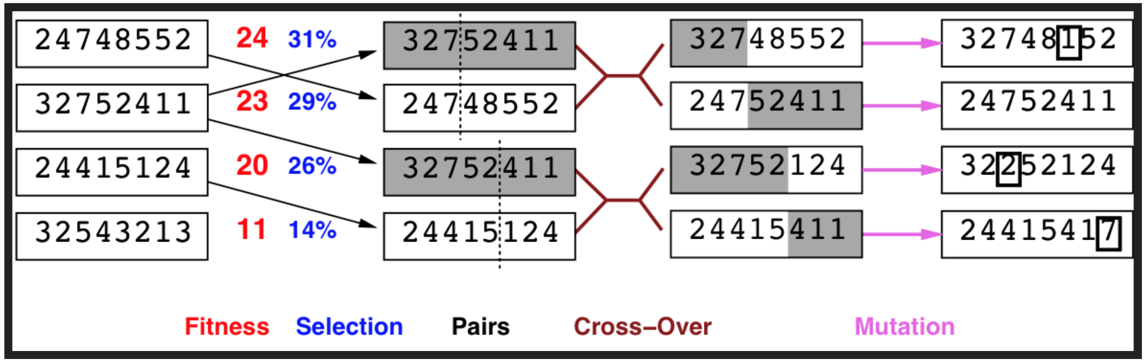
For each generation:

- Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
- For each pair, perform a cross-over: form two offspring each taking different parts of their parents:
- Mutate some values.

Stop when a solution is found.

n QUEENS ENCODED AS A GENETIC ALGORITHM

The n queens problem can be encoded as n numbers $1 \dots n$:

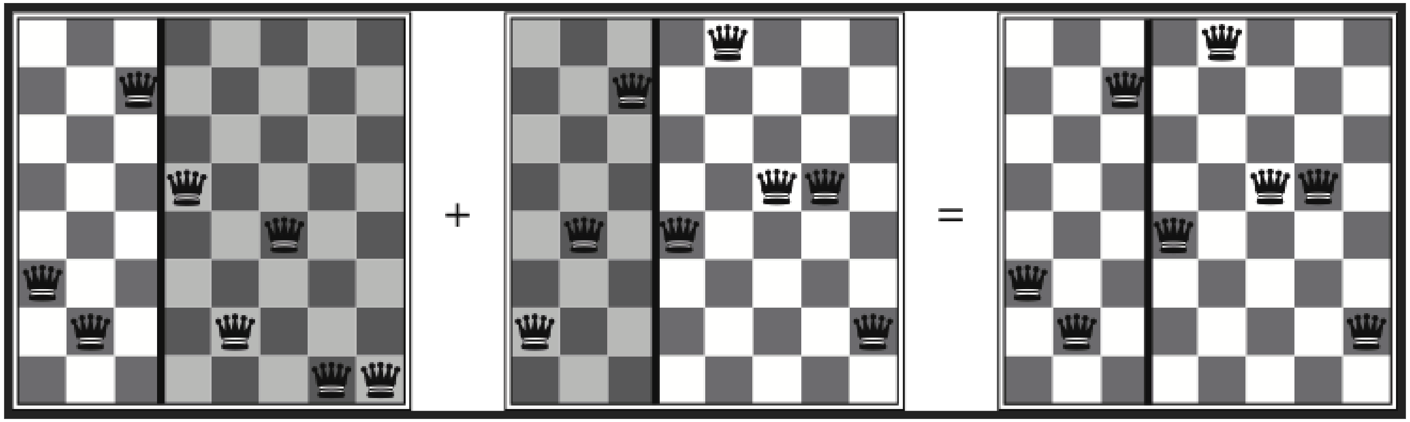

24748552 24 31%
32752411 23 29%
24415124 20 26%
32543213 11 14%

32752411
24748552
32752411
24415124

32748552
24752411
32752124
24415411

32748152
24752411
32252124
24415417

Fitness **Selection** **Pairs** **Cross–Over** **Mutation**



EVALUATING RANDOMIZED ALGORITHMS (NOT IN R&N)

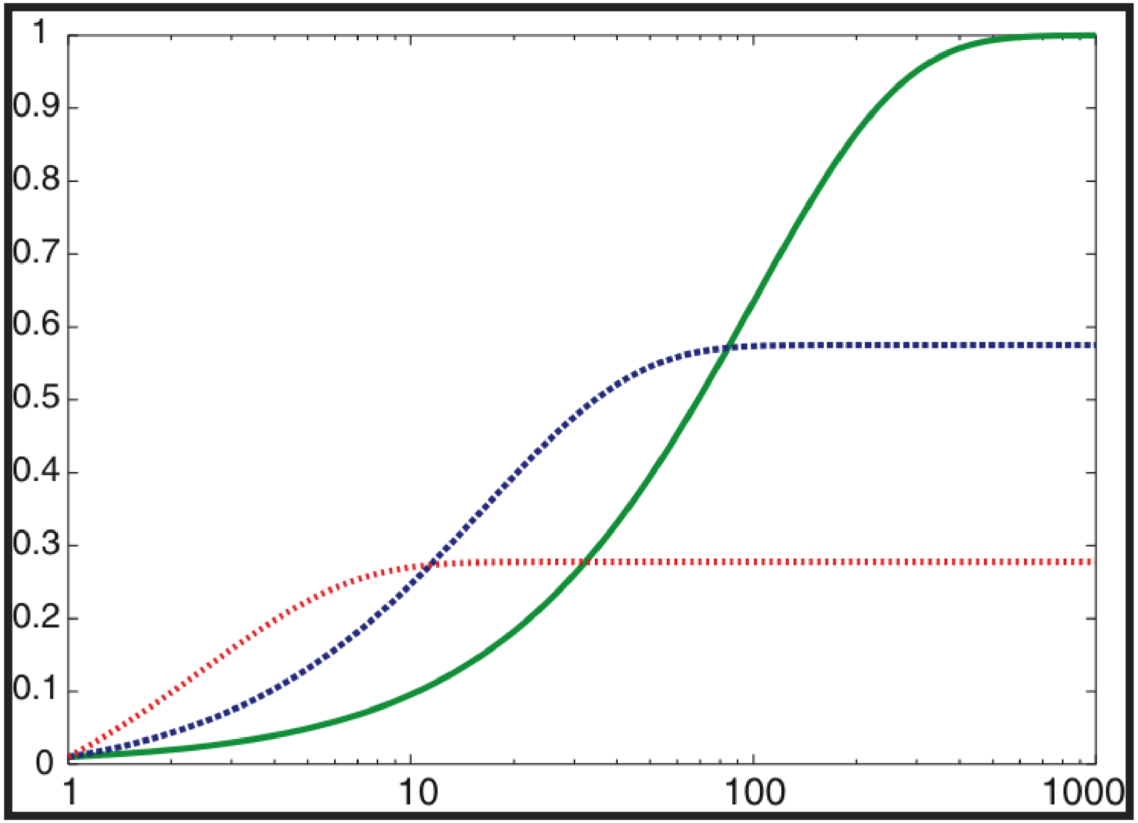
How can you compare three algorithms A, B and C, when

- A solves the problem 30% of the time very quickly but doesn’t halt for the other 70% of the cases
- B solves 60% of the cases reasonably quickly but doesn’t solve the rest
- C solves the problem in 100% of the cases, but slowly?

Summary statistics, such as mean run time or median run time
don’t make much sense.

RUNTIME DISTRIBUTION

Plots the runtime and the proportion of the runs that are solved within that runtime.



CHAPTER 6: CONSTRAINT SATISFACTION PROBLEMS

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

31 March, 2017

TABLE OF CONTENTS

CSP: CONSTRAINT SATISFACTION PROBLEMS (R&N 6.1)

FORMULATING A CSP

CONSTRAINT GRAPH

CONSTRAINT SATISFACTION PROBLEMS (CSP)

Standard search problem:

- the *state* is a “black box”,
  any data structure that supports: goal test, cost evaluation, successor

CSP is a more specific search problem:

- the *state* is defined by *variables* X_i, taking values from the domain $\mathbf{D}_i$
- the *goal test* is a set of *constraints* specifying allowable combinations of values for subsets of variables

Since CSP is more specific, it allows useful algorithms with more power than standard search algorithms

STATES AND VARIABLES

Just a few variables can describe many states:

n	binary variables can describe	2^n states
10	binary variables can describe	2^{10} = 1,024
20	binary variables can describe	2^{20} = 1,048,576
30	binary variables can describe	2^{30} = 1,073,741,824
100	binary variables can describe	2^{100} = 1,267,650,600,228,229, 401,496,703,205,376

HARD AND SOFT CONSTRAINTS

Given a set of variables, assign a value to each variable that either

- satisfies some set of constraints:
    - *satisfiability problems* — “hard constraints”
- or minimizes some cost function,
  where each assignment of values to variables has some cost:
    - *optimization problems* — “soft constraints” — “preferences”

Many problems are a mix of hard constraints and preferences
(constraint optimization problems)

RELATIONSHIP TO SEARCH

CSP differences to general search problems:

- The path to a goal isn't important, only the solution is.
- There are no predefined starting nodes.
- Often these problems are huge, with thousands of variables, so systematically searching the space is infeasible.
- For optimization problems, there are no well-defined goal nodes.

FORMULATING A CSP

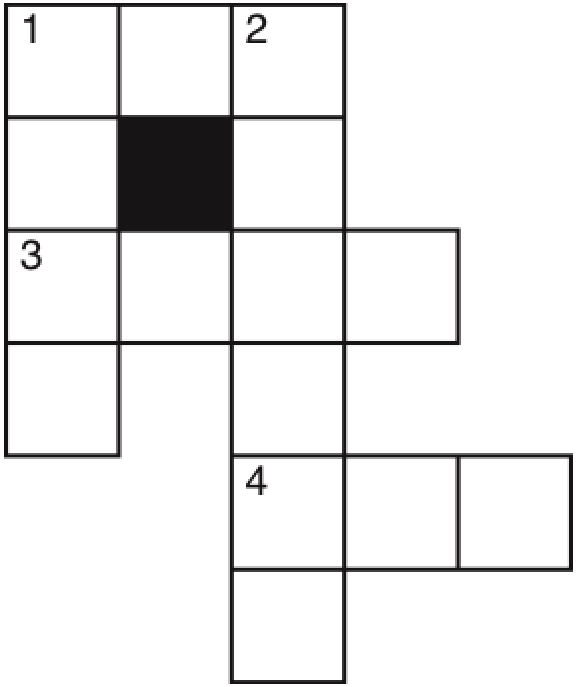
A CSP is characterized by

- A set of variables $X_1, X_2, \ldots, X_n$.
- Each variable X_i has an associated domain $\mathbf{D}_i$ of possible values.
- There are hard constraints $C_{X_i,\ldots,X_j}$ on various subsets of the variables which specify legal combinations of values for these variables.
- A solution to the CSP is an *assignment* of a value to each variable that satisfies all the constraints.

EXAMPLE: SCHEDULING ACTIVITIES

Variables:	A, B, C, D, E representing starting times of various activities. (e.g., courses and their study periods)
Domains:	$\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1, 2, 3, 4\}$
Constraints:	$(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D),$ $(E < A), (E < B), (E < C), (E < D), (B \neq D)$

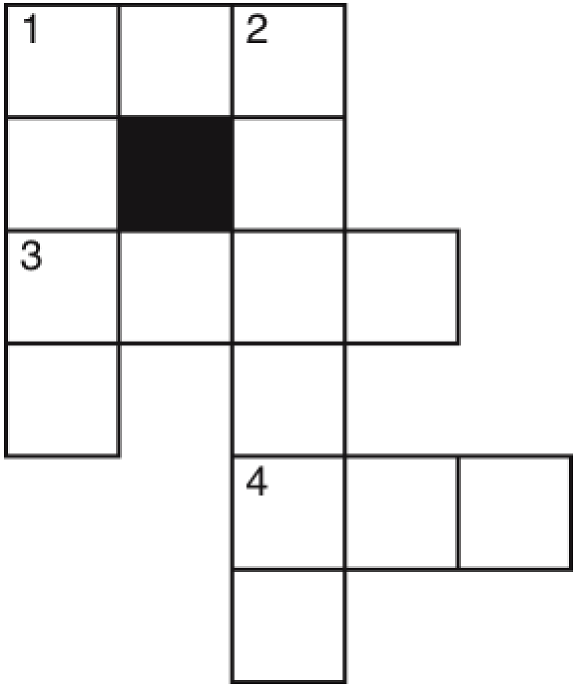
EXAMPLE: CROSSWORD PUZZLE



Words: ant, big, bus, car, has, book, buys, hold, lane, year, beast, ginger, search, symbol, syntax, …

DUAL REPRESENTATIONS

Many problems can be represented in different ways as a CSP, e.g., the crossword puzzle:



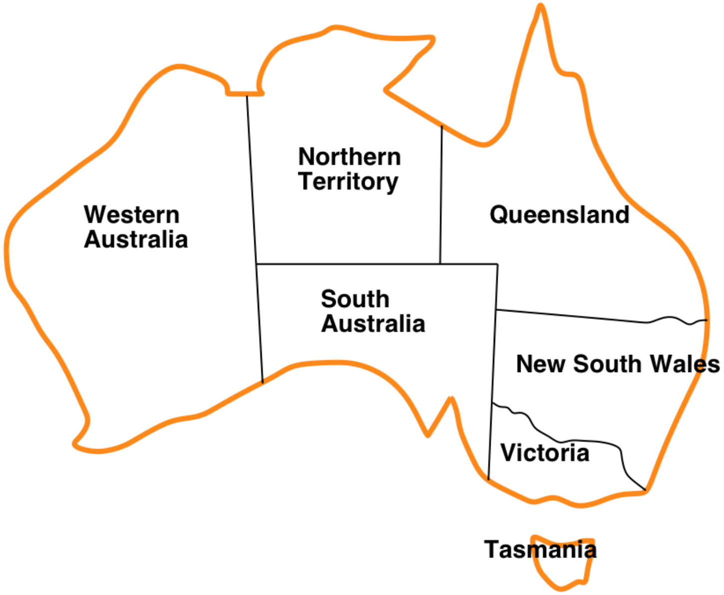
One representation:

- nodes represent word positions: 1-down…6-across
- domains are the words
- constraints specify that the letters on the intersections must be the same

Dual representation:

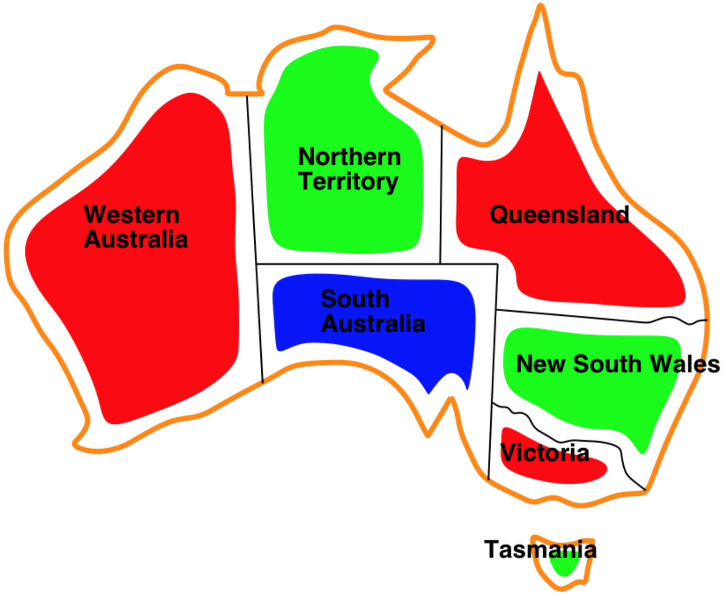
- nodes represent the individual squares
- domains are the letters
- constraints specify that the words must fit

EXAMPLE: MAP COLOURING



Variables:	WA, NT, Q, NSW, V, SA, T
Domains:	$\mathbf{D}_i = \{red, green, blue\}$
Constraints:	adjacent regions must have different colors, i.e., $WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, \ldots$

EXAMPLE: MAP COLOURING

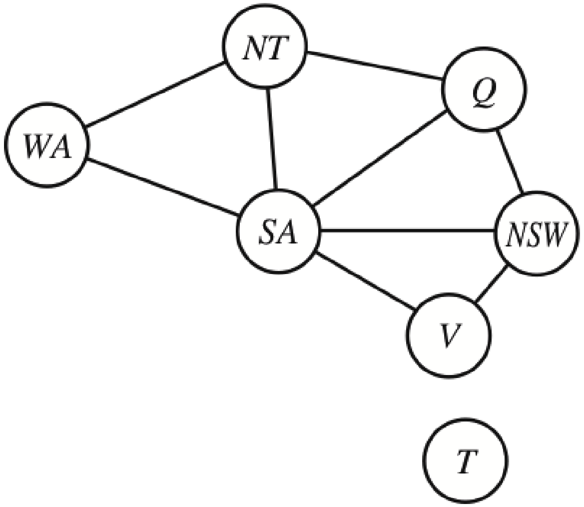


Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green,$
$V = red, SA = blue, T = green\}$

CONSTRAINT GRAPH

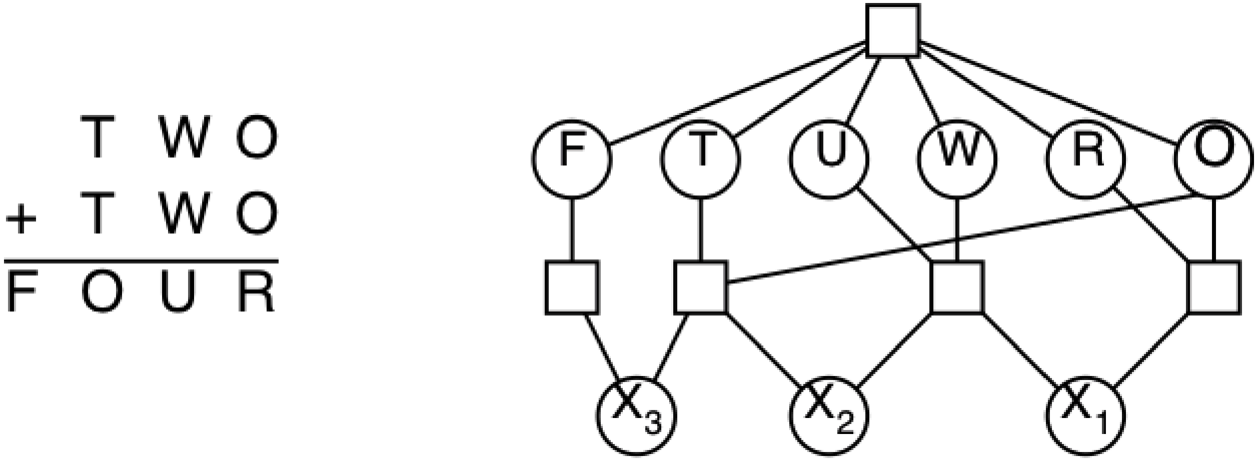
Binary CSP: each constraint relates at most two variables
(note: this does not say anything about the domains)

Constraint graph: every variable is a node, every binary constraint is an arc



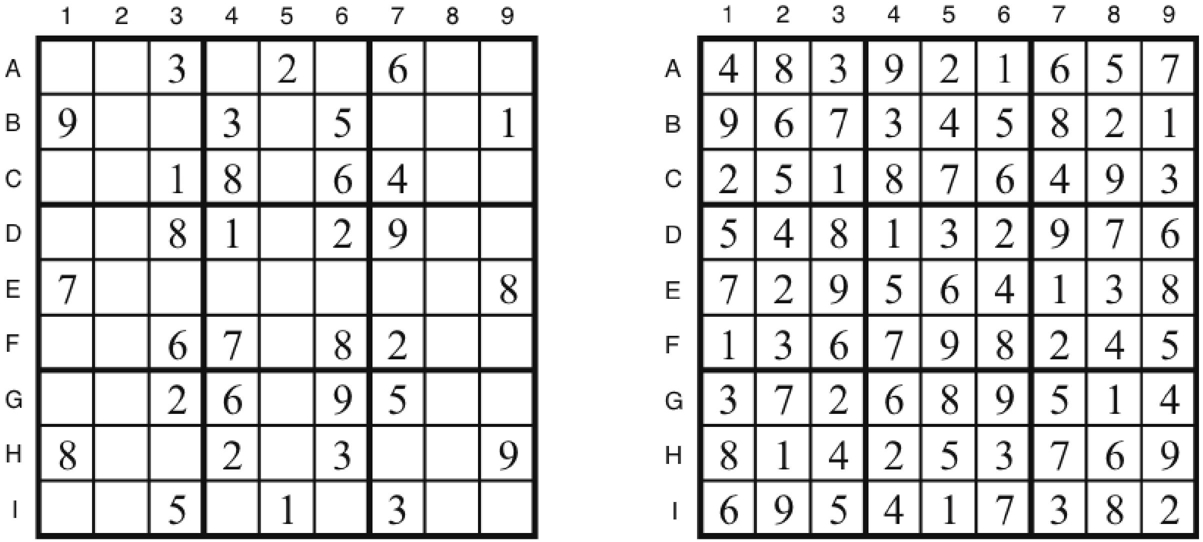
CSP algorithms can use the graph structure to speed up search,
e.g., Tasmania is an independent subproblem.

EXAMPLE: CRYPTARITHMETIC PUZZLE



Variables:	$F, T, U, W, R, O, X_1, X_2, X_3$
Domains:	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints:	$Alldiff(F, T, U, W, R, O)$, $O + O = R + 10 \cdot X_1$, etc.
Note:	This is not a binary CSP! The graph is a *constraint hypergraph*

EXAMPLE: SUDOKU



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

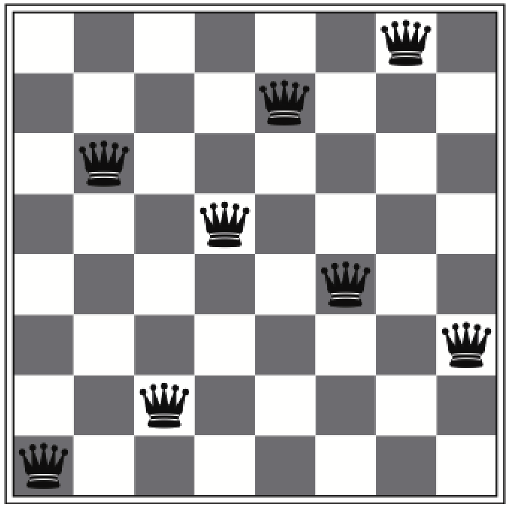
(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Variables:	$A_1 \ldots A_9, B_1, \ldots, E_5, \ldots, I_9$
Domains:	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints:	$\mathit{Alldiff}(A_1, \ldots, A_9)$, …, $\mathit{Alldiff}(A_5, \ldots, I_5)$, …, $\mathit{Alldiff}(D_1, \ldots, F_3)$, …, $B_1 = 9$, …, $F_6 = 8$, …, $I_7 = 3$

EXAMPLE: N-QUEENS



Variables:	$Q_1, Q_2, \ldots, Q_n$
Domains:	$\{1, 2, 3, \ldots, n\}$
Constraints:	$\mathit{Alldiff}(Q_1, Q_2, \ldots, Q_n)$, $Q_i - Q_j \neq \lvert i - j \rvert \;\; (1 \leq i < j \leq n)$

CSP VARIETIES

Discrete variables, *finite domains*:

- n variables, domain size $d \Rightarrow O(d^n)$ complete assignments
- what we discuss in this course

Discrete variables, *infinite domains* (integers, strings, etc.)

- e.g., job scheduling — variables are start/end times for each job
- we need a *constraint language* for formulating the constraints (e.g., $T_1 + d_1 \leq T_2$)
- *linear* constraints are solvable — *nonlinear* are undecidable

Continuous variables:

- e.g., scheduling for Hubble Telescope observations and manouvers
- linear constraints (*linear programming*) — solvable in polynomial time!

DIFFERENT KINDS OF CONSTRAINTS

Unary constraints involve a single variable:
- e.g., $SA \neq green$

Binary constraints involve pairs of variables:
- e.g., $SA \neq WA$

Global constraints (or *higher-order*) involve 3 or more variables:
- e.g., $\mathit{Alldiff}(WA, NT, SA)$
- all global constraints can be reduced to a number of binary constraints (but this might lead to an explosion of the number of constraints)

Preferences (or soft constraints):
- "constraint optimization problems"
- often representable by a cost for each variable assignment
- not discussed in this course

CSP AS A SEARCH PROBLEM
(R&N 6.3–6.3.2)

BACKTRACKING SEARCH

HEURISTICS: IMPROVING BACKTRACKING EFFICIENCY

GENERATE-AND-TEST ALGORITHM

Generate the assignment space $\mathbf{D} = \mathbf{D}_{V_1} \times \mathbf{D}_{V_2} \times \cdots \times \mathbf{D}_{V_n}$
Test each assignment with the constraints.

Example:

$$\begin{aligned}
\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\
&= \{1,2,3,4\} \times \cdots \times \{1,2,3,4\} \\
&= \{(1,1,1,1,1),(1,1,1,1,2),\ldots,(4,4,4,4,4)\}
\end{aligned}$$

How many assignments need to be tested for n variables, each with domain size $d = |\mathbf{D_i}|$?

CSP AS A SEARCH PROBLEM

Let’s start with the straightforward, dumb approach.

