Artificial Intelligence 1
Winter Semester 2024/25

– Lecture Notes –
Part II: General Problem Solving

Prof. Dr. Michael Kohlhase

Professur für Wissensrepräsentation und -verarbeitung
Informatik, FAU Erlangen-Nürnberg
`Michael.Kohlhase@FAU.de`

2025-02-06

Chapter 6
Problem Solving and Search

6.1 Problem Solving

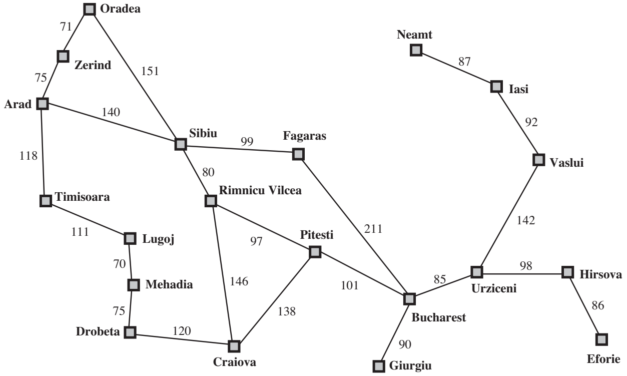
Problem Solving: Introduction

- **Recap:** Agents perceive the environment and compute an action.
- **In other words:** Agents continually solve "the problem of what to do next".
- **AI Goal:** Find algorithms that help solving problems in general.
- **Idea:** If we can describe/represent problems in a standardized way, we may have a chance to find general algorithms.
- **Concretely:** We will use the following two concepts to describe problems
  - States: A set of possible situations in our problem domain ($\widehat{=}$ environments)
  - Actions: that get us from one state to another. ($\widehat{=}$ agents)

  A sequence of actions is a solution, if it brings us from an initial state to a goal state. Problem solving computes solutions from problem formulations.
- **Definition 1.1.** In offline problem solving an agent computing an action sequence based complete knowledge of the environment.
- *Remark 1.2.* Offline problem solving only works in fully observable, deterministic, static, and episodic environments.
- **Definition 1.3.** In online problem solving an agent computes one action at a time based on incoming perceptions.
- **This Semester:** We largely restrict ourselves to offline problem solving.(easier)

Example: Traveling in Romania

- **Scenario:** An agent is on holiday in Romania; currently in Arad; flight home leaves tomorrow from Bucharest; how to get there? We have a map:



- **Formulate the Problem:**
  - States: various cities.
  - Actions: drive between cities.
- **Solution:** Appropriate sequence of cities, e.g.: Arad, Sibiu, Fagaras, Bucharest

Problem Formulation

- **Definition 1.4.** A problem formulation models a situation using states and actions at an appropriate level of abstraction. (do not model things like "put on my left sock", etc.)
  - it describes the initial state (we are in Arad)
  - it also limits the objectives by specifying goal states. (excludes, e.g. to stay another couple of weeks.)

  A solution is a sequence of actions that leads from the initial state to a goal state.

  Problem solving computes solutions from problem formulations.
- Finding the right level of abstraction and the required (not more!) information is often the key to success.

The Math of Problem Formulation: Search Problems

- **Definition 1.5.** A search problem $\Pi := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set $\mathcal{S}$ of states, a set $\mathcal{A}$ of actions, and a transition model $\mathcal{T} \colon \mathcal{A} \times \mathcal{S} \to \mathcal{P}(\mathcal{S})$ that assigns to any action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$ a set of successor states.
  Certain states in $\mathcal{S}$ are designated as goal states (also called terminal state) ($\mathcal{G} \subseteq \mathcal{S}$ with $\mathcal{G} \neq \emptyset$) and initial states $\mathcal{I} \subseteq \mathcal{S}$.
- **Definition 1.6.** We say that an action $a \in \mathcal{A}$ is applicable in state $s \in \mathcal{S}$, iff $\mathcal{T}(a, s) \neq \emptyset$ and that any $s' \in \mathcal{T}(a, s)$ is a result of applying action a to state s.
  We call $\mathcal{T}_a \colon \mathcal{S} \to \mathcal{P}(\mathcal{S})$ with $\mathcal{T}_a(s) := \mathcal{T}(a, s)$ the result relation for a and $\mathcal{T}_{\mathcal{A}} := \bigcup_{a \in \mathcal{A}} \mathcal{T}_a$ the result relation of Π.
- **Definition 1.7.** The graph $\langle \mathcal{S}, \mathcal{T}_{\mathcal{A}} \rangle$ is called the state space induced by Π.
- **Definition 1.8.** A solution for Π consists of a sequence $a_1, \ldots, a_n$ of actions such that for all $1 < i \leq n$
  - a_i is applicable to state s_{i-1}, where $s_0 \in \mathcal{I}$ and
  - $s_i \in \mathcal{T}_{a_i}(s_{i-1})$, and $s_n \in \mathcal{G}$.
- **Idea:** A solution bring us from $\mathcal{I}$ to a goal state via applicable actions.
- **Definition 1.9.** Often we add a cost function $c \colon \mathcal{A} \to \mathbb{R}_0^+$ that associates a step cost $c(a)$ to an action $a \in \mathcal{A}$. The cost of a solution is the sum of the step costs of its actions.

Structure Overview: Search Problem

- The structure overview for search problems:

$$\text{search problem} = \left\langle \begin{array}{lll} \mathcal{S} & \text{Set} & \text{states}, \\ \mathcal{A} & \text{Set} & \text{actions}, \\ \mathcal{T} & \mathcal{A}\times\mathcal{S}\rightarrow\mathcal{P}(\mathcal{S}) & \text{transition model}, \\ \mathcal{I} & \mathcal{S} & \text{initial state}, \\ \mathcal{G} & \mathcal{P}(\mathcal{S}) & \text{goal states} \end{array} \right\rangle$$

Search Problems in deterministic, fully observable Environments

- This semester, we will restrict ourselves to search problems, where(extend in AI II)
  - $|\mathcal{T}(a,s)| \leq 1$ for the transition models and (⇜ deterministic environment)
  - $\mathcal{I} = \{s_0\}$ (⇜ fully observable environment)

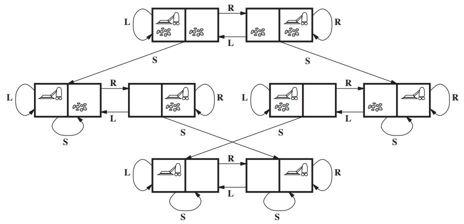
  Definition 1.11. We call a search problem with transition model $\mathcal{T}$ deterministic, iff $|\mathcal{T}(a,s)| \leq 1$.
-
- **Definition 1.12.** In a deterministic search problem, $\mathcal{T}_a$ induces partial function $\mathcal{S}_a : \mathcal{S} \rightharpoonup \mathcal{S}$ whose natural domain is the set of states where a is applicable: $\mathcal{S}_a(s) := s'$ if $\mathcal{T}_a = \{s'\}$ and undefined at s otherwise. We call $\mathcal{S}_a$ the successor function for a and $\mathcal{S}_a(s)$ the successor state of s.
- **Definition 1.13.** The predicate that tests for goal states is called a goal test.

6.2 Problem Types

Problem types

- **Definition 2.1.** A search problem is called a single state problem, iff it is
  - fully observable (at least the initial state)
  - deterministic (unique successor states)
  - static (states do not change other than by our own actions)
  - discrete (a countable number of states)
- **Definition 2.2.** A search problem is called a multi state problem
  - states partially observable (e.g. multiple initial states)
  - deterministic, static, discrete
- **Definition 2.3.** A search problem is called a contingency problem, iff
  - the environment is non deterministic (solution can branch, depending on contingencies)
  - the state space is unknown(like a baby, agent has to learn about states and actions)

Example: vacuum-cleaner world



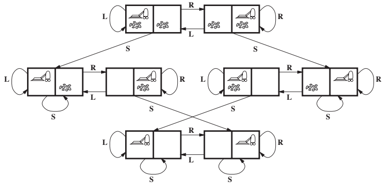
- **Single-state Problem:**
  - Start in 5
  - **Solution**: [*right*, *suck*]
- **Multiple-state Problem:**
  - Start in $\{1,2,3,4,5,6,7,8\}$
  - **Solution**: [*right*, *suck*, *left*, *suck*]

right	$\rightarrow$	$\{2,4,6,8\}$
suck	$\rightarrow$	$\{4,8\}$
left	$\rightarrow$	$\{3,7\}$
suck	$\rightarrow$	$\{7\}$

Example: Vacuum-Cleaner World (continued)

- **Contingency Problem:**
  - Murphy's Law: *suck* can dirty a clean carpet
  - Local sensing: *dirty*/*notdirty* at location only
  - Start in: $\{1,3\}$
  - **Solution**: $[suck, right, suck]$

suck	$\rightarrow$	$\{5,7\}$
right	$\rightarrow$	$\{6,8\}$
suck	$\rightarrow$	$\{6,8\}$



- **better:** $[suck, right, \text{if } dirt \text{ then } suck]$ (decide whether in 6 or 8 using local sensing)

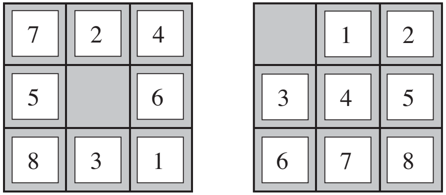
Single-state problem formulation

- Defined by the following four items
  1. Initial state: (e.g. *Arad*)
  2. Successor function $S_a(s)$: (e.g. $S_{goZer} = \{(Arad, Zerind), (goSib, Sibiu), \ldots\}$)
  3. Goal test: (e.g. $x = Bucharest$ (explicit test)
     $noDirt(x)$ (implicit test))
  4. Path cost (optional): (e.g. sum of distances, number of operators executed, etc.)
- Solution: A sequence of actions leading from the initial state to a goal state.

Selecting a state space

- **Abstraction:** Real world is absurdly complex!
  State space must be abstracted for problem solving.
- **(Abstract) state:** Set of real states.
- **(Abstract) operator:** Complex combination of real actions.
- **Example:** $\mathit{Arad} \to \mathit{Zerind}$ represents complex set of possible routes.
- **(Abstract) solution:** Set of real paths that are solutions in the real world.

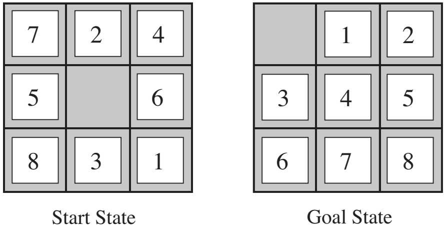
Example: The 8-puzzle



Start State Goal State

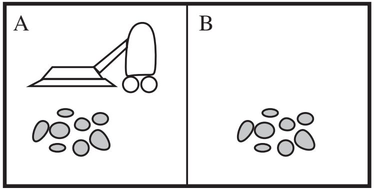
States? Actions?...

Example: The 8-puzzle



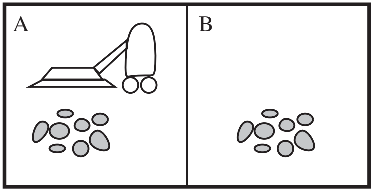
States	integer locations of tiles
Actions	*left*, *right*, *up*, *down*
Goal test	= goal state?
Path cost	1 per move

Example: Vacuum-cleaner



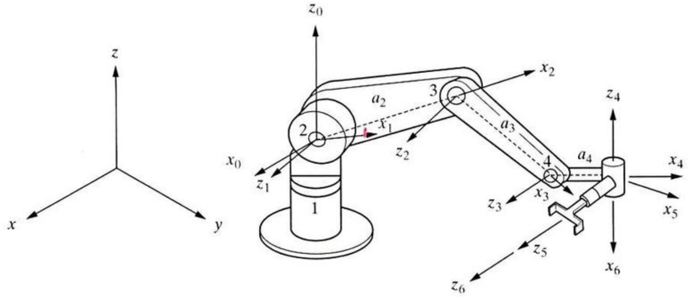
States? Actions?...

Example: Vacuum-cleaner



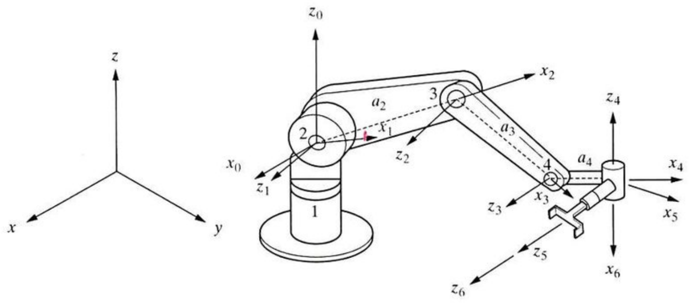
States	integer dirt and robot locations
Actions	*left*, *right*, *suck*, *noOp*
Goal test	*notdirty*?
Path cost	1 per operation (0 for *noOp*)

Example: Robotic assembly



States? Actions?. . .

Example: Robotic assembly



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

General Problems

- **Question:** Which are "Problems"?
  (A) You didn't understand any of the lecture.
  (B) Your bus today will probably be late.
  (C) Your vacuum cleaner wants to clean your apartment.
  (D) You want to win a chess game.

General Problems

- **Question:** Which are “Problems”?
  - (A) You didn’t understand any of the lecture.
  - (B) Your bus today will probably be late.
  - (C) Your vacuum cleaner wants to clean your apartment.
  - (D) You want to win a chess game.
- **Answer:**
  - (A/B) These are problems in the natural language use of the word, but not “problems” in the sense defined here.

General Problems

- **Question:** Which are "Problems"?
  - (A) You didn't understand any of the lecture.
  - (B) Your bus today will probably be late.
  - (C) Your vacuum cleaner wants to clean your apartment.
  - (D) You want to win a chess game.
- **Answer:**
  - (A/B) These are problems in the natural language use of the word, but not "problems" in the sense defined here.
  - (C) Yes, presuming that this is a robot, an autonomous vacuum cleaner, and that the robot has perfect knowledge about your apartment (else, it's not a classical search problem).

General Problems

- **Question:** Which are "Problems"?
  - (A) You didn't understand any of the lecture.
  - (B) Your bus today will probably be late.
  - (C) Your vacuum cleaner wants to clean your apartment.
  - (D) You want to win a chess game.
- **Answer:**
  - (A/B) These are problems in the natural language use of the word, but not "problems" in the sense defined here.
  - (C) Yes, presuming that this is a robot, an autonomous vacuum cleaner, and that the robot has perfect knowledge about your apartment (else, it's not a classical search problem).
  - (D) That's a search problem, but not a classical search problem (because it's multi-agent). We'll tackle this kind of problem in **??**

6.3 Search

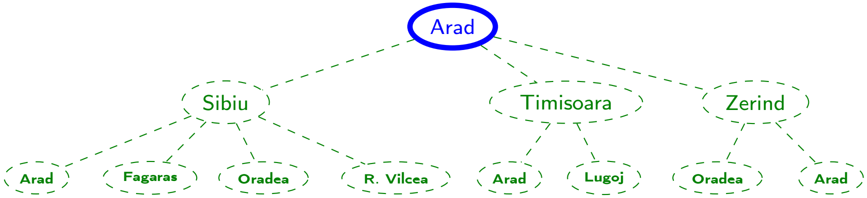
Tree Search Algorithms

- **Note:** The state space of a search problem $\langle\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{I},\mathcal{G}\rangle$ is a graph $\langle\mathcal{S},\mathcal{T}_{\mathcal{A}}\rangle$.
- As graphs are difficult to compute with, we often compute a corresponding tree and work on that. (standard trick in graph algorithms)
- **Definition 3.1.** Given a search problem $\mathcal{P} := \langle\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{I},\mathcal{G}\rangle$, the tree search algorithm consists of the simulated exploration of state space $\langle\mathcal{S},\mathcal{T}_{\mathcal{A}}\rangle$ in a search tree formed by successively expanding already explored states. (offline algorithm)

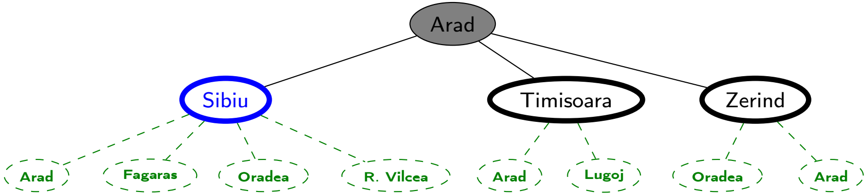
```
procedure Tree—Search (problem, strategy) : <a solution or failure>
  <initialize the search tree using the initial state of problem>
  loop
     if <there are no candidates for expansion> <return failure> end if
    <choose a leaf node for expansion according to strategy>
     if <the node contains a goal state> return <the corresponding solution>
     else <expand the node and add the resulting nodes to the search tree>
     end if
  end loop
end procedure
```

We expand a node *n* by generating all successors of *n* and inserting them as children of *n* in the search tree.

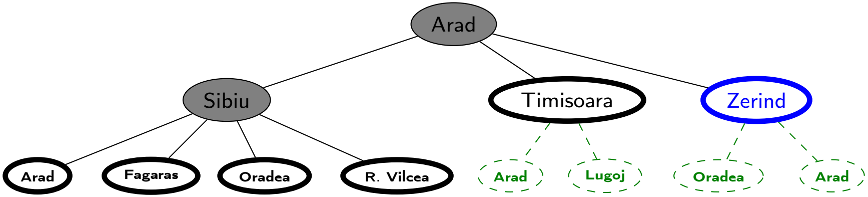
Tree Search: Example



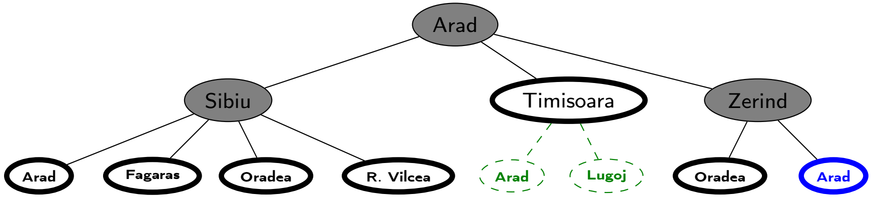
Tree Search: Example



Tree Search: Example



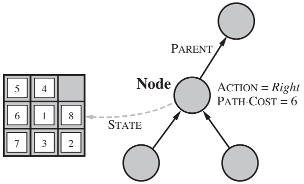
Tree Search: Example



Implementation: States vs. nodes

- **Recap:** A state is a (representation of) a physical configuration.
- **Definition 3.2 (Implementing a Search Tree).**

A search tree node is a data structure that includes accessors for parent, children, depth, path cost, insertion order, etc.
A goal node (initial node) is a search tree node labeled with a goal state (initial state).



- **Observation:** A set of search tree nodes that can all (recursively) reach a single initial node form a search tree. (they implement it)
- **Observation:** Paths in the search tree correspond to paths in the state space.
- **Definition 3.3.** We define the path cost of a node n in a search tree T to be the sum of the step costs on the path from n to the root of T.
- **Observation:** As a search tree node has access to parents, we can read off the solution from a goal node.

Implementation of Search Algorithms

- **Definition 3.4 (Implemented Tree Search Algorithm).**

```
procedure Tree_Search (problem,strategy)
  fringe := insert(make_node(initial_state(problem)))
    loop
     if empty(fringe) fail end if
      node := first(fringe,strategy)
      if GoalTest(node) return node
      else fringe := insert(expand(node,problem))
      end if
   end loop
end procedure
```

  The fringe is the set of search tree nodes not yet expanded in tree search.

- **Idea:** We treat the fringe as an abstract data type with three accessors: the
  - binary function first retrieves an element from the fringe according to a strategy.
  - binary function insert adds a (set of) search tree node into a fringe.
  - unary predicate empty to determine whether a fringe is the empty set.
- The strategy determines the behavior of the fringe (data structure) (see below)

Search strategies

- **Definition 3.5.** A strategy is a function that picks a node from the fringe of a search tree. (equivalently, orders the fringe and picks the first.)
- **Definition 3.6 (Important Properties of Strategies).**

completeness	does it always find a solution if one exists?
time complexity	number of nodes generated/expanded
space complexity	maximum number of nodes in memory
optimality	does it always find a least cost solution?

- **Time and space complexity measured in terms of:**

b	maximum branching factor of the search tree
d	minimal graph depth of a solution in the search tree
m	maximum graph depth of the search tree (may be ∞)

Complexity means here always *worst-case* complexity!

6.4 Uninformed Search Strategies

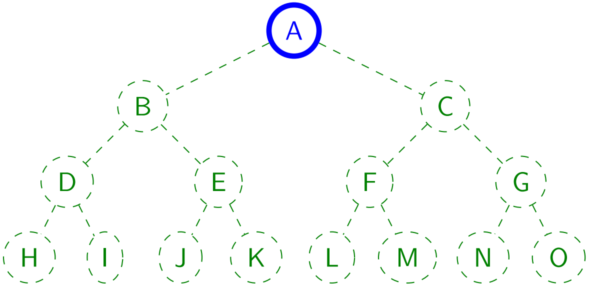
Uninformed search strategies

- **Definition 4.1.** We speak of an uninformed search algorithm, if it only uses the information available in the problem definition.
- **Next:** Frequently used search algorithms
  - Breadth first search
  - Uniform cost search
  - Depth first search
  - Depth limited search
  - Iterative deepening search

6.4.1 Breadth-First Search Strategies

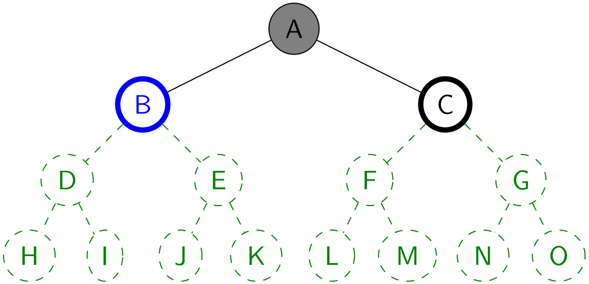
Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.2.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.3 (Synthetic).**



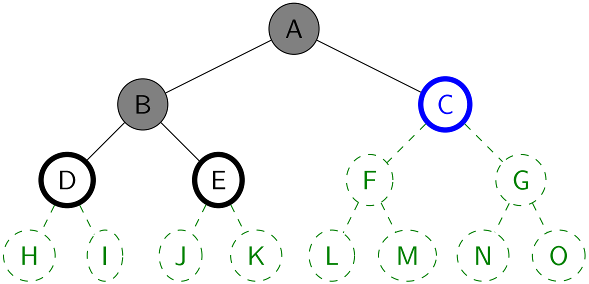
Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.4.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.5 (Synthetic).**



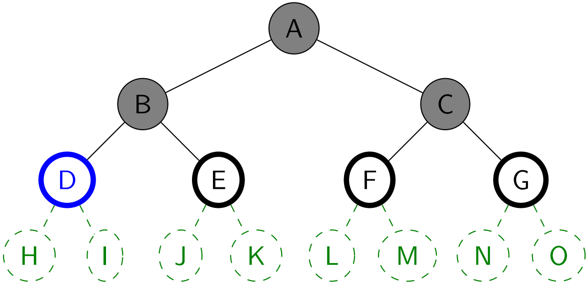
Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.6.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.7 (Synthetic).**



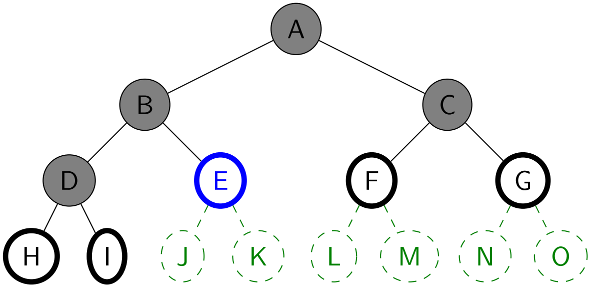
Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.8.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.9 (Synthetic).**



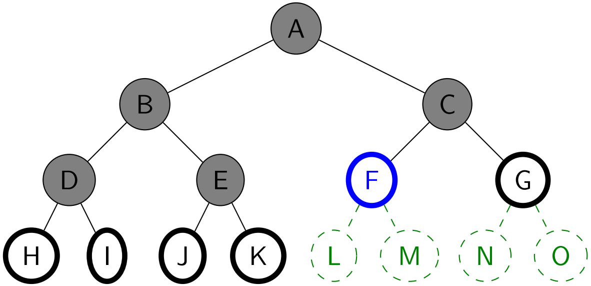
Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.10.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.11 (Synthetic).**



Breadth-First Search

- **Idea:** Expand the shallowest unexpanded node.
- **Definition 4.12.** The breadth first search (BFS) strategy treats the fringe as a FIFO queue, i.e. successors go in at the end of the fringe.
- **Example 4.13 (Synthetic).**