States are defined by the values assigned so far:

- *Initial state*: the empty assignment, { }
- *Successor function*: assign a value to an unassigned variable that does not conflict with current assignment
  $\Longrightarrow$ fail if there are no legal assignments
- *Goal test*: the current assignment is complete

Every solution appears at depth n (assuming n variables)
$\Longrightarrow$ we can use depth-first-search, no risk for infinite loops

At search depth k, the branching factor is $b = (n-k)d$
(where $d = |\mathbf{D}_i|$ is the domain size and $n-k$ is the number of unassigned variables)
$\Longrightarrow$ hence there are $n!d^n$ leaves

BACKTRACKING SEARCH

Variable assignments are commutative:

- $\{WA = red, NT = green\}$ is the same as $\{NT = green, WA = red\}$

It's unnecessary work to assign WA followed by NT in one branch,
and NT followed by WA in another branch.

Instead, at each depth level, we can decide on one single variable to assign:

- this gives branching factor $b = d$, so there are d^n leaves (instead of $n!d^n$)

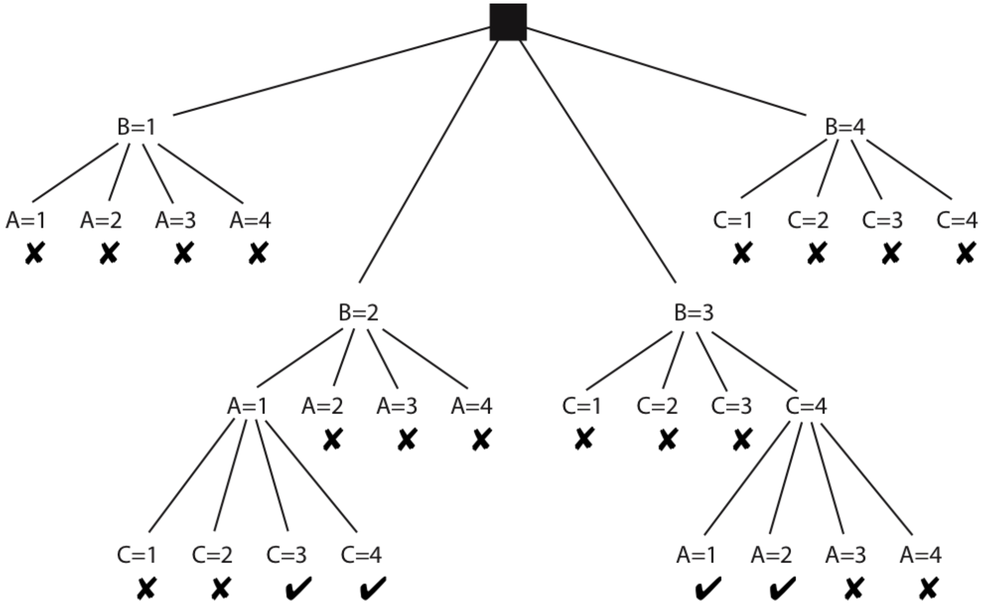
Depth-first search with single-variable assignments is called *backtracking search*:

- backtracking search is the basic uninformed CSP algorithm
- it can solve n-queens for $n \approx 25$

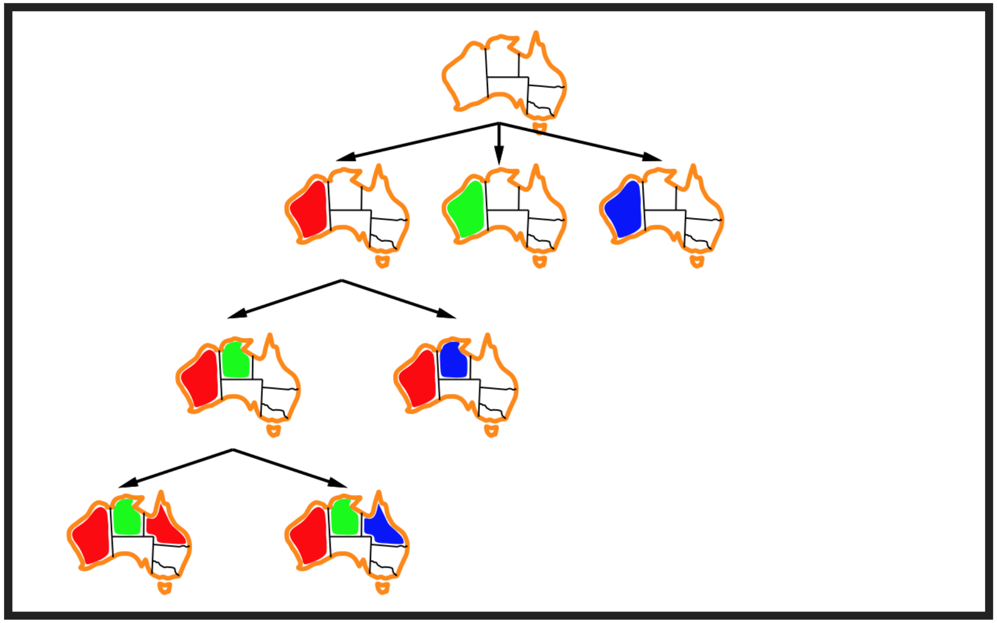
Why not use breadth-first search?

SIMPLE BACKTRACKING EXAMPLE

Variables:	A, B, C
Domains:	$\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \{1, 2, 3, 4\}$
Constraints:	$(A < B), (B < C)$



EXAMPLE: AUSTRALIA MAP COLOURS



Assign variable: *Q*

ALGORITHM FOR BACKTRACKING SEARCH

```
function BacktrackingSearch(csp):
    return Backtrack(csp, { })

function Backtrack(csp, assignment):
    if assignment is complete then return assignment
    var := SelectUnassignedVariable(csp, assignment)
    for each value in OrderDomainValues(csp, var, assignment):
        if value is consistent with assignment:
            inferences := Inference(csp, var, value)
            if inferences ≠ failure:
                result := Backtrack(csp, assignment ∪ {var=value} ∪ inferences)
                if result ≠ failure then return result
    return failure
```

HEURISTICS: IMPROVING BACKTRACKING EFFICIENCY

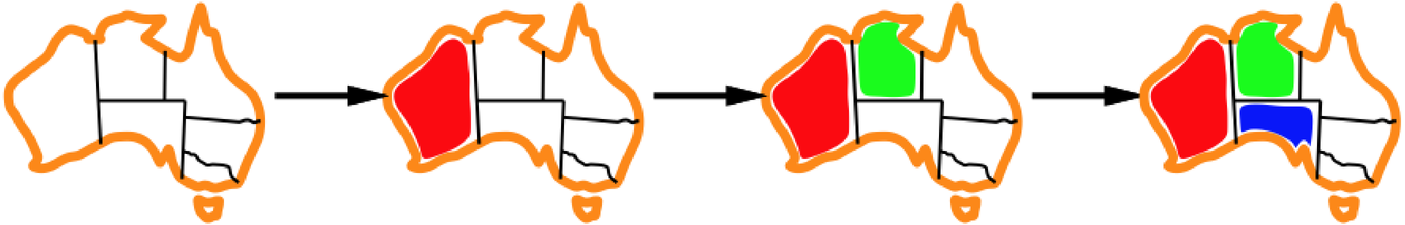
The general-purpose algorithm gives rise to several questions:

- Which variable should be assigned next?
    - SelectUnassignedVariable($csp, assignment$)
- In what order should its values be tried?
    - OrderDomainValues($csp, var, assignment$)
- What inferences should be performed at each step?
    - Inference($csp, var, value$)
- Can the search avoid repeating failures?
    - Conflict-directed backjumping, constraint learning, no-good sets (R&N 6.3.3, not covered in this course)

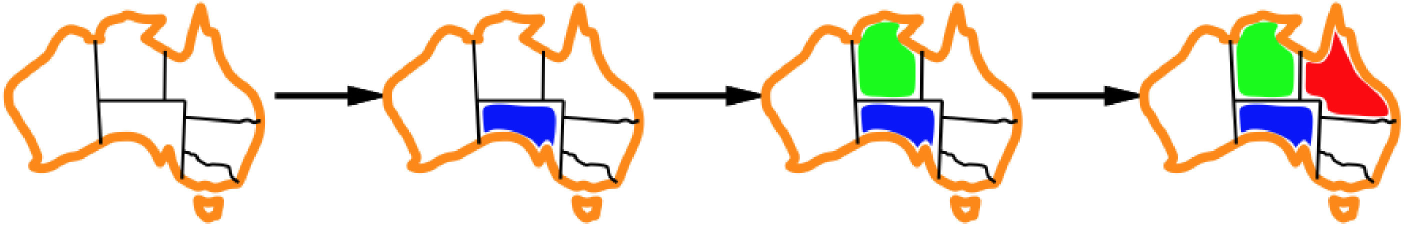
SELECTING UNASSIGNED VARIABLES

Heuristics for selecting the next unassigned variable:

- Minimum remaining values (MRV):
  ⟹ choose the variable with the fewest legal values



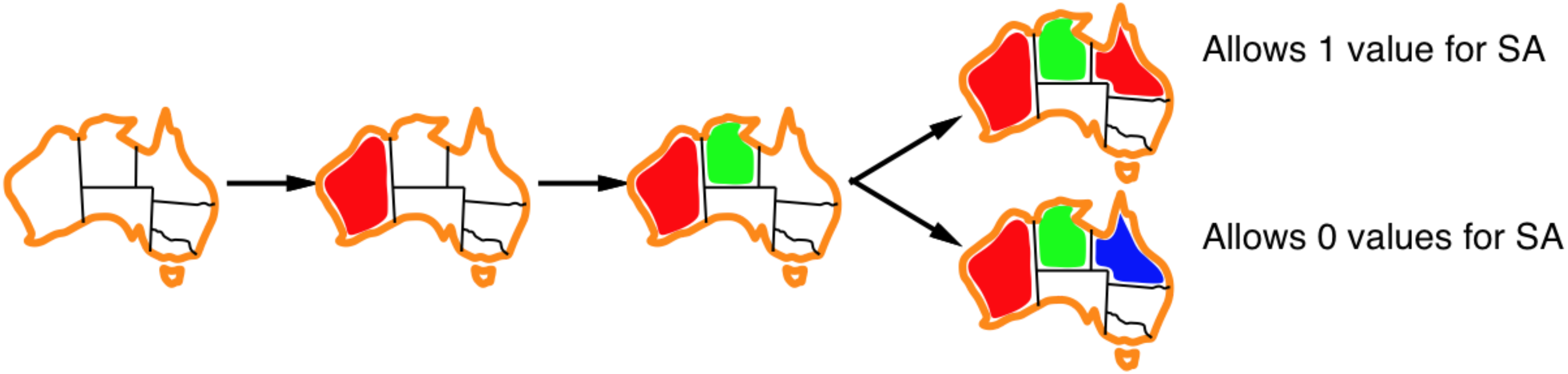
- Degree heuristic (if there are several MRV variables):
  ⟹ choose the variable with most constraints on remaining variables



ORDERING DOMAIN VALUES

Heuristics for ordering the values of a selected variable:

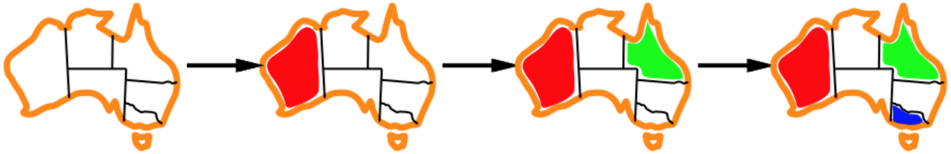
- Least constraining value:
  $\Longrightarrow$ prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph

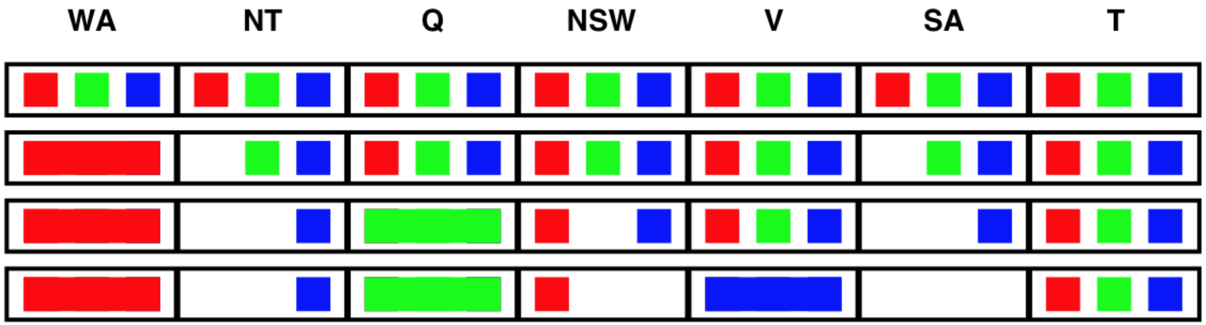


INFERENCE: FORWARD CHECKING

Forward checking is a simple form of inference:

- Keep track of remaining legal values for unassigned variables
  — terminate when any variable has no legal values left
- When a new variable is assigned, recalculate the legal values for its neighbors





WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥	🟦		🟥 🟩 🟦

INFERENCE: CONSTRAINT PROPAGATION

Forward checking propagates information from assigned to unassigned variables, but doesn't detect all failures early:

WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦

NT and SA cannot both be blue!

- *Forward checking* enforces local constraints
- *Constraint propagation* enforces local constraints, *repeatedly* until reaching a fixed point

CONSTRAINT PROGAGATION (R&N 6.2–6.2.2)

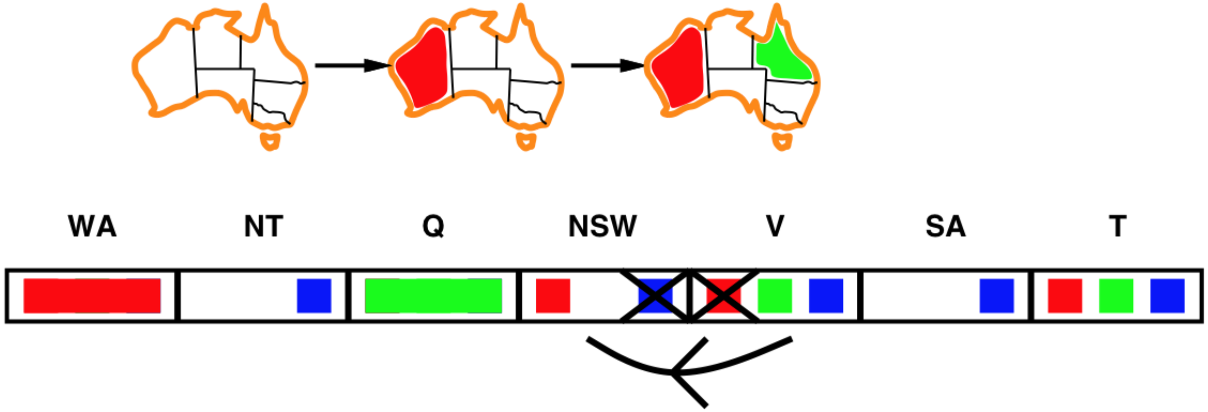
ARC CONSISTENCY

MAINTAINING ARC CONSISTENCY

CONSTRAINT PROPAGATION: ARC CONSISTENCY

The simplest form of propagation is to make each arc consistent:

- $X \rightarrow Y$ is *arc consistent* iff:
  for every value x of X, there is some allowed value y in Y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking

CONSISTENCY

Different variants of constistency:

- A variable is *node-consistent* if all values in its domain satisfy its own unary constraints,
- a variable is *arc-consistent* if every value in its domain satisfies the variable's binary constraints,
- *Generalised arc-consistency* is the same, but for n-ary constraints,
- *Path consistency* is arc-consistency, but for 3 variables at the same time.
- k*-consistency* is arc-consistency, but for k variables,
- …and there are consistency checks for several global constraints, such as $Alldiff$ and $Atmost$.

A network is X-consistent if every variable is X-consistent with every other variable.

SCHEDULING EXAMPLE (AGAIN)

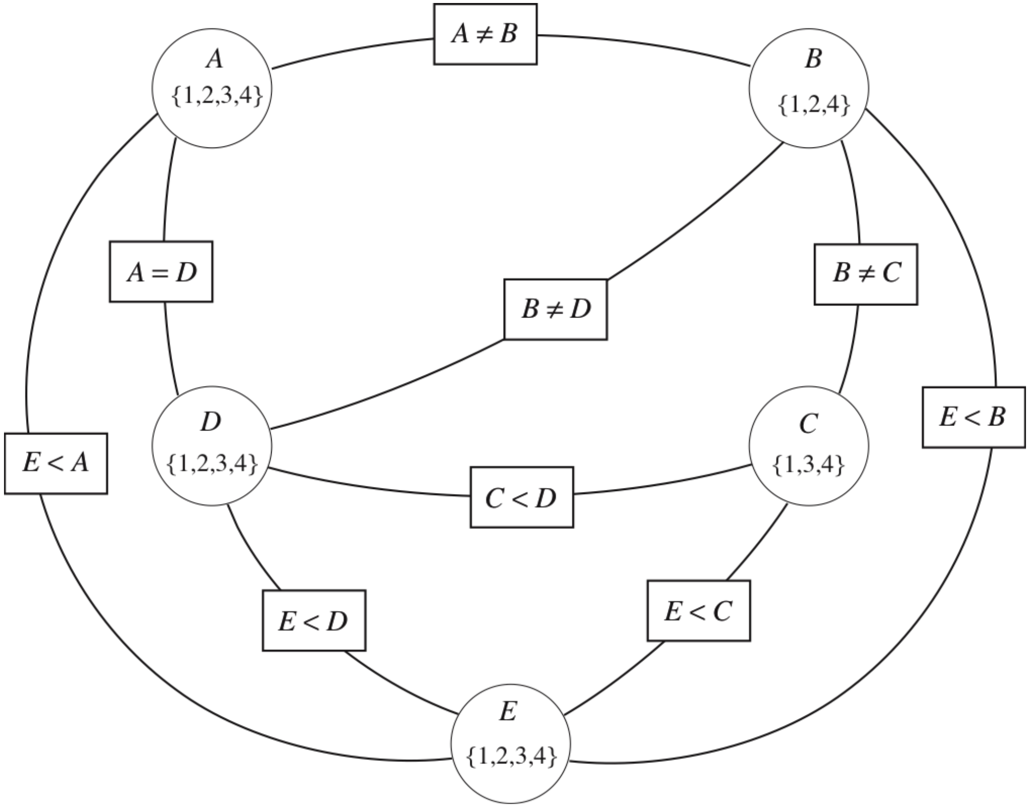
Variables:	A, B, C, D, E representing starting times of various activities.
Domains:	$\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1, 2, 3, 4\}$
Constraints:	$(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D),$ $(E < A), (E < B), (E < C), (E < D), (B \neq D)$

Is this example node consistent?

- $\mathbf{D}_B = \{1, 2, 3, 4\}$ is not node consistent, since $B = 3$ violates the constraint $B \neq 3$ $\Longrightarrow$ reduce the domain $\mathbf{D}_B = \{1, 2, 4\}$
- $\mathbf{D}_C = \{1, 2, 3, 4\}$ is not node consistent, since $C = 2$ violates the constraint $C \neq 2$ $\Longrightarrow$ reduce the domain $\mathbf{D}_C = \{1, 3, 4\}$

SCHEDULING EXAMPLE AS A CONSTRAINT GRAPH

If we reduce the domains for B and C, then the constraint graph is node consistent.



ARC CONSISTENCY

A variable X is binary *arc-consistent* with respect to another variables (Y) if:

- For each value $x \in \mathbf{D}_X$, there is some $y \in \mathbf{D}_Y$ such that the binary constraint $C_{XY}(x, y)$ is satisfied.

A variable X is *generalised arc-consistent* with respect to variables $(Y, Z, \dots)$ if:

- For each value $x \in \mathbf{D}_X$, there is some assignment $y, z, \dots \in \mathbf{D}_Y, \mathbf{D}_Z, \dots$ such that $C_{XYZ\dots}(x, y, z, \dots)$ is satisfied.

What if X is not arc consistent to Y?

- All values $x \in \mathbf{D}_X$ for which there is no corresponding $y \in \mathbf{D}_Y$ can be deleted from $\mathbf{D}_X$ to make X arc consistent.

Note! The arcs in a constraint graph are directed:

- (X, Y) and (Y, X) are considered as two different arcs,
- i.e., X can be arc consistent to Y, but Y not arc consistent to X.

ARC CONSISTENCY ALGORITHM

Keep a set of arcs to be considered: pick one arc (X, Y) at the time and make it consistent (i.e., make X arc consistent to Y).

- Start with the set of all arcs $\{(X, Y), (Y, X), (X, Z), (Z, X), \ldots\}$.

When an arc has been made arc consistent, does it ever need to be checked again?

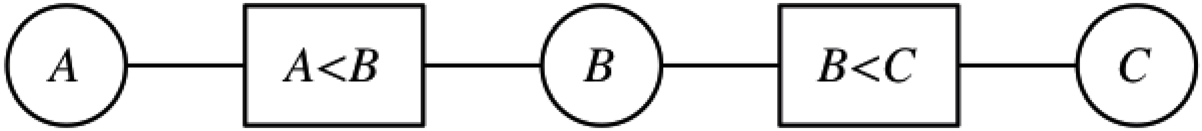
- An arc (X, Y) needs to be revisited if the domain of Y is revised.

Three possible outcomes when all arcs are made arc consistent:
(Is there a solution?)

- One domain is empty $\Longrightarrow$ *no solution*
- Each domain has a single value $\Longrightarrow$ *unique solution*
- Some domains have more than one value $\Longrightarrow$ *maybe a solution, maybe not*

QUIZ: ARC CONSISTENCY

The variables and constraints are in the constraint graph:



Assume the initial domains are $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \{1, 2, 3, 4\}$

How will the domains look like after making the graph arc consistent?

THE ARC CONSISTENCY ALGORITHM AC-3

```
function AC-3(inout csp):
    initialise queue to all arcs in csp
    while queue is not empty:
        $(X, Y)$ := RemoveOne(queue)
        if Revise(csp, $X$, $Y$):
            if  $\mathbf{D}_X = \emptyset$  then return false
            for each $Z$ in $X$.neighbors–$\{Y\}$:
                add $(Z, X)$ to queue
    return true

function Revise(inout csp, $X$, $Y$):
    revised := false
    for each $x$ in $\mathbf{D}_X$:
        if there is no value $y$ in $\mathbf{D}_Y$ satisfying the csp constraint $C_{XY}(x, y)$:
            delete $x$ from $\mathbf{D}_X$
            revised := true
    return revised
```

Note: This algorithm destructively updates the domains of the CSP! You might need to copy the CSP before calling AC-3.

MAINTAINING ARC-CONSISTENCY (MAC)

What if some domains have more than one element after AC?

We can always resort to backtracking search:

- Select a variable and a value using some heuristics (e.g., minimum-remaining-values, degree-heuristic, least-constraining-value)
- Make the graph arc-consistent again
- Backtrack and try new values/variables, if AC fails
- Select a new variable/value, perform arc-consistency, etc.

Do we need to restart AC from scratch?

- no, only some arcs risk becoming inconsistent after a new assignment
- restart AC with the queue $\{(Y_i, X) | X \rightarrow Y_i\}$, i.e., only the arcs (Y_i, X) where Y_i are the neighbors of X
- this algorithm is called *Maintaining Arc Consistency* (MAC)

DOMAIN SPLITTING (NOT IN R&N)

What if some domains are very big?

- Instead of assigning every possible value to a variable, we can split its domain
- Split one of the domains, then recursively solve each half, i.e.:
    - perform AC on the resulting graph, then split a domain, perform AC, split a domain, perform AC, split, etc.
- It is often good to split a domain in half, i.e.:
    - if $\mathbf{D}_X = \{1, \dots, 1000\}$, split into $\{1, \dots 500\}$ and $\{501, \dots, 1000\}$

NATURAL LANGUAGE PROCESSING

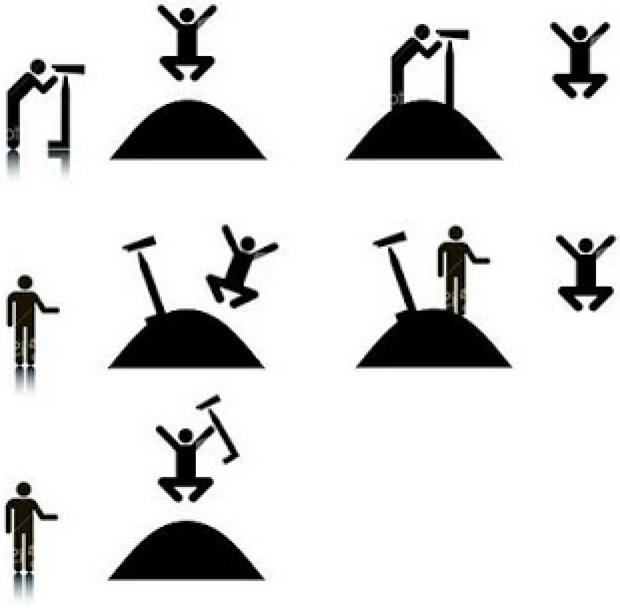
DIT410/TIN174, Artificial Intelligence

John J. Camilleri

4 April, 2017

STUDENT PARTICIPATION LINK

https://b.socrative.com/student/
or go to socrative.com and click **Student login**
Room name: **HORSEY**



Source: http://www.denizyuret.com/2010/12/research-focus.html

NATURAL LANGUAGE

FORMAL LANGUAGE

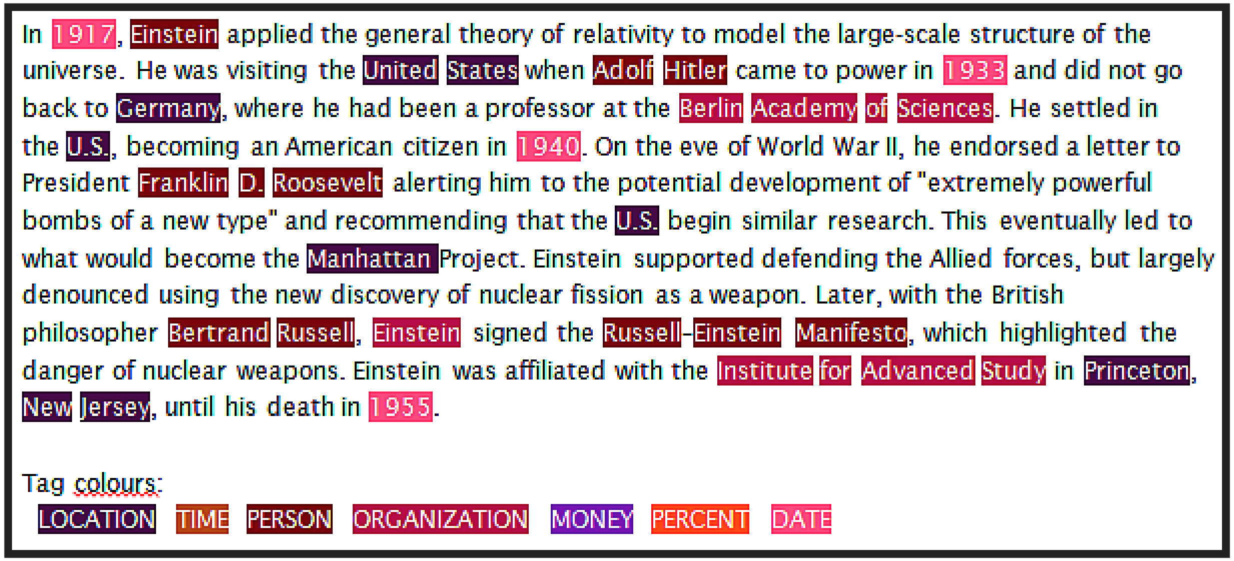
NATURAL LANGUAGE *UNDERSTANDING*

http://i.huffpost.com/gen/1403845/images/o-SPIKE-JONZE-HER-facebook.jpg

NATURAL LANGUAGE *PROCESSING*

INFORMATION EXTRACTION

Named entity recognition



In 1917, Einstein applied the general theory of relativity to model the large-scale structure of the universe. He was visiting the United States when Adolf Hitler came to power in 1933 and did not go back to Germany, where he had been a professor at the Berlin Academy of Sciences. He settled in the U.S., becoming an American citizen in 1940. On the eve of World War II, he endorsed a letter to President Franklin D. Roosevelt alerting him to the potential development of "extremely powerful bombs of a new type" and recommending that the U.S. begin similar research. This eventually led to what would become the Manhattan Project. Einstein supported defending the Allied forces, but largely denounced using the new discovery of nuclear fission as a weapon. Later, with the British philosopher Bertrand Russell, Einstein signed the Russell–Einstein Manifesto, which highlighted the danger of nuclear weapons. Einstein was affiliated with the Institute for Advanced Study in Princeton, New Jersey, until his death in 1955.

Tag colours:

LOCATION TIME PERSON ORGANIZATION MONEY PERCENT DATE

http://www.europeana-newspapers.eu/named-entity-recognition-for-digitised-newspapers/

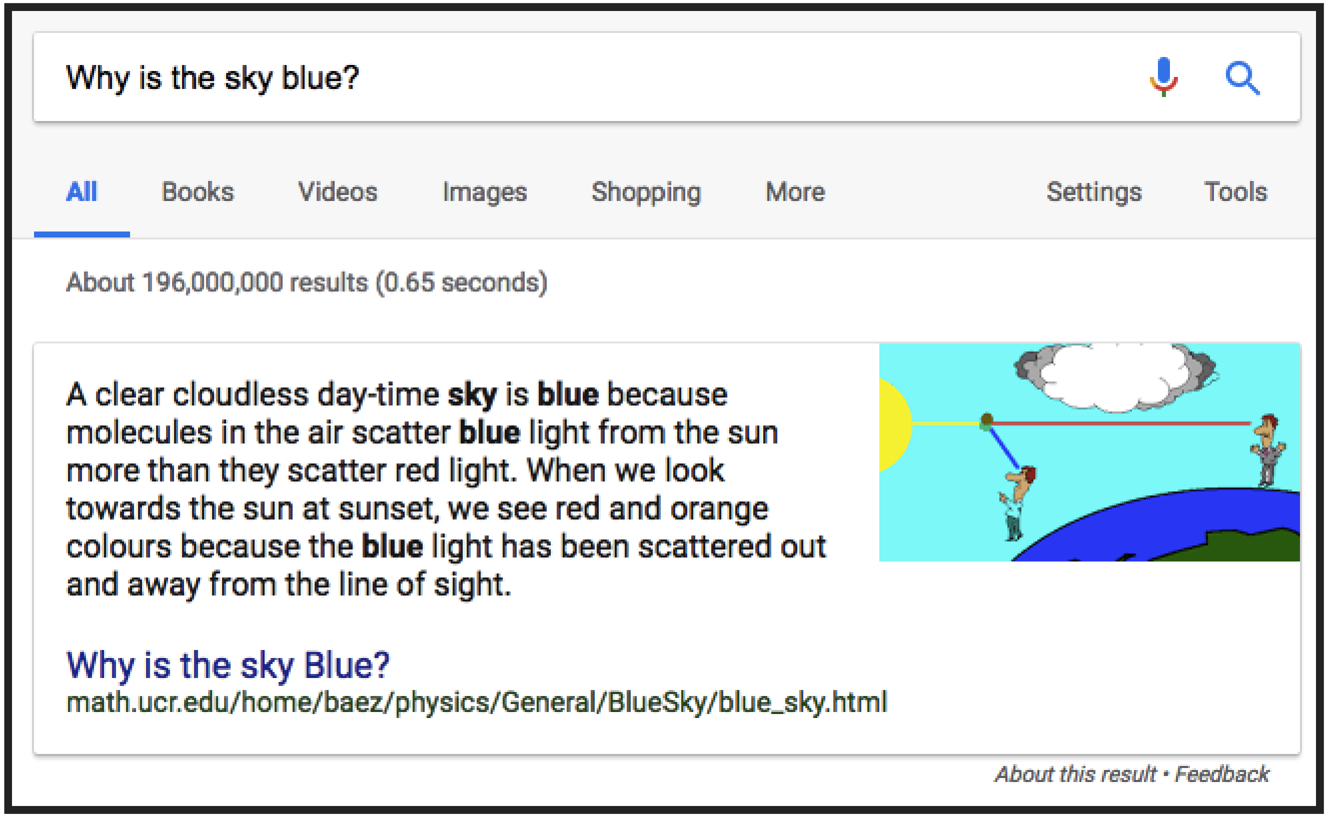
CLASSIFICATION

Sentiment analysis

https://www.csc.ncsu.edu/faculty/healey/tweet_viz/

INFORMATION RETRIEVAL

Search



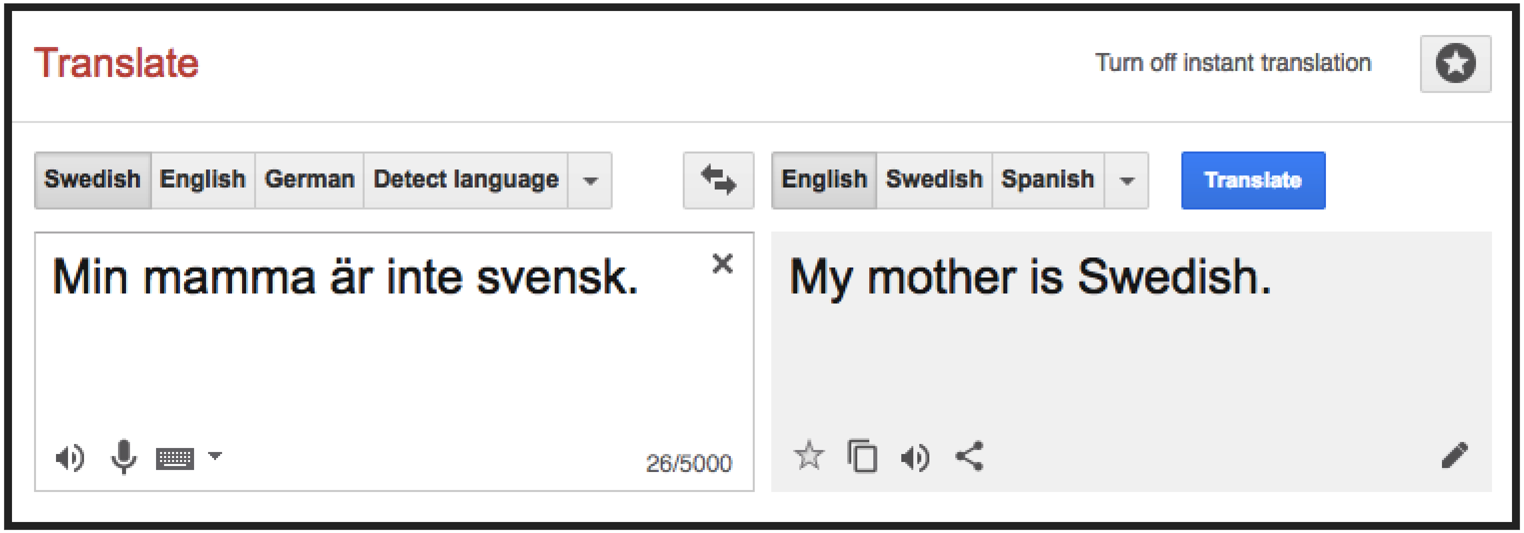
Why is the sky blue?

About 196,000,000 results (0.65 seconds)

A clear cloudless day-time **sky** is **blue** because molecules in the air scatter **blue** light from the sun more than they scatter red light. When we look towards the sun at sunset, we see red and orange colours because the **blue** light has been scattered out and away from the line of sight.

Why is the sky Blue?
math.ucr.edu/home/baez/physics/General/BlueSky/blue_sky.html

MACHINE TRANSLATION



Min mamma är inte svensk.

My mother is Swedish.

APPROACHES

RULE-BASED

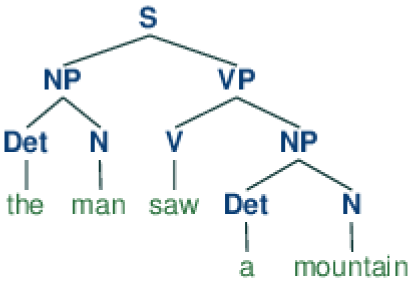
STATISTICAL

DEEP LEARNING

PHRASE-STRUCTURE GRAMMARS

“the man saw a mountain”

↓



CONTEXT-FREE GRAMMAR (CFG)

terminals, non-terminals, rules

S → NP VP
NP → Det N
VP → V NP
N → man | mountain
V → saw
Det → a | the

SOCRATIVE QUESTION

http://www.triblocal.com/highland-park-highwood/files/2012/03/stock-photo-17181584-mountain-man.jpg

AMBIGUITY

extending the grammar with prepositions

PARSING

input string → parse tree(s)

CYK ALGORITHM

PROBABILISTIC PARSING

OVERGENERATION

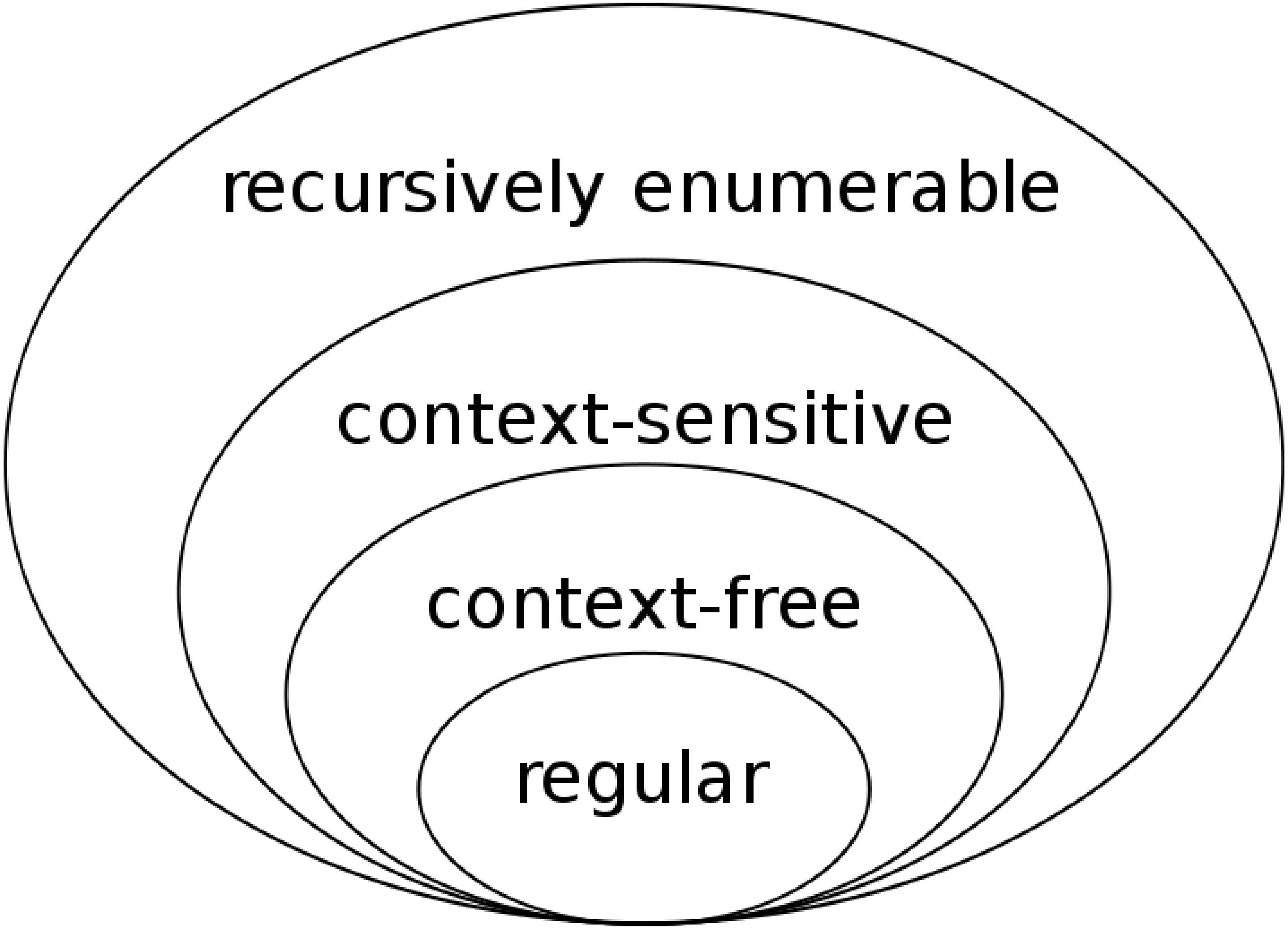
“All grammars leak”

SOLUTIONS TO OVERGENERATION

In CFG
Other formalisms

GENERATIVE CAPACITY

Chomsky hierarchy



SOCRATIVE QUESTION

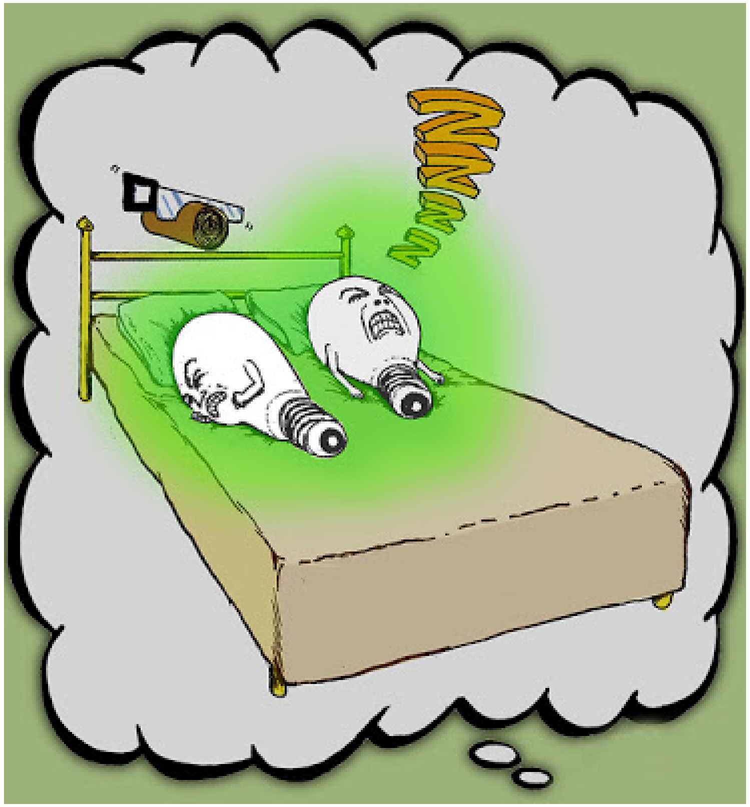
LEVELS OF AMBIGUITY

Lexical
Syntactic
Semantic

MODELS FOR DISAMBIGUATION

Acoustic model
Language model
Mental model
World model

SOCRATIVE QUESTION



http://wmjasco.blogspot.se/2008/11/colorless-green-ideas-do-not-sleep.html

THAT’S ALL FOR TODAY

FRIDAY:

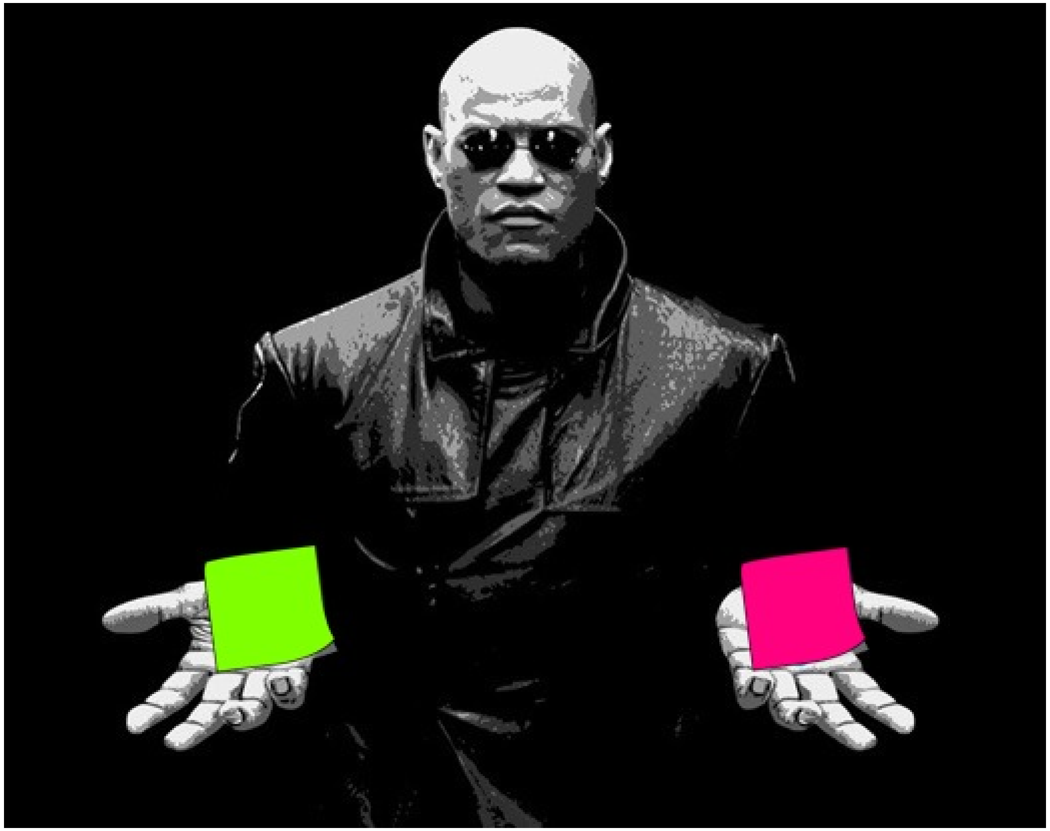
Semantics, Interpretation, NLP in Shrdlite

NATURAL LANGUAGE INTERPRETATION

DIT410/TIN174, Artificial Intelligence

John J. Camilleri

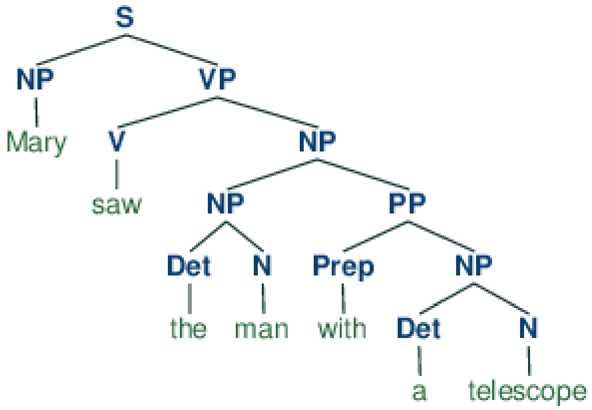
7 April, 2017

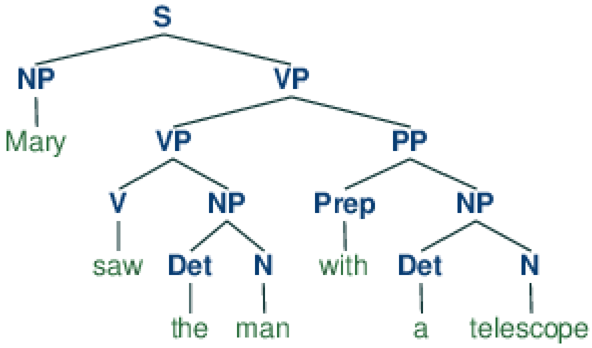


https://img.memesuper.com/7ad355dacca363617cdfcff7defc07ed_-of-morpheus-offering-the-morpheus-pill-meme_520-412.jpeg

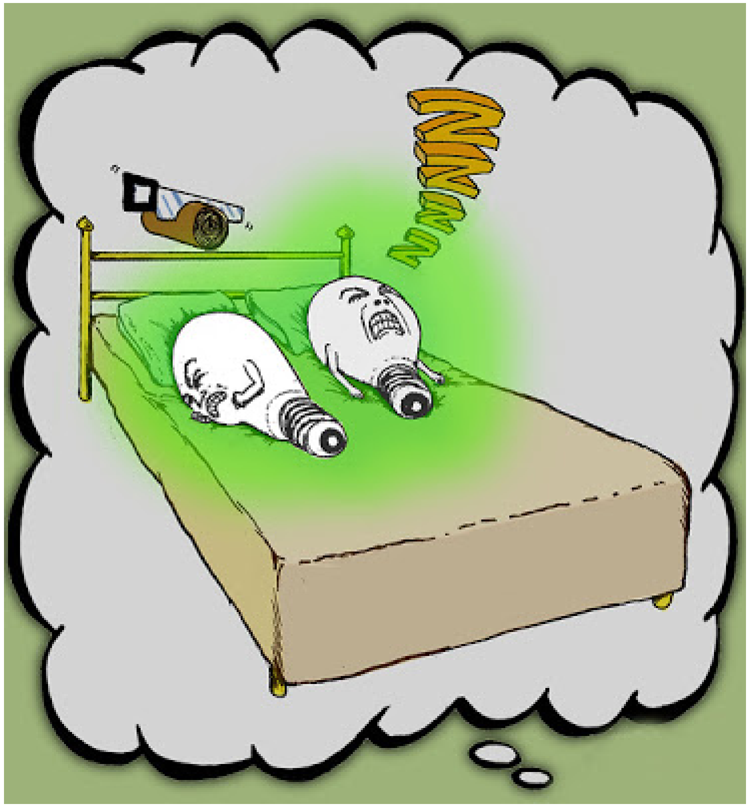
LAST TIME...

“Mary saw the man with a telescope”





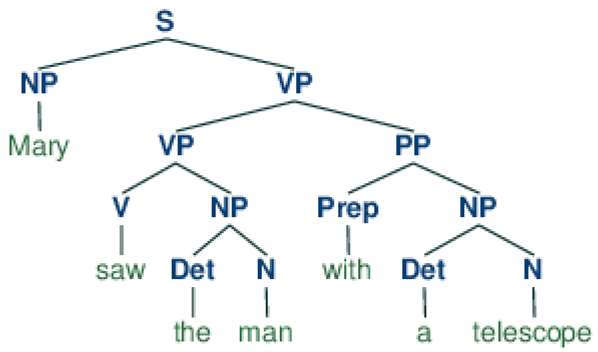
“Colourless green ideas sleep furiously”



http://wmjasco.blogspot.se/2008/11/colorless-green-ideas-do-not-sleep.html

Is this sentence valid? Yes or No

WHY SYNTAX?