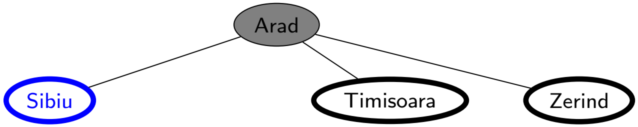


Breadth-First Search: Romania

- **Example 4.14.**

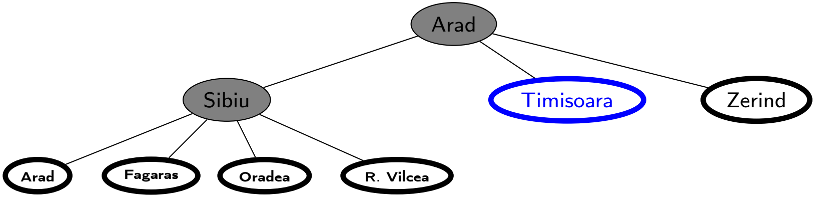
Breadth-First Search: Romania

- **Example 4.15.**



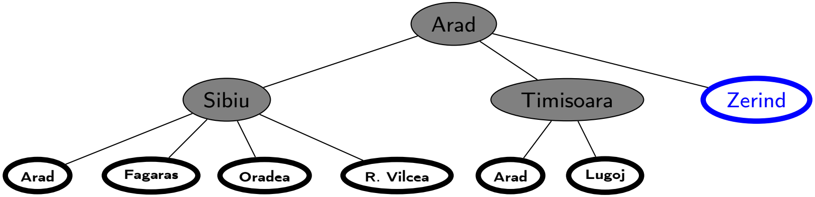
Breadth-First Search: Romania

- **Example 4.16.**



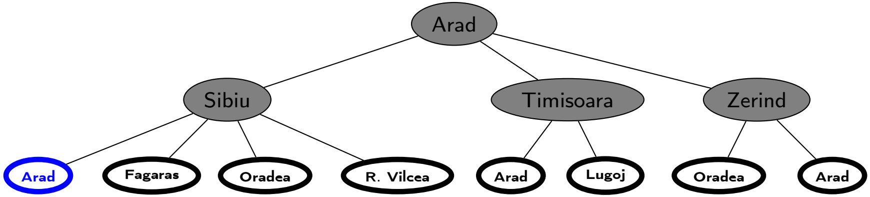
Breadth-First Search: Romania

- **Example 4.17.**



Breadth-First Search: Romania

- **Example 4.18.**



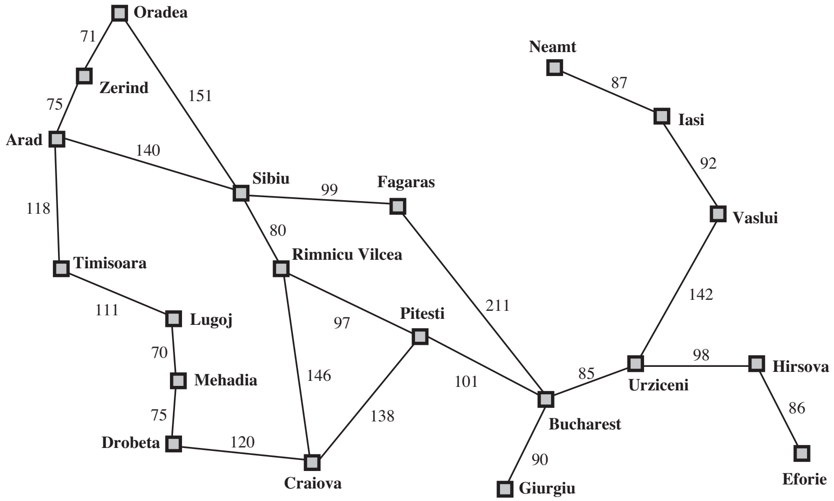
Breadth-first search: Properties

▶

Completeness	Yes (if b is finite)
Time complexity	$1+b+b^2+b^3+\ldots+b^d$, so $\mathcal{O}(b^d)$, i.e. exponential in d
Space complexity	$\mathcal{O}(b^d)$ (fringe may be whole level)
Optimality	Yes (if cost $=1$ per step), not optimal in general

- **Disadvantage:** Space is the big problem (can easily generate nodes at 500MB/sec $\widehat{=}$ 1.8TB/h)
- **Optimal?:** No! If cost varies for different steps, there might be better solutions below the level of the first one.
- An alternative is to generate *all* solutions and then pick an optimal one. This works only, if m is finite.

Romania with Step Costs as Distances

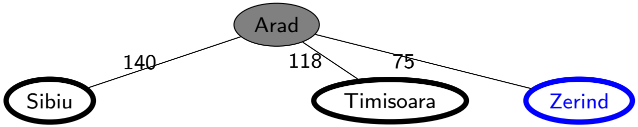


Uniform-cost search

- **Idea:** Expand least cost unexpanded node.
- **Definition 4.19.** Uniform-cost search (UCS) is the strategy where the fringe is ordered by increasing path cost.
- **Note:** Equivalent to breadth first search if all step costs are equal.
- **Synthetic Example:**

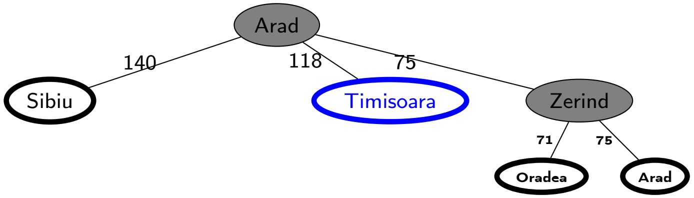
Uniform-cost search

- **Idea:** Expand least cost unexpanded node.
- **Definition 4.20.** Uniform-cost search (UCS) is the strategy where the fringe is ordered by increasing path cost.
- **Note:** Equivalent to breadth first search if all step costs are equal.
- **Synthetic Example:**



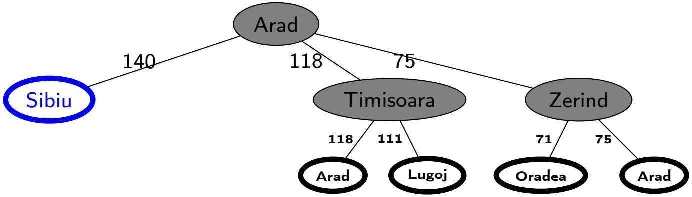
Uniform-cost search

- **Idea:** Expand least cost unexpanded node.
- **Definition 4.21.** Uniform-cost search (UCS) is the strategy where the fringe is ordered by increasing path cost.
- **Note:** Equivalent to breadth first search if all step costs are equal.
- **Synthetic Example:**



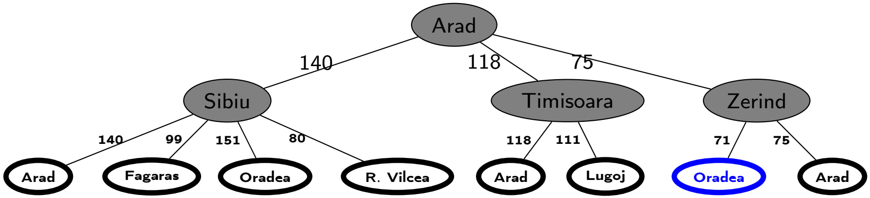
Uniform-cost search

- **Idea:** Expand least cost unexpanded node.
- **Definition 4.22.** Uniform-cost search (UCS) is the strategy where the fringe is ordered by increasing path cost.
- **Note:** Equivalent to breadth first search if all step costs are equal.
- **Synthetic Example:**



Uniform-cost search

- **Idea:** Expand least cost unexpanded node.
- **Definition 4.23.** Uniform-cost search (UCS) is the strategy where the fringe is ordered by increasing path cost.
- **Note:** Equivalent to breadth first search if all step costs are equal.
- **Synthetic Example:**



Uniform-cost search: Properties

Completeness	Yes (if step costs $\geq \epsilon > 0$)
Time complexity	number of nodes with path cost less than that of optimal solution
Space complexity	ditto
Optimality	Yes

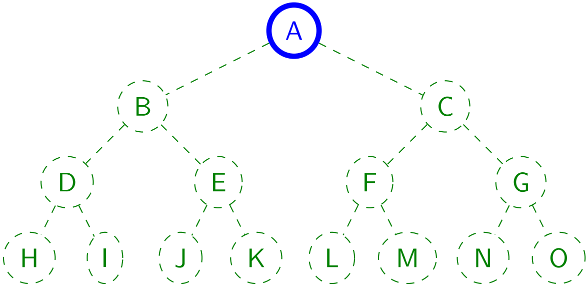
6.4.2 Depth-First Search Strategies

Depth-first Search

- **Idea:** Expand deepest unexpanded node.
- **Definition 4.24.** Depth-first search (DFS) is the strategy where the fringe is organized as a (LIFO) stack i.e. successors go in at front of the fringe.
- **Definition 4.25.** Every node that is pushed to the stack is called a backtrack point. The action of popping a non-goal node from the stack and continuing the search with the new top element of the stack (a backtrack point by construction) is called backtracking, and correspondingly the DFS algorithm backtracking search.
- **Note:** Depth first search can perform infinite cyclic excursions
  Need a finite, non cyclic state space (or repeated state checking)

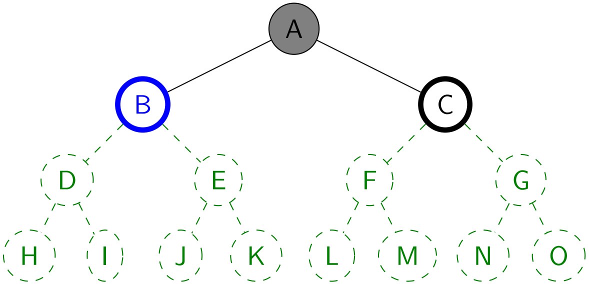
Depth-First Search

- **Example 4.26 (Synthetic).**



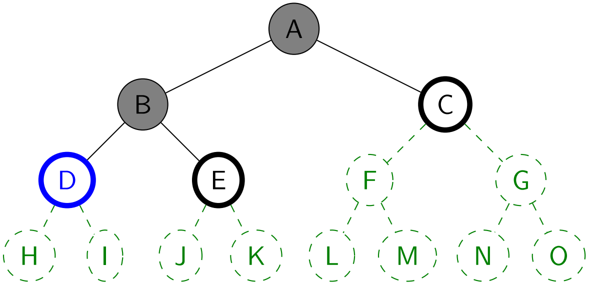
Depth-First Search

- **Example 4.27 (Synthetic).**



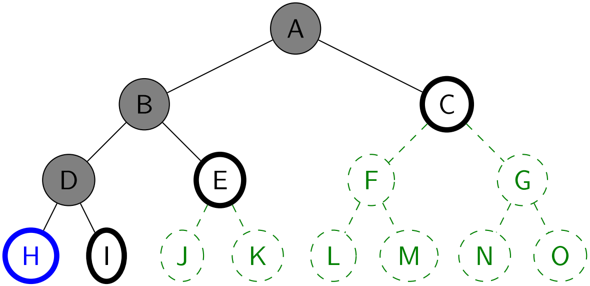
Depth-First Search

- **Example 4.28 (Synthetic).**



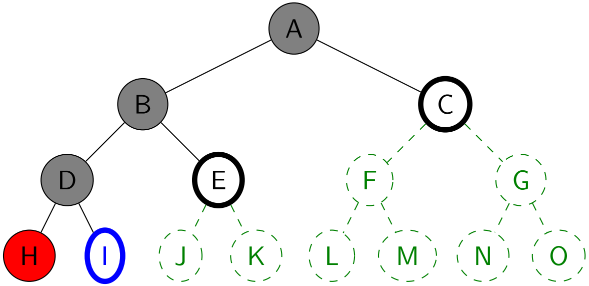
Depth-First Search

- **Example 4.29 (Synthetic).**



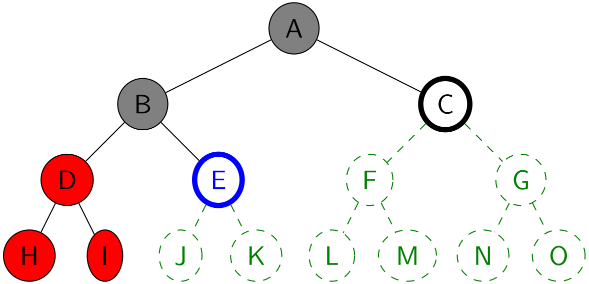
Depth-First Search

- **Example 4.30 (Synthetic).**



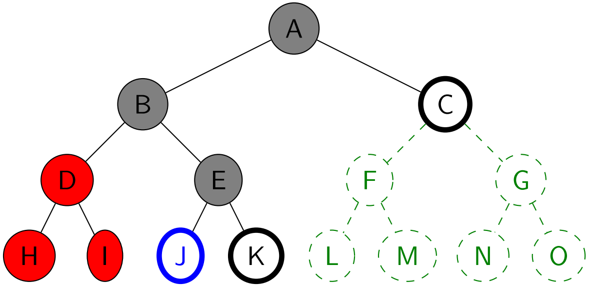
Depth-First Search

- **Example 4.31 (Synthetic).**



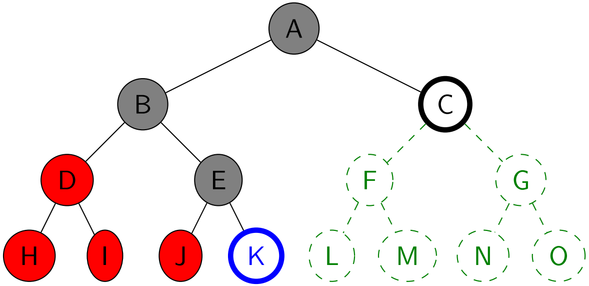
Depth-First Search

- **Example 4.32 (Synthetic).**



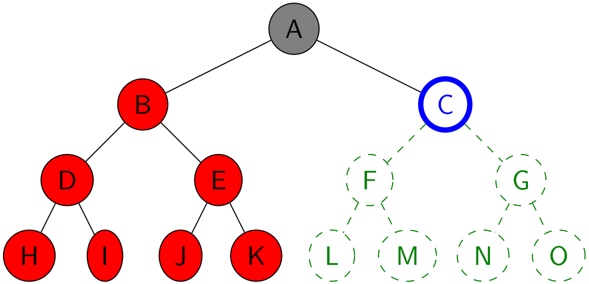
Depth-First Search

- **Example 4.33 (Synthetic).**



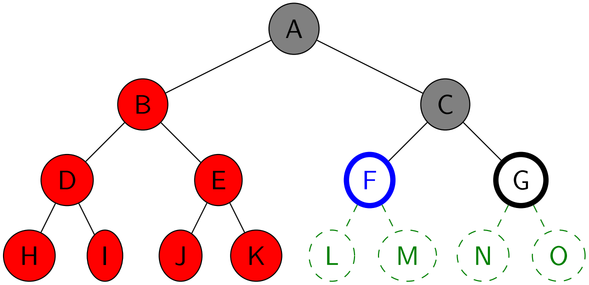
Depth-First Search

- **Example 4.34 (Synthetic).**



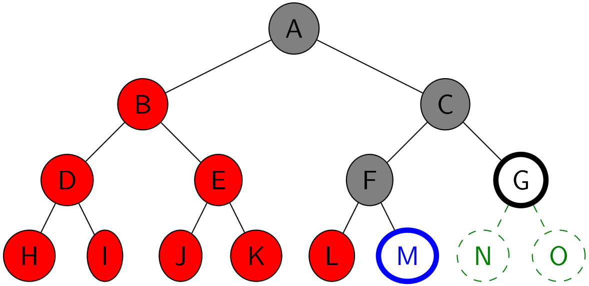
Depth-First Search

- **Example 4.35 (Synthetic).**



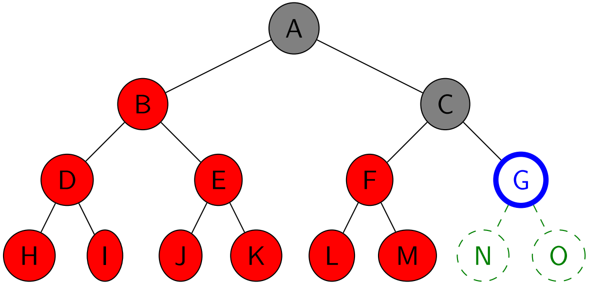
Depth-First Search

- **Example 4.36 (Synthetic).**



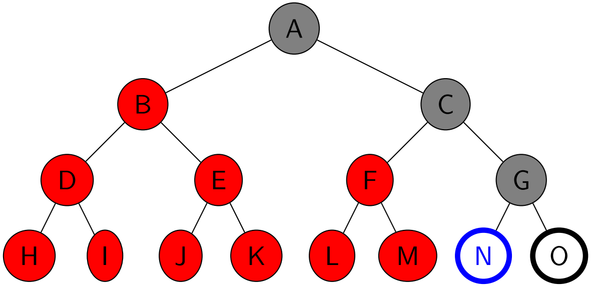
Depth-First Search

- **Example 4.37 (Synthetic).**



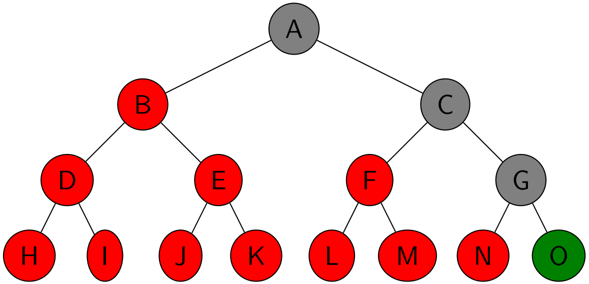
Depth-First Search

- **Example 4.38 (Synthetic).**



Depth-First Search

- **Example 4.39 (Synthetic).**

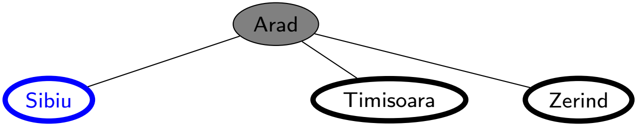


Depth-First Search: Romania

- **Example 4.40 (Romania).**

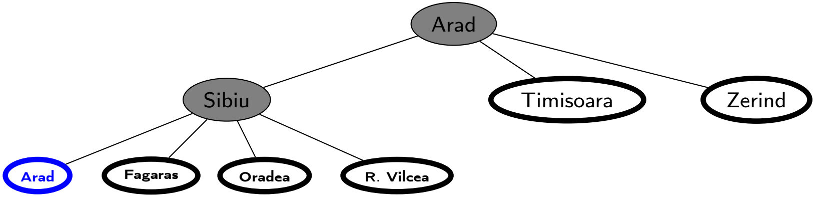
Depth-First Search: Romania

- **Example 4.41 (Romania).**



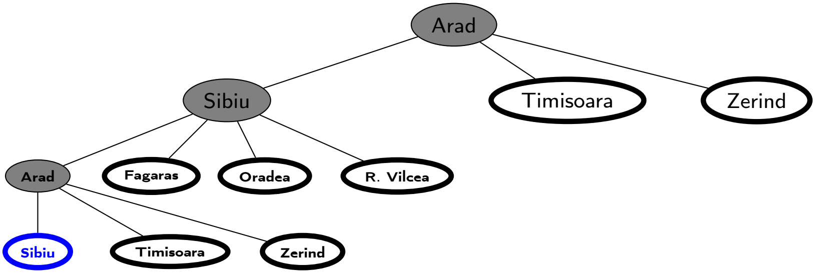
Depth-First Search: Romania

- **Example 4.42 (Romania).**



Depth-First Search: Romania

- **Example 4.43 (Romania).**



Depth-first search: Properties

▶

Completeness	Yes: if search tree finite No: if search tree contains infinite paths or loops
Time complexity	$\mathcal{O}(b^m)$ (we need to explore until max depth m in any case!)
Space complexity	$\mathcal{O}(bm)$ (i.e. linear space) (need at most store m levels and at each level at most b nodes)
Optimality	No (there can be many better solutions in the unexplored part of the search tree)

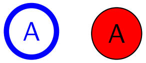
- **Disadvantage:** Time terrible if m much larger than d.
- **Advantage:** Time may be much less than breadth first search if solutions are dense.

Iterative deepening search

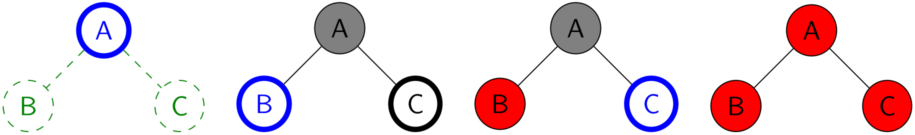
- **Definition 4.44.** Depth limited search is depth first search with a depth limit.
- **Definition 4.45.** Iterative deepening search (IDS) is depth limited search with ever increasing depth limits. We call the difference between successive depth limits the step size.
-

```
procedure Tree_Search (problem)
  <initialize the search tree using the initial state of problem>
  for depth = 0 to $\infty$
    result := Depth_Limited_search(problem,depth)
    if depth $\neq$ cutoff return result end if
  end for
end procedure
```

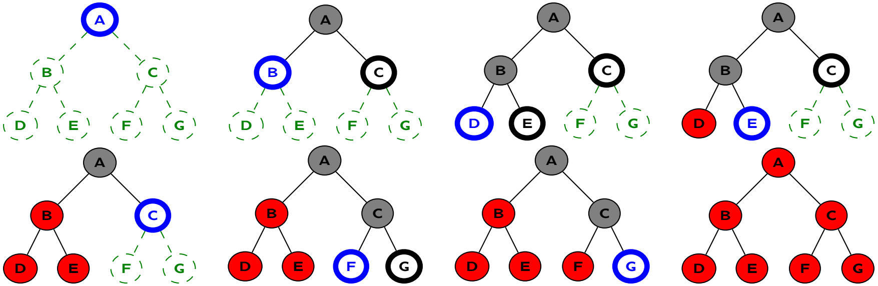
Ilustration: Iterative Deepening Search at various Limit Depths



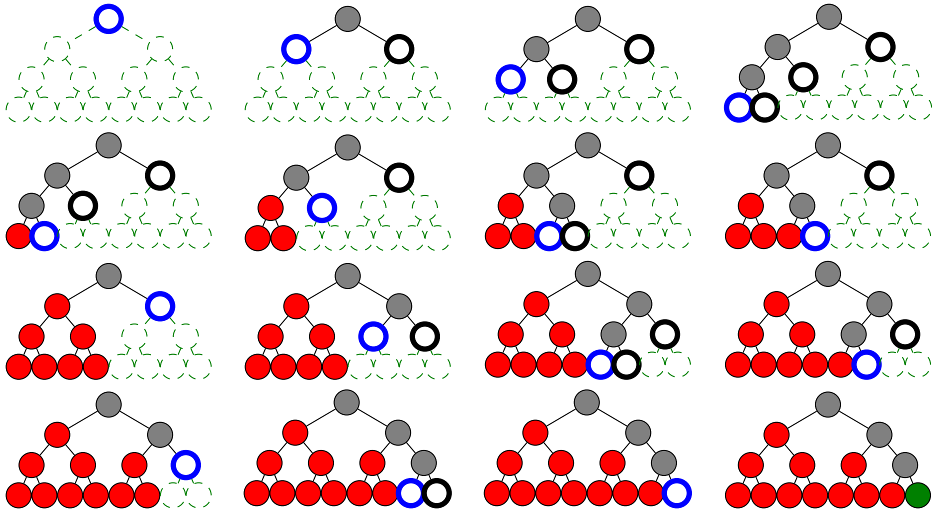
Ilustration: Iterative Deepening Search at various Limit Depths



Ilustration: Iterative Deepening Search at various Limit Depths



Ilustration: Iterative Deepening Search at various Limit Depths



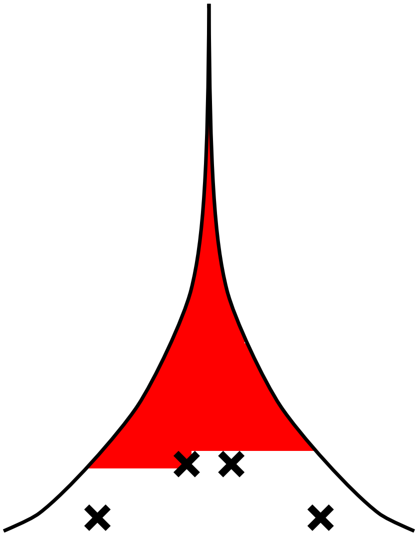
Iterative deepening search: Properties

Completeness	Yes
Time complexity	$(d+1)\cdot b^0+d\cdot b^1+(d-1)\cdot b^2+\ldots+b^d \in \mathcal{O}(b^{d+1})$
Space complexity	$\mathcal{O}(b\cdot d)$
Optimality	Yes (if step cost = 1)
- **Consequence:** IDS used in practice for search spaces of large, infinite, or unknown depth.

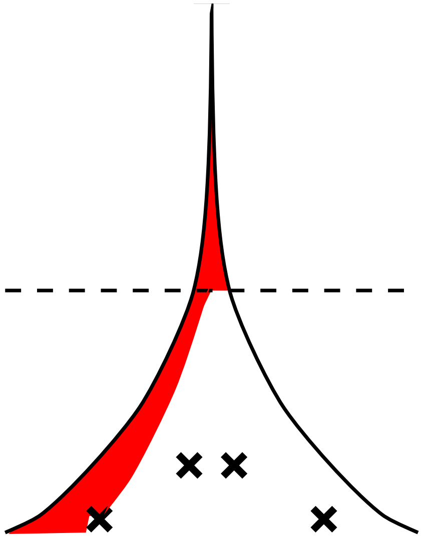
Comparison BFS (optimal) and IDS (not)

- **Example 4.46.** IDS may fail to be be optimal at step sizes > 1.