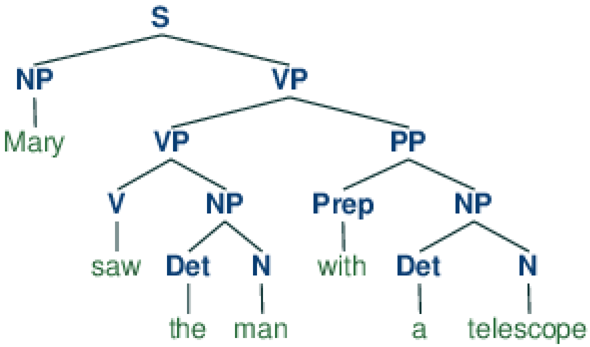

S
NP
VP
Mary
VP
PP
V
NP
Prep
NP
saw
Det
N
with
Det
N
the
man
a
telescope

SEMANTIC REPRESENTATION

Introducing logical terms

Mary = Mary
the man = Man
Mary saw the man = Saw(Mary, Man)

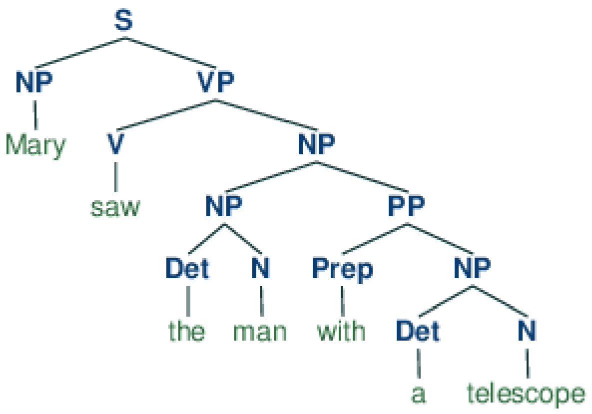
SEMANTIC INTERPRETATION (1)



↓

With(Saw(Mary, Man), Telescope)

SEMANTIC INTERPRETATION (2)



↓

Saw(Mary, With(Man, Telescope))

COMPOSITIONAL SEMANTICS

Mary = Mary

the man = Man

Mary saw the man = Saw(Mary, Man)

saw = $\lambda y\ \lambda x \cdot \mathrm{Saw}(x, y)$

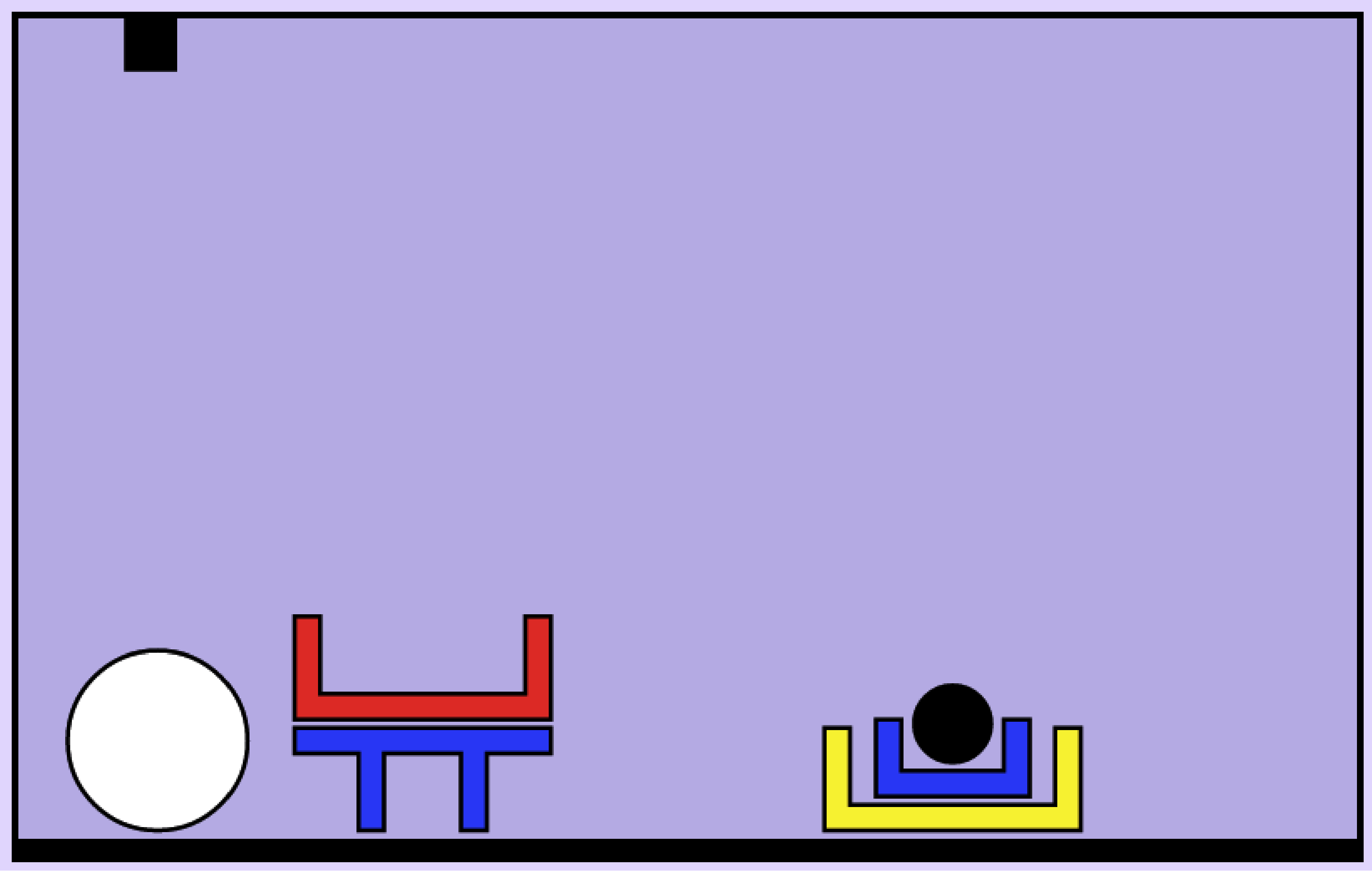
saw the man = $\lambda x \cdot \mathrm{Saw}(x, \mathrm{Man})$

INTERPRETATION

syntactic representation → *semantic* representation

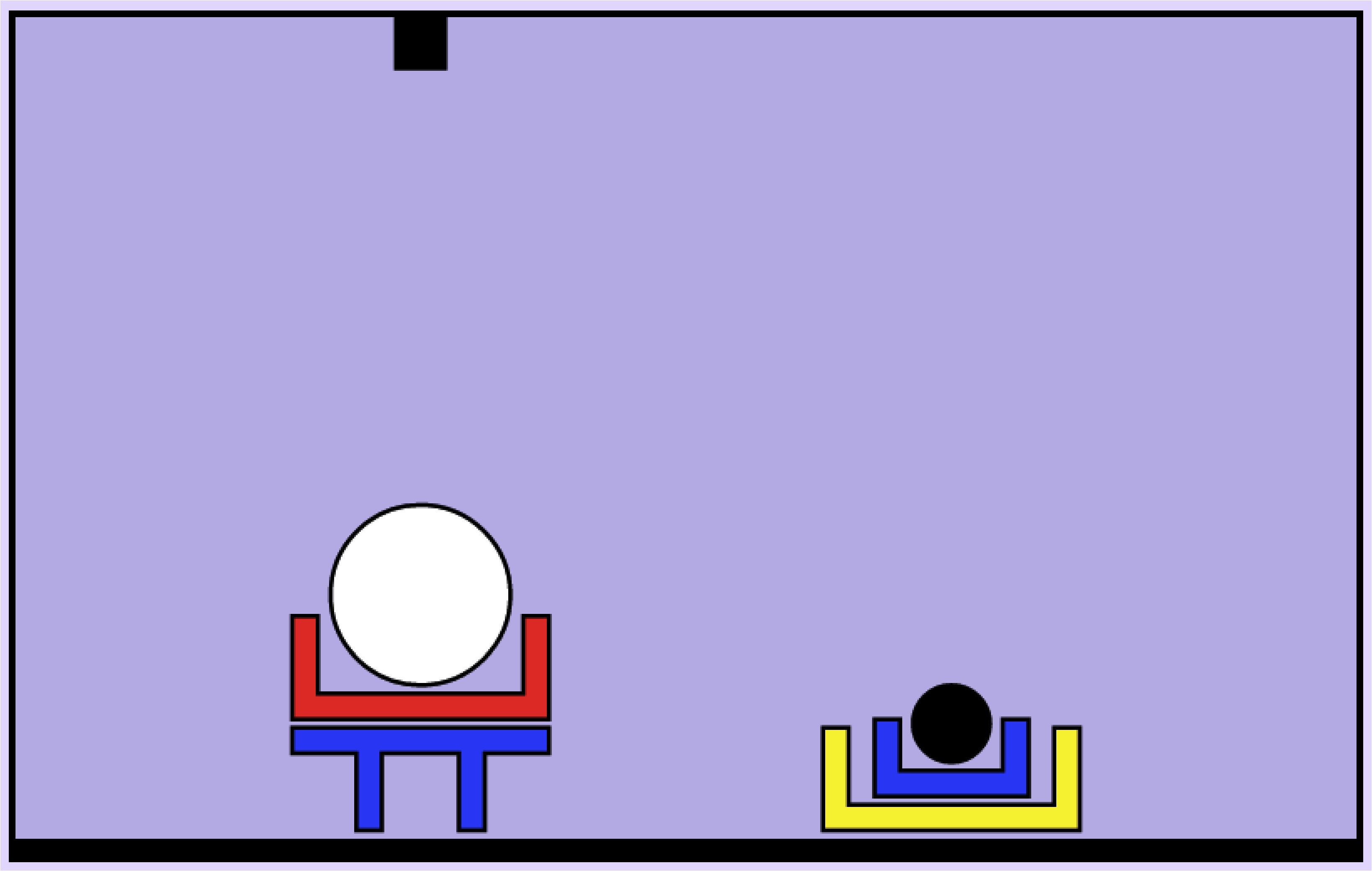
parse tree → logical term

Utterance: *“move the white ball into the red box”*

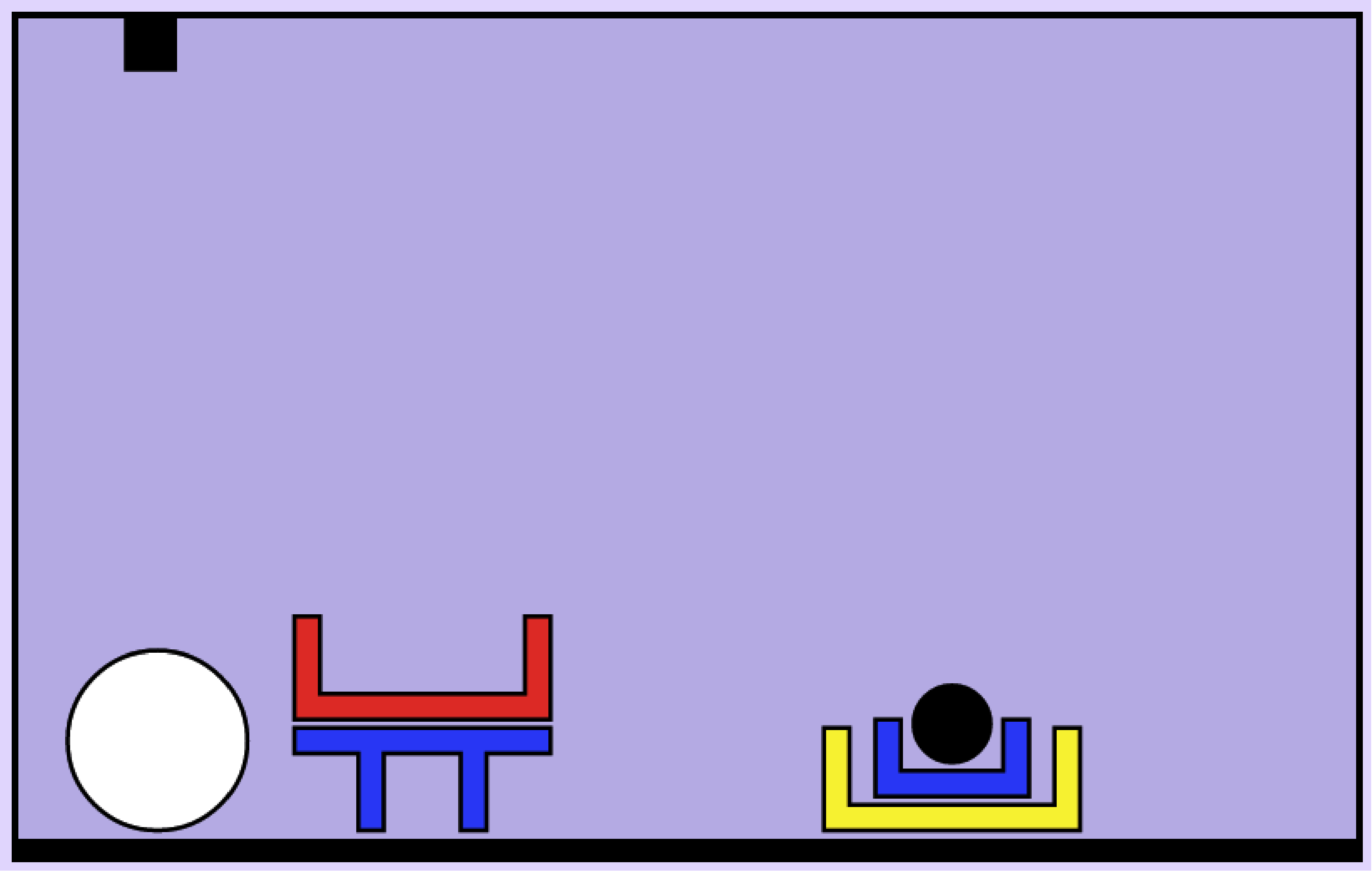


Is this ambiguous? Yes or No

Goal: inside(white_ball, red_box)



Utterance: *“move the ball into the red box”*



Is this ambiguous? Yes or No

SHRDLITE PIPELINE

1. *Parsing*: text input → parse trees
2. *Interpretation*: parse tree + world → goals
3. *Ambiguity resolution*: many goals → one goal
4. *Planning*: goal → robot movements

PARSING

text input → parse trees

` ` `function parse(input:string) : string | ShrdliteResult[]

~~~
{: .code}

```interface ShrdliteResult {
    input : string
    parse : Command
    interpretation? : DNFFormula
    plan? : string[]
}
~~~

GRAMMAR (SIMPLIFIED)

From file Grammar.ne

` ` `command –> “put” entity location entity –> quantifier object object –> size:? color:? form object –> object location location –> relation entity

```
{: .code}

Notes:

- Recursion
- Draw a tree top-down on the board

---

_“put the white ball in a box on the floor”_
![](img/nlp/shrdlite-small.png){:.noborder}
   Is this ambiguous?
<span style="background:lime;color:white;padding:3px 6px;">Yes</span> or
<span style="background:magenta;color:white;padding:3px 6px;">No</span>

---

_“put the white ball in a box on the floor”_
![](img/nlp/shrdlite-small.png){:.noborder}
   Is the ambiguity
<span style="background:lime;color:white;padding:3px 6px;">syntactic</span> or
<span style="background:magenta;color:white;padding:3px 6px;">semantic</span>?

Notes:
```

LOGICAL INTERPRETATIONS (“GOALS”)

` ` `type DNFFormula = Conjunction[] type Conjunction = Literal[]

~~~
{: .code}

DNF = Disjunctive Normal Form

Example: `(x ∧ y) ∨ (z)`

```DNFFormula([Conjunction([x, y]), Conjunction(z)])
~~~

LITERALS

` ` `interface Literal { relation : string; args : string[]; polarity : boolean; }

~~~
{: .code}

Example: `ontop(a,b)`

```{ relation:"ontop", args:["a","b"], polarity:true }
~~~

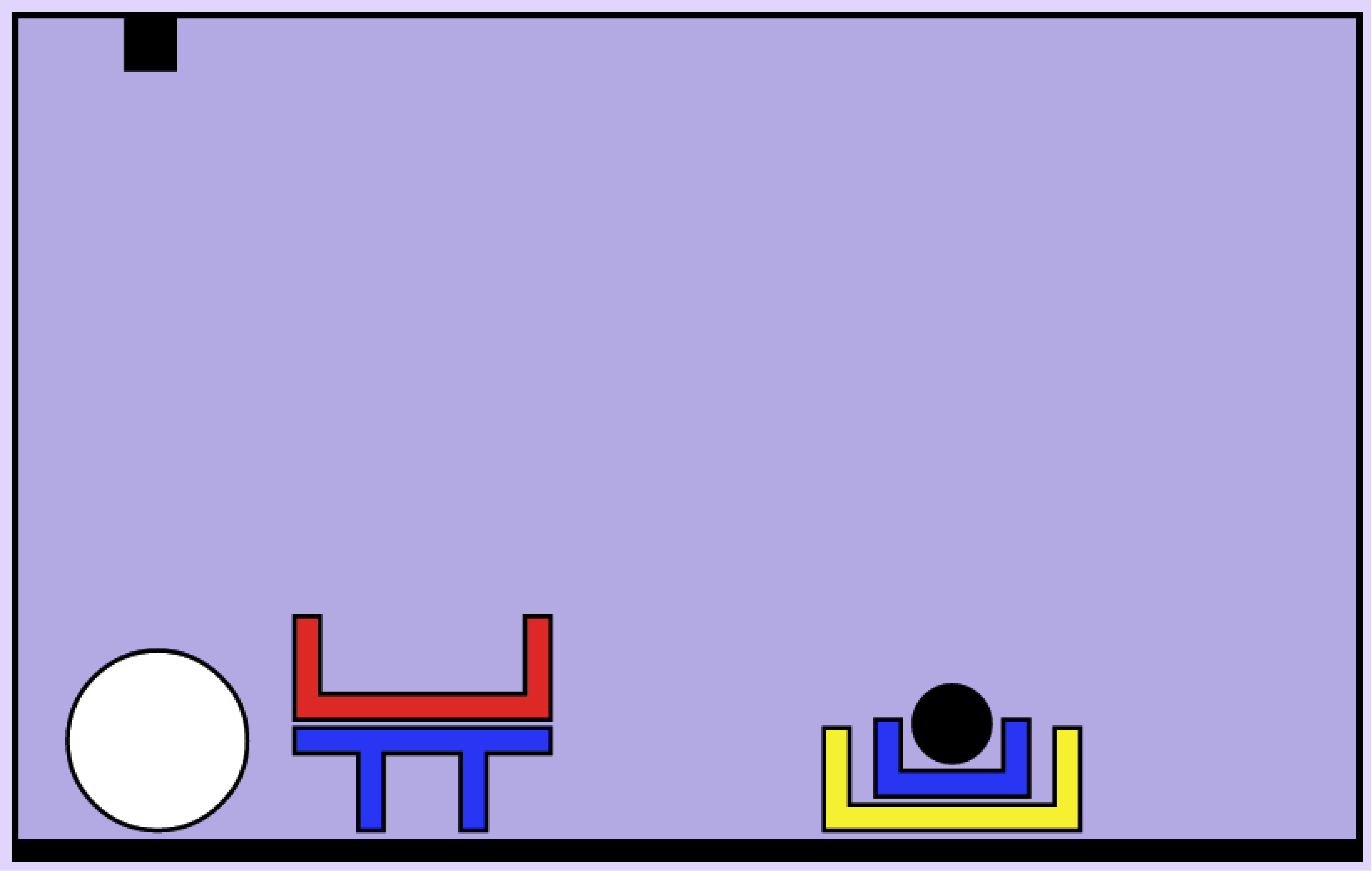
SPATIAL RELATIONS

- x is **on top** of y if it is directly on top
- x is **above** y if it is somewhere above
- …

AMBIGUITY

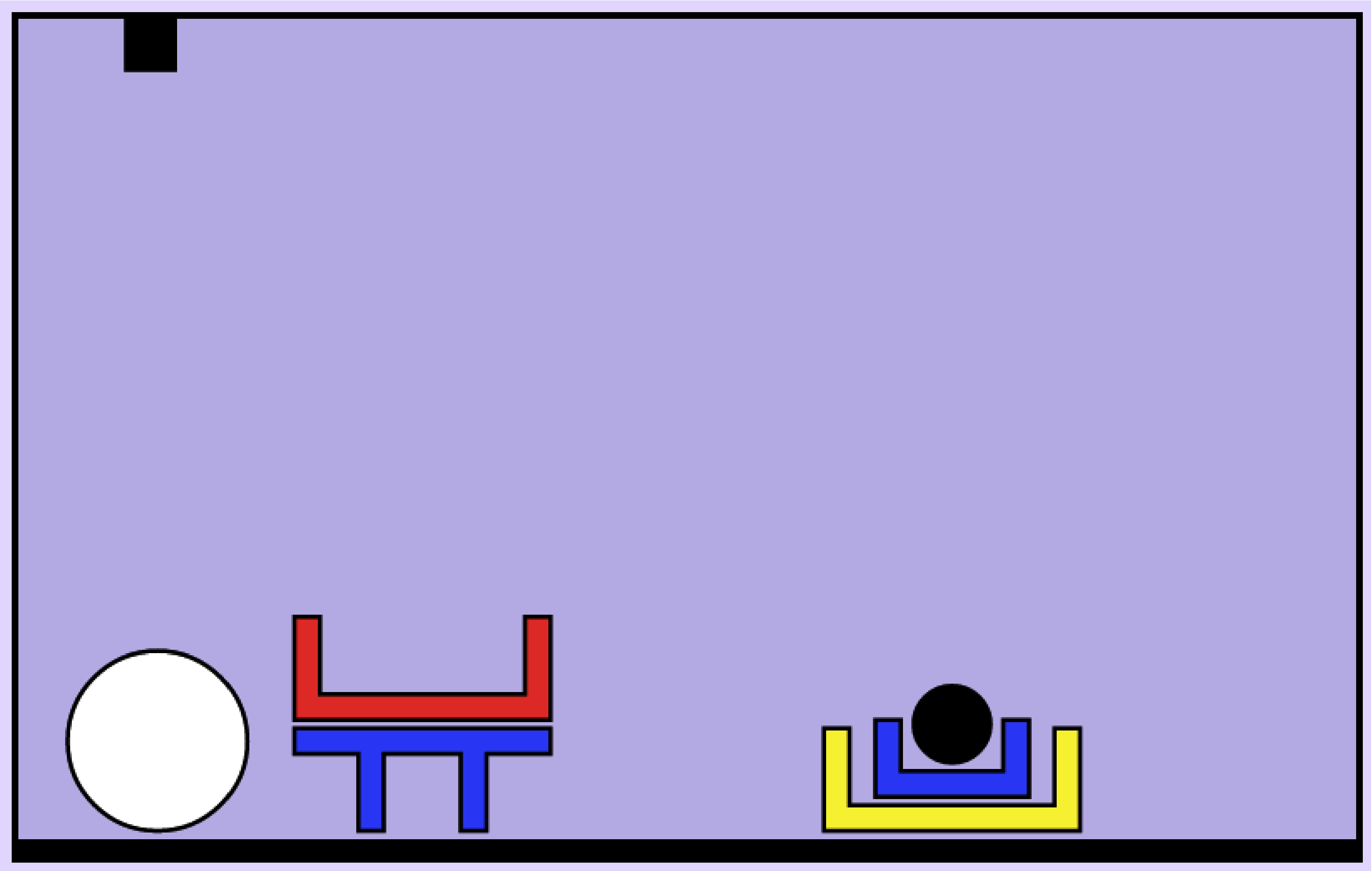
DNF inherently captures ambiguity
But impossible interperetations should be removed

“put the white ball ***that is*** *in a box on the floor”*



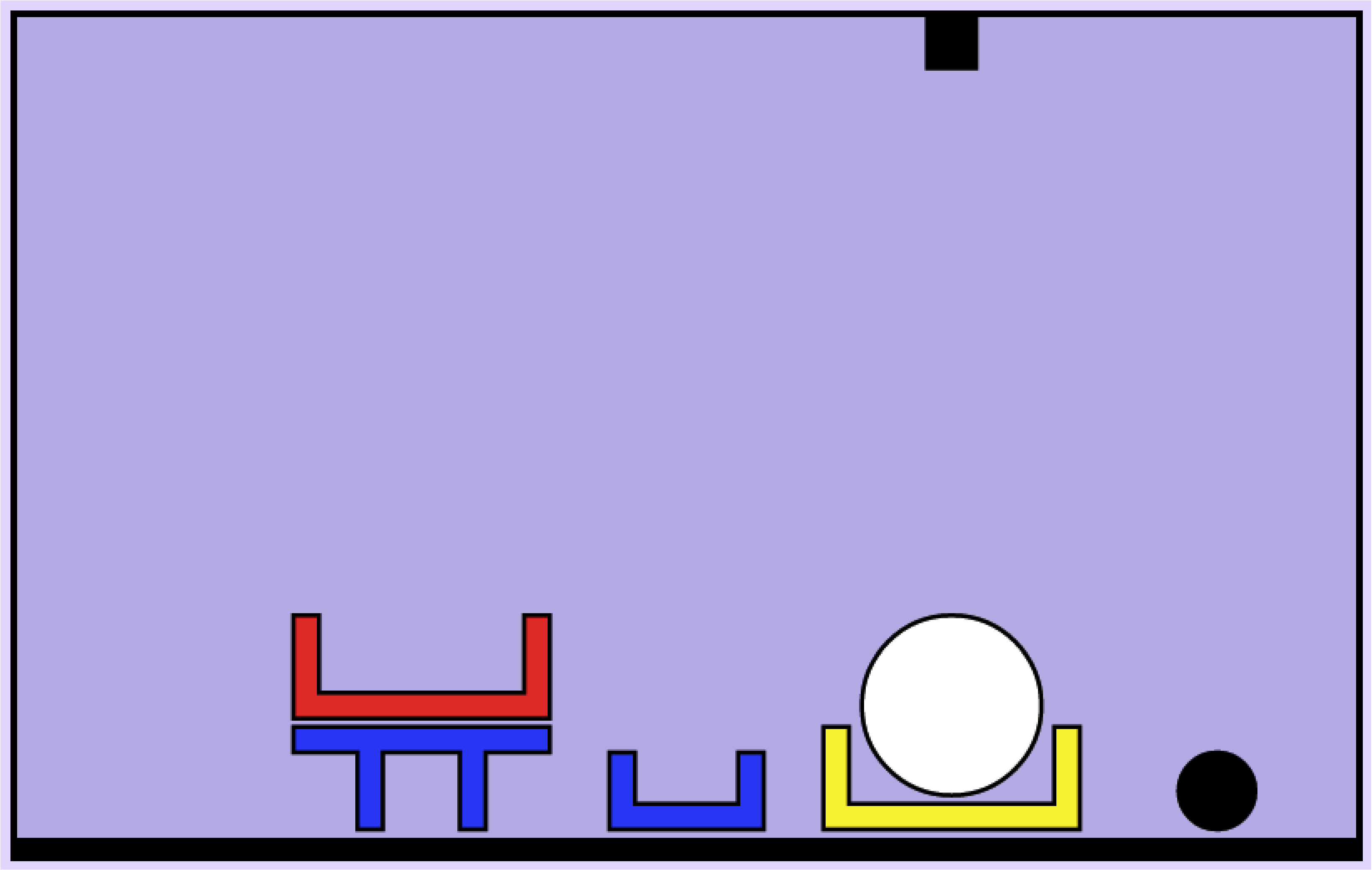
There is no ~~spoon~~ *white ball in a box*.

“put the white ball in a box on the floor”



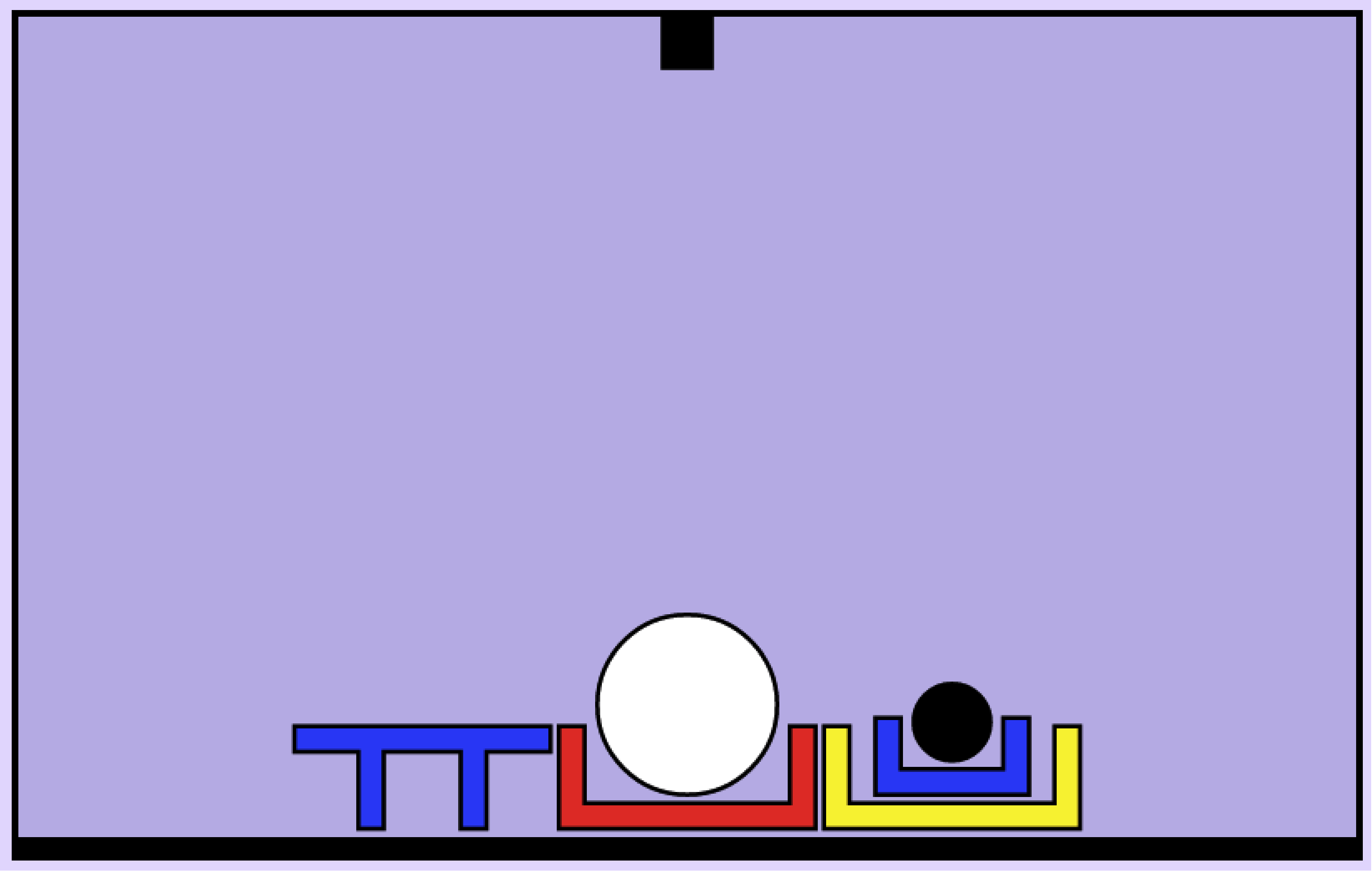
inside(WhiteBall, YellowBox)

Yellow box is already on floor: 3 moves



$\text{inside}(\text{WhiteBall}, \text{RedBox}) \land \text{on}(\text{RedBox}, \text{floor})$

Red box can be placed on floor first: 2 moves



FINAL INTERPRETATION

$\text{inside}(\text{WhiteBall}, \text{YellowBox}) \lor (\text{inside}(\text{WhiteBall}, \text{RedBox}) \land \text{on}(\text{RedBox}, \text{floor}))$

PHYSICAL LAWS

- Balls must be in boxes or on the floor, otherwise they roll away.
- Small objects cannot support large objects.
- …

INTERPRETER TEST CASES

Each test case contains a *list of interpretations*
Each interpretation is already a list (a disunction of conjunctions)

```
  world: "small",
  utterance: "take a blue object",
  interpretations: [["holding(BlueTable)","holding(BlueBox)"]]
}
```

```
  world: "small",
  utterance: "put a black ball in a box on the floor",
  interpretations: [["inside(BlackBall,YellowBox)"],
                    ["ontop(BlackBall,floor)"]]
}
```

CONJUNCTION

```
  world: "small",
  utterance: "put all balls on the floor",
  interpretations: [["ontop(WhiteBall,floor) & ontop(BlackBall,floor)"]]
}
```

NO VALID INTERPRETATIONS

```
  world: "small",
  utterance: "put a ball on a table",
  interpretations: []
}
```

Breaks the laws of nature!

SOME INTERPRETATIONS ARE MISSING

```
  world: "small",
  utterance: "put a ball in a box on the floor",
  interpretations: [["COME UP WITH YOUR OWN INTERPRETATION"]]
}
```

TIPS FOR INTERPRETER IN SHRDLITE

- Sub-functions based on grammar types
- Use `instanceof` when traversing parse tree (`Command`)
- Use recursion to handle nesting
  “put a box in a box on a table on the floor”

AMBIGUITY RESOLUTION

Handling multiple interpretations

- Fail
- Pick “first”
- Use some rules of thumb
  e.g. prefer box already on floor
- Ask the user for clarification (extension)

PLANNING

goal → robot movements

- Movements: *left, right, pick, drop*
- Use graph search
- Given a disjunction of goals, should find the easiest to satisfy

AUDIENCE PARTICIPATION META-QUESTION

Do you prefer Socrative or post-it notes?

Thank you for returning your post-it notes!

CHAPTERS 4–5: NON-CLASSICAL AND ADVERSARIAL SEARCH

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

21 April, 2017

TABLE OF CONTENTS

REPETITION

UNINFORMED SEARCH (R&N 3.4)

Search problems, graphs, states, arcs, goal test, generic search algorithm, tree search, graph search, depth-first search, breadth-first search, uniform cost search, iterative deepending, bidirectional search, …

HEURISTIC SEARCH (R&N 3.5–3.6)

Greedy best-first search, A* search, heuristics, admissibility, consistency, dominating heuristics, …

LOCAL SEARCH (R&N 4.1)

Hill climbing / gradient descent, random moves, random restarts, beam search, simulated annealing, …

NON-CLASSICAL SEARCH

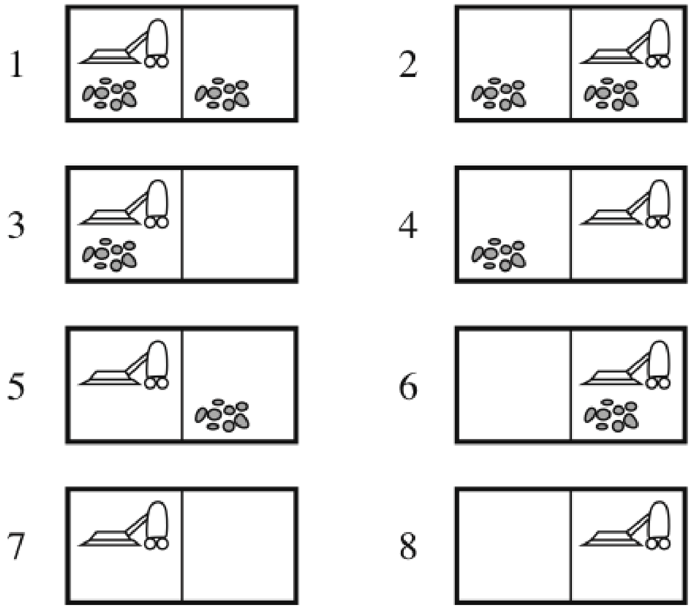
NONDETERMINISTIC SEARCH (R&N 4.3)

PARTIAL OBSERVATIONS (R&N 4.4)

NONDETERMINISTIC SEARCH (R&N 4.3)

- Contingency plan (strategy)
- ***And-or*** search trees
- And-or graph search algorithm

THE VACUUM CLEANER WORLD, AGAIN



The eight possible states of the vacuum world; states 7 and 8 are goal states.

There are three actions: *Left, Right, Suck*

AN ERRATIC VACUUM CLEANER

Assume that the *Suck* action works as follows:

- if the square is dirty, it is cleaned but sometimes also the adjacent square is
- if the square is clean, the vacuum cleaner sometimes deposists dirt

Now we need a more general *result* function:

- instead of returning a single state, it returns a set of possible outcome states
- e.g., $\text{Results}(\text{Suck}, 1) = \{5, 7\}$ and $\text{Results}(\text{Suck}, 5) = \{1, 5\}$

We also need to generalise the notion of a *solution*:

- instead of a single sequence (path) from the start to the goal, we need a *strategy* (or a *contingency plan*)
- i.e., we need **if-then-else** constructs
- this is a possible solution from state 1:
  - [*Suck*, if *State*=5 then [*Right*, *Suck*] else []]

HOW TO FIND CONTINGENCY PLANS

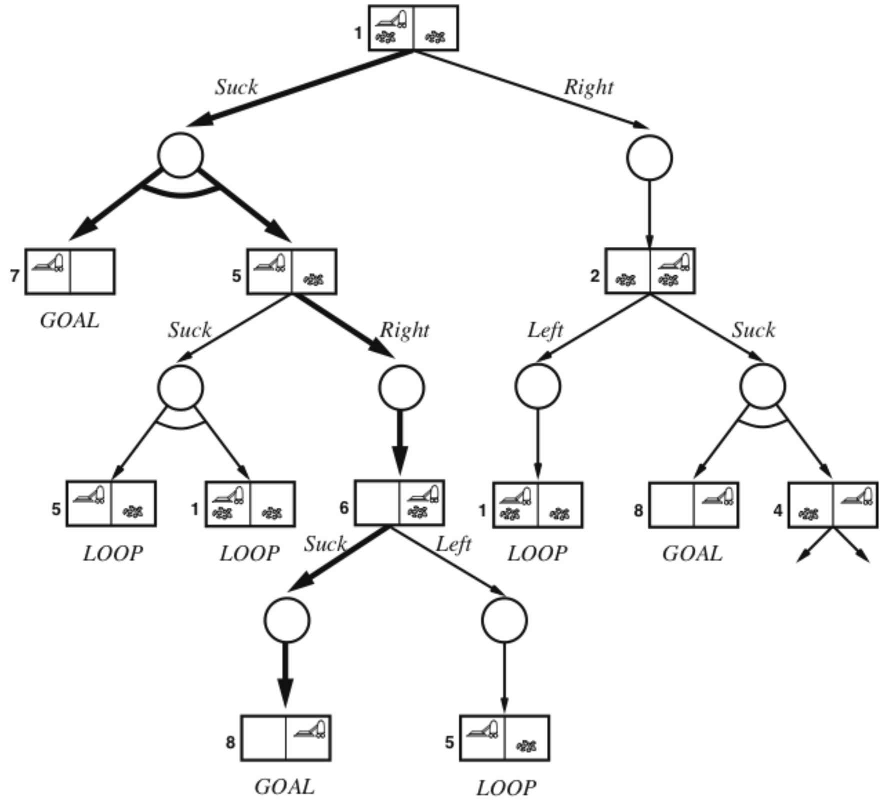
We need a new kind of nodes in the search tree:

- ***and*** *nodes*:
  these are used whenever an action is nondeterministic
- normal nodes are called ***or*** *nodes*:
  they are used when we have several possible actions in a state

A solution for an ***and-or*** search problem is a subtree that:

- has a goal node at every leaf
- specifies exactly one action at each of its ***or*** *node*
- includes every branch at each of its ***and*** *node*

A SOLUTION TO THE ERRATIC VACUUM CLEANER



The solution subtree is shown in bold, and corresponds to the plan:
[*Suck*, if *State*=5 then [*Right*, *Suck*] else []]

AN ALGORITHM FOR FINDING A CONTINGENCY PLAN

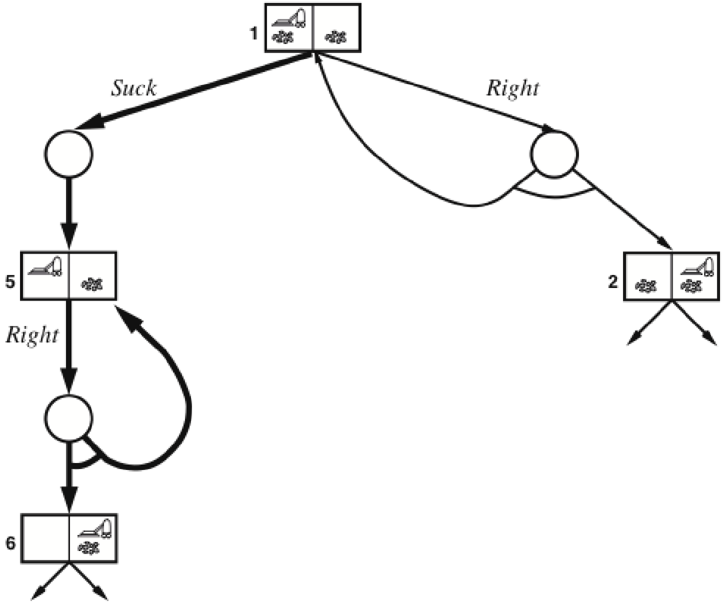
This algorithm does a depth-first search in the *and-or* tree,
so it is not guaranteed to find the best or shortest plan:

```
function AndOrGraphSearch(problem):
    return OrSearch(problem.InitialState, problem, [])

function OrSearch(state, problem, path):
    if problem.GoalTest(state) then return []
    if state is on path then return failure
    for each action in problem.Actions(state):
        plan := AndSearch(problem.Results(state, action), problem, [state] ++ path)
        if plan ≠ failure then return [action] ++ plan
    return failure