Breadth first search



Iterative deepening search



6.4.3 Further Topics

Tree Search vs. Graph Search

- We have only covered tree search algorithms.
- States duplicated in nodes are a huge problem for efficiency.
- **Definition 4.47.** A graph search algorithm is a variant of a tree search algorithm that prunes nodes whose state has already been considered (duplicate pruning), essentially using a DAG data structure.
- **Observation 4.48.** *Tree search is memory intensive it has to store the fringe so keeping a list of "explored states" does not lose much.*
- Graph versions of all the tree search algorithms considered here exist, but are more difficult to understand (and to prove properties about).
- The (time complexity) properties are largely stable under duplicate pruning. (no gain in the worst case)
- **Definition 4.49.** We speak of a search algorithm, when we do not want to distinguish whether it is a tree or graph search algorithm. (difference considered an implementation detail)

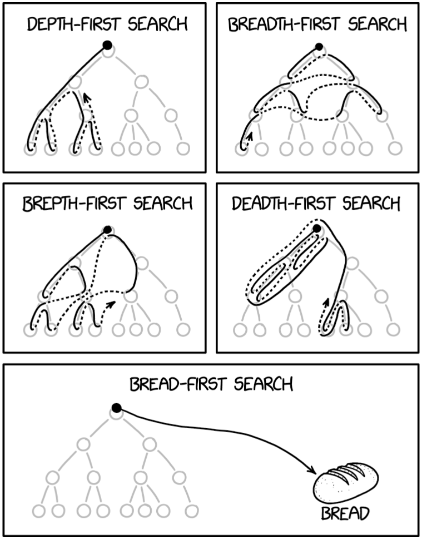
Uninformed Search Summary

- **Tree/Graph Search Algorithms:** Systematically explore the state tree/graph induced by a search problem in search of a goal state. Search strategies only differ by the treatment of the fringe.
- **Search Strategies and their Properties:** We have discussed

Criterion	Breadth first	Uniform cost	Depth first	Iterative deepening
Completeness	Yes[1]	Yes[2]	No	Yes
Time complexity	b^d	$\approx b^d$	b^m	b^{d+1}
Space complexity	b^d	$\approx b^d$	bm	bd
Optimality	Yes*	Yes	No	Yes*
Conditions	[1] b finite	[2] $0 < \epsilon \leq \mathrm{cost}$		

Search Strategies; the XKCD Take

- **More Search Strategies?:** (from https://xkcd.com/2407/)



6.5 Informed Search Strategies

Summary: Uninformed Search/Informed Search

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- Variety of uninformed search strategies.
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.
- **Next Step:** Introduce additional knowledge about the problem (heuristic search)
  - Best-first-, A^*-strategies (guide the search by heuristics)
  - Iterative improvement algorithms.
- **Definition 5.1.** A search algorithm is called informed, iff it uses some form of external information – that is not part of the search problem – to guide the search.

6.5.1 Greedy Search

Best-first search

- **Idea:** Order the fringe by estimated "desirability" (Expand most desirable unexpanded node)
- **Definition 5.2.** An evaluation function assigns a desirability value to each node of the search tree.
- **Note:** A evaluation function is not part of the search problem, but must be added externally.
- **Definition 5.3.** In best first search, the fringe is a queue sorted in decreasing order of desirability.
- **Special cases:** Greedy search, A^* search

Greedy search

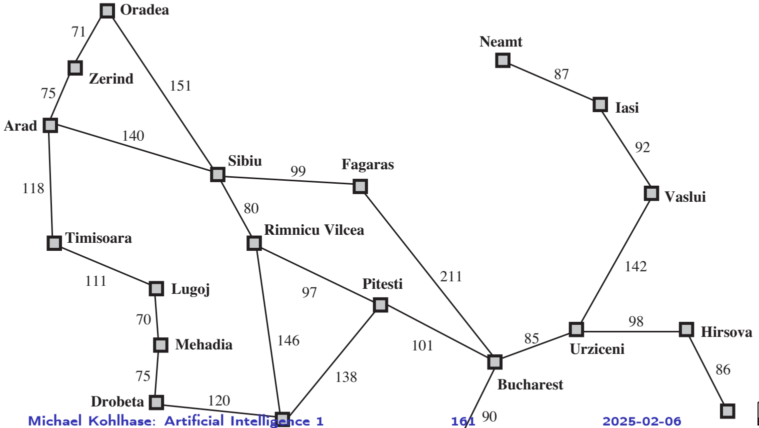
- **Idea:** Expand the node that *appears* to be closest to the goal.
- **Definition 5.4.** A heuristic is an evaluation function h on states that estimates the cost from n to the nearest goal state. We speak of heuristic search if the search algorithm uses a heuristic in some way.
- **Note:** All nodes for the same state must have the same h-value!
- **Definition 5.5.** Given a heuristic h, greedy search is the strategy where the fringe is organized as a queue sorted by increasing h value.
- **Example 5.6.** Straight-line distance from/to Bucharest.
- **Note:** Unlike uniform cost search the node evaluation function has nothing to do with the nodes expanded so far

$$\begin{aligned} \text{internal search control} &\rightsquigarrow \text{external search control} \\ \text{partial solution cost} &\rightsquigarrow \text{goal cost estimation} \end{aligned}$$

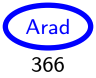
Romania with Straight-Line Distances

- **Example 5.7 (Informed Travel).**
  $h_{\mathrm{SLD}}(n) = \mathit{straight} - \mathit{line\ distance\ to\ Bucharest}$

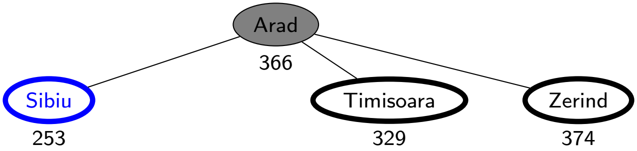
Arad	366	Mehadia	241	Bucharest	0	Neamt	234
Craiova	160	Oradea	380	Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193	Fragaras	176	Sibiu	253
Giurgiu	77	Timisoara	329	Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199	Lugoj	244	Zerind	374



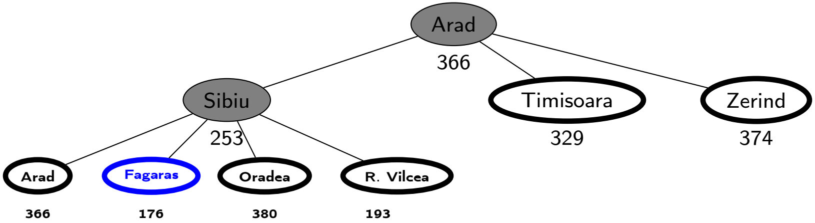
Greedy Search: Romania



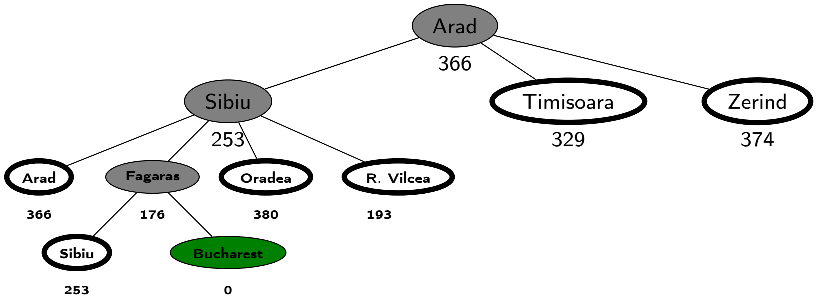
Greedy Search: Romania



Greedy Search: Romania

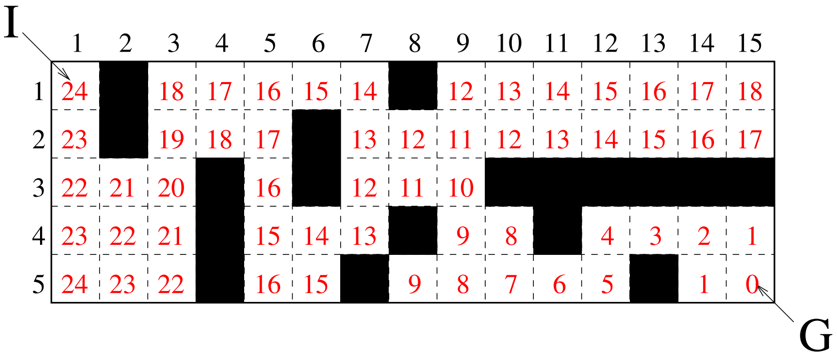


Greedy Search: Romania



Heuristic Functions in Path Planning

- **Example 5.8 (The maze solved).** We indicate h^* by giving the goal distance:

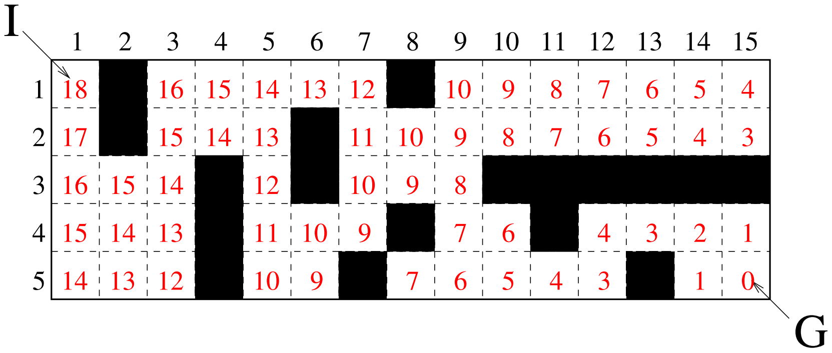


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	24	■	18	17	16	15	14	■	12	13	14	15	16	17	18
2	23	■	19	18	17	■	13	12	11	12	13	14	15	16	17
3	22	21	20	■	16	■	12	11	10	■	■	■	■	■	■
4	23	22	21	■	15	14	13	■	9	8	■	4	3	2	1
5	24	23	22	■	16	15	■	9	8	7	6	5	■	1	0

- **Example 5.9 (Maze Heuristic: The good case).** We use the Manhattan distance to the goal as a heuristic:

Heuristic Functions in Path Planning

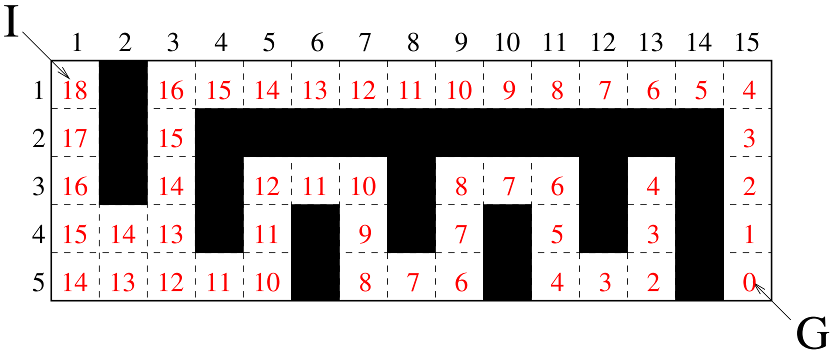
- **Example 5.11 (The maze solved).** We indicate h^* by giving the goal distance:
- **Example 5.12 (Maze Heuristic: The good case).** We use the Manhattan distance to the goal as a heuristic:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	18	■	16	15	14	13	12	■	10	9	8	7	6	5	4
2	17	■	15	14	13	■	11	10	9	8	7	6	5	4	3
3	16	15	14	■	12	■	10	9	8	■	■	■	■	■	■
4	15	14	13	■	11	10	9	■	7	6	■	4	3	2	1
5	14	13	12	■	10	9	■	7	6	5	4	3	■	1	0

Heuristic Functions in Path Planning

- **Example 5.14 (The maze solved).** We indicate h^* by giving the goal distance:
- **Example 5.15 (Maze Heuristic: The good case).** We use the Manhattan distance to the goal as a heuristic:
- **Example 5.16 (Maze Heuristic: The bad case).** We use the Manhattan distance to the goal as a heuristic again:



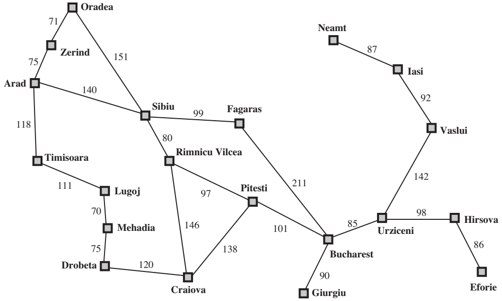
Greedy search: Properties

Completeness	No: Can get stuck in infinite loops. Complete in finite state spaces with repeated state checking
Time complexity	$\mathcal{O}(b^m)$
Space complexity	$\mathcal{O}(b^m)$
Optimality	No

Greedy search: Properties

Completeness	No: Can get stuck in infinite loops. Complete in finite state spaces with repeated state checking
Time complexity	$\mathcal{O}(b^m)$
Space complexity	$\mathcal{O}(b^m)$
Optimality	No

- **Example 5.18.** Greedy search can get stuck going from Iasi to Oradea: Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \cdots$



Greedy search: Properties

Completeness	No: Can get stuck in infinite loops. Complete in finite state spaces with repeated state checking
Time complexity	$\mathcal{O}(b^m)$
Space complexity	$\mathcal{O}(b^m)$
Optimality	No

- **Example 5.19.** Greedy search can get stuck going from Iasi to Oradea: Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \cdots$
- **Worst-case Time:** Same as depth first search.
- **Worst-case Space:** Same as breadth first search. ($\leftsquigarrow$ repeated state checking)
- **But:** A good heuristic can give dramatic improvements.

6.5.2 Heuristics and their Properties

Heuristic Functions

- **Definition 5.20.** Let Π be a search problem with states $\mathcal{S}$. A heuristic function (or short heuristic) for Π is a function $h\colon \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$ so that $h(s) = 0$ whenever s is a goal state.
- $h(s)$ is intended as an estimate the distance between state s and the nearest goal state.
- **Definition 5.21.** Let Π be a search problem with states $\mathcal{S}$, then the function $h^*\colon S \to \mathbb{R}_0^+ \cup \{\infty\}$, where $h^*(s)$ is the cost of a cheapest path from s to a goal state, or ∞ if no such path exists, is called the goal distance function for Π.
- **Notes:**
  - $h(s) = 0$ on goal states: If your estimator returns "I think it's still a long way" on a goal state, then its intelligence is, um …
  - Return value ∞: To indicate dead ends, from which the goal state can't be reached anymore.
  - The distance estimate depends only on the state s, not on the node (i.e., the path we took to reach s).

Where does the word “Heuristic” come from?

- Ancient Greek word $\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$ ($\widehat{=}$ “I find”) (aka. $\epsilon\upsilon\rho\epsilon\kappa\alpha$!)
- Popularized in modern science by George Polya: “How to solve it” [Pól73]
- Same word often used for “rule of thumb” or “imprecise solution method”.

Heuristic Functions: The Eternal Trade-Off

- "Distance Estimate"? (h is an arbitrary function in principle)
  - In practice, we want it to be *accurate* (aka: *informative*), i.e., close to the actual goal distance.
  - We also want it to be fast, i.e., a small overhead for computing h.
  - These two wishes are in contradiction!
- **Example 5.22 (Extreme cases).**
  - $h = 0$: no overhead at all, completely un-informative.
  - $h = h^*$: perfectly accurate, overhead $\widehat{=}$ solving the problem in the first place.
- **Observation 5.23.** *We need to trade off the accuracy of h against the overhead for computing it.*

Properties of Heuristic Functions

- **Definition 5.24.** Let Π be a search problem with states S and actions A. We say that a heuristic h for Π is admissible if $h(s) \leq h^*(s)$ for all $s \in S$.
  We say that h is consistent if $h(s) - h(s') \leq c(a)$ for all $s \in S$, $a \in A$, and $s' \in \mathcal{T}(s, a)$.
- **In other words . . . :**
  - h is admissible if it is a lower bound on goal distance.
  - h is consistent if, when applying an action a, the heuristic value cannot decrease by more than the cost of a.

Properties of Heuristic Functions, ctd.

- Let Π be a search problem, and let h be a heuristic for Π. If h is consistent, then h is admissible.
- *Proof:* we prove $h(s) \leq h^*(s)$ for all $s \in S$ by induction over the length of the cheapest path to a goal node.
  1. base case
     1.1. $h(s) = 0$ by definition of heuristic, so $h(s) \leq h^*(s)$ as desired.
  2. step case
     2.1. We assume that $h(s') \leq h^*(s)$ for all states s' with a cheapest goal node path of length n.
     2.2. Let s be a state whose cheapest goal path has length $n+1$ and the first transition is $o = (s,s')$.
     2.3. By consistency, we have $h(s) - h(s') \leq c(o)$ and thus $h(s) \leq h(s') + c(o)$.
     2.4. By construction, $h^*(s)$ has a cheapest goal path of length n and thus, by induction hypothesis $h(s') \leq h^*(s')$.
     2.5. By construction, $h^*(s) = h^*(s') + c(o)$.
     2.6. Together this gives us $h(s) \leq h^*(s)$ as desired.
- Consistency is a sufficient condition for admissibility (easier to check)

Properties of Heuristic Functions: Examples

- **Example 5.25.** Straight line distance is admissible and consistent by the triangle inequality.
  If you drive 100km, then the straight line distance to Rome can't decrease by more than 100km.
- **Observation:** In practice, admissible heuristics are typically consistent.
- **Example 5.26 (An admissible, but inconsistent heuristic).** When traveling to Rome, let $h(Munich) = 300$ and $h(Innsbruck) = 100$.
- **Inadmissible heuristics** typically arise as approximations of admissible heuristics that are too costly to compute. (see later)

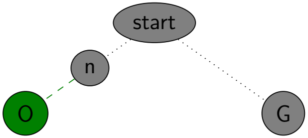
6.5.3 A-Star Search

A^* Search: Evaluation Function

- **Idea:** Avoid expanding paths that are already expensive (make use of actual cost)
  The simplest way to combine heuristic and path cost is to simply add them.
- **Definition 5.27.** The evaluation function for A^* search is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost for n and $h(n)$ is the estimated cost to the nearest goal from n.
- Thus $f(n)$ is the estimated total cost of the path through n to a goal.
- **Definition 5.28.** Best first search with evaluation function $g + h$ is called A^* search.

A^* Search: Optimality

- **Theorem 5.29.** *A^* search with admissible heuristic is optimal.*
- *Proof:* We show that sub-optimal nodes are never expanded by A^*
  1. Suppose a suboptimal goal node G has been generated then we are in the following situation:

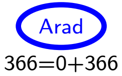


  2. Let n be an unexpanded node on a path to an optimality goal node O, then

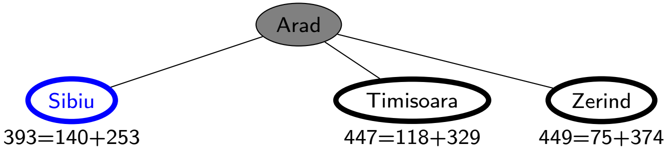
$$\begin{array}{ll} f(G) = g(G) & \text{since } h(G) = 0 \\ g(G) > g(O) & \text{since } G \text{ suboptimal} \\ g(O) = g(n) + h^*(n) & n \text{ on optimal path} \\ g(n) + h^*(n) \geq g(n) + h(n) & \text{since } h \text{ is admissible} \\ g(n) + h(n) = f(n) & \end{array}$$

  3. Thus, $f(G) > f(n)$ and A^* never expands G.

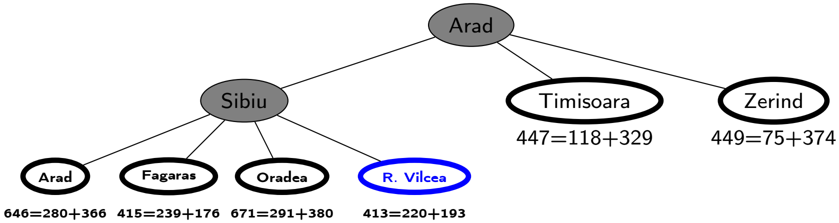
A^* Search Example



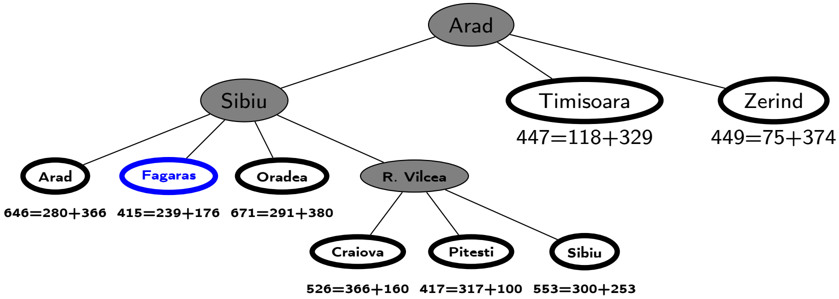
A^* Search Example



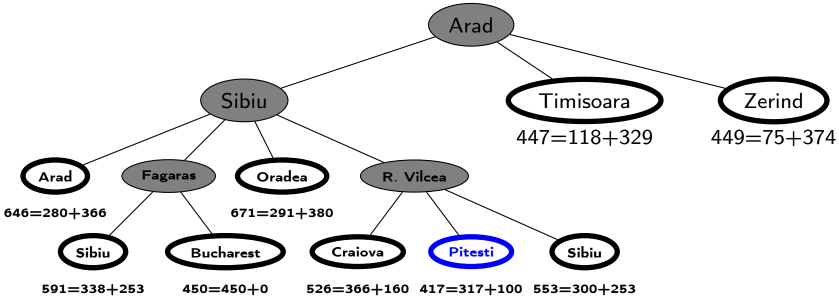
A^* Search Example



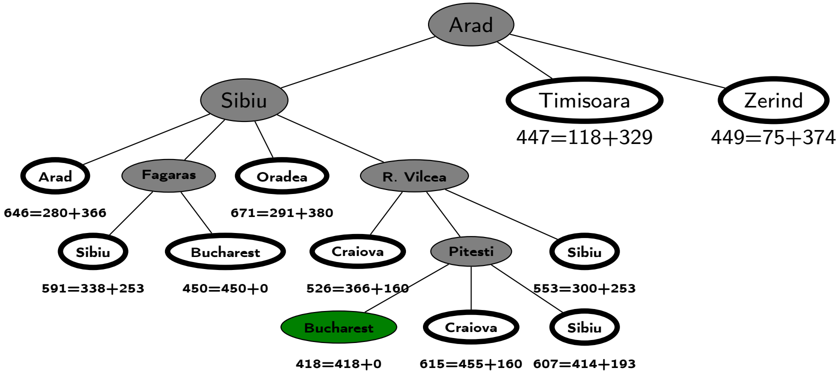
A^* Search Example



A^* Search Example

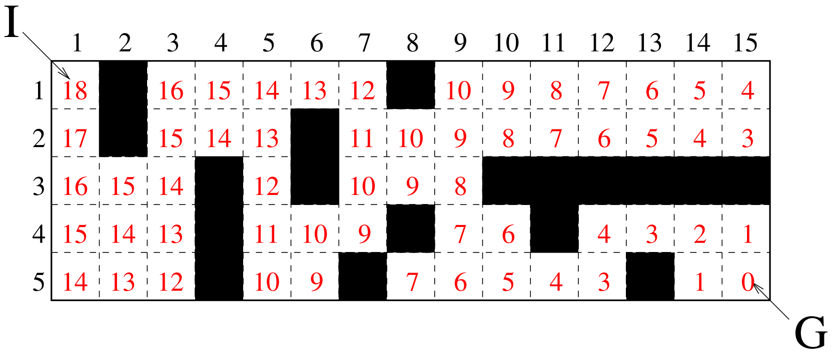


A^* Search Example



Additional Observations (Not Limited to Path Planning)

- **Example 5.30 (Greedy best-first search, "good case").**

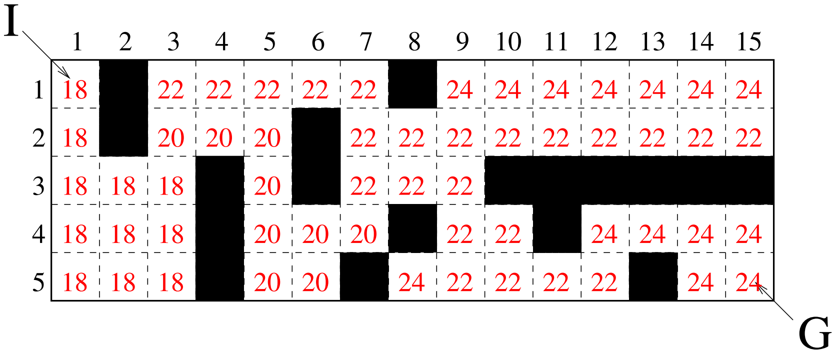


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	18	■	16	15	14	13	12	■	10	9	8	7	6	5	4
2	17	■	15	14	13	■	11	10	9	8	7	6	5	4	3
3	16	15	14	■	12	■	10	9	8	■	■	■	■	■	■
4	15	14	13	■	11	10	9	■	7	6	■	4	3	2	1
5	14	13	12	■	10	9	■	7	6	5	4	3	■	1	0

We will find a solution with little search.

Additional Observations (Not Limited to Path Planning)

- **Example 5.31 (A^* ($g + h$), "good case").**

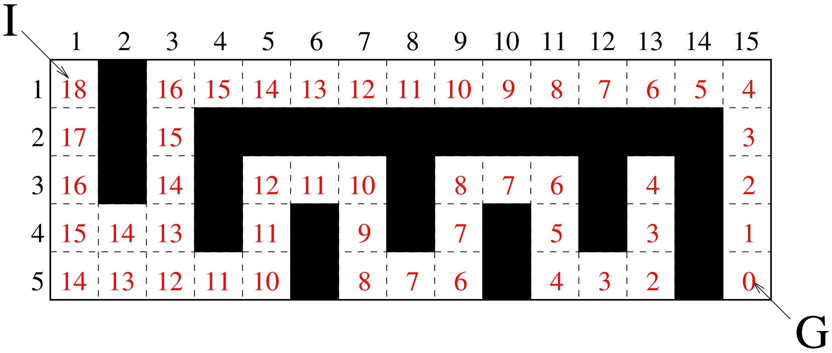


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	18	■	22	22	22	22	22	■	24	24	24	24	24	24	24
2	18	■	20	20	20	■	22	22	22	22	22	22	22	22	22
3	18	18	18	■	20	■	22	22	22	■	■	■	■	■	■
4	18	18	18	■	20	20	20	■	22	22	■	24	24	24	24
5	18	18	18	■	20	20	■	24	22	22	22	22	■	24	24

  - In A^* with a consistent heuristic, $g + h$ always increases monotonically (h cannot decrease more than g increases)
  - We need more search, in the "right upper half". This is typical: Greedy best first search tends to be faster than A^*.

Additional Observations (Not Limited to Path Planning)

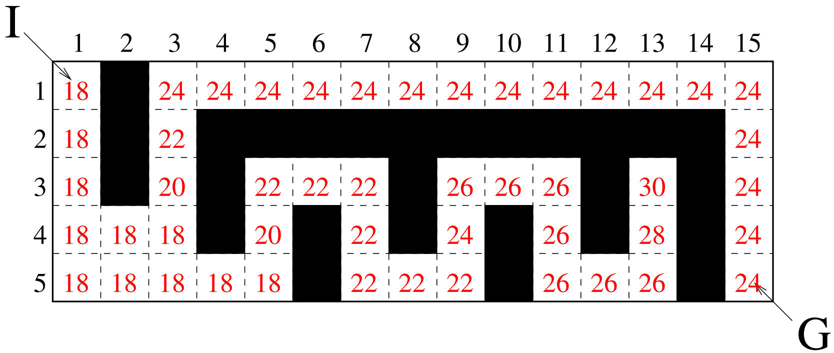
- **Example 5.32 (Greedy best-first search, "bad case").**



Search will be mis-guided into the "dead-end street".

Additional Observations (Not Limited to Path Planning)

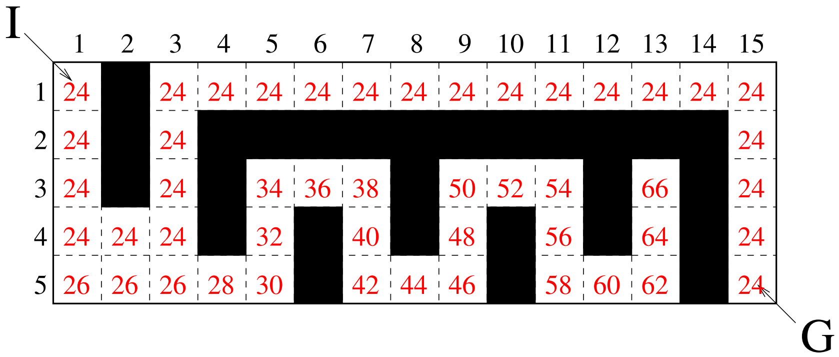
- **Example 5.33 (A^* ($g+h$), "bad case").**



We will search less of the "dead-end street". Sometimes $g+h$ gives better search guidance than h. ($\rightsquigarrow$ A^* is faster there)

Additional Observations (Not Limited to Path Planning)

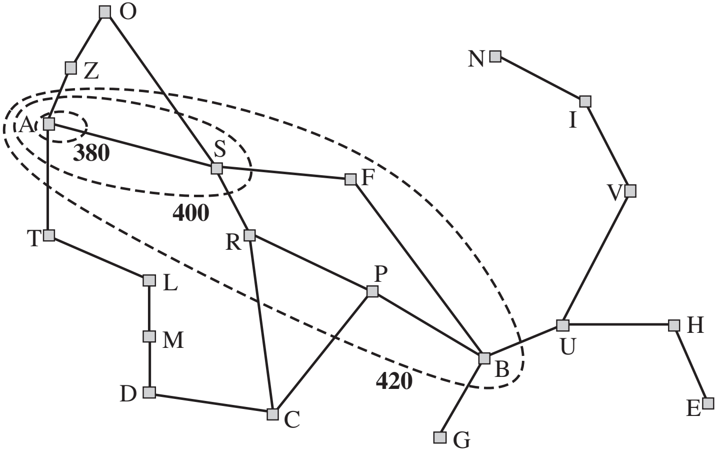
- **Example 5.34 (A^* $(g + h)$ using h^*).**



In A^*, node values always increase monotonically (with any heuristic). If the heuristic is perfect, they remain constant on optimal paths.

A^* search: f-contours

- **Intuition:** A^*-search gradually adds "f-contours" (areas of the same f-value) to the search.



A^* search: Properties

- Properties or A^*-search:

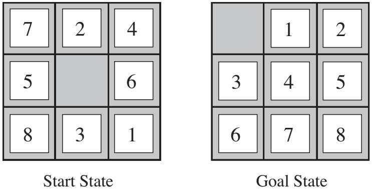
Completeness	Yes (unless there are infinitely many nodes n with $f(n) \leq f(0)$)
Time complexity	Exponential in [relative error in h $\times$ length of solution]
Space complexity	Same as time (variant of BFS)
Optimality	Yes

n

- A^*-search expands all (some/no) nodes with $f(n) < h^*(n)$
- The run-time depends on how well we approximated the real cost h^* with h.

6.5.4 Finding Good Heuristics

Admissible heuristics: Example 8-puzzle



- **Example 5.35.** Let $h_1(n)$ be the number of misplaced tiles in node n. $(h_1(S) = 9)$
- **Example 5.36.** Let $h_2(n)$ be the total Manhattan distance from desired location of each tile. $(h_2(S) = 3 + 1 + 2 + 2 + 2 + 3 + 2 + 2 + 3 = 20)$
- **Observation 5.37 (Typical search costs).** (*IDS* $\widehat{=}$ *iterative deepening search*)

nodes explored	*IDS*	$A^*(h_1)$	$A^*(h_2)$
$d = 14$	*3,473,941*	*539*	*113*
$d = 24$	*too many*	*39,135*	*1,641*

Dominance

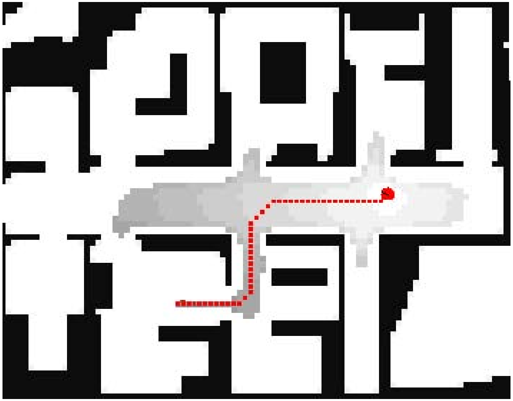
- **Definition 5.38.** Let h_1 and h_2 be two admissible heuristics we say that h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all n.
- **Theorem 5.39.** *If h_2 dominates h_1, then h_2 is better for search than h_1.*
- *Proof sketch:* If h_2 dominates h_1, then h_2 is "closer to h^*" than h_1, which means better search performance.

Relaxed problems

- **Observation:** Finding good admissible heuristics is an art!
- **Idea:** Admissible heuristics can be derived from the *exact* solution cost of a relaxed version of the problem.
- **Example 5.40.** If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then we get heuristic h_1.
- **Example 5.41.** If the rules are relaxed so that a tile can move to *any adjacent square*, then we get heuristic h_2. (Manhattan distance)
- **Definition 5.42.** Let $\Pi := \langle\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{I},\mathcal{G}\rangle$ be a search problem, then we call a search problem $\mathcal{P}^r := \langle\mathcal{S},\mathcal{A}^r,\mathcal{T}^r,\mathcal{I}^r,\mathcal{G}^r\rangle$ a relaxed problem (wrt. Π; or simply relaxation of Π), iff $\mathcal{A}\subseteq\mathcal{A}^r$, $\mathcal{T}\subseteq\mathcal{T}^r$, $\mathcal{I}\subseteq\mathcal{I}^r$, and $\mathcal{G}\subseteq\mathcal{G}^r$.
- **Lemma 5.43.** *If $\mathcal{P}^r$ relaxes Π, then every solution for Π is one for $\mathcal{P}^r$.*
- **Key point:** The optimal solution cost of a relaxed problem is not greater than the optimal solution cost of the real problem.

Empirical Performance: A^* in Path Planning

- **Example 5.44 (Live Demo vs. Breadth-First Search).**



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

- **Difference to Breadth-first Search?:** That would explore all grid cells in a *circle* around the initial state!

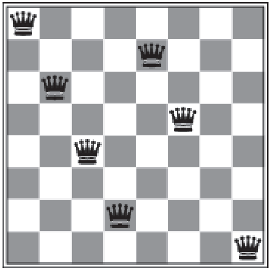
6.6 Local Search

Systematic Search vs. Local Search

- **Definition 6.1.** We call a search algorithm systematic, if it considers all states at some point.
- **Example 6.2.** All tree search algorithms (except pure depth first search) are systematic. (given reasonable assumptions e.g. about costs.)
- **Observation 6.3.** *Systematic search algorithms are complete.*
- **Observation 6.4.** *In systematic search algorithms there is no limit of the number of nodes that are kept in memory at any time.*
- **Alternative:** Keep only one (or a few) nodes at a time
  - $\rightsquigarrow$ no systematic exploration of all options, $\rightsquigarrow$ incomplete.

Local Search Problems

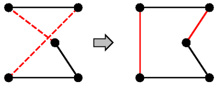
- **Idea:** Sometimes the path to the solution is irrelevant.
- **Example 6.5 (8 Queens Problem).** Place 8 queens on a chess board, so that no two queens threaten each other.
- This problem has various solutions (the one of the right isn't one of them)
- **Definition 6.6.** A local search algorithm is a search algorithm that operates on a single state, the current state (rather than multiple paths). (advantage: constant space)



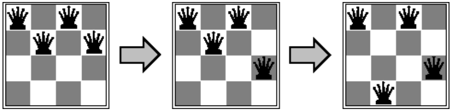
- Typically local search algorithms only move to successor of the current state, and do not retain search paths.
- Applications include: integrated circuit design, factory-floor layout, job-shop scheduling, portfolio management, fleet deployment,...

Local Search: Iterative improvement algorithms

- **Definition 6.7.** The traveling salesman problem (TSP is to find shortest trip through set of cities such that each city is visited exactly once.
- **Idea:** Start with any complete tour, perform pairwise exchanges



- **Definition 6.8.** The n-queens problem is to put n queens on $n \times n$ board such that no two queen in the same row, columns, or diagonal.
- **Idea:** Move a queen to reduce number of conflicts



Hill-climbing (gradient ascent/descent)

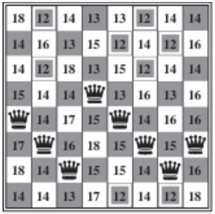
- **Idea:** Start anywhere and go in the direction of the steepest ascent.
- **Definition 6.9.** Hill climbing (also gradient ascent) is a local search algorithm that iteratively selects the best successor:

```
procedure Hill−Climbing (problem) /∗ a state that is a local minimum ∗/
  local current, neighbor /∗ nodes ∗/
  current := Make−Node(Initial−State[problem])
  loop
    neighbor := <a highest−valued successor of current>
    if Value[neighbor] < Value[current] return [current] end if
    current := neighbor
  end loop
end procedure
```

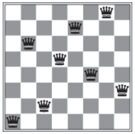
- **Intuition:** Like best first search without memory.
- Works, if solutions are dense and local maxima can be escaped.

Example Hill Climbing with 8 Queens

- **Idea:** Consider $h \mathrel{\widehat{=}}$ number of queens that threaten each other.
- **Example 6.10.** An 8-queens state with heuristic cost estimate $h = 17$ showing h-values for moving a queen within its column:

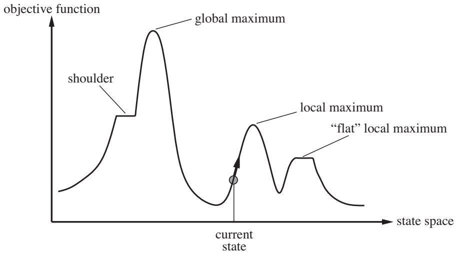


- **Problem:** The state space has local minima. e.g. the board on the right has $h = 1$ but every successor has $h > 1$.



Hill-climbing

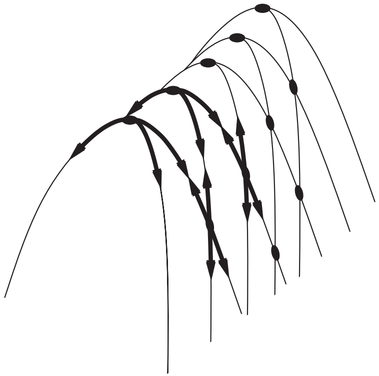
- **Problem:** Depending on initial state, can get stuck on local maxima/minima and plateaux.
- "Hill-climbing search is like climbing Everest in thick fog with amnesia".



- **Idea:** Escape local maxima by allowing some "bad" or random moves.
- **Example 6.11.** local search, simulated annealing, ...
- **Properties:** All are incomplete, nonoptimal.
- Sometimes performs well in practice (if (optimal) solutions are dense)

Simulated annealing (Idea)

- **Definition 6.12.** Ridges are ascending successions of local maxima.
- **Problem:** They are extremely difficult to bv navigate for local search algorithms.
- **Idea:** Escape local maxima by allowing some "bad" moves, but gradually decrease their size and frequency.



- Annealing is the process of heating steel and let it cool gradually to give it time to grow an optimal cristal structure.
- Simulated annealing is like shaking a ping pong ball occasionally on a bumpy surface to free it. (so it does not get stuck)
- Devised by Metropolis et al for physical process modelling [Met+53]
- Widely used in VLSI layout, airline scheduling, etc.

Simulated annealing (Implementation)

- **Definition 6.13.** The following algorithm is called simulated annealing:

```
procedure Simulated−Annealing (problem,schedule) /* a solution state */
  local node, next /* nodes */
   local T /* a ‘‘temperature’’ controlling prob.~of downward steps */
   current := Make−Node(Initial−State[problem])
   for t :=1 to $\infty$
    T := schedule[t]
      if T = 0 return current end if
      next := <a randomly selected successor of current>
     $\Delta(E)$ := Value[next]−Value[current]
      if $\Delta(E) > 0$ current := next
      else
       current := next <only with probability> $e^{\Delta(E)/T}$
     end if
  end for
end procedure
```

A schedule is a mapping from time to “temperature”.

Properties of simulated annealing

- At fixed “temperature” T, state occupation probability reaches Boltzman distribution
$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$
T decreased slowly enough $\rightsquigarrow$ always reach best state x^* because
$$\frac{e^{\frac{E(x^*)}{kT}}}{e^{\frac{E(x)}{kT}}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$$
for small T.
- **Question:** Is this necessarily an interesting guarantee?

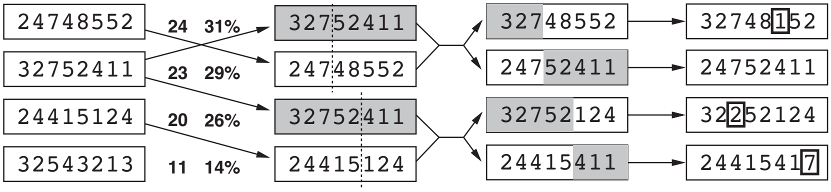
Local beam search

- **Definition 6.14.** Local beam search is a search algorithm that keep k states instead of 1 and chooses the top k of all their successors.
- **Observation:** Local beam search is not the same as k searches run in parallel! (Searches that find good states recruit other searches to join them)
- **Problem:** Quite often, all k searches end up on the same local hill!
- **Idea:** Choose k successors randomly, biased towards good ones. (Observe the close analogy to natural selection!)

Genetic algorithms (very briefly)

- **Definition 6.15.** A genetic algorithm is a variant of local beam search that generates successors by
  - randomly modifying states (mutation)
  - mixing pairs of states (sexual reproduction or crossover)

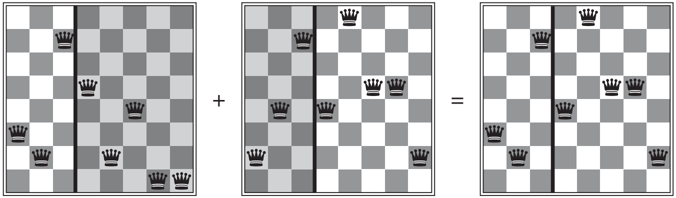
  to optimize a fitness function. (survival of the fittest)
- **Example 6.16.** Generating successors for 8 queens



(a) Initial Population (b) Fitness Function (c) Selection (d) Crossover (e) Mutation

Genetic algorithms (continued)

- **Problem:** Genetic algorithms require states encoded as strings.
- Crossover only helps iff substrings are meaningful components.
- **Example 6.17 (Evolving 8 Queens).** First crossover



- **Note:** Genetic algorithms $\neq$ evolution: e.g., real genes also encode replication machinery!

Chapter 7
Adversarial Search for Game Playing

7.1 Introduction

The Problem

- **The Problem of Game-Play:** cf. **??**
- **Example 1.1.**

- **Definition 1.2.** Adversarial search $\widehat{=}$ Game playing against an opponent.

Why Game Playing?

- **What do you think?**
  - Playing a game well clearly requires a form of "intelligence".
  - Games capture a pure form of competition between opponents.
  - Games are abstract and precisely defined, thus very easy to formalize.
- Game playing is one of the oldest sub-areas of AI (ca. 1950).
- The dream of a machine that plays chess is, indeed, *much* older than AI!

"Schachtürke" (1769)

"El Ajedrecista" (1912)

"Game" Playing? *Which* Games?

- ...sorry, we're not gonna do soccer here.
- **Definition 1.3 (Restrictions).** A game in the sense of AI-1 is one where
  - Game state discrete, number of game state finite.
  - Finite number of possible moves.
  - The game state is fully observable.
  - The outcome of each move is deterministic.
  - Two players: Max and Min.
  - Turn-taking: It's each player's turn alternatingly. Max begins.
  - Terminal game states have a utility u. Max tries to maximize u, Min tries to minimize u.
  - In that sense, the utility for Min is the exact opposite of the utility for Max ("zero sum").
  - There are no infinite runs of the game (no matter what moves are chosen, a terminal state is reached after a finite number of moves).

An Example Game

- Game states: Positions of figures.
- Moves: Given by rules.
- Players: white (Max), black (Min).
- Terminal states: checkmate.
- Utility of terminal states, e.g.:
  - $+100$ if black is checkmated.
  - 0 if stalemate.
  - -100 if white is checkmated.

“Game” Playing? Which Games *Not*?

- Soccer (sorry guys; not even RoboCup)
- Important types of games that we don’t tackle here:
  - Chance. (E.g., backgammon)
  - More than two players. (E.g., Halma)
  - Hidden information. (E.g., most card games)
  - Simultaneous moves. (E.g., Diplomacy)
  - Not zero-sum, i.e., outcomes may be beneficial (or detrimental) for both players. (cf. Game theory: Auctions, elections, economy, politics, . . .)
- Many of these more general game types can be handled by similar/extended algorithms.

(A Brief Note On) Formalization

- **Definition 1.4.** An adversarial search problem is a search problem $\langle\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{I},\mathcal{G}\rangle$, where
  1. $\mathcal{S} = \mathcal{S}^{\mathrm{Max}} \uplus \mathcal{S}^{\mathrm{Min}} \uplus \mathcal{G}$ and $\mathcal{A} = \mathcal{A}^{\mathrm{Max}} \uplus \mathcal{A}^{\mathrm{Min}}$
  2. For $a \in \mathcal{A}^{\mathrm{Max}}$, if $s \xrightarrow{a} s'$ then $s \in \mathcal{S}^{\mathrm{Max}}$ and $s' \in (\mathcal{S}^{\mathrm{Min}} \cup \mathcal{G})$.
  3. For $a \in \mathcal{A}^{\mathrm{Min}}$, if $s \xrightarrow{a} s'$ then $s \in \mathcal{S}^{\mathrm{Min}}$ and $s' \in (\mathcal{S}^{\mathrm{Max}} \cup \mathcal{G})$.

  together with a game utility function $u\colon \mathcal{G} \to \mathbb{R}$. (the “score” of the game)
- **Definition 1.5 (Commonly used terminology).**
  position $\widehat{=}$ state, move $\widehat{=}$ action, end state $\widehat{=}$ terminal state $\widehat{=}$ goal state.
- **Remark:** A round of the game – one move Max, one move Min – is often referred to as a “move”, and individual actions as “half-moves” (we *don’t* in AI-1)

Why Games are Hard to Solve: I

- What is a "solution" here?
- **Definition 1.6.** Let Θ be an adversarial search problem, and let $X \in \{\mathrm{Max}, \mathrm{Min}\}$. A strategy for X is a function $\sigma^X : \mathcal{S}^X \to \mathcal{A}^X$ so that a is applicable to s whenever $\sigma^X(s) = a$.
- We don't know how the opponent will react, and need to prepare for all possibilities.
- **Definition 1.7.** A strategy is called optimal if it yields the best possible utility for X assuming perfect opponent play (not formalized here).
- **Problem:** In (almost) all games, computing an optimal strategy is infeasible. (state/search tree too huge)
- **Solution:** Compute the next move "on demand", given the current state instead.

Why Games are hard to solve II

- **Example 1.8.** Number of reachable states in chess: 10^{40}.
- **Example 1.9.** Number of reachable states in go: 10^{100}.
- **It's even worse:** Our algorithms here look at search trees (game trees), no duplicate pruning.
- **Example 1.10.**
  - Chess without duplicate pruning: $35^{100} \simeq 10^{154}$.
  - Go without duplicate pruning: $200^{300} \simeq 10^{690}$.

How To Describe a Game State Space?

- Like for classical search problems, there are three possible ways to describe a game: blackbox/API description, declarative description, explicit game state space.
- **Question:** Which ones do humans use?
  - Explicit $\approx$ Hand over a book with all 10^{40} moves in chess.
  - Blackbox $\approx$ Give possible chess moves on demand but don't say how they are generated.
- **Answer:** Declarative!
  With "game description language" $\widehat{=}$ natural language.

Specialized vs. General Game Playing

- And which game descriptions do computers use?
  - Explicit: Only in illustrations.
  - Blackbox/API: Assumed description in (This Chapter)
    - Method of choice for all those game players out there in the market (Chess computers, video game opponents, you name it).
    - Programs designed for, and specialized to, a particular game.
    - Human knowledge is key: evaluation functions (see later), opening databases (chess!!), end game databases.
  - Declarative: General game playing, active area of research in AI.
    - Generic game description language (GDL), based on logic.
    - Solvers are given only "the rules of the game", no other knowledge/input whatsoever (cf. **??**).
    - Regular academic competitions since 2005.

Our Agenda for This Chapter

- **Minimax Search:** How to compute an optimal strategy?
  - Minimax is the canonical (and easiest to understand) algorithm for *solving* games, i.e., computing an optimal strategy.
- **Evaluation functions:** But what if we don't have the time/memory to solve the entire game?
  - Given limited time, the best we can do is look ahead as far as we can. Evaluation functions tell us how to evaluate the leaf states at the cut off.
- **Alphabeta search:** How to prune unnecessary parts of the tree?
  - Often, we can detect early on that a particular action choice cannot be part of the optimal strategy. We can then stop considering this part of the game tree.
- **State of the art:** What is the state of affairs, for prominent games, of computer game playing vs. human experts?
  - Just FYI (not part of the technical content of this course).

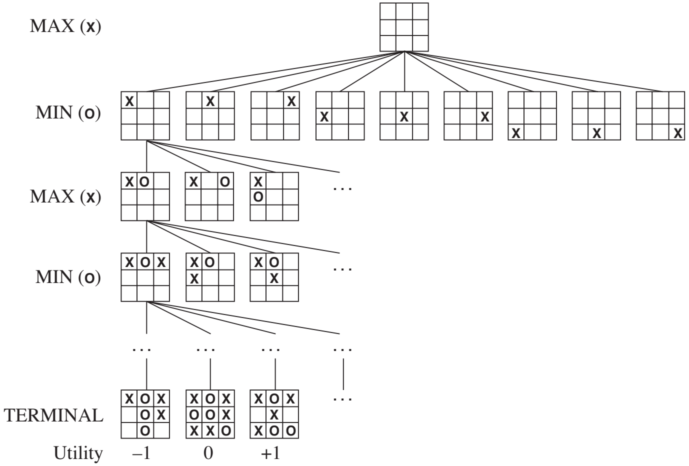
7.2 Minimax Search

“Minimax”?