function AndSearch(states, problem, path):
    for each $s_i$ in states:
        $plan_i$ := OrSearch($s_i$, problem, path)
        if $plan_i$ = failure then return failure
    return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else … if $s_n$ then $plan_n$]
```

WHILE LOOPS IN CONTINGENCY PLANS



If the search graph contains cycles, **if-then-else** is not enough in a contingency plan:

- we need **while** loops instead

In the slippery vacuum world above, the cleaner don't always move when told:

- the solution is a sub-graph (not a subtree), shown in bold above
- this solution translates to [*Suck*, while *State*=5 do *Right*, *Suck*]

PARTIAL OBSERVATIONS (R&N 4.4)

- Belief states: goal test, transitions, …
- Sensor-less (conformant) problems
- Partially observable problems

OBSERVABILITY VS DETERMINISM

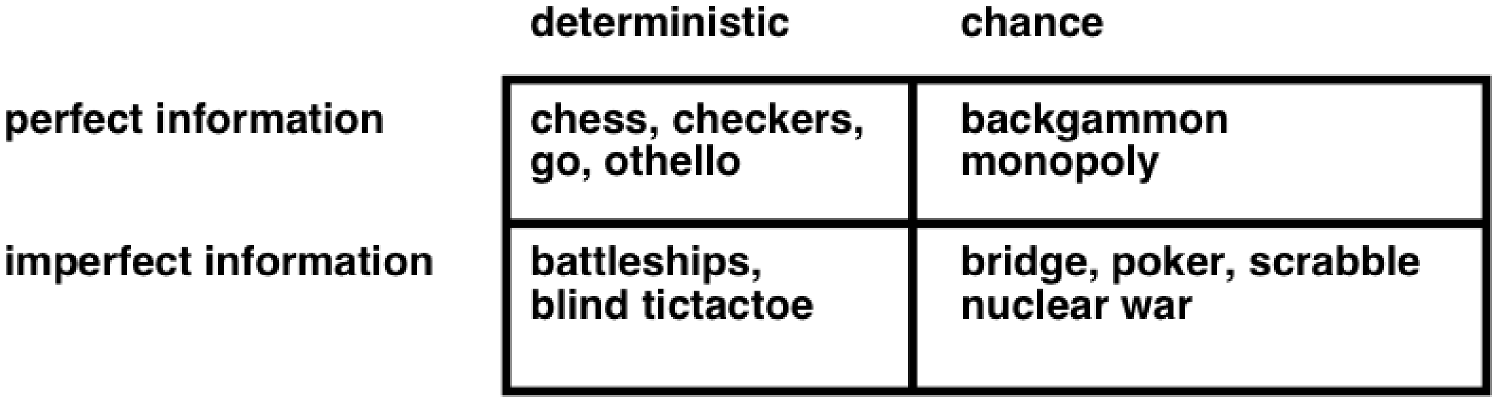
A problem is *nondeterministic* if there are several possible outcomes of an action

- deterministic — nondeterministic (chance)

It is *partially observable* if the agent cannot tell exactly which state it is in

- fully observable (perfect info.) — partially observable (imperfect info.)

A problem can be either nondeterministic, or partially observable, or both:



	deterministic	**chance**
perfect information	**chess, checkers, go, othello**	**backgammon monopoly**
imperfect information	**battleships, blind tictactoe**	**bridge, poker, scrabble nuclear war**

BELIEF STATES

Instead of searching in a graph of states, we use *belief states*

- A belief state is a *set of states*

In a sensor-less (or conformant) problem, the agent has *no information at all*

- The initial belief state is the set of all problem states
    - e.g., for the vacuum world the initial state is {1,2,3,4,5,6,7,8}

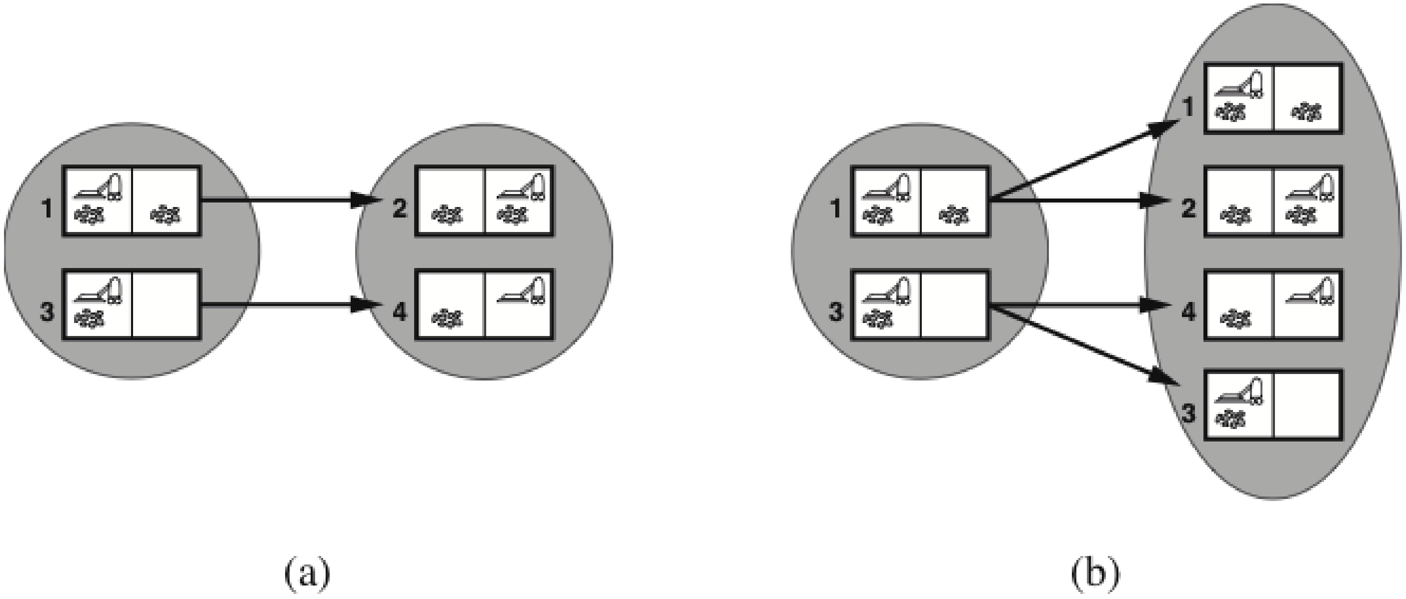
The goal test has to check that *all* members in the belief state is a goal

- e.g., for the vacuum world, the following are goal states: {7}, {8}, and {7,8}

The result of performing an action is the *union* of all possible results

- i.e., $\text{Predict}(b, a) = \{\text{Result}(s, a)$ for each $s \in b\}$
- if the problem is also nondeterministic:
    - $\text{Predict}(b, a) = \bigcup\{\text{Results}(s, a)$ for each $s \in b\}$

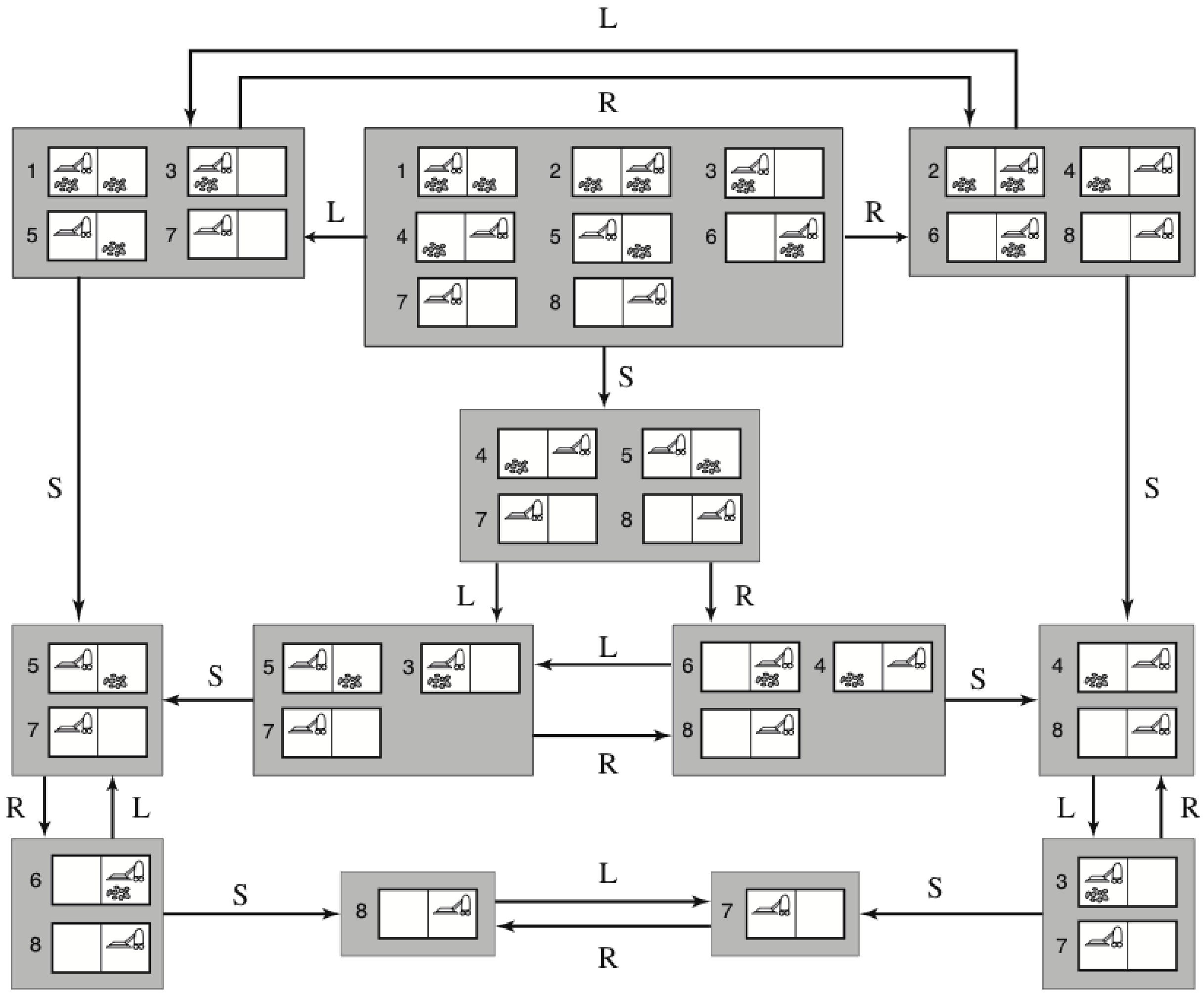
PREDICTING BELIEF STATES IN THE VACUUM WORLD



(a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*.

(b) Prediction for the same belief state and action in the nondeterministic slippery version of the sensorless vacuum world.

THE DETERMINISTIC SENSORLESS VACUUM WORLD



PARTIAL OBSERVATIONS: STATE TRANSITIONS

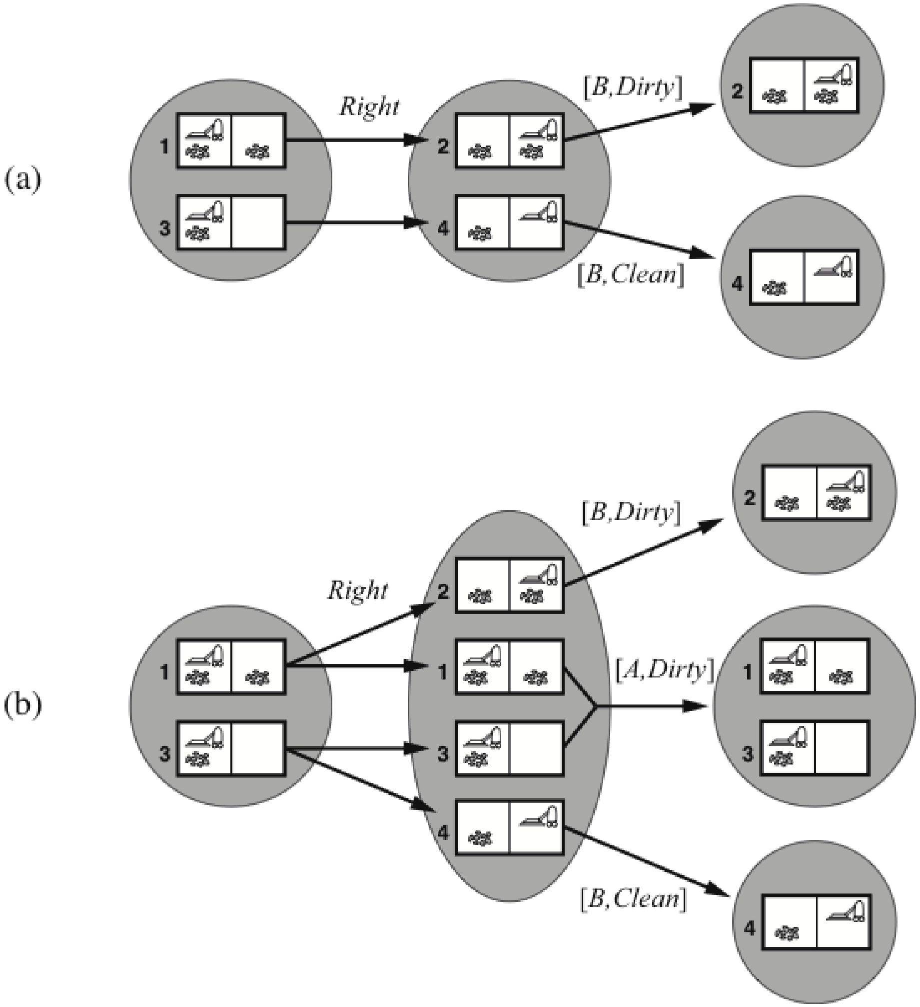
With partial observations, we can think of belief state transitions in three stages:

- **Prediction**, the same as for sensorless problems:
  - $b' = \text{Predict}(b, a) = \{\text{Result}(s, a) \text{ for each } s \in b\}$
- **Observation prediction**, determines the percepts that can be observed:
  - $\text{PossiblePercepts}(b') = \{\text{Percept}(s) \text{ for each } s \in b'\}$
- **Update**, filters the predicted states according to the percepts:
  - $\text{Update}(b', o) = \{s \text{ for each } s \in b' \text{ such that } o = \text{Percept}(s)\}$

Belief state transitions:

- $\text{Results}(b, a) = \{\text{Update}(b', o) \text{ for each } o \in \text{PossiblePercepts}(b')\}$
  where $b' = \text{Predict}(b, a)$

TRANSITIONS IN PARTIALLY OBSERVABLE VACUUM WORLDS

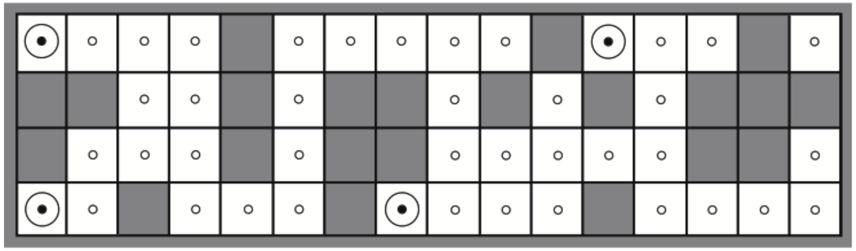


The percepts return the current position and the dirtyness of that square.

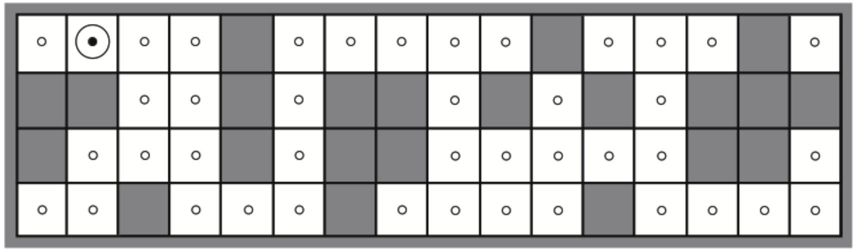
(a) The deterministic world: *Right* always succeeds.

(b) The slippery world: *Right* sometimes fails.

EXAMPLE: ROBOT LOCALISATION



(a) Possible locations of robot after $E_1 = NSW$



(b) Possible locations of robot After $E_1 = NSW$, $E_2 = NS$

The percepts return if there is a wall in each of the directions.

(a) Possible initial positions of the robot, after one observation.

(b) After moving right and a new observation, there is only one possible position left.

ADVERSARIAL SEARCH

TYPES OF GAMES (R&N 5.1)

MINIMAX SEARCH (R&N 5.2–5.3)

IMPERFECT DECISIONS (R&N 5.4–5.4.2)

STOCHASTIC GAMES (R&N 5.5)

TYPES OF GAMES (R&N 5.1)

- cooperative, competetive, zero-sum games
- game trees, ply/plies, utility functions

MULTIPLE AGENTS

Let’s consider problems with multiple agents, where:

- the agents select actions autonomously
- each agent has its own information state
    - they can have different information (even conflicting)
- the outcome depends on the actions of all agents
- each agent has its own utility function (that depends on the total outcome)

TYPES OF AGENTS

There are two extremes of multiagent systems:

- **Cooperative**: The agents share the same utility function
    - *Example*: Automatic trucks in a warehouse
- **Competetive**: When one agent wins all other agents lose
    - A common special case is when $\sum_a u_a(o) = 0$ for any outcome o. This is called a zero-sum game.
    - *Example*: Most board games

Many multiagent systems are between these two extremes.

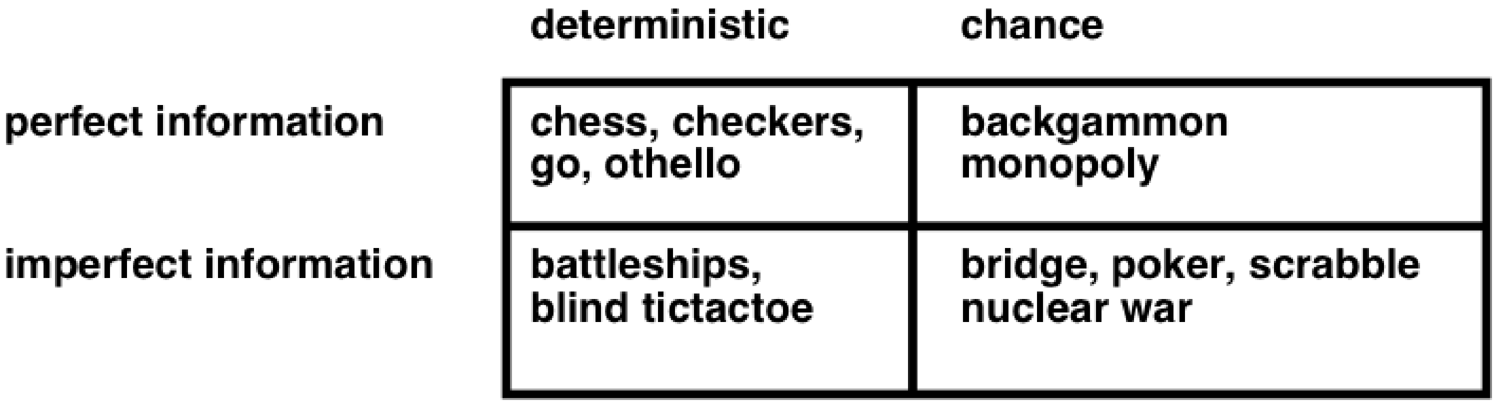
- *Example*: Long-distance bike races are usually both cooperative (bikers usually form clusters where they take turns in leading a group), and competetive (only one of them can win in the end).

GAMES AS SEARCH PROBLEMS

The main difference to chapters 3–4:
now we have more than one agent that have different goals.

- All possible game sequences are represented in a game tree.
- The nodes are states of the game, e.g. board positions in chess.
- Initial state (root) and terminal nodes (leaves).
- States are connected if there is a legal move/ply.
  (a ply is a move by one player, i.e., one layer in the game tree)
- Utility function (payoff function). Terminal nodes have utility values $+x$ (player 1 wins), $-x$ (player 2 wins) and 0 (draw).

TYPES OF GAMES (AGAIN)



	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

PERFECT INFORMATION GAMES: ZERO-SUM GAMES

Perfect information games are solvable in a manner similar to
fully observable single-agent systems, e.g., using forward search.

If two agents are competing so that a positive reward for one is a negative reward for the other agent, we have a two-agent *zero-sum game*.

The value of a game zero-sum game can be characterized by a single number that one agent is trying to maximize and the other agent is trying to minimize.

This leads to a *minimax strategy*:

- A node is either a MAX node (if it is controlled by the maximising agent),
- or is a MIN node (if it is controlled by the minimising agent).

MINIMAX SEARCH (R&N 5.2–5.3)

- Minimax algorithm
- α-β pruning

MINIMAX SEARCH FOR ZERO-SUM GAMES

Given two players called MAX and MIN:

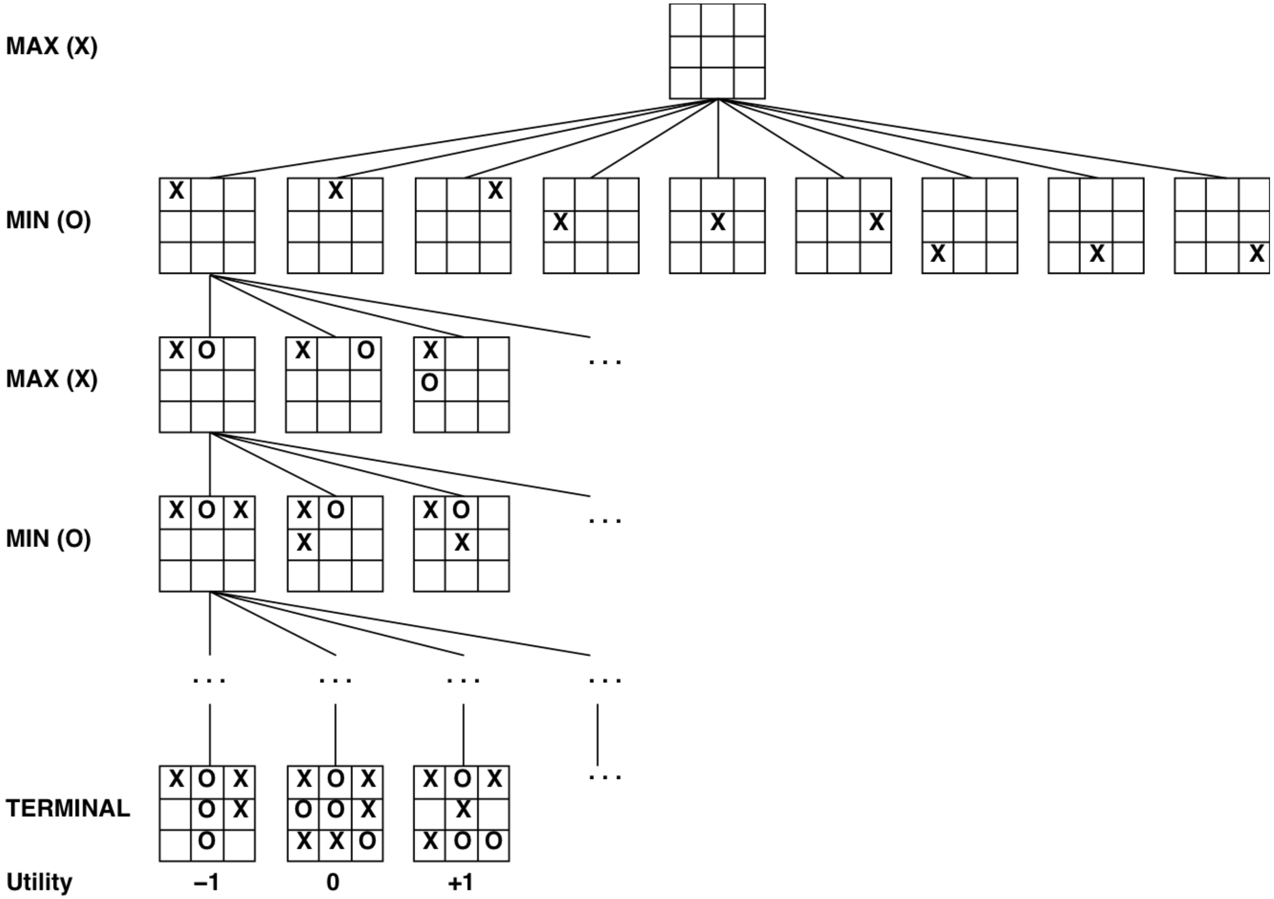
- MAX wants to maximize the utility value,
- MIN wants to minimize the same value.

⇒ MAX should choose the alternative that maximizes assuming that MIN minimizes.

Minimax gives perfect play for deterministic, perfect-information games:

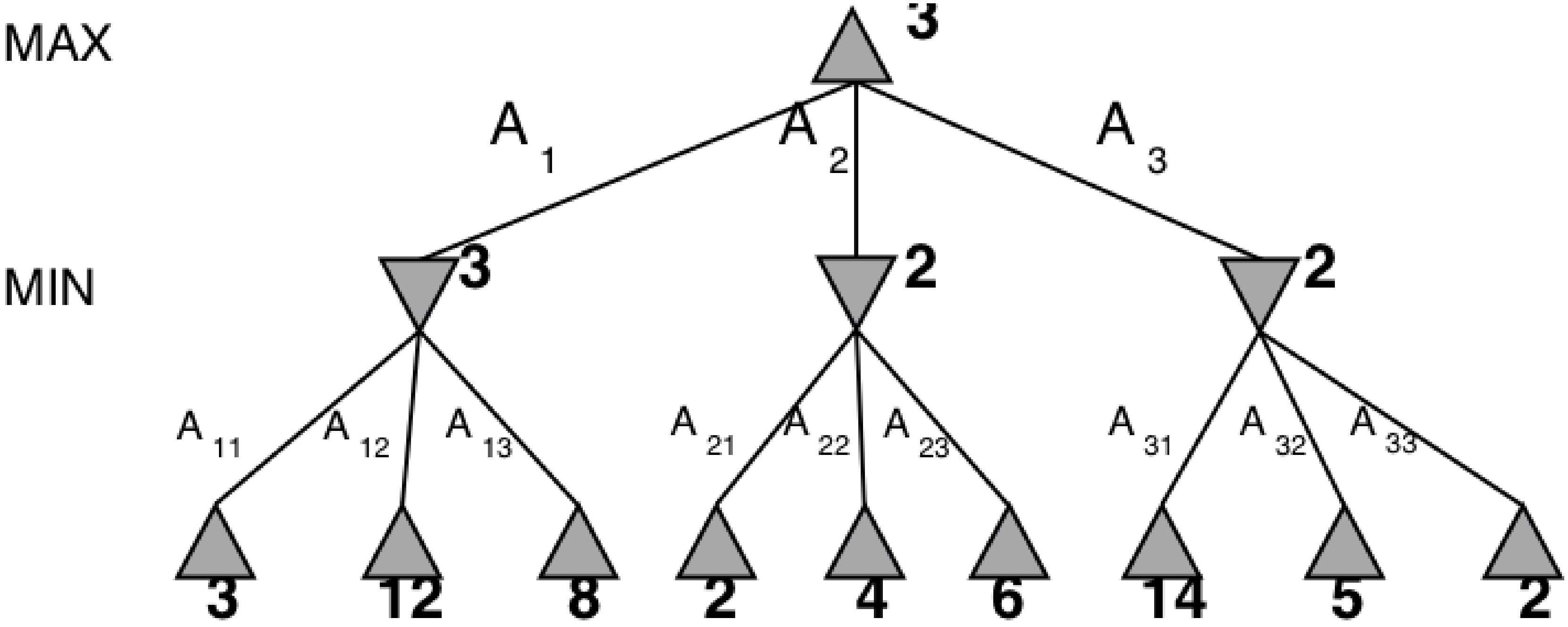
```
function Minimax(state):
    if TerminalTest(state) then return Utility(state)
    $A$ := Actions(state)
    if state is a MAX node then return $\max_{a \in A}$ Minimax(Result(state, a))
    if state is a MIN node then return $\min_{a \in A}$ Minimax(Result(state, a))
```

MINIMAX SEARCH: TIC-TAC-TOE



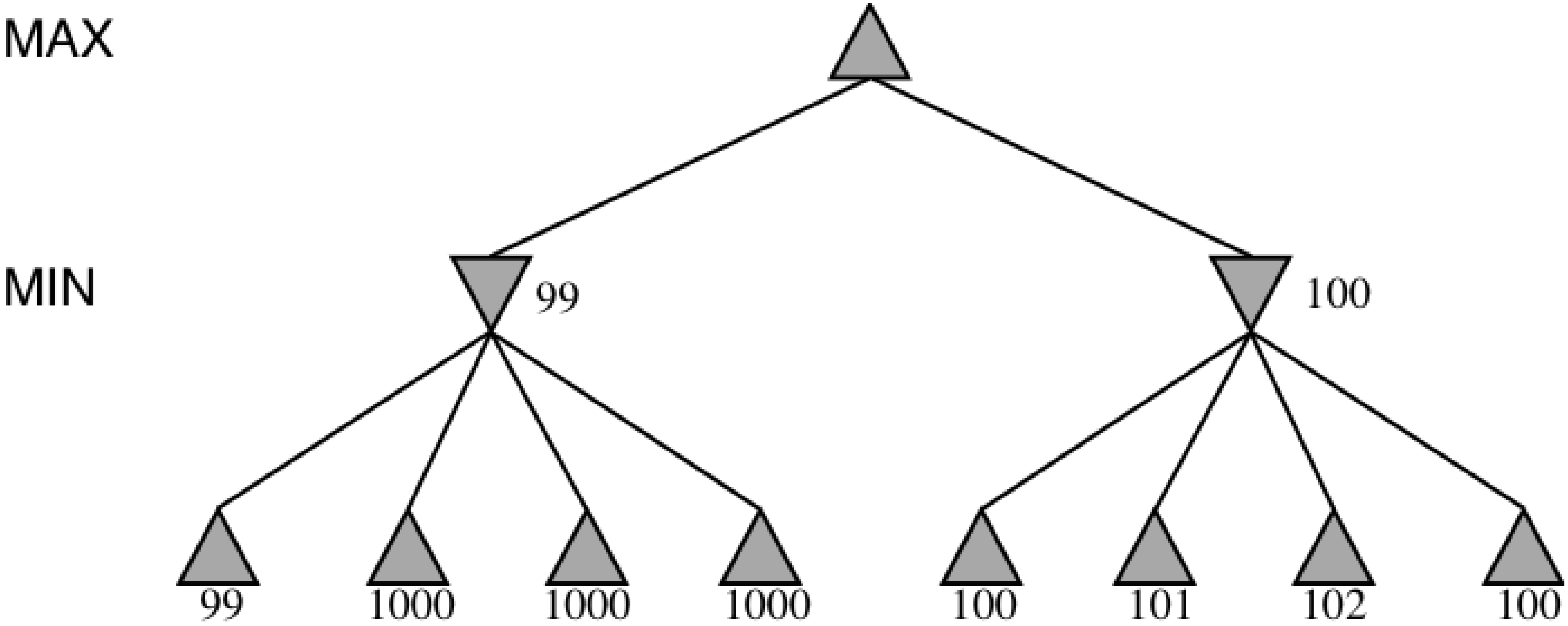
MINIMAX EXAMPLE

The Minimax algorithm gives perfect play for deterministic, perfect-information games.



CAN MINIMAX BE WRONG?

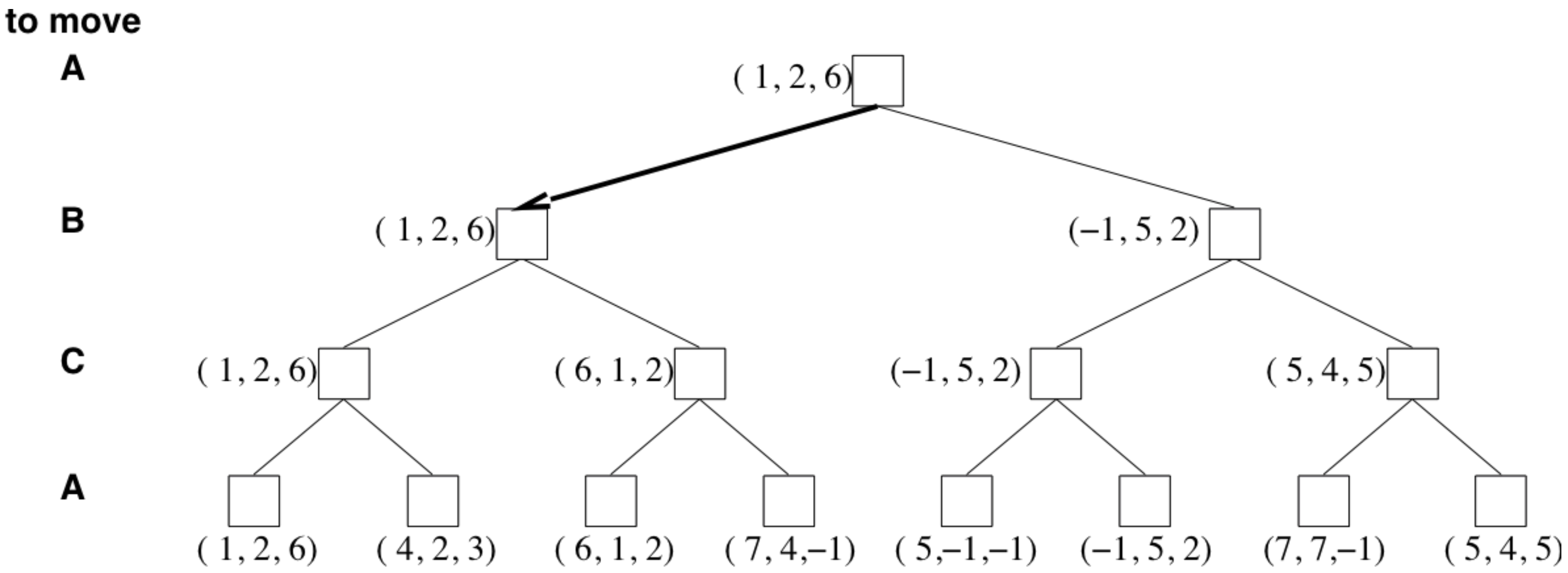
Minimax gives perfect play, but is that always the best strategy?



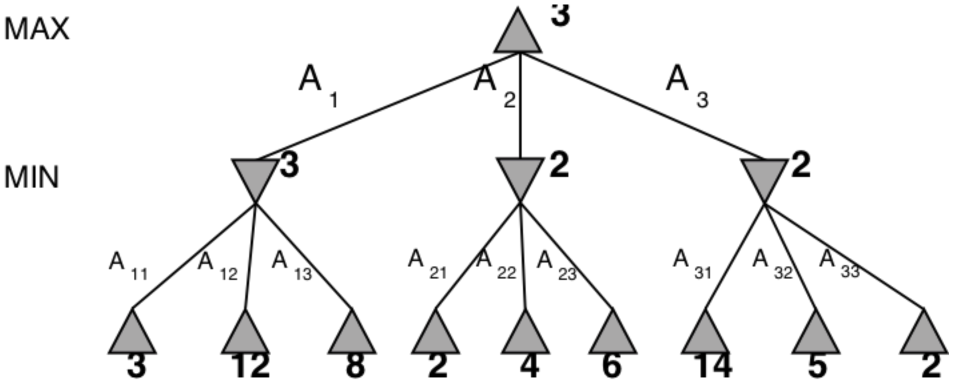
Perfect play assumes that the opponent is also a perfect player!

3-PLAYER MINIMAX

Minimax can also be used on multiplayer games



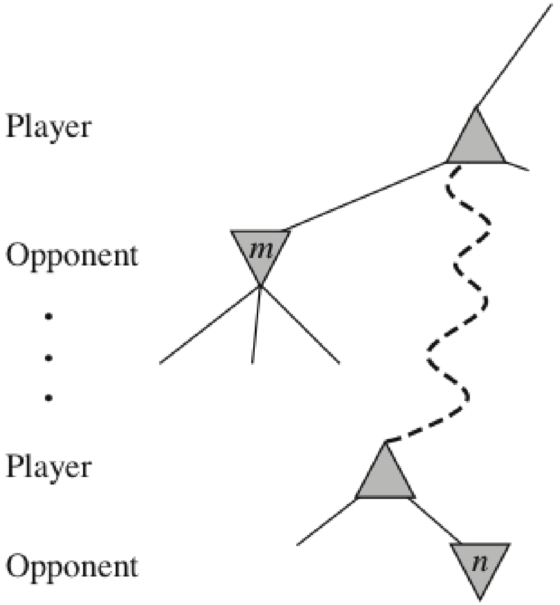
α–β PRUNING



$$
\begin{aligned}
\text{Minimax}(\textit{root}) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \quad \text{where } z \le 2 \\
&= 3
\end{aligned}
$$

I.e., we don't need to know the values of x and y!

α–β PRUNING, GENERAL IDEA

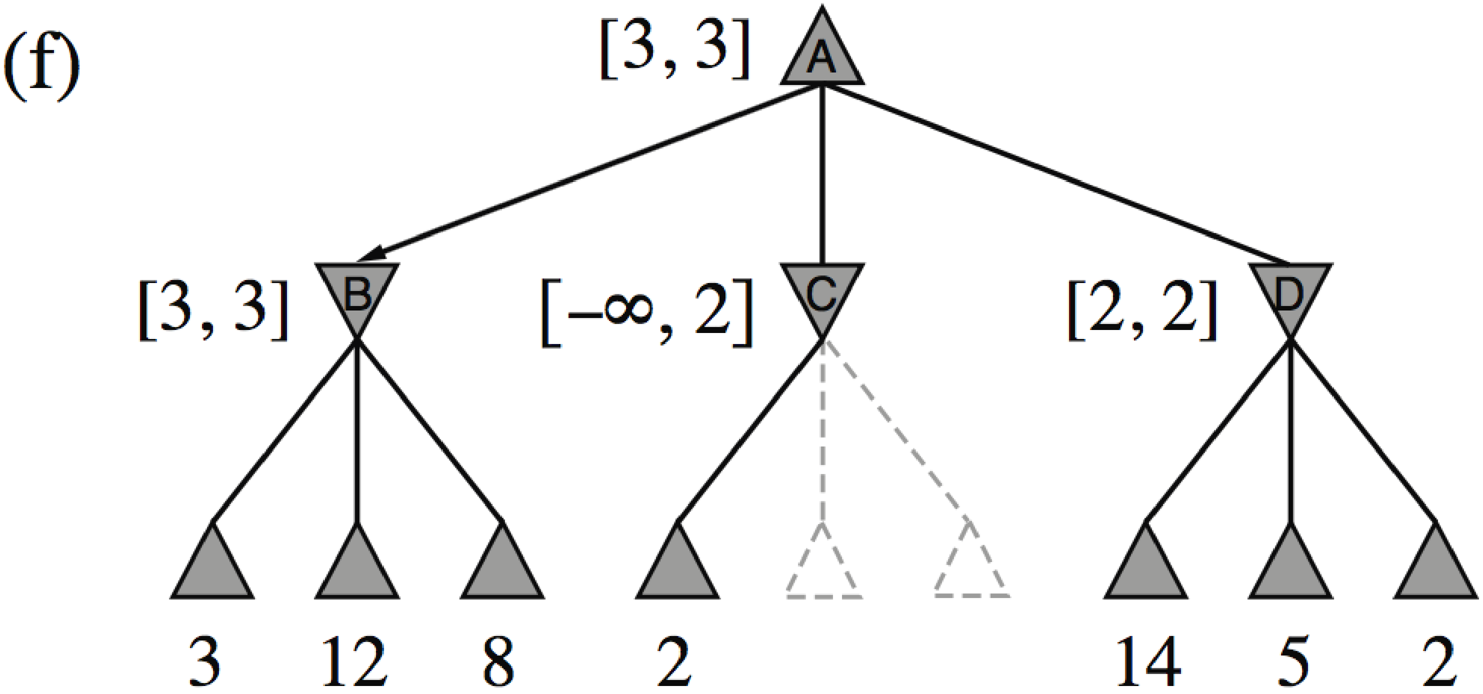


The general idea of α-β pruning is this:

- if *m* is better than *n* for Player, we don't want to pursue *n*
- so, once we know enough about *n* we can prune it
- sometimes it's enough to examine just one of *n*'s descendants

α-β pruning keeps track of the possible range of values for every node it visits;
the parent range is updated when the child has been visited.

MINIMAX EXAMPLE, WITH α—β PRUNING



THE α–β ALGORITHM