- We want to compute an optimal strategy for player “Max”.
  - In other words: *We are Max, and our opponent is Min.*
- **Recall:** We compute the strategy offline, before the game begins. During the game, whenever it’s our turn, we just look up the corresponding action.
- **Idea:** Use tree search using an extension $\hat{u}$ of the utility function u to inner nodes. $\hat{u}$ is computed recursively from u during search:
  - Max attempts to maximize $\hat{u}(s)$ of the terminal states reachable during play.
  - Min attempts to minimize $\hat{u}(s)$.
- The computation alternates between minimization and maximization $\leadsto$ hence “minimax”.

Example Tic-Tac-Toe

- **Example 2.1.** A full game tree for tic-tac-toe

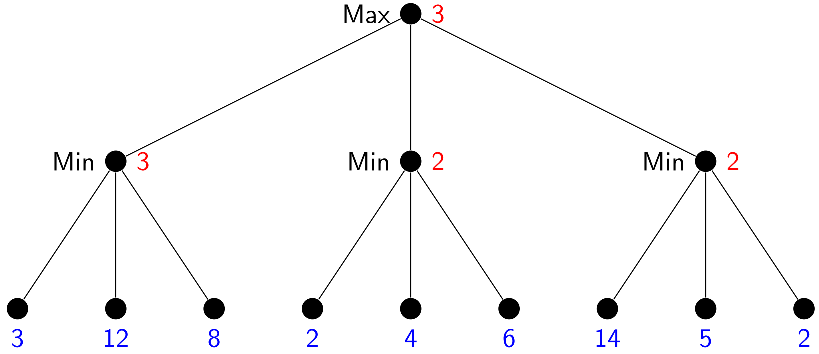


  - current player and action marked on the left.
  - Last row: terminal positions with their utility.

Minimax: Outline

- **We max, we min, we max, we min . . .**
  1. Depth first search in game tree, with Max in the root.
  2. Apply game utility function to terminal positions.
  3. Bottom-up for each inner node *n* in the search tree, compute the utility $\hat{u}(n)$ of *n* as follows:
     - If it's Max's turn: Set $\hat{u}(n)$ to the maximum of the utilities of *n*'s successor nodes.
     - If it's Min's turn: Set $\hat{u}(n)$ to the minimum of the utilities of *n*'s successor nodes.
  4. Selecting a move for Max at the root: Choose one move that leads to a successor node with maximal utility.

Minimax: Example



- **Blue numbers:** Utility function u applied to terminal positions.
- **Red numbers:** Utilities of inner nodes, as computed by the minimax algorithm.

The Minimax Algorithm: Pseudo-Code

- **Definition 2.2.** The minimax algorithm (often just called minimax) is given by the following functions whose argument is a state $s \in \mathcal{S}^{\mathrm{Max}}$, in which Max is to move.

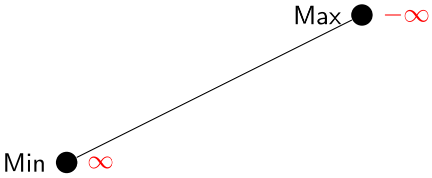
```
function Minimax−Decision(s) returns an action
  v := Max−Value(s)
  return an action yielding value v in the previous function call

function Max−Value(s) returns a utility value
  if Terminal−Test(s) then return u(s)
  v := −∞
  for each a ∈ Actions(s) do
    v := max(v,Min−Value(ChildState(s,a)))
  return v

function Min−Value(s) returns a utility value
  if Terminal−Test(s) then return u(s)
  v := +∞
  for each a ∈ Actions(s) do
    v := min(v,Max−Value(ChildState(s,a)))
  return v
```

We call nodes, where Max/Min acts Max-nodes/Min-nodes.

Minimax: Example, Now in Detail

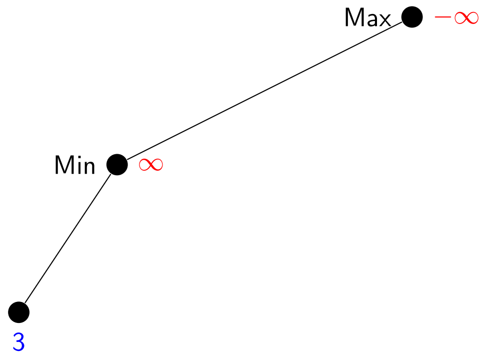
- So which action for Max is returned?

Minimax: Example, Now in Detail



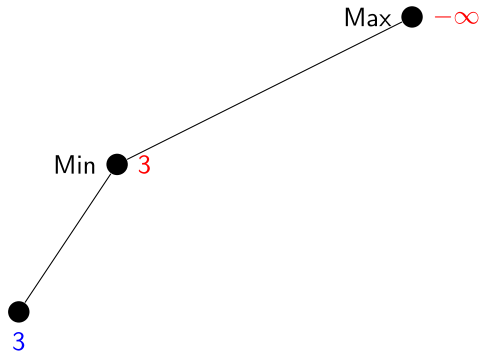
- So which action for Max is returned?

Minimax: Example, Now in Detail



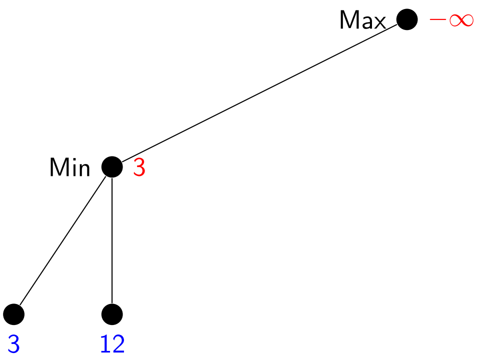
- So which action for Max is returned?

Minimax: Example, Now in Detail



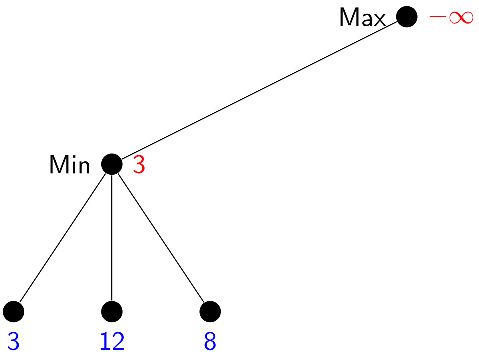
- So which action for Max is returned?

Minimax: Example, Now in Detail



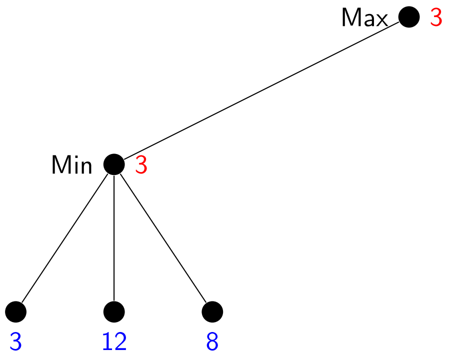
- So which action for Max is returned?

Minimax: Example, Now in Detail



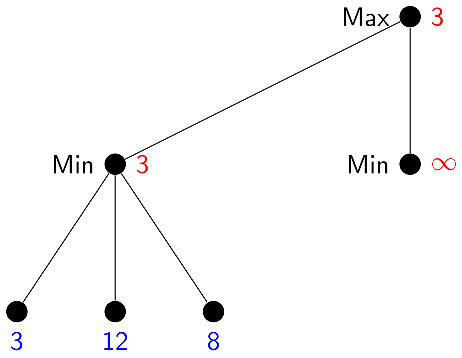
- So which action for Max is returned?

Minimax: Example, Now in Detail



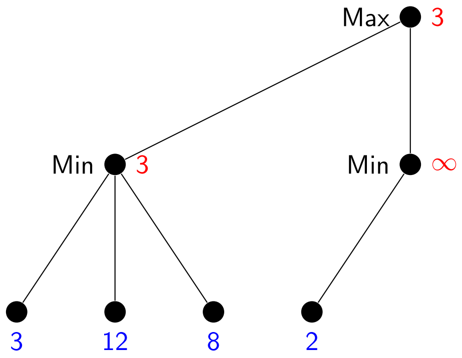
- So which action for Max is returned?

Minimax: Example, Now in Detail



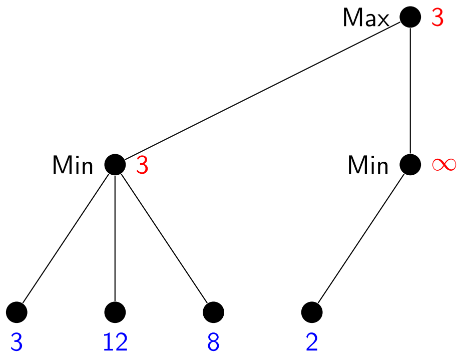
- So which action for Max is returned?

Minimax: Example, Now in Detail



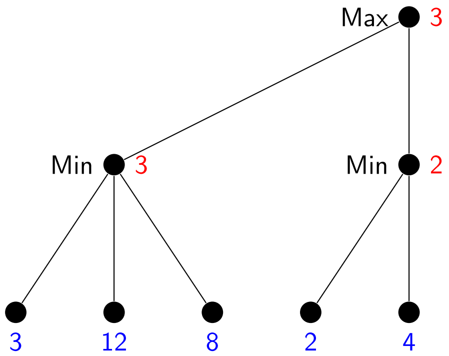
- So which action for Max is returned?

Minimax: Example, Now in Detail



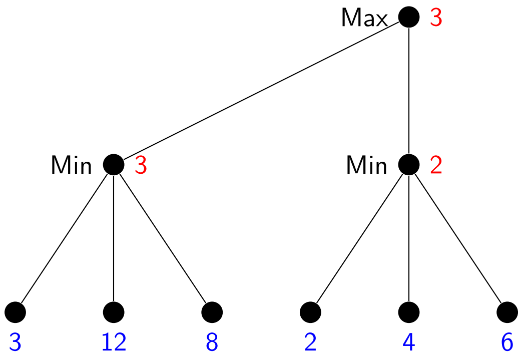
- So which action for Max is returned?

Minimax: Example, Now in Detail



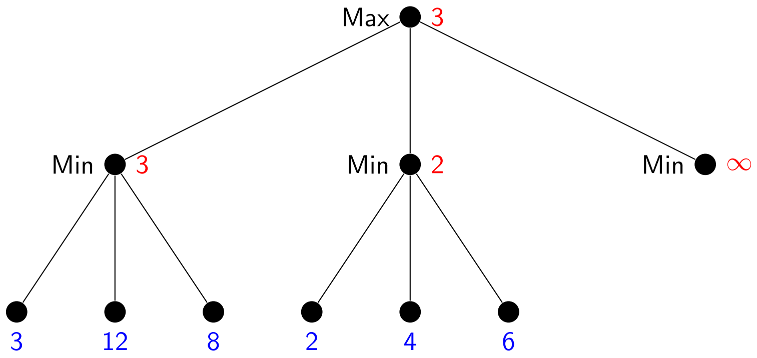
- So which action for Max is returned?

Minimax: Example, Now in Detail



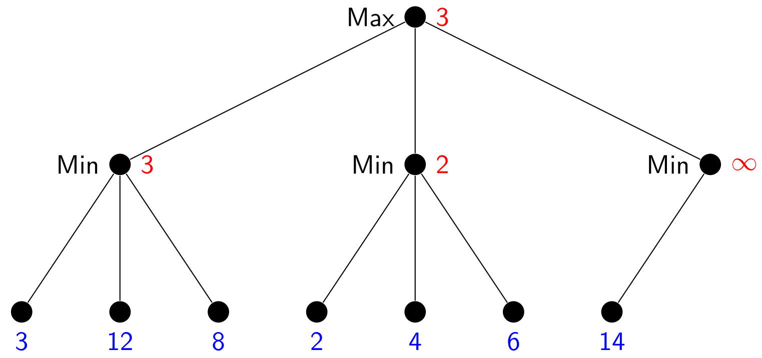
- So which action for Max is returned?

Minimax: Example, Now in Detail



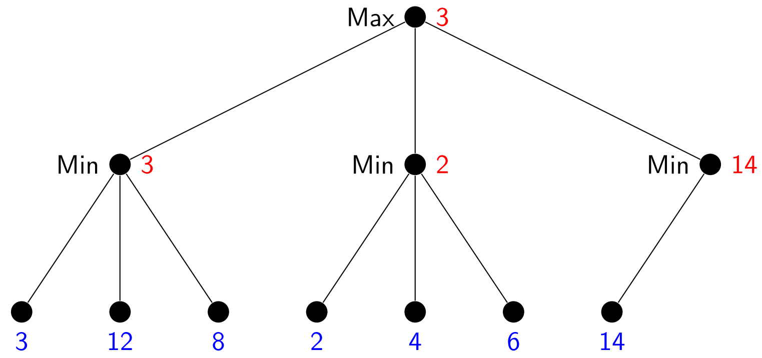
- So which action for Max is returned?

Minimax: Example, Now in Detail



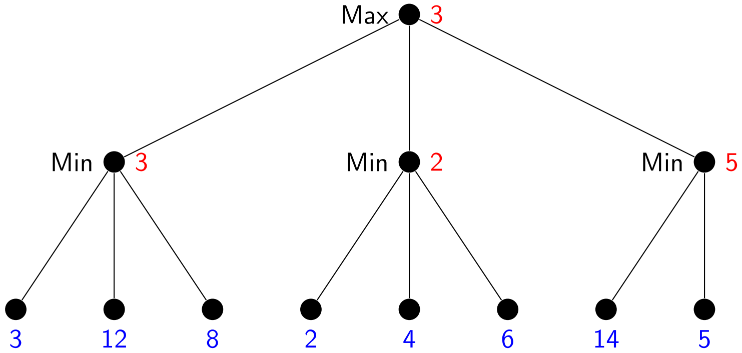
- So which action for Max is returned?

Minimax: Example, Now in Detail



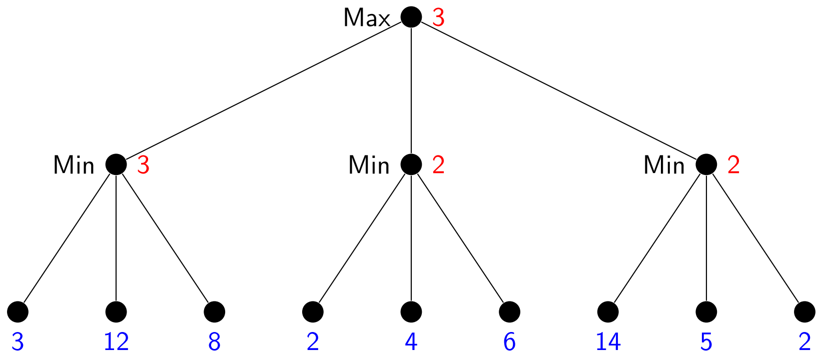
- So which action for Max is returned?

Minimax: Example, Now in Detail



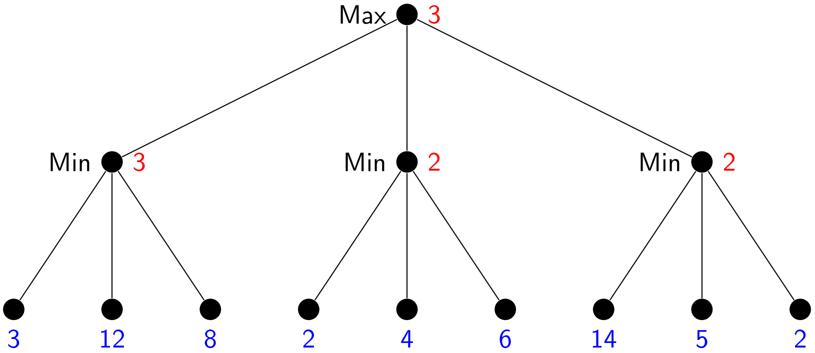
- So which action for Max is returned?

Minimax: Example, Now in Detail



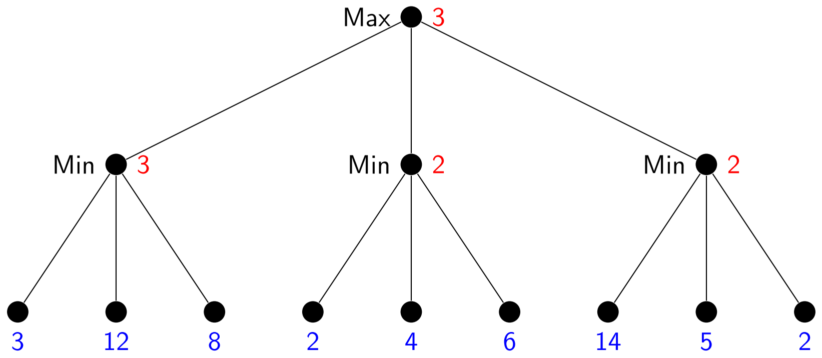
- So which action for Max is returned?

Minimax: Example, Now in Detail



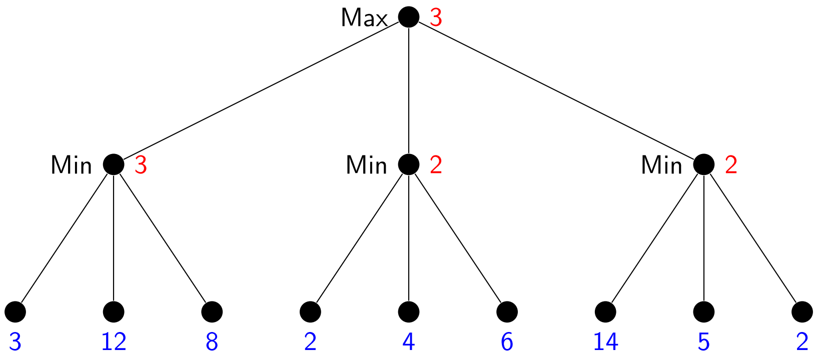
- So which action for Max is returned?

Minimax: Example, Now in Detail



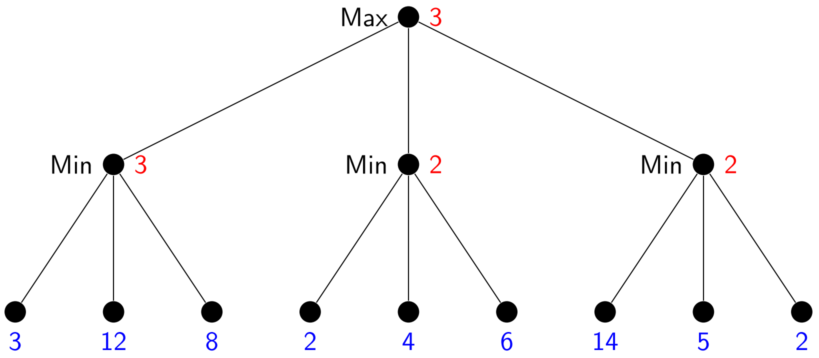
- So which action for Max is returned?

Minimax: Example, Now in Detail



- So which action for Max is returned?
- Leftmost branch.

Minimax: Example, Now in Detail



- So which action for Max is returned?
- Leftmost branch.
- **Note:** The maximal possible pay-off is higher for the rightmost branch, but assuming perfect play of Min, it's better to go left. (Going right would be "relying on your opponent to do something stupid".)

Minimax, Pro and Contra

- **Minimax advantages:**
  - Minimax is the simplest possible (reasonable) search algorithm for games.
    (If any of you sat down, prior to this lecture, to implement a Tic-Tac-Toe player, chances are you either looked this up on Wikipedia, or invented it in the process.)
  - Returns an optimal action, assuming perfect opponent play.
    - No matter how the opponent plays, the utility of the terminal state reached will be at least the value computed for the root.
    - If the opponent plays perfectly, exactly that value will be reached.
  - There's no need to re-run minimax for every game state: Run it once, offline before the game starts. During the actual game, just follow the branches taken in the tree. Whenever it's your turn, choose an action maximizing the value of the successor states.

Minimax, Pro and Contra

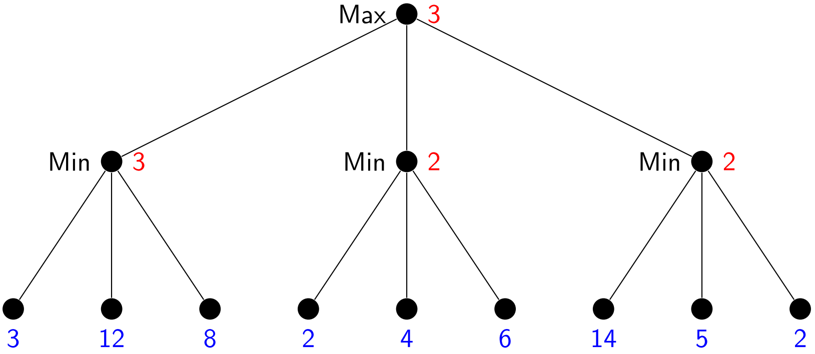
- **Minimax advantages:**
  - Minimax is the simplest possible (reasonable) search algorithm for games. (If any of you sat down, prior to this lecture, to implement a Tic-Tac-Toe player, chances are you either looked this up on Wikipedia, or invented it in the process.)
  - Returns an optimal action, assuming perfect opponent play.
    - No matter how the opponent plays, the utility of the terminal state reached will be at least the value computed for the root.
    - If the opponent plays perfectly, exactly that value will be reached.
  - There's no need to re-run minimax for every game state: Run it once, offline before the game starts. During the actual game, just follow the branches taken in the tree. Whenever it's your turn, choose an action maximizing the value of the successor states.
- **Minimax disadvantages:** It's completely infeasible in practice.
  - When the search tree is too large, we need to limit the search depth and apply an evaluation function to the cut off states.

7.3 Evaluation Functions

Evaluation Functions for Minimax

- **Problem:** Search tree are too big to search through in minimax.
- **Solution:** We impose a search depth limit (also called horizon) d, and apply an evaluation function to the cut-off states, i.e. states s with $\mathrm{dp}(s) = d$.
- **Definition 3.1.** An evaluation function f maps game states to numbers:
  - $f(s)$ is an estimate of the actual value of s (as would be computed by unlimited-depth minimax for s).
  - If cut-off state is terminal: Just use $\hat{u}$ instead of f.
- Analogy to heuristic functions (cf. **??**): We want f to be both (a) accurate and (b) fast.
- Another analogy: (a) and (b) are in contradiction $\rightsquigarrow$ need to trade-off accuracy against overhead.
  - In typical game playing algorithms today, f is inaccurate but very fast. (usually no good methods known for computing accurate f)

Example Revisited: Minimax With Depth Limit $d = 2$



- **Blue numbers:** evaluation function f, applied to the cut-off states at $d = 2$.
- **Red numbers:** utilities of inner node, as computed by minimax using f.

Example Chess

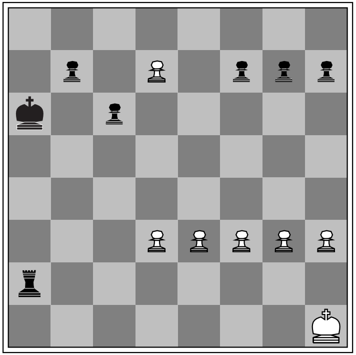
- Evaluation function in chess:
  - **Material**: Pawn 1, Knight 3, Bishop 3, Rook 5, Queen 9.
  - 3 points advantage $\rightsquigarrow$ safe win.
  - **Mobility**: How many fields do you control?
  - King safety, Pawn structure, . . .
- Note how simple this is! (probably is not how Kasparov evaluates his positions)

Linear Evaluation Functions

- **Problem:** How to come up with evaluation functions?
- **Definition 3.2.** A common approach is to use a weighted linear function for f, i.e. given a sequence of features $f_i \colon S \to \mathbb{R}$ and a corresponding sequence of weights $w_i \in \mathbb{R}$, f is of the form $f(s) := w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \cdots + w_n \cdot f_n(s)$
- **Problem:** How to obtain these weighted linear functions?
  - Weights w_i can be learned automatically. (learning agent)
  - The features f_i, however, have to be designed by human experts.
- **Note:** Very fast, very simplistic.
- **Observation:** Can be computed incrementally: In transition $s \xrightarrow{a} s'$, adapt $f(s)$ to $f(s')$ by considering only those features whose values have changed.

The Horizon Problem

- **Problem:** Critical aspects of the game can be cut off by the horizon. We call this the horizon problem.
- **Example 3.3.**



Black to move

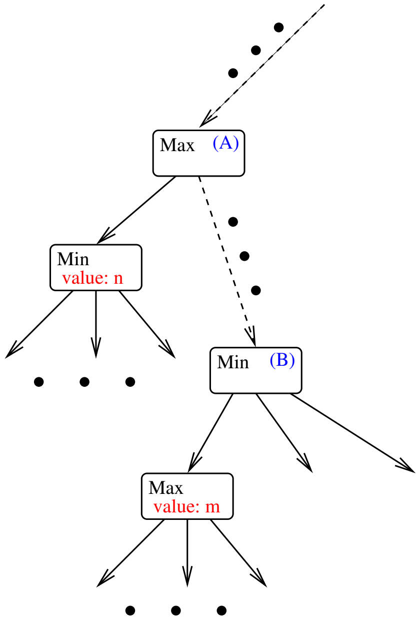
- Who's gonna win here?
  - White wins (pawn cannot be prevented from becoming a queen.)
  - Black has a +4 advantage in material, so if we cut-off here then our evaluation function will say "100%, black wins".
  - The loss for black is "beyond our horizon" unless we search extremely deeply: black can hold off the end by repeatedly giving check to white's king.

So, How Deeply to Search?

- **Goal:** In given time, search as deeply as possible.
- **Problem:** Very difficult to predict search running time. (need an anytime algorithm)
- **Solution:** Iterative deepening search.
  - Search with depth limit $d = 1, 2, 3, \ldots$
  - When time is up: return result of deepest completed search.
- **Definition 3.4 (Better Solution).** The quiescent search algorithm uses a dynamically adapted search depth d: It searches more deeply in unquiet positions, where value of evaluation function changes a lot in neighboring states.
- **Example 3.5.** In quiescent search for chess:
  - piece exchange situations (“you take mine, I take yours”) are very unquiet
  - $\rightsquigarrow$ Keep searching until the end of the piece exchange is reached.

7.4 Alpha-Beta Search

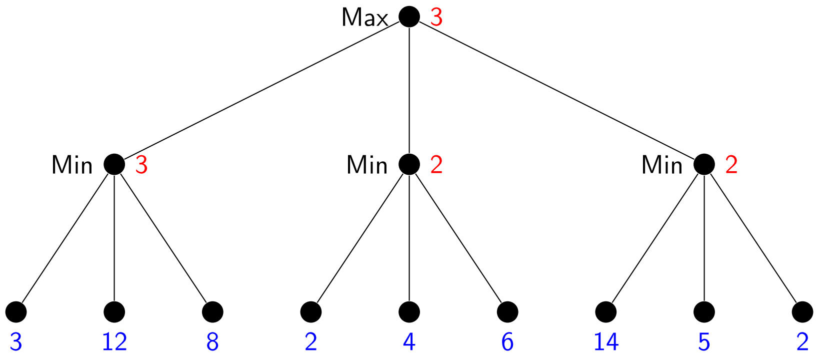
When We Already Know We Can Do Better Than This



- Say $n > m$.
- By choosing to go to the left in search node (A), Max already can get utility of at least n in this part of the game.
- So, if "later on" (further down in the same subtree), in search node (B) we already know that Min can force Max to get value $m < n$.
- Then Max will play differently in (A) so we will never actually get to (B).

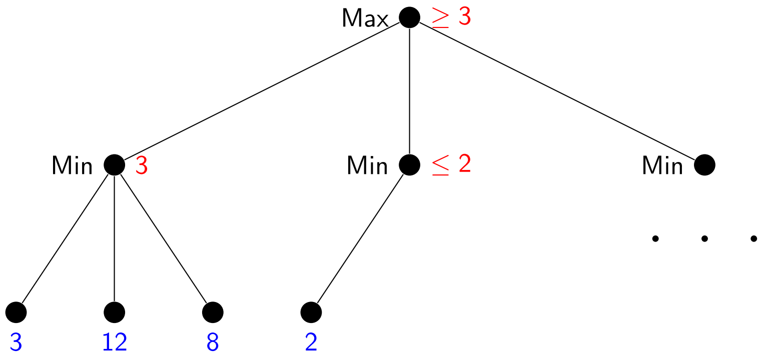
Alpha Pruning: Basic Idea

- **Question:** Can we save some work here?



Alpha Pruning: Basic Idea (Continued)

- **Answer:** Yes! We already know at this point that the middle action won't be taken by Max.



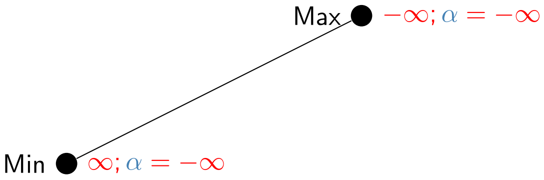
- **Idea:** We can use this to prune the search tree $\rightsquigarrow$ better algorithm

Alpha Pruning

- **Definition 4.1.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.

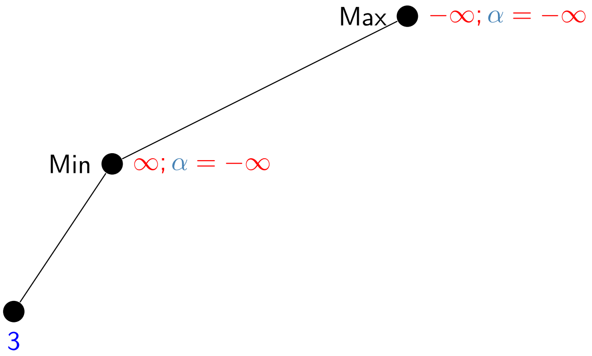
Alpha Pruning

- **Definition 4.3.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.4 (Computing alpha values).**



Alpha Pruning

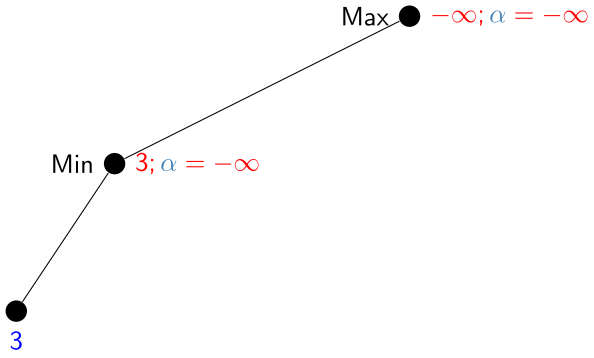
- **Definition 4.5.** For each node *n* in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to *n*.
- **Example 4.6 (Computing alpha values).**



- **How to use α?:** In a Min-node *n*, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering *n*. (pruning out its remaining successors)

Alpha Pruning

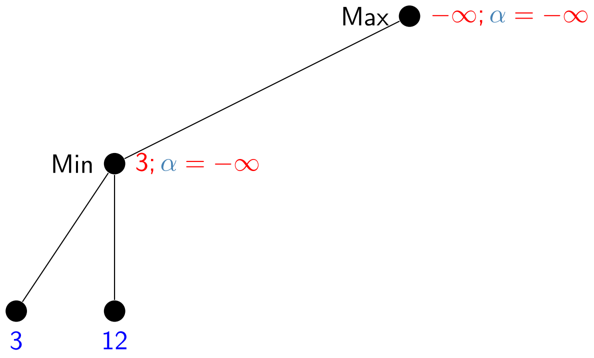
- **Definition 4.7.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.8 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

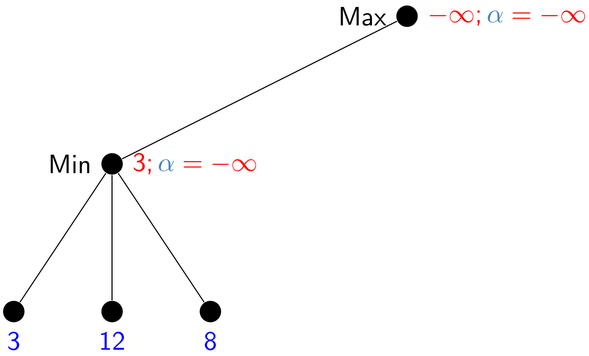
- **Definition 4.9.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.10 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

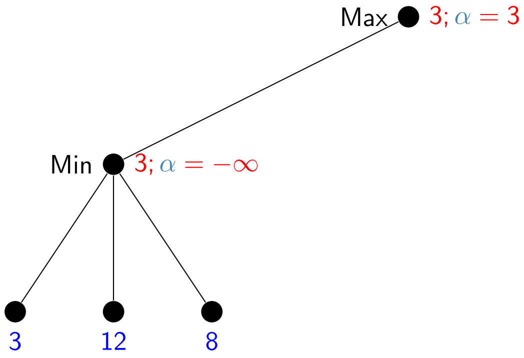
- **Definition 4.11.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.12 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

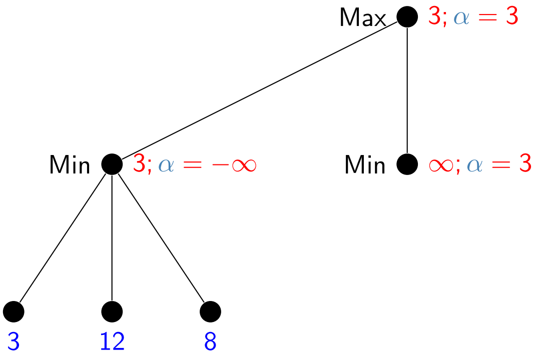
- **Definition 4.13.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.14 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

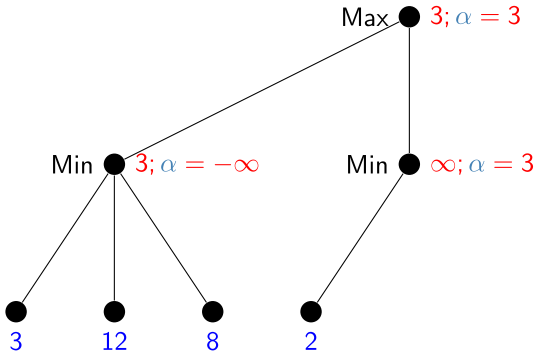
- **Definition 4.15.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.16 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

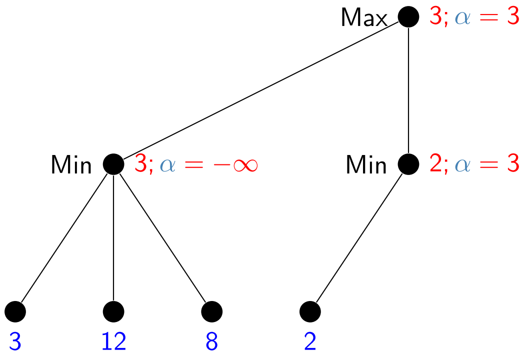
- **Definition 4.17.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.18 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

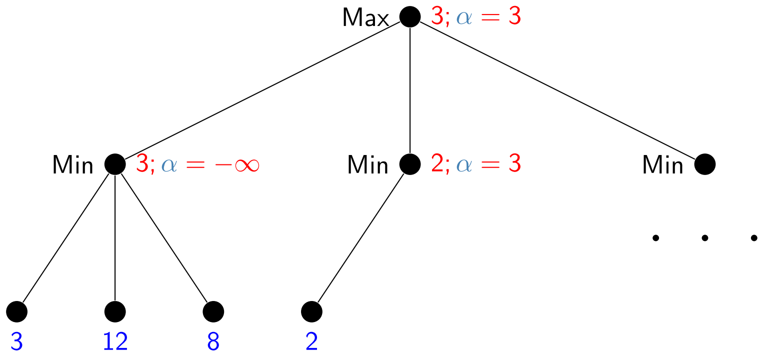
- **Definition 4.19.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.20 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha Pruning

- **Definition 4.21.** For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.
- **Example 4.22 (Computing alpha values).**



- **How to use α?:** In a Min-node n, if $\hat{u}(n') \leq \alpha(n)$ for one of the successors, then stop considering n. (pruning out its remaining successors)

Alpha-Beta Pruning

- **Recall:**
  - **What is α:** For each search node *n*, the highest Max-node utility that search has encountered on its path from the root to *n*.
  - **How to use α:** In a Min-node *n*, if one of the successors already has utility $\leq \alpha(n)$, then stop considering *n*. (Pruning out its remaining successors)
- **Idea:** We can use a dual method for Min!
- **Definition 4.23.** For each node *n* in a minimax search tree, the beta value $\beta(n)$ is the highest Min-node utility that search has encountered on its path from the root to *n*.
- **How to use β:** In a Max-node *n*, if one of the successors already has utility $\geq \beta(n)$, then stop considering *n*. (pruning out its remaining successors)
- . . . and of course we can use α and β together! $\leadsto$ alphabeta-pruning

Alpha-Beta Search: Pseudocode

- **Definition 4.24.** The alphabeta search algorithm is given by the following pseudocode

```
function Alpha−Beta−Search ($s$) returns an action
  $v$ := Max−Value($s$, $-\infty$, $+\infty$)
  return an action yielding value $v$ in the previous function call

function Max−Value($s$, $\alpha$, $\beta$) returns a utility value
  if Terminal−Test($s$) then return $u(s)$
  $v := -\infty$
  for each $a \in$ Actions($s$) do
    $v$ := max($v$,Min−Value(ChildState($s$,$a$), $\alpha$, $\beta$))
     $\alpha$ := max($\alpha$, $v$)
     if $v \geq \beta$ then return $v$ /* Here: $v \geq \beta \Leftrightarrow \alpha \geq \beta$ */
  return $v$

function Min−Value($s$, $\alpha$, $\beta$) returns a utility value
  if Terminal−Test($s$) then return $u(s)$
  $v := +\infty$
  for each $a \in$ Actions($s$) do
     $v$ := min($v$,Max−Value(ChildState($s$,$a$), $\alpha$, $\beta$))
     $\beta$ := min($\beta$, $v$)
     if $v \leq \alpha$ then return $v$ /* Here: $v \leq \alpha \Leftrightarrow \alpha \geq \beta$ */
  return $v$
```

$\widehat{=}$ Minimax (slide 212) + α/β book-keeping and pruning.

Alpha-Beta Search: Example

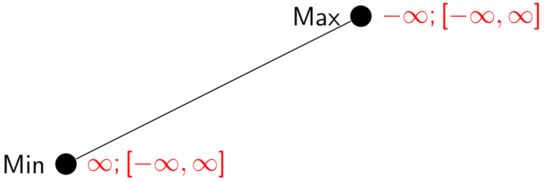
- **Notation:** $v; [\alpha, \beta]$

Max ● $-\infty; [-\infty, \infty]$

- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

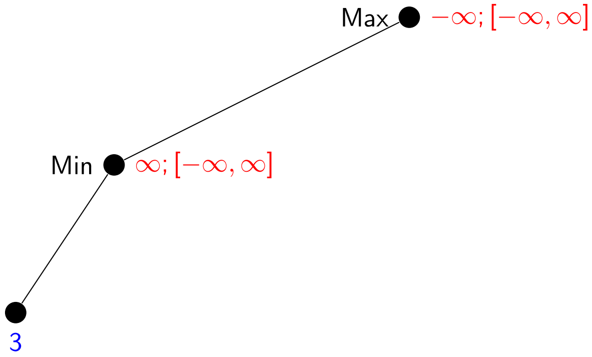
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

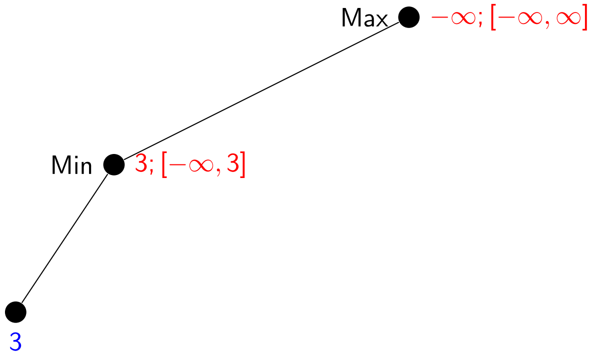
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

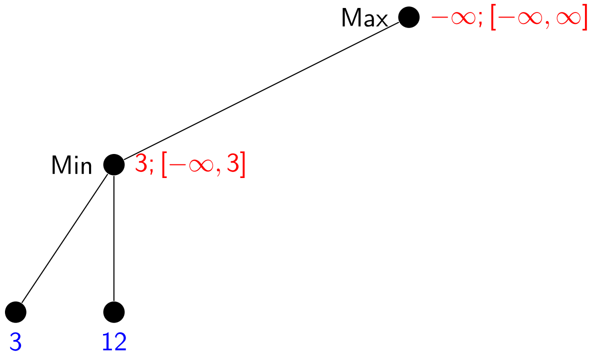
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

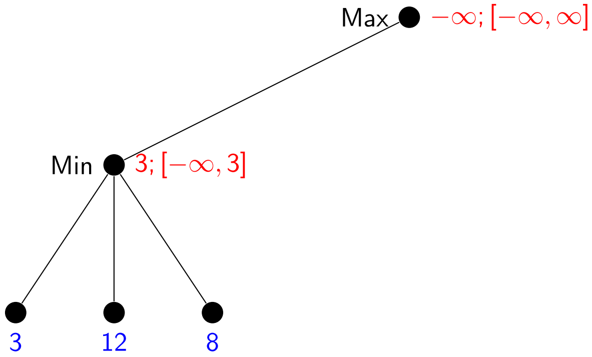
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

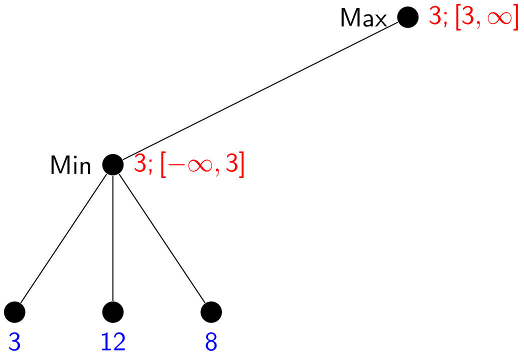
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

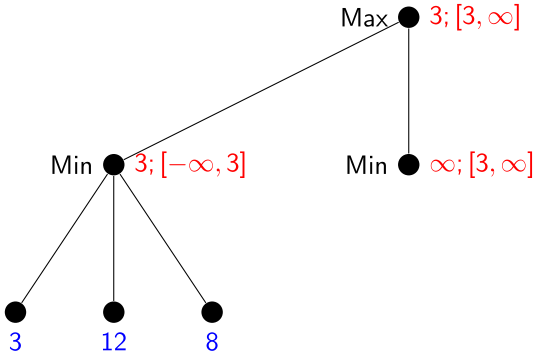
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

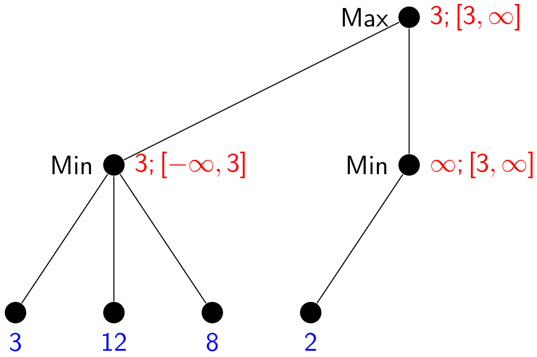
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

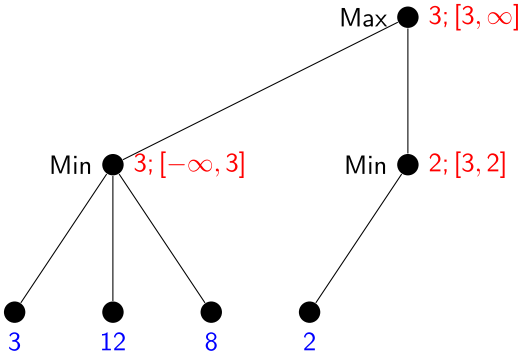
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

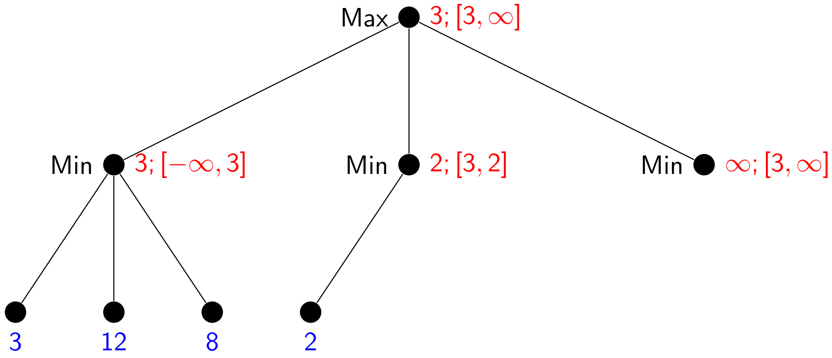
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

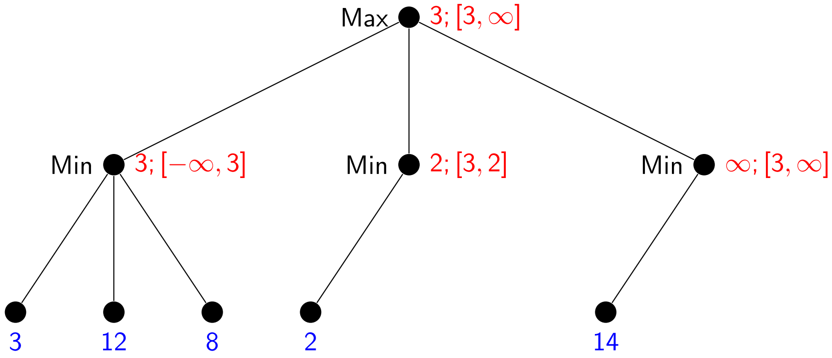
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

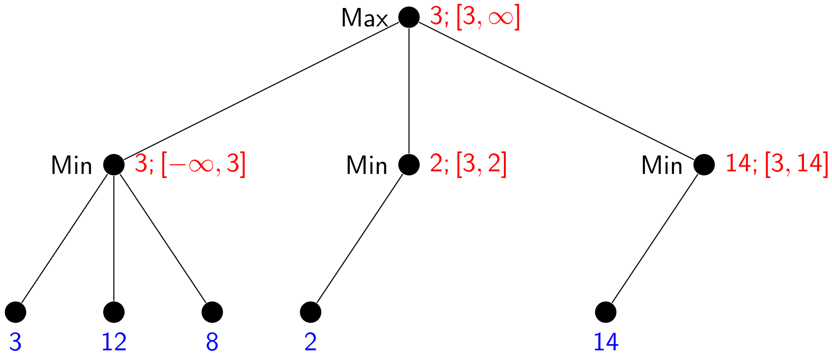
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

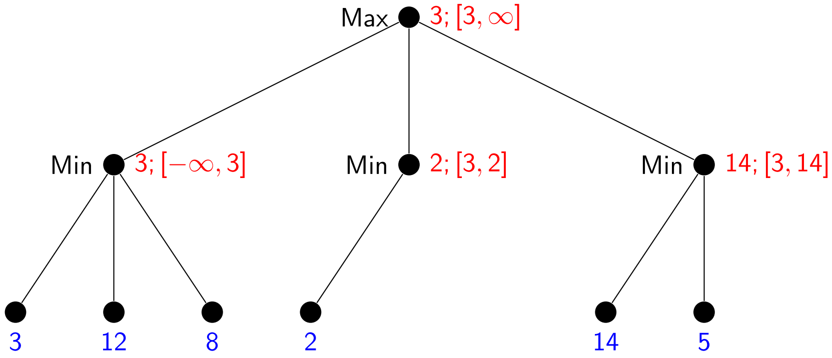
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

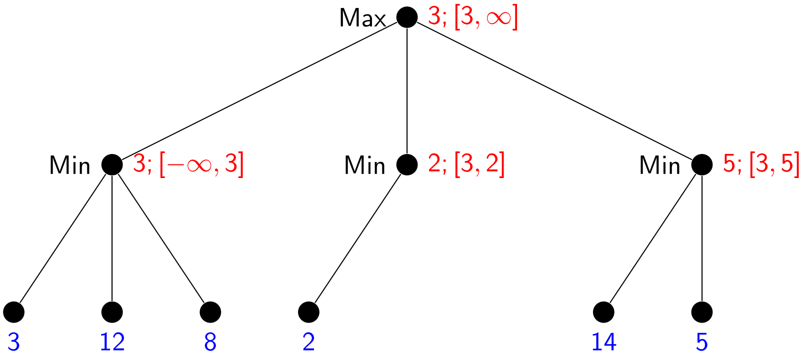
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

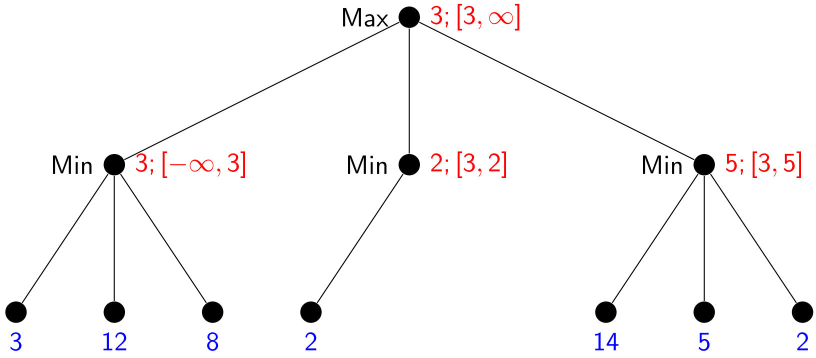
- **Notation:** v; $[\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

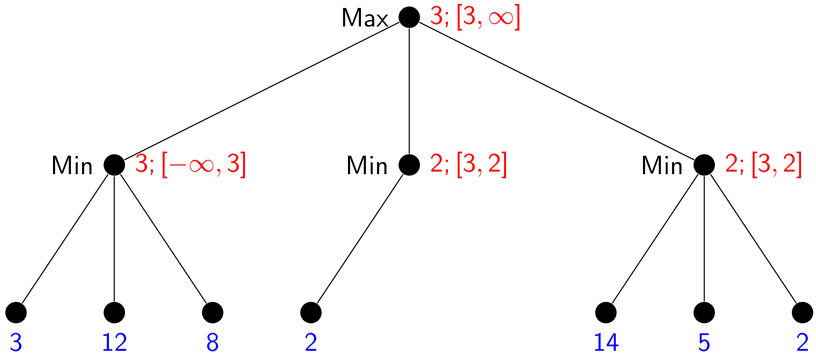
- **Notation:** v; $[\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

Alpha-Beta Search: Example

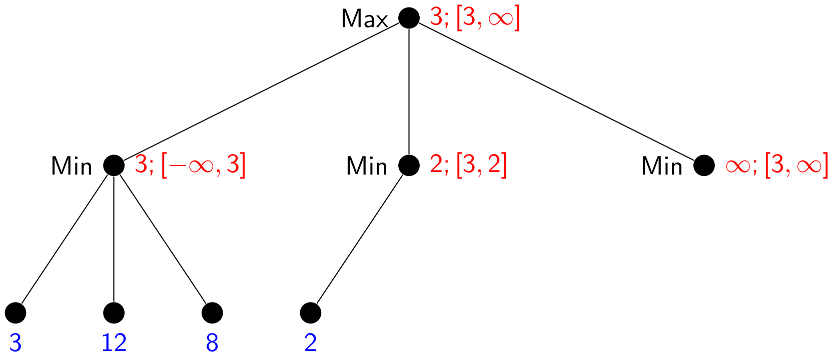
- **Notation:** $v; [\alpha, \beta]$



- **Note:** We could have saved work by choosing the opposite order for the successors of the rightmost Min-node.
  Choosing the best moves (for each of Max and Min) first yields more pruning!