```
function AlphaBetaSearch(state):
    $v$ := MaxValue(state, $-\infty$, $+\infty$))
    return the action in Actions(state) that has value $v$

function MaxValue(state, $\alpha$, $\beta$):
    if TerminalTest(state) then return Utility(state)
    $v$ := $-\infty$
    for each action in Actions(state):
        $v$ := max($v$, MinValue(Result(state, action), $\alpha$, $\beta$))
        if $v \geq \beta$ then return $v$
        $\alpha$ := max($\alpha$, $v$)
    return $v$

function MinValue(state, $\alpha$, $\beta$):
    same as MaxValue but reverse the roles of $\alpha/\beta$ and min/max and $-\infty/+\infty$
```

HOW EFFICIENT IS $\alpha-\beta$ PRUNING?

The amount of pruning provided by the α-β algorithm depends on the ordering of the children of each node.

- It works best if a highest-valued child of a MAX node is selected first and if a lowest-valued child of a MIN node is returned first.
- In real games, much of the effort is made to optimise the search order.
- With a “perfect ordering”, the time complexity becomes $O(b^{m/2})$
    - this doubles the solvable search depth
    - however, $35^{80/2}$ (for chess) or $250^{160/2}$ (for go) is still impossible…

MINIMAX AND REAL GAMES

Most real games are too big to carry out minimax search, even with α-β pruning.

- For these games, instead of stopping at leaf nodes,
  we have to use a cutoff test to decide when to stop.
- The value returned at the node where the algorithm stops
  is an estimate of the value for this node.
- The function used to estimate the value is an evaluation function.
- Much work goes into finding good evaluation functions.
- There is a trade-off between the amount of computation required
  to compute the evaluation function and the size of the search space
  that can be explored in any given time.

IMPERFECT DECISIONS (R&N 5.4–5.4.2)

STOCHASTIC GAMES (R&N 5.5)

Note: these two sections were presented Tuesday 25th April!

CHAPTER 6: SEARCH PART IV, AND CONSTRAINT SATISFACTION PROBLEMS, PART II

DIT410/TIN174, Artificial Intelligence

Peter Ljunglöf

25 April, 2017

TABLE OF CONTENTS

REPETITION OF SEARCH

CLASSICAL SEARCH (R&N 3.1–3.6)

Generic search algorithm, tree search, graph search, depth-first search, breadth-first search, uniform cost search, iterative deepending, bidirectional search, greedy best-first search, A* search, heuristics, admissibility, consistency, dominating heuristics, …

NON-CLASSICAL SEARCH (R&N 4.1, 4.3–4.4)

Hill climbing, random moves, random restarts, beam search, nondeterministic actions, contingency plan, and-or search trees, partial observations, belief states, sensor-less problems, …

ADVERSARIAL SEARCH (R&N 5.1–5.3)

Cooperative, competetive, zero-sum games, game trees, minimax, α-β pruning, …

MORE GAMES

IMPERFECT DECISIONS (R&N 5.4–5.4.2)

STOCHASTIC GAMES (R&N 5.5)

IMPERFECT DECISIONS (R&N 5.4–5.4.2)

- H-minimax algorithm
- evaluation function, cutoff test
- features, weighted linear function
- quiescence search, horizon effect

REPETITION: MINIMAX SEARCH FOR ZERO-SUM GAMES

Given two players called MAX and MIN:

- MAX wants to maximize the utility value,
- MIN wants to minimize the same value.

⇒ MAX should choose the alternative that maximizes assuming that MIN minimizes.

```
function Minimax($state$):
    if TerminalTest($state$) then return Utility($state$)
    $A$ := Actions($state$)
    if $state$ is a MAX node then return $\max_{a \in A}$ Minimax(Result($state$, $a$))
    if $state$ is a MIN node then return $\min_{a \in A}$ Minimax(Result($state$, $a$))
```

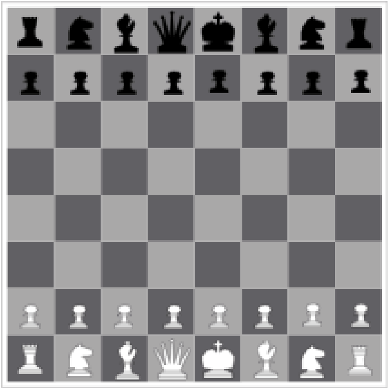
H-MINIMAX ALGORITHM

The *Heuristic* Minimax algorithm is similar to normal Minimax

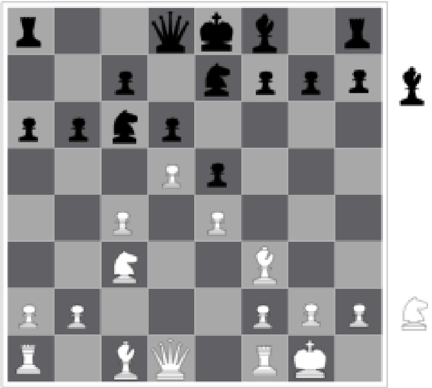
- it replaces **TerminalTest** and **Utility** with **CutoffTest** and **Eval**

```
function H-Minimax(state, depth):
    if CutoffTest(state, depth) then return Eval(state)
    $A$ := Actions(state)
    if state is a MAX node then return $\max_{a \in A}$ H-Minimax(Result(state, a), depth+1)
    if state is a MIN node then return $\min_{a \in A}$ H-Minimax(Result(state, a), depth+1)
```

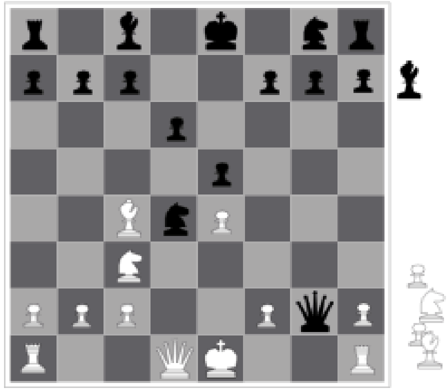
CHESS POSITIONS: HOW TO EVALUATE



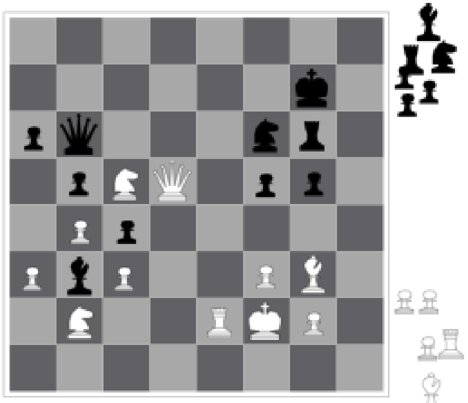
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning



(d) Black to move
White about to lose

WEIGHTED LINEAR EVALUATION FUNCTIONS

A very common evaluation function is to use a weighted sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

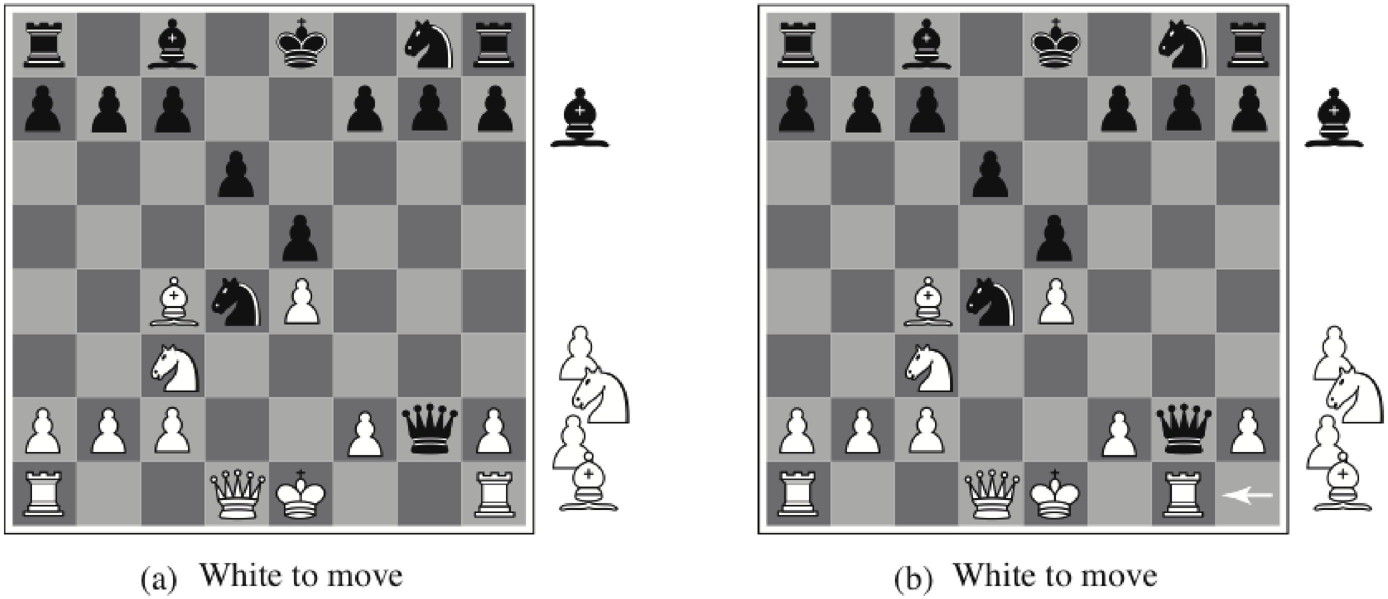
This relies on a strong assumption: all features are *independent of each other*

- which is usually not true, so the best programs for chess (and other games) also use nonlinear feature combinations

The weights can be calculated using machine learning algorithms, but a human still has to come up with the features.

- using recent advances in deep machine learning, the computer can learn the features too

EVALUATION FUNCTIONS



A naive weighted sum of features will not see the difference between these two states.

PROBLEMS WITH CUTOFF TESTS

Too simplistic cutoff tests and evaluation functions can be problematic:

- e.g., if the cutoff is only based on the current depth
- then it might cut off the search in unfortunate positions (such as (b) on the previous slide)

We want more sophisticated cutoff tests:

- only cut off search in *quiescent* positions
- i.e., in positions that are “stable”, unlikely to exhibit wild swings in value
- non-quiescent positions should be expanded further

Another problem is the *horizon effect*:

- if a bad position is unavoidable (e.g., loss of a piece), but the system can delay it from happening, it might push the bad position “over the horizon”
- in the end, the resulting delayed position might be even worse

DETERMINISTIC GAMES IN PRACTICE

Chess:

- DeepBlue (IBM) beats world champion Garry Kasparov, 1997.
- Modern chess programs: Houdini, Critter, Stockfish.

Checkers/Othello/Reversi:

- Logistello beats the world champion in Othello/Reversi, 1997.
- Chinook plays checkers perfectly, 2007. It uses an endgame database defining perfect play for all 8-piece positions on the board, (a total of 443,748,401,247 positions).

Go:

- AlphaGo (Google DeepMind) beats one of the world’s best players, Lee Sedol by 4–1, in April 2016.
- Modern programs: MoGo, Zen, GNU Go, AlphaGo.

GAMES OF IMPERFECT INFORMATION

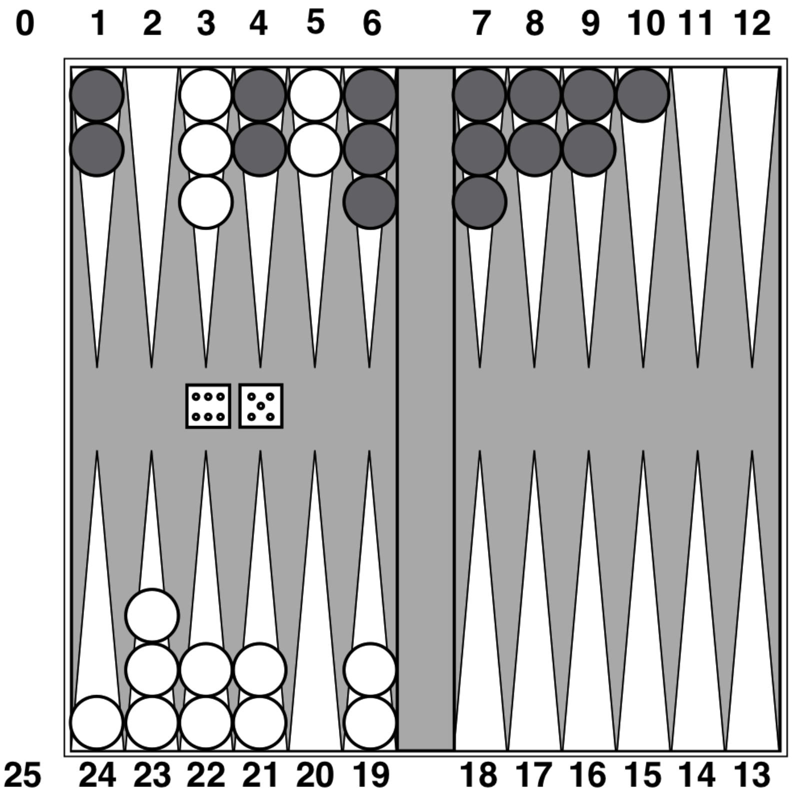
Imperfect information games

- e.g., card games, where the opponent’s initial cards are unknown
- typically we can calculate a probability for each possible deal
- seems just like having one big dice roll at the beginning of the game
- main idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals

STOCHASTIC GAMES (R&N 5.5)

- chance nodes
- expected value
- expecti-minimax algorithm

STOCHASTIC GAME EXAMPLE: BACKGAMMON

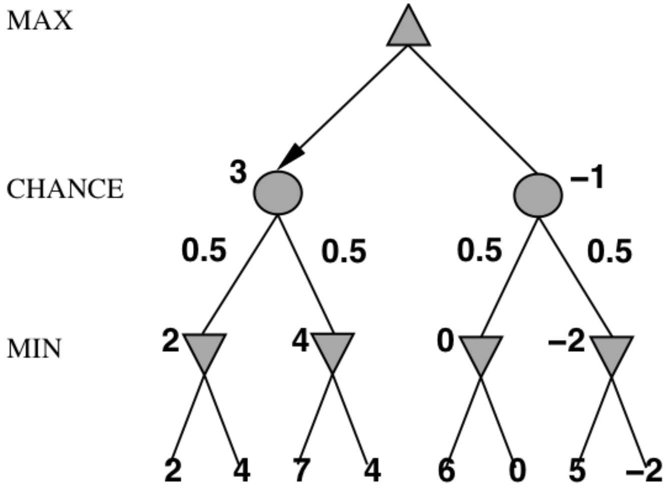


STOCHASTIC GAMES IN GENERAL

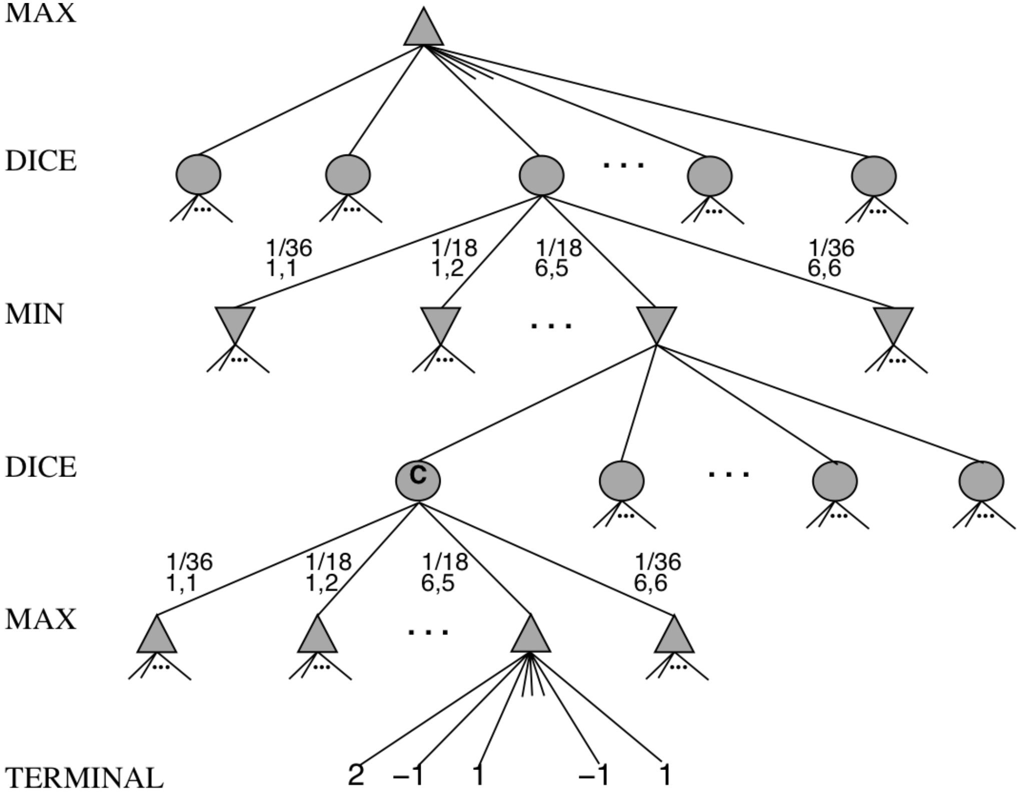
In stochastic games, chance is introduced by dice, card-shuffling, etc.

- We introduce *chance nodes* to the game tree.
- We can’t calculate a definite minimax value,
  instead we calculate the *expected value* of a position.
- The expected value is the average of all possible outcomes.

A very simple example with coin-flipping and arbitrary values:



BACKGAMMON GAME TREE



ALGORITHM FOR STOCHASTIC GAMES

The ExpectiMinimax algorithm gives perfect play;
it’s just like Minimax, except we must also handle chance nodes:

```
function ExpectiMinimax(state):
    if TerminalTest(state) then return Utility(state)
    $A$ := Actions(state)
    if state is a MAX node then return $\max_{a \in A}$ Minimax(state, $a$)
    if state is a MAX node then return $\min_{a \in A}$ Minimax(state, $a$)
    if state is a chance node then return $\sum_{a \in A} P(a)$ Minimax(state, $a$)
```

where $P(a)$ is the probability that action a occurs.

STOCHASTIC GAMES IN PRACTICE

Dice rolls increase the branching factor *b*:

- there are 21 possible rolls with 2 dice

Backgammon has ≈20 legal moves:

- depth $4 \Rightarrow 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$ nodes

As depth increases, the probability of reaching a given node shrinks:

- value of lookahead is diminished
- α-β pruning is much less effective

TDGammon (1995) used depth-2 search + very good Eval:

- the evaluation function was learned by self-play
- world-champion level

REPETITION OF CSP

CONSTRAINT SATISFACTION PROBLEMS (R&N 6.1)

Variables, domains, constraints (unary, binary, n-ary), constraint graph

CSP AS A SEARCH PROBLEM (R&N 6.3–6.3.2)

Backtracking search, heuristics (minimum remaining values, degree, least constraining value), forward checking, maintaining arc-consistency (MAC)

CONSTRAINT PROGAGATION (R&N 6.2–6.2.2)

Consistency (node, arc, path, k, …), global constratints, the AC-3 algorithm

CSP: CONSTRAINT SATISFACTION PROBLEMS (R&N 6.1)

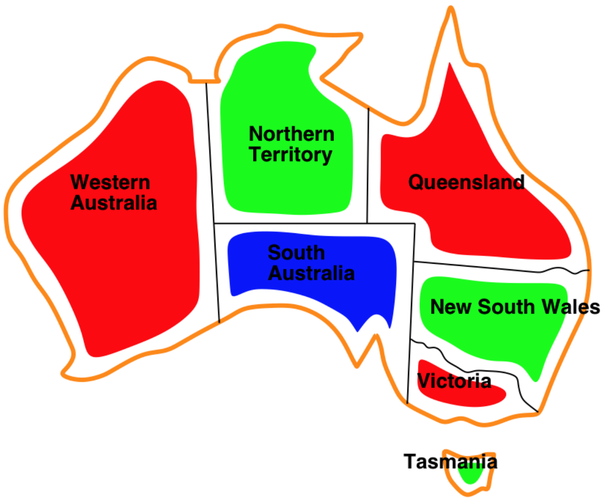
CSP is a specific kind of search problem:

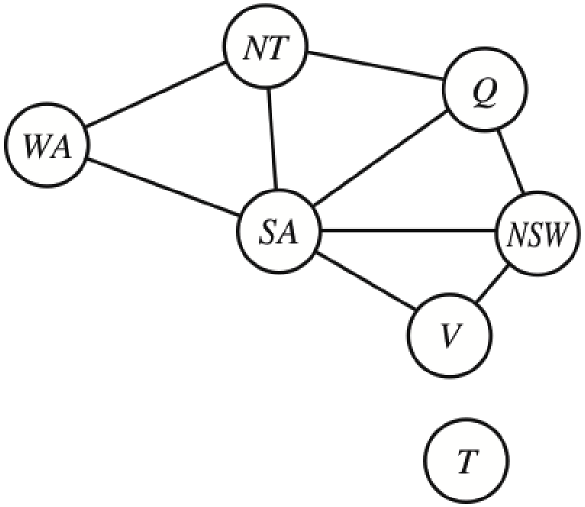
- the *state* is defined by *variables* X_i, each taking values from the domain D_i
- the *goal test* is a set of *constraints*:
  - each constraint specifies allowed values for a subset of variables
  - all constraints must be satisfied

Differences to general search problems:

- the path to a goal isn't important, only the solution is.
- there are no predefined starting state
- often these problems are huge, with thousands of variables, so systematically searching the space is infeasible

EXAMPLE: MAP COLOURING (BINARY CSP)

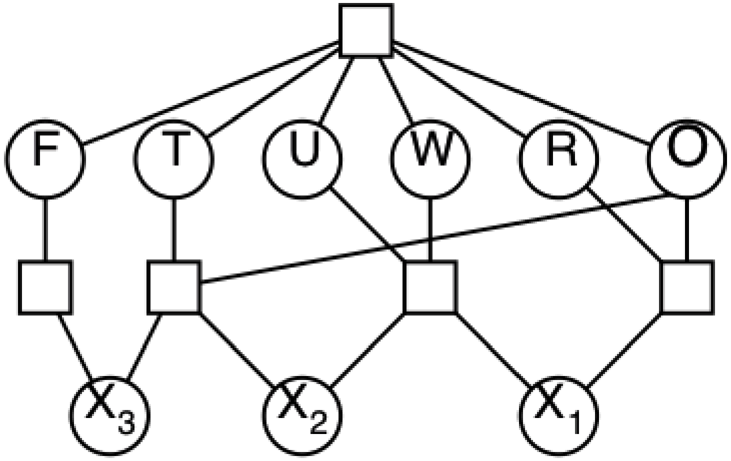




Variables:	WA, NT, Q, NSW, V, SA, T
Domains:	D_i = {red, green, blue}
Constraints:	$SA \neq WA$, $SA \neq NT$, $SA \neq Q$, $SA \neq NSW$, $SA \neq V$, $WA \neq NT$, $NT \neq Q$, $Q \neq NSW$, $NSW \neq V$
Constraint graph:	Every variable is a node, every binary constraint is an arc.

EXAMPLE: CRYPTARITHMETIC PUZZLE (HIGHER-ORDER CSP)