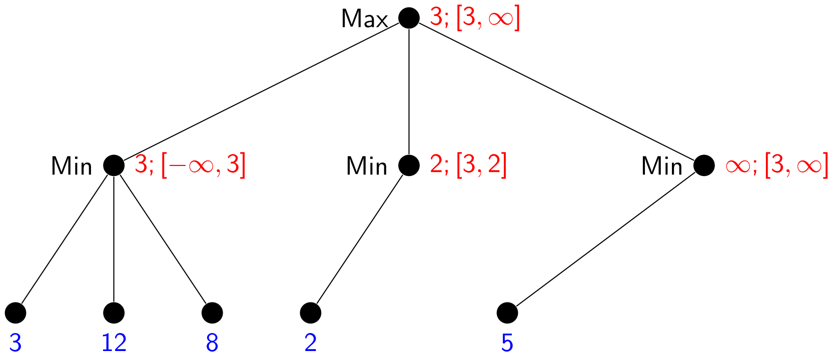
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



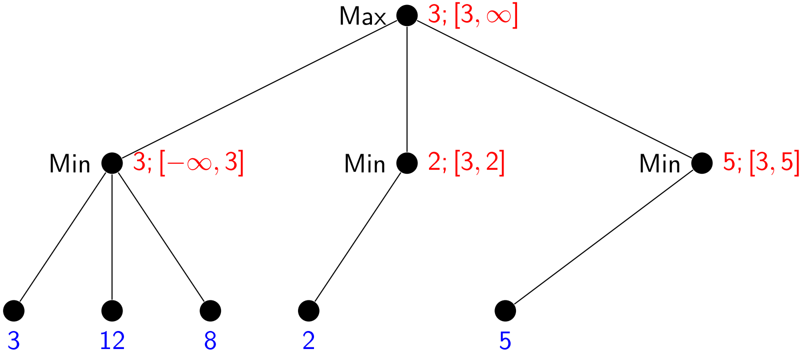
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



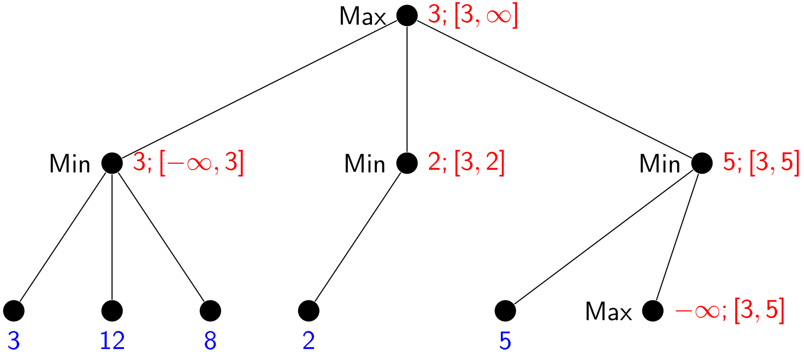
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



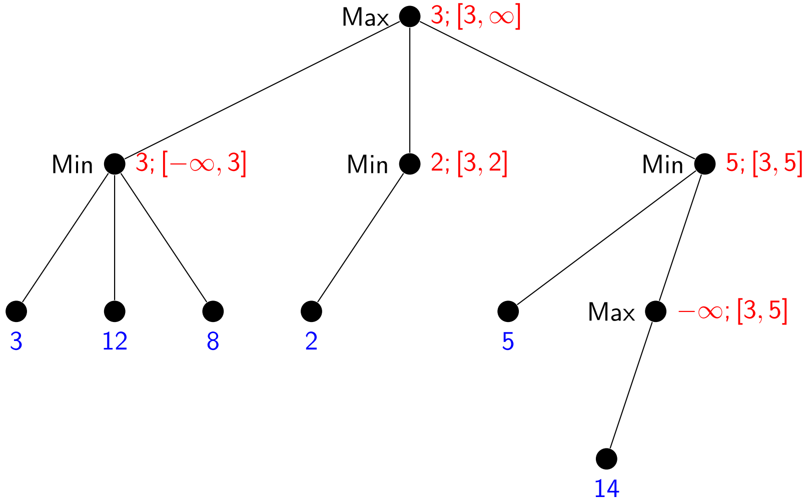
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



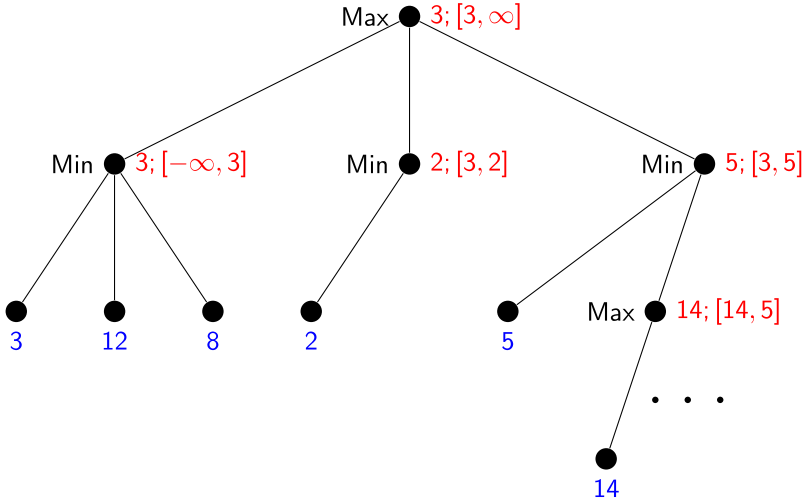
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



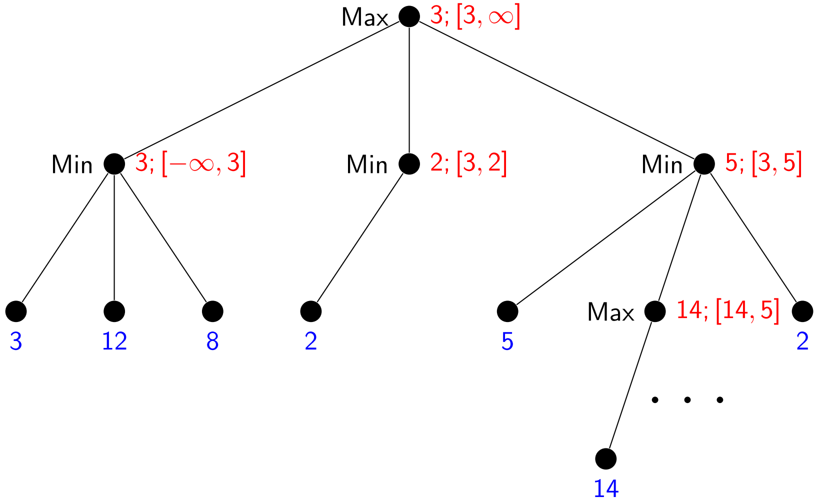
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



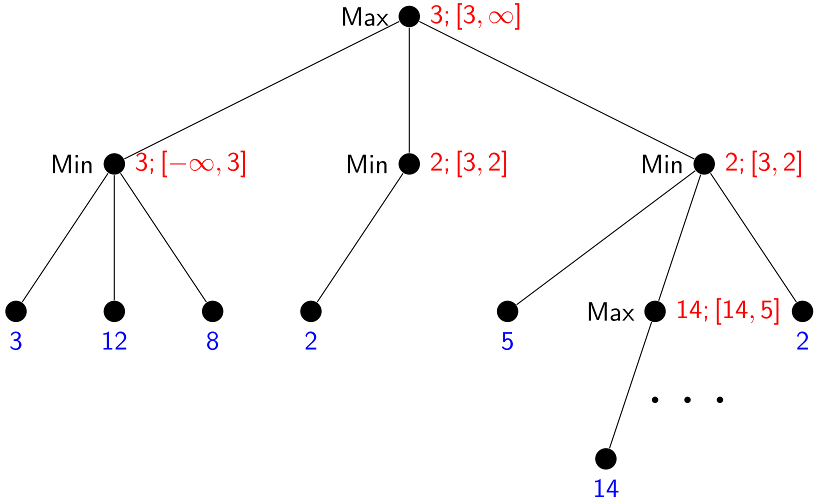
Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



Alpha-Beta Search: Modified Example

- **Showing off some actual β pruning:**



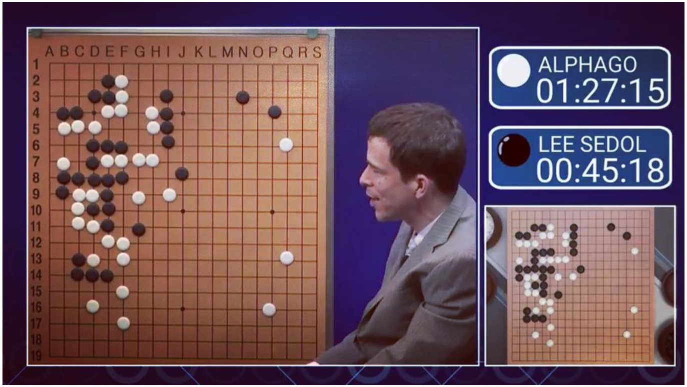
How Much Pruning Do We Get?

- Choosing the best moves first yields most pruning in alphabeta search.
  - The maximizing moves for Max, the minimizing moves for Min.
- **Observation:** Assuming game tree with branching factor b and depth limit d:
  - Minimax would have to search b^d nodes.
  - **Best case:** If we always choose the best moves first, then the search tree is reduced to $b^{\frac{d}{2}}$ nodes!
  - Practice: It is often possible to get very close to the best case by simple move-ordering methods.
- **Example 4.25 (Chess).**
  - Move ordering: Try captures first, then threats, then forward moves, then backward moves.
  - From 35^d to $35^{\frac{d}{2}}$. E.g., if we have the time to search a billion (10^9) nodes, then minimax looks ahead $d = 6$ moves, i.e., 3 rounds (white-black) of the game. Alpha-beta search looks ahead 6 rounds.

7.5 Monte-Carlo Tree Search (MCTS)

And now . . .

- AlphaGo = Monte Carlo tree search **(AI-1)** + neural networks **(AI-2)**



CC-BY-SA: Buster Benson© `https://www.flickr.com/photos/erikbenson/25717574115`

Monte-Carlo Tree Search: Basic Ideas

- **Observation:** We do not always have good evaluation functions.
- **Definition 5.1.** For Monte Carlo sampling we evaluate actions through sampling.
  - When deciding which action to take on game state s:

```
while time not up do
  select action $a$ applicable to $s$
  run a random sample from $a$ until terminal state $t$
return an $a$ for $s$ with maximal average $u(t)$
```

- **Definition 5.2.** For the Monte Carlo tree search algorithm (MCTS) we maintain a search tree T, the MCTS tree.

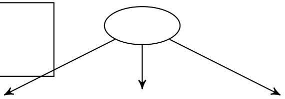
```
while time not up do
  apply actions within $T$ to select a leaf state $s'$
  select action $a'$ applicable to $s'$, run random sample from $a'$
  add $s'$ to $T$, update averages etc.
return an $a$ for $s$ with maximal average $u(t)$
When executing $a$, keep the part of $T$ below $a$.
```

- Compared to alphabeta search: no exhaustive enumeration.
  - **Pro**: running time & memory.
  - **Contra**: need good guidance how to select and sample.

Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.3 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

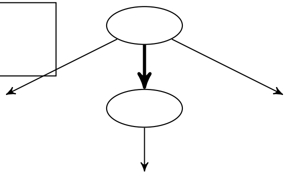
Expansions: 0, 0, 0
avg. reward: 0, 0, 0



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.4 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

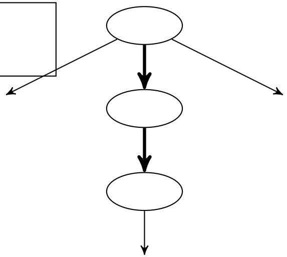
Expansions: 0, 0, 0
avg. reward: 0, 0, 0



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.5 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

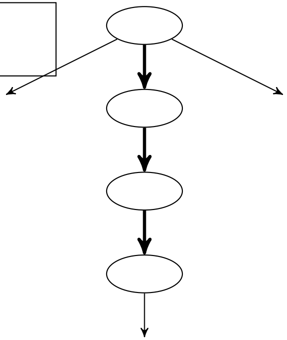
Expansions: 0, 0, 0
avg. reward: 0, 0, 0



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.6 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

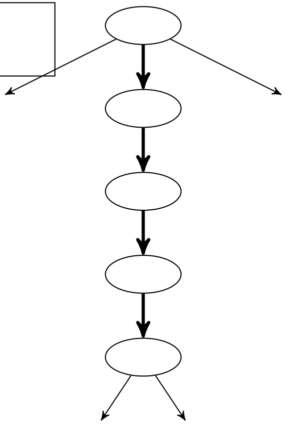
Expansions: 0, 0, 0
avg. reward: 0, 0, 0



Monte-Carlo Sampling: Illustration of Sampling

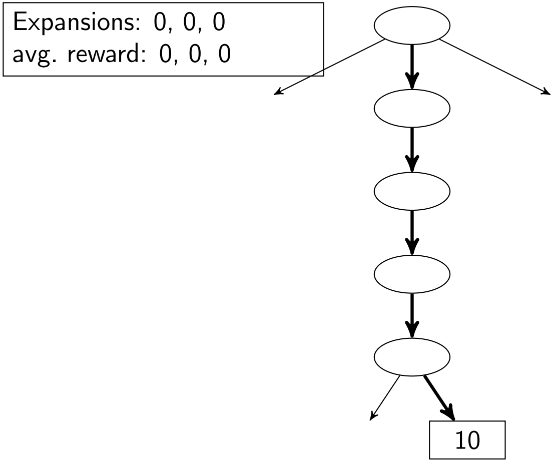
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.7 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 0, 0, 0
avg. reward: 0, 0, 0



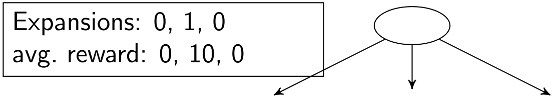
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.8 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



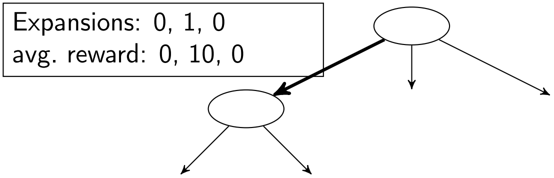
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.9 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



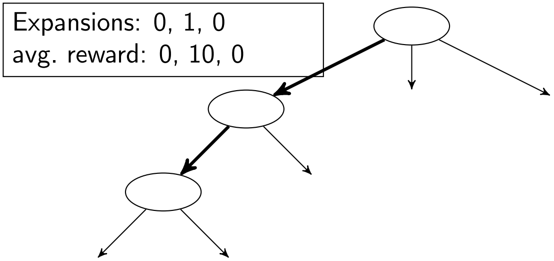
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.10 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

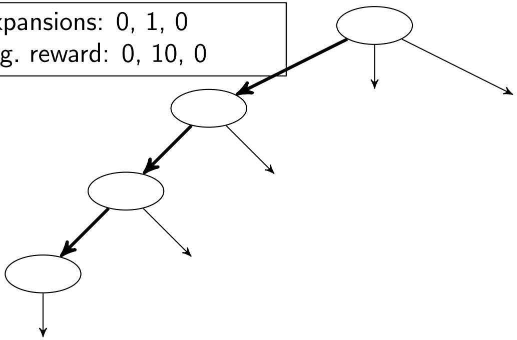
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.11 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

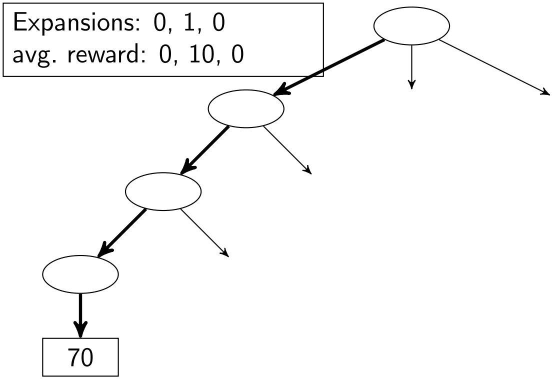
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.12 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 0, 1, 0
avg. reward: 0, 10, 0



Monte-Carlo Sampling: Illustration of Sampling

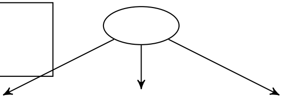
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.13 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

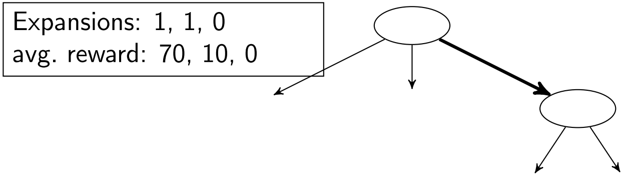
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.14 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 1, 1, 0
avg. reward: 70, 10, 0



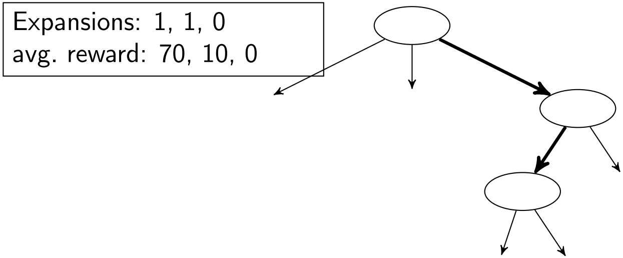
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.15 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



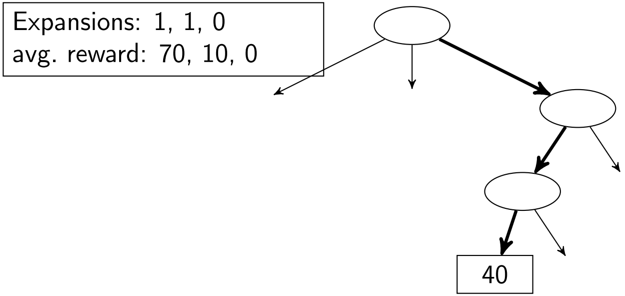
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.16 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



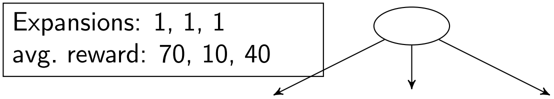
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.17 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



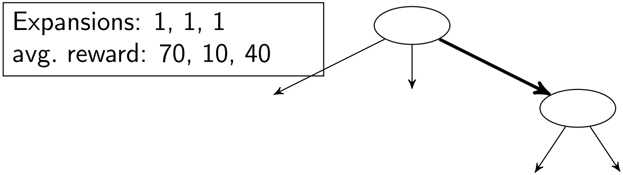
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.18 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



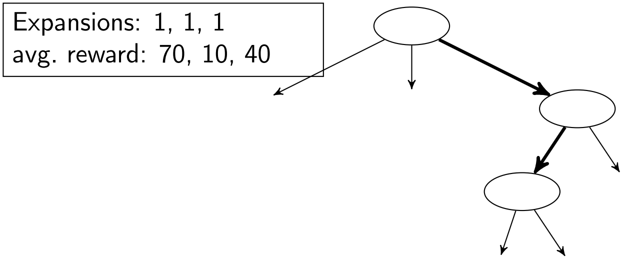
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.19 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



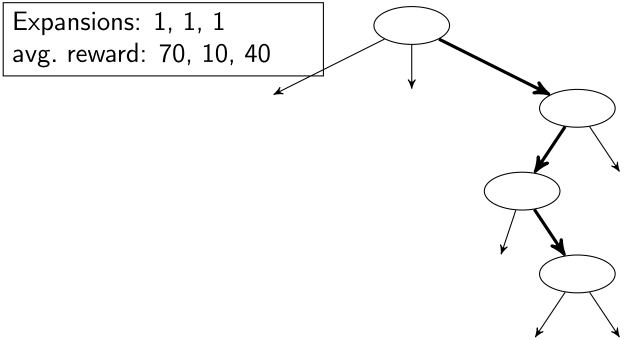
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.20 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

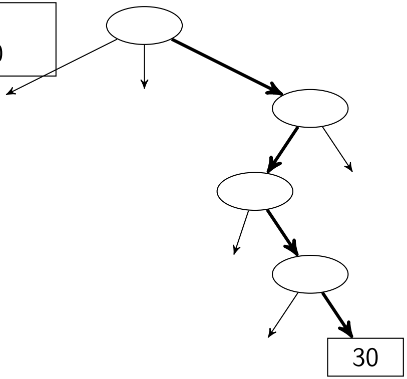
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.21 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)


Expansions: 1, 1, 1
avg. reward: 70, 10, 40

Monte-Carlo Sampling: Illustration of Sampling

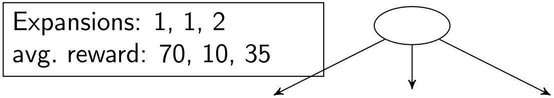
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.22 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 1, 1, 1
avg. reward: 70, 10, 40



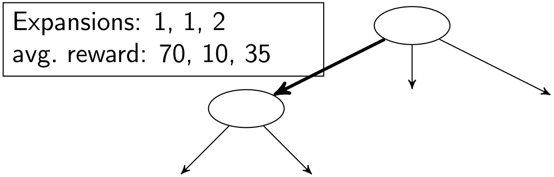
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.23 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



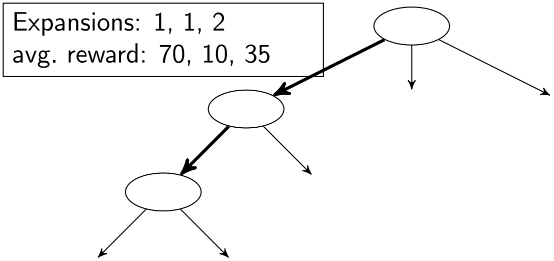
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.24 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

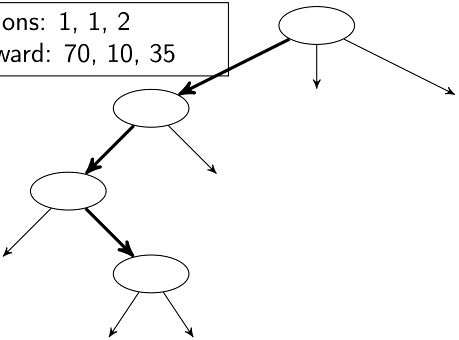
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.25 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

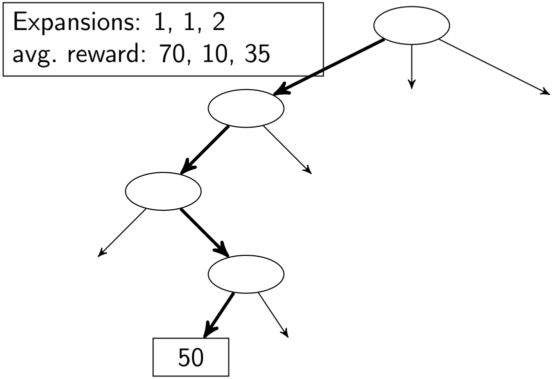
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.26 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 1, 1, 2
avg. reward: 70, 10, 35



Monte-Carlo Sampling: Illustration of Sampling

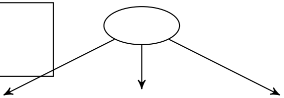
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.27 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.28 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