```
   T W O
 + T W O
 -------
 F O U R
```



Variables:	*F, T, U, W, R, O,* X_1, X_2, X_3
Domains:	D_i = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
Constraints:	*Alldiff*(*F,T,U,W,R,O*), *O*+*O*=*R*+10·X_1, etc.
Constraint graph:	This is not a binary CSP! The graph is a *constraint hypergraph*.

CSP AS A SEARCH PROBLEM (R&N 6.3–6.3.2)

- backtracking search
- select variable: minimum remaining values, degree heuristic
- order domain values: least constraining value
- inference: forward checking and arc consistency

ALGORITHM FOR BACKTRACKING SEARCH

At each depth level, decide on one single variable to assign:

- this gives branching factor $b = d$, so there are d^n leaves

Depth-first search with single-variable assignments is called *backtracking search*:

```
function BacktrackingSearch(csp):
    return Backtrack(csp, { })

function Backtrack(csp, assignment):
    if assignment is complete then return assignment
    var := SelectUnassignedVariable(csp, assignment)
    for each value in OrderDomainValues(csp, var, assignment):
        if value is consistent with assignment:
            inferences := Inference(csp, var, value)
            if inferences ≠ failure:
                result := Backtrack(csp, assignment ∪ {var=value} ∪ inferences)
                if result ≠ failure then return result
    return failure
```

IMPROVING BACKTRACKING EFFICIENCY

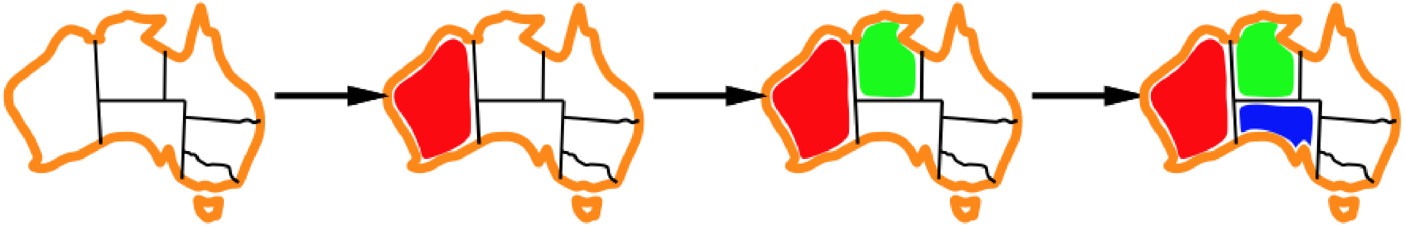
The general-purpose algorithm gives rise to several questions:

- Which variable should be assigned next?
    - SelectUnassignedVariable(*csp*, *assignment*)
- In what order should its values be tried?
    - OrderDomainValues(*csp*, *var*, *assignment*)
- What inferences should be performed at each step?
    - Inference(*csp*, *var*, *value*)
- Can the search avoid repeating failures?
    - Conflict-directed backjumping, constraint learning, no-good sets (R&N 6.3.3, not covered in this course)

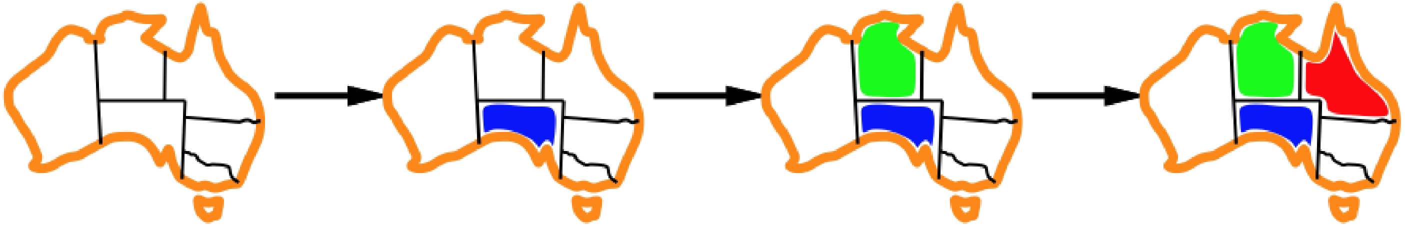
SELECTING UNASSIGNED VARIABLES

Heuristics for selecting the next unassigned variable:

- Minimum remaining values (MRV):
  $\Longrightarrow$ choose the variable with the fewest legal values



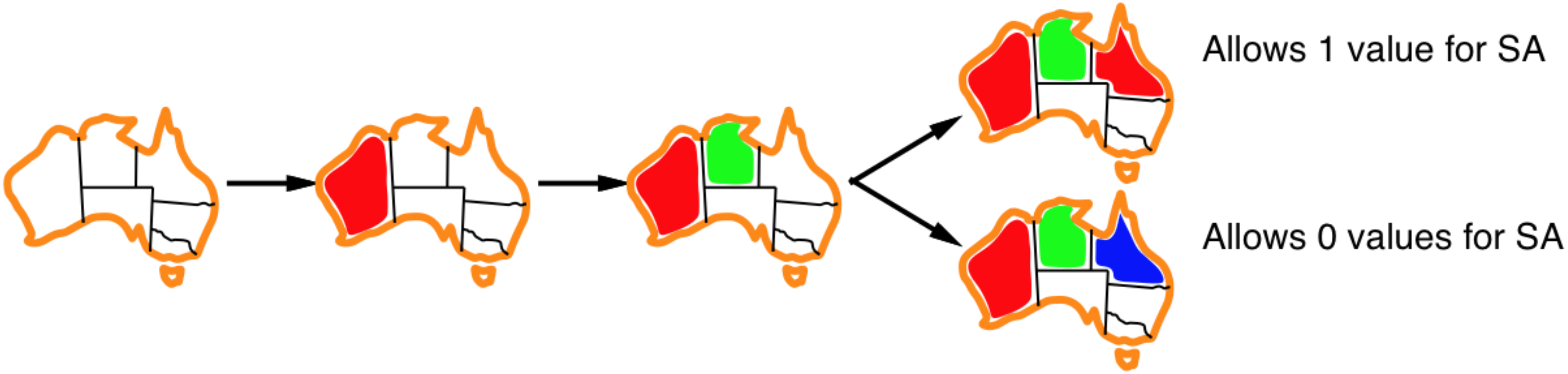
- Degree heuristic (if there are several MRV variables):
  $\Longrightarrow$ choose the variable with most constraints on remaining variables



ORDERING DOMAIN VALUES

Heuristics for ordering the values of a selected variable:

- Least constraining value:
  $\Longrightarrow$ prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph



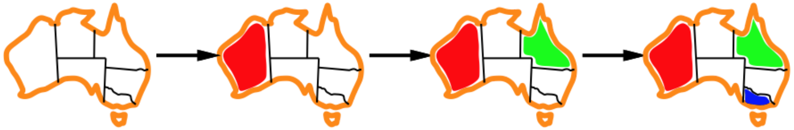
CONSTRAINT PROGAGATION (R&N 6.2–6.2.2)

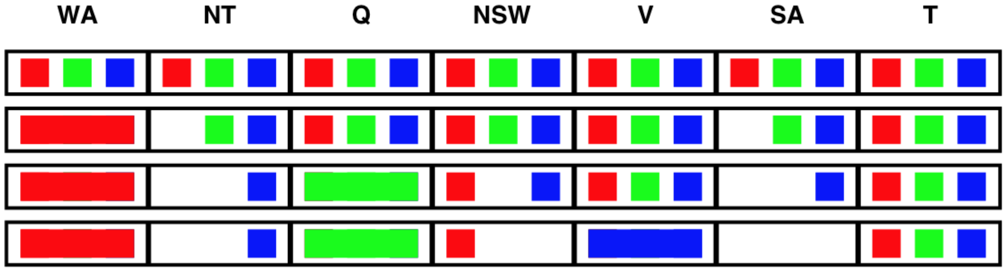
- consistency (node, arc, path, *k*, …)
- global constratints
- the AC-3 algorithm
- maintaining arc consistency

INFERENCE: FORWARD CHECKING AND ARC CONSISTENCY

Forward checking is a simple form of inference:

- Keep track of remaining legal values for unassigned variables
- When a new variable is assigned, recalculate the legal values for its neighbors





WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥	🟦		🟥 🟩 🟦

Arc consistency: $X \rightarrow Y$ is ac iff for every x in X, there is some allowed y in Y

- since NT and SA cannot both be blue, the problem becomes arc inconsistent before forward checking notices
- arc consistency detects failure earlier than forward checking

ARC CONSISTENCY ALGORITHM, AC-3

Keep a set of arcs to be considered: pick one arc (X, Y) at the time and make it consistent (i.e., make X arc consistent to Y).

- Start with the set of all arcs $\{(X, Y), (Y, X), (X, Z), (Z, X), \ldots\}$.

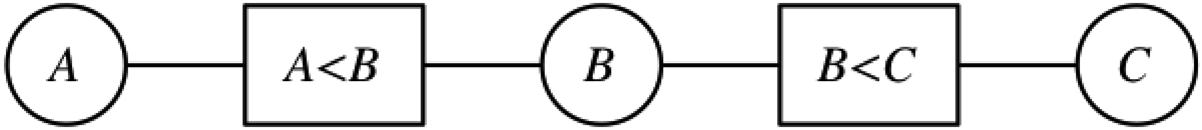
When an arc has been made arc consistent, does it ever need to be checked again?

- An arc (X, Y) needs to be revisited if the domain of Y is revised.

```
function AC-3(inout csp):
    initialise queue to all arcs in csp
    while queue is not empty:
        (X, Y) := RemoveOne(queue)
        if Revise(csp, X, Y):
            if  $D_X = \emptyset$  then return failure
            for each Z in X.neighbors–{Y} do add (Z, X) to queue

function Revise(inout csp, X, Y):
    delete every x from $D_X$ such that there is no value y in $D_Y$ satisfying the constraint $C_{XY}$
```

AC-3 EXAMPLE



remove	D_A	D_B	D_C	add	queue
	1234	1234	1234		*A<B, B<C, C>B, B>A*
A<B	***123***	1234	1234		*B<C, C>B, B>A*
B<C	123	***123***	1234	*A<B*	*C>B, B>A,* ***A<B***
C>B	123	123	***234***		*B>A, A<B*
B>A	123	***23***	234	*C>B*	*A<B,* ***C>B***
A<B	***12***	23	234		*C>B*
C>B	12	23	***34***		$\emptyset$

COMBINING BACKTRACKING WITH AC-3

What if some domains have more than one element after AC?

We can resort to backtracking search:

- Select a variable and a value using some heuristics
  (e.g., minimum-remaining-values, degree-heuristic, least-constraining-value)
- Make the graph arc-consistent again
- Backtrack and try new values/variables, if AC fails
- Select a new variable/value, perform arc-consistency, etc.

Do we need to restart AC from scratch?

- no, only some arcs risk becoming inconsistent after a new assignment
- restart AC with the queue $\{(Y_i, X) | X \rightarrow Y_i\}$,
  i.e., only the arcs (Y_i, X) where Y_i are the neighbors of X
- this algorithm is called *Maintaining Arc Consistency* (MAC)

CONSISTENCY PROPERTIES

There are several kinds of consistency properties and algorithms:

- *Node consistency*: single variable, unary constraints (straightforward)
- *Arc consistency*: pairs of variables, binary constraints (AC-3 algorithm)
- *Path consistency*: triples of variables, binary constraints (PC-2 algorithm)
- *k-consistency*: k variables, k-ary constraints (algorithms exponential in k)
- Consistency for global constraints:
    - special-purpose algorithms for different constraints, e.g.:
    - *Alldiff*($X_1, \dots, X_m$) is inconsistent if $m > |D_1 \cup \cdots \cup D_m|$
    - *Atmost*($n, X_1, \dots, X_m$) is inconsistent if $n < \sum_i \min(D_i)$

MORE ABOUT CSP

LOCAL SEARCH FOR CSPS (R&N 6.4)

PROBLEM STRUCTURE (R&N 6.5)

LOCAL SEARCH FOR CSPS (R&N 6.4)

Given an assignment of a value to each variable:

- A conflict is an unsatisfied constraint.
- The goal is an assignment with zero conflicts.

Local search / Greedy descent algorithm:

- Start with a complete assignment.
- Repeat until a satisfying assignment is found:
  - select a variable to change
  - select a new value for that variable

MIN CONFLICTS ALGORITHM

Heuristic function to be minimized: the number of conflicts.

- this is the *min-conflicts* heuristics

Note: this does not always work!

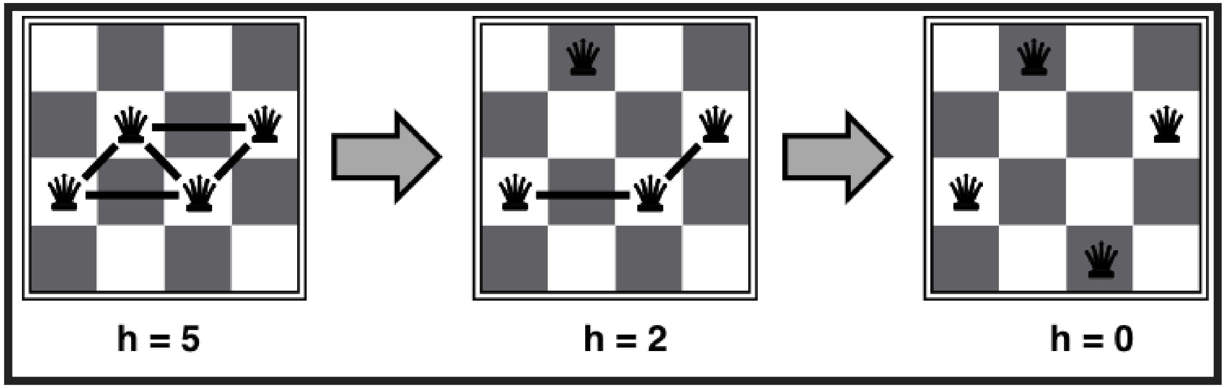
- it can get stuck in a *local minimum*

```
function MinConflicts(csp, max_steps)
    current := an initial complete assignment for csp
    repeat max_steps times:
        if current is a solution for csp then return current
        var := a randomly chosen conflicted variable from csp
        value := the value v for var that minimises Conflicts(var, v, current, csp)
        current[var] = value
    return failure
```

EXAMPLE: $\mathbf{n}$-QUEENS (REVISITED)

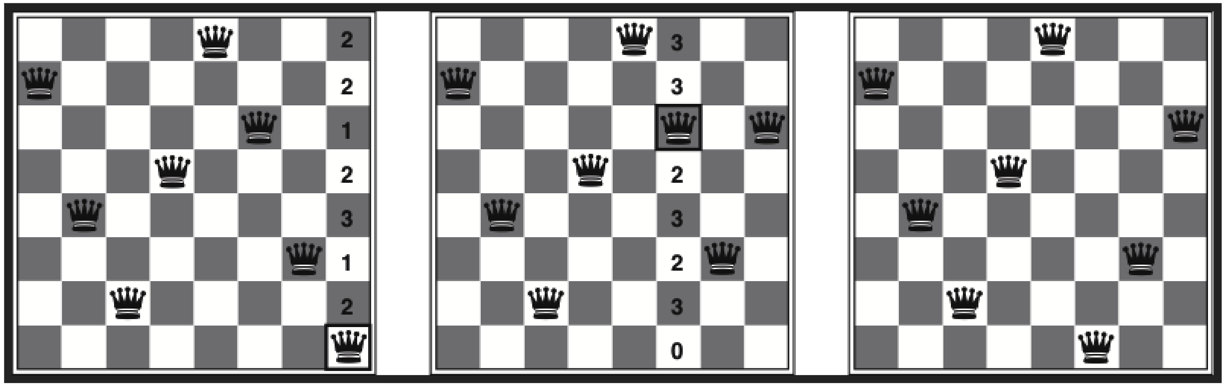
Do you remember this example?

- Put n queens on an $n \times n$ board, in separate columns
- Conflicts = unsatisfied constraints = n:o of threatened queens
- Move a queen to reduce the number of conflicts
    - repeat until we cannot move any queen anymore
    - then we are at a local maximum — hopefully it is global too



EASY AND HARD PROBLEMS

Two-step solution using min-conflicts for an 8-queens problem:



The runtime of min-conflicts on *n-queens* is *independent of problem size*!

- it solves even the *million*-queens problem ≈50 steps

Why is *n*-queens easy for local search?

- because solutions are *densely distributed* throughout the state space!

VARIANTS OF GREEDY DESCENT

To choose a variable to change and a new value for it:

- Find a variable-value pair that minimizes the number of conflicts.
- Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
- Select a variable that appears in any conflict.
  Select a value that minimizes the number of conflicts.
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
- Select a variable and value at random;
  accept this change if it doesn’t increase the number of conflicts.

All local search techniques from section 4.1 can be applied to CSPs, e.g.:

- random walk, random restarts, simulated annealing, beam search, …

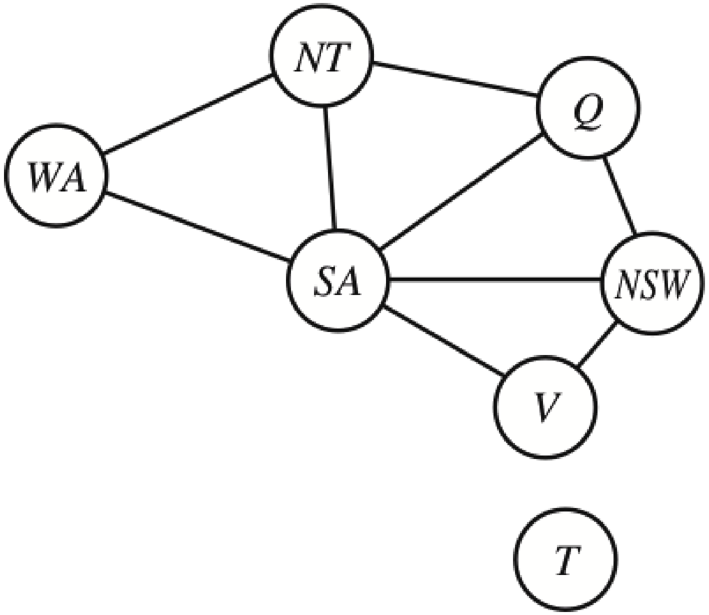
PROBLEM STRUCTURE (R&N 6.5)

- independent subproblems, connected components
- tree-structured CSP, topological sort
- converting to tree-structured CSP, cycle cutset, tree decomposition

INDEPENDENT SUBPROBLEMS

Tasmania is an *independent subproblem*:

- there are efficient algorithms for finding *connected components* in a graph



Suppose that each subproblem has c variables out of n total. The cost of the worst-case solution is $n/c \cdot d^c$, which is linear in n.

E.g., $n = 80, d = 2, c = 20$:

- 2^{80} = 4 billion years at 10 million nodes/sec

If we divide it into 4 equal-size subproblems:

- $4 \cdot 2^{20}$ =0.4 seconds at 10 million nodes/sec

Note: this only has a real effect if the subproblems are (roughly) equal size!

TREE-STRUCTURED CSP

A constraint graph is a tree when any two variables are connected by only one path.

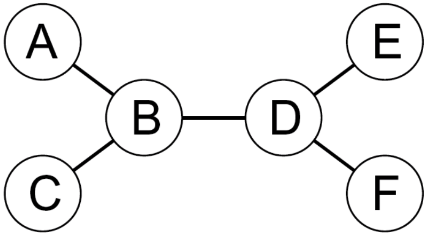
- then any variable can act as root in the tree
- tree-structured CSP can be solved in *linear time*, in the number of variables!

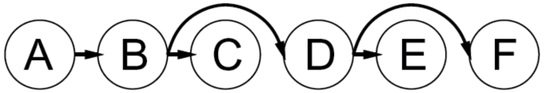
CSP is directed arc-consistent if:

- there is an ordering of variables $X_1, X_2, \ldots, X_n$ such that
- every X_i is arc-consistent with each X_j for all $j > i$

To solve a tree-structured CSP:

- first pick a variable to be the root of the tree
- then find a *topological sort* of the variables (with the root first)
- finally, make each arc consistent, in reverse topological order





SOLVING TREE-STRUCTURED CSP

```
function TreeCSPSolver($csp$)
    $n$ := number of variables in $csp$
    $root$ := any variable in $csp$
    $X_1 \ldots X_n$ := TopologicalSort($csp$, $root$)
    for $j := n, n-1, \ldots, 2$:
        MakeArcConsistent(Parent($X_j$), $X_j$)
        if it could not be made consistent then return failure
    $assignment$ := an empty assignment
    for $i := 1, 2, \ldots, n$:
        $assignment[X_i]$ := any consistent value from $D_i$
    return $assignment$
```

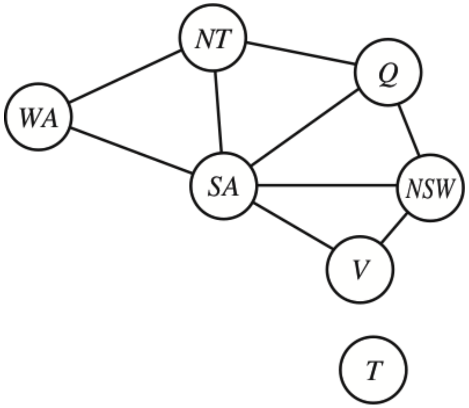
What is the runtime?

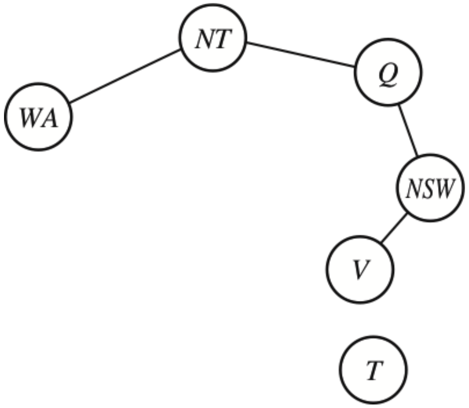
- to make an arc consistent, we must compare up to d^2 domain value pairs
- there are $n-1$ arcs, so the total runtime is $O(nd^2)$

CONVERTING TO TREE-STRUCTURED CSP

Most CSPs are *not* tree-structured, but sometimes we can reduce a problem to a tree

- one approach is to assign values to some variables,
  so that the remaining variables form a tree





If we assign a colour to South Australia, then the remaining variables form a tree

- a (worse) alternative is to assign values to {*NT,Q,V*}

Why is {*NT,Q,V*} a worse alternative?

- because then we have to try 3×3×3 different assignments,
  and for each of them solve the remaining tree-CSP

SOLVING ALMOST-TREE-STRUCTURED CSP

```
function SolveByReducingToTreeCSP(csp):
    S := a cycle cutset of variables, such that csp–S becomes a tree
    for each assignment for S that satisfies all constraints on S:
        remove any inconsistent values from neighboring variables of S
        solve the remaining tree-CSP (i.e., csp–S)
        if there is a solution then return it together with the assignment for S
    return failure
```

The set of variables that we have to assign is called a *cycle cutset*

- for Australia, {*SA*} is a cycle cutset and {*NT,Q,V*} is also a cycle cutset
- finding the smallest cycle cutset is NP-hard,
  but there are efficient approximation algorithms

TREE DECOMPOSITION

Another approach for reducing to a tree-CSP is *tree decomposition*:

- divide the original CSP into a set of connected subproblems, such that the connections form a *tree-structured graph*
- solve each subproblem independently
- since the decomposition is a tree, we can solve the main problem using directed arc consistency (the TreeCSPSolver algorithm)