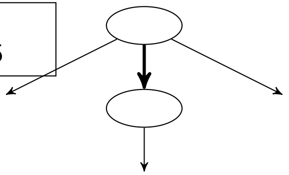
Expansions: 2, 1, 2
avg. reward: 60, 10, 35



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.29 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

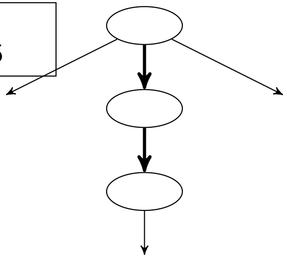
Expansions: 2, 1, 2
avg. reward: 60, 10, 35



Monte-Carlo Sampling: Illustration of Sampling

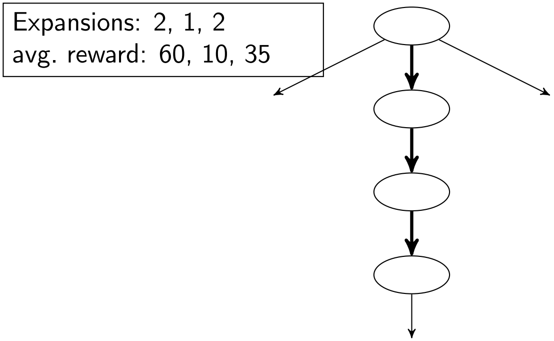
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.30 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 2, 1, 2
avg. reward: 60, 10, 35



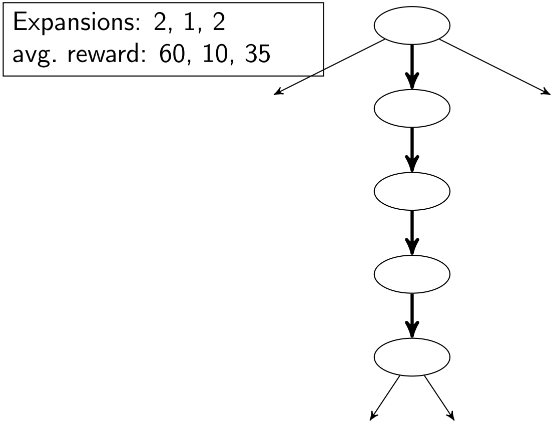
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.31 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

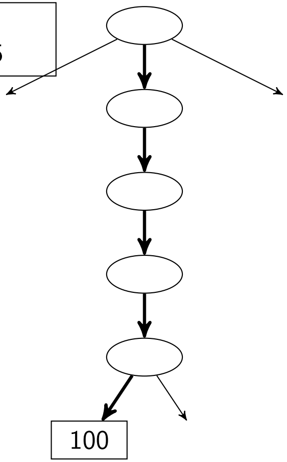
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.32 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.33 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

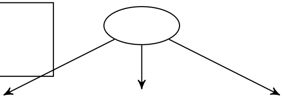
Expansions: 2, 1, 2
avg. reward: 60, 10, 35



Monte-Carlo Sampling: Illustration of Sampling

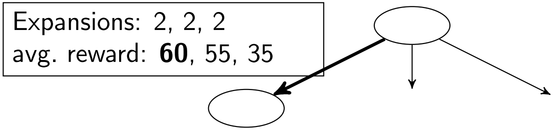
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.34 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)

Expansions: 2, 2, 2
avg. reward: 60, 55, 35



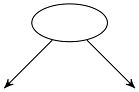
Monte-Carlo Sampling: Illustration of Sampling

- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.35 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Monte-Carlo Sampling: Illustration of Sampling

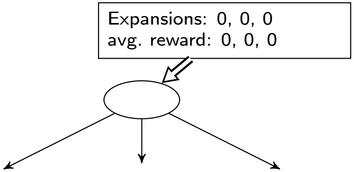
- **Idea:** Sample the search tree keeping track of the average utilities.
- **Example 5.36 (Single-player, for simplicity).** (with adversary, distinguish max/min nodes)



Expansions: 0, 0
avg. reward: 0, 0

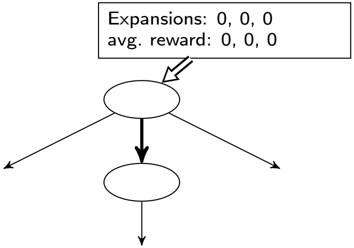
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.37 (Redoing the previous example).**



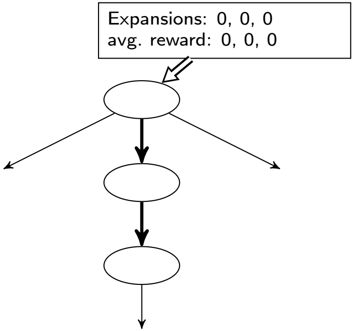
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.38 (Redoing the previous example).**



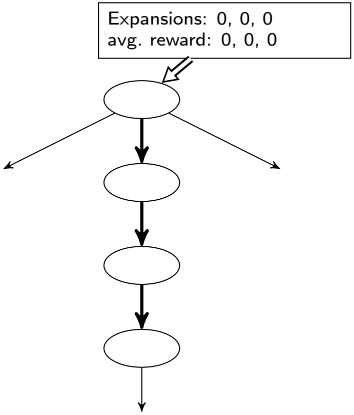
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.39 (Redoing the previous example).**



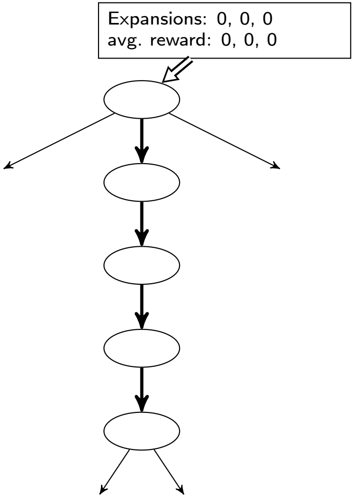
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.40 (Redoing the previous example).**



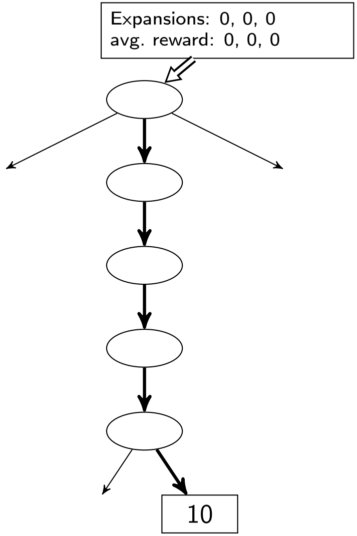
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.41 (Redoing the previous example).**



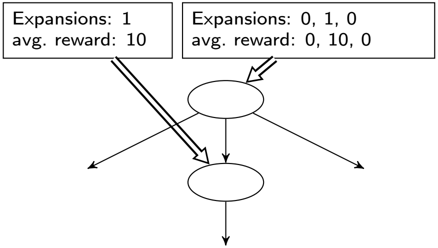
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.42 (Redoing the previous example).**



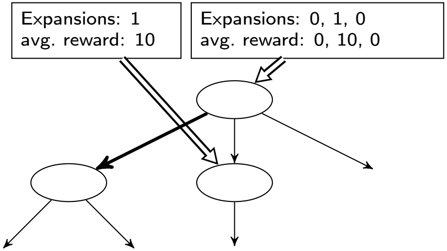
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.43 (Redoing the previous example).**



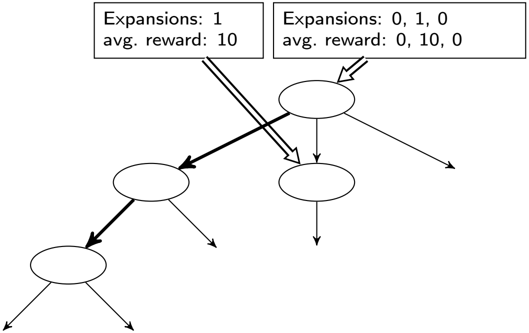
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.44 (Redoing the previous example).**



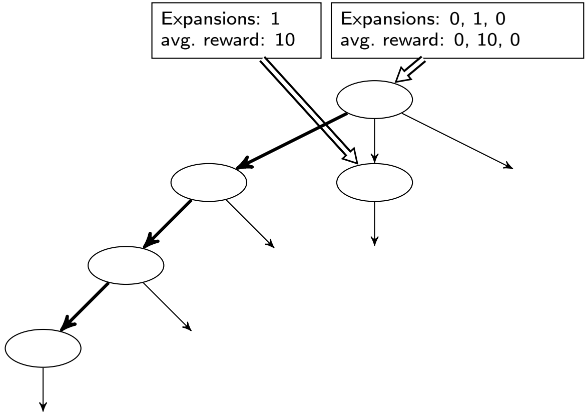
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.45 (Redoing the previous example).**



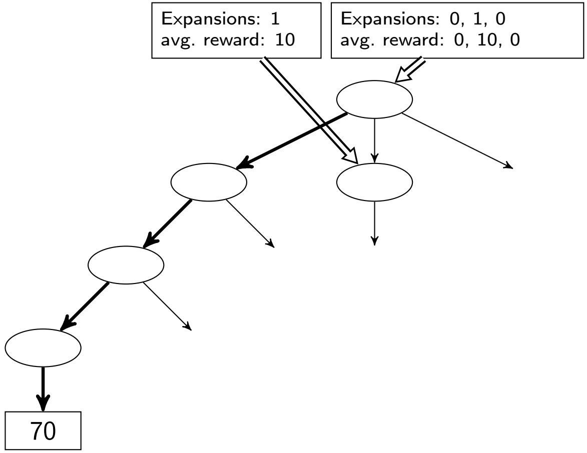
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.46 (Redoing the previous example).**



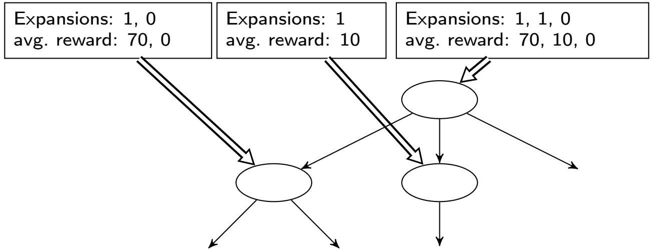
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.47 (Redoing the previous example).**



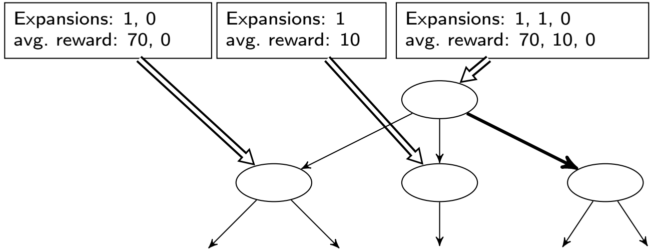
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.48 (Redoing the previous example).**



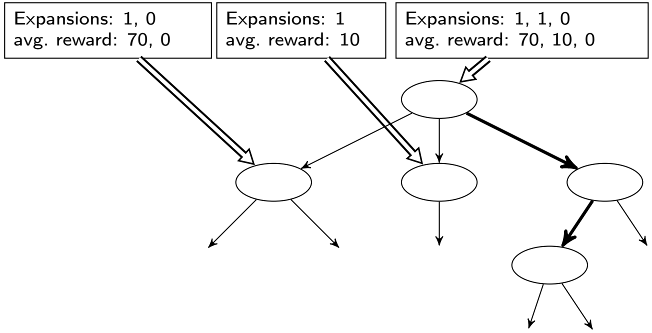
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.49 (Redoing the previous example).**



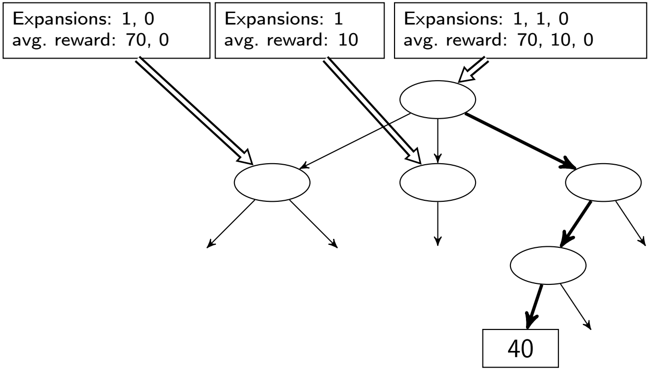
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.50 (Redoing the previous example).**



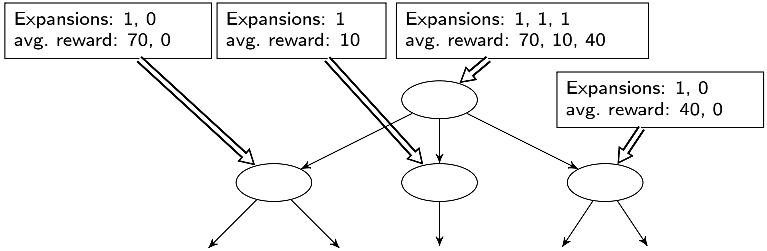
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.51 (Redoing the previous example).**



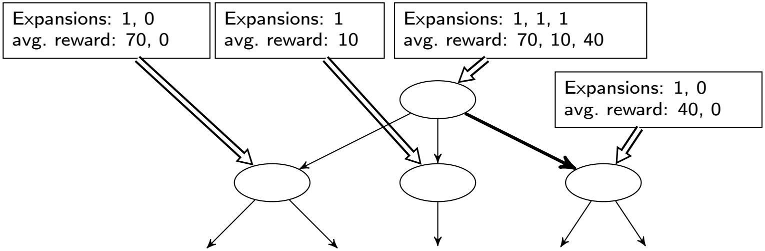
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.52 (Redoing the previous example).**



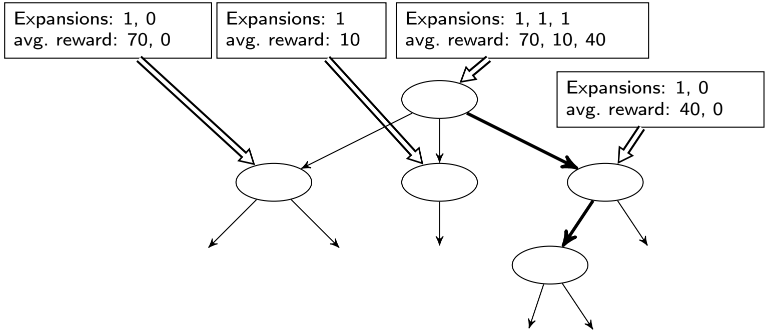
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.53 (Redoing the previous example).**



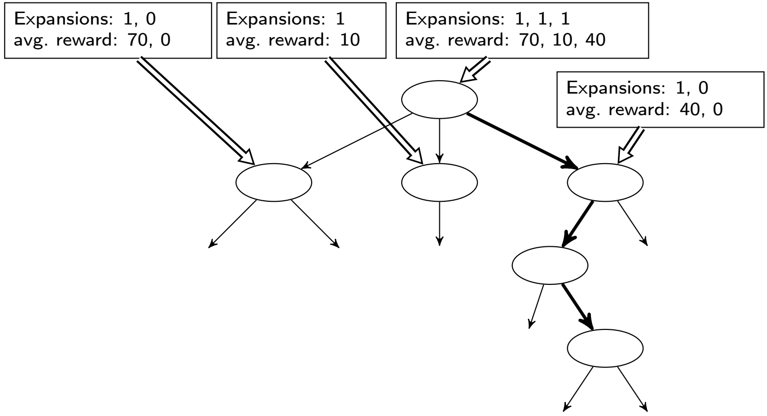
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.54 (Redoing the previous example).**



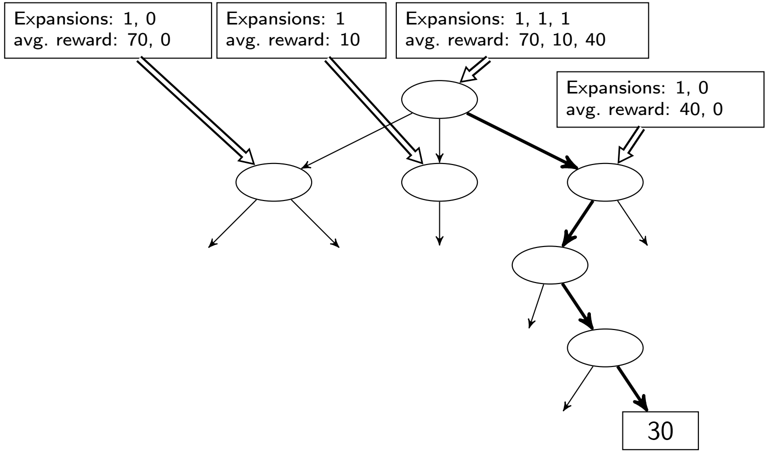
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.55 (Redoing the previous example).**



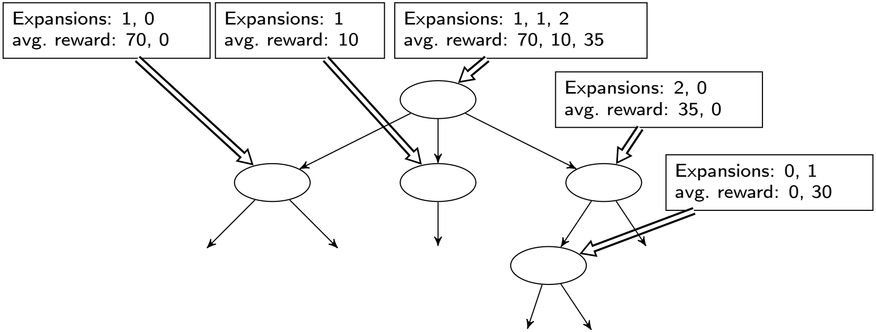
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.56 (Redoing the previous example).**



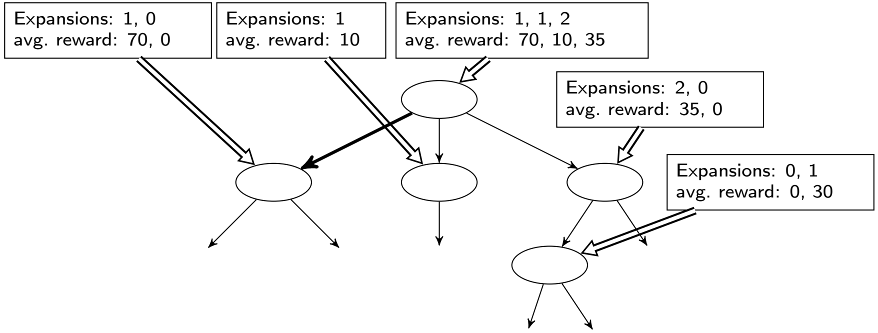
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.57 (Redoing the previous example).**



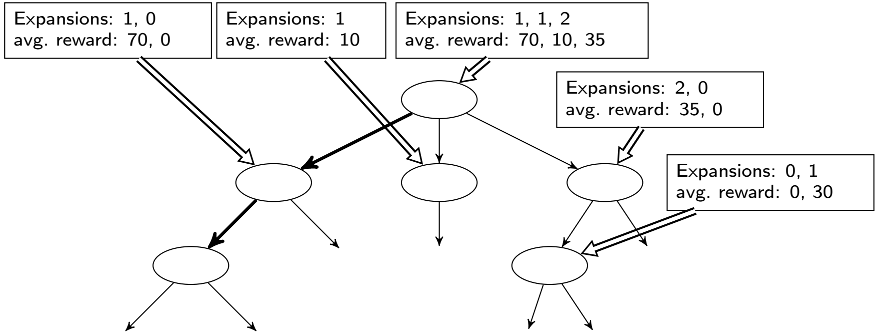
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.58 (Redoing the previous example).**



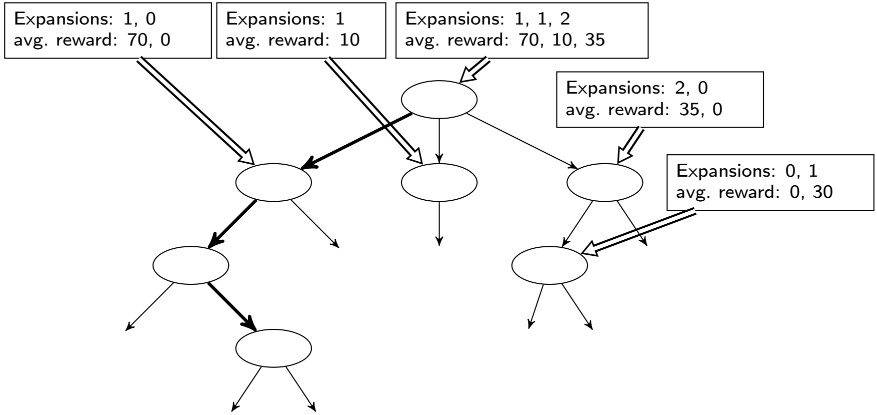
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.59 (Redoing the previous example).**



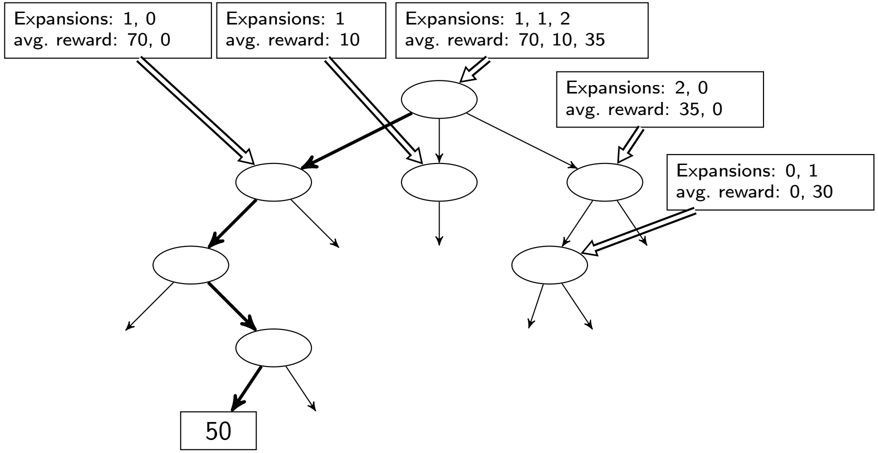
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.60 (Redoing the previous example).**



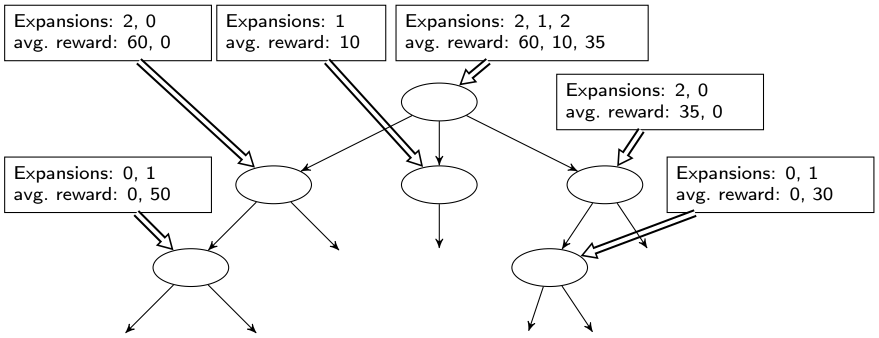
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.61 (Redoing the previous example).**



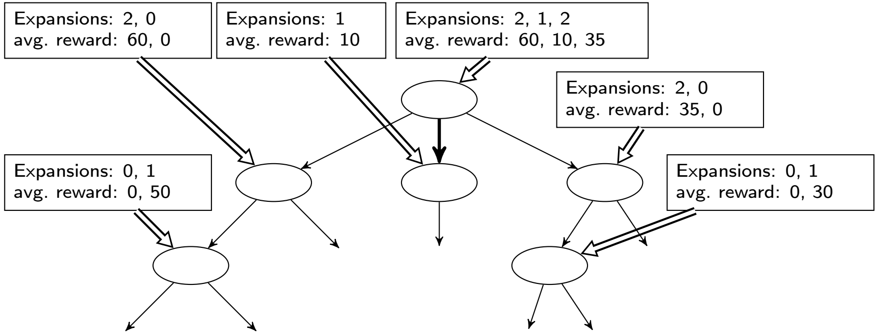
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.62 (Redoing the previous example).**



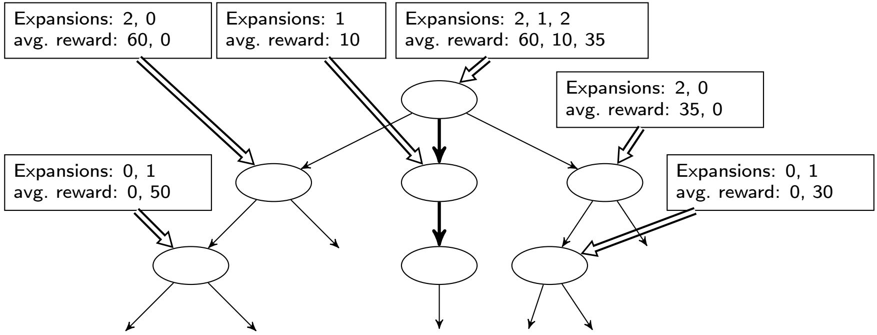
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.63 (Redoing the previous example).**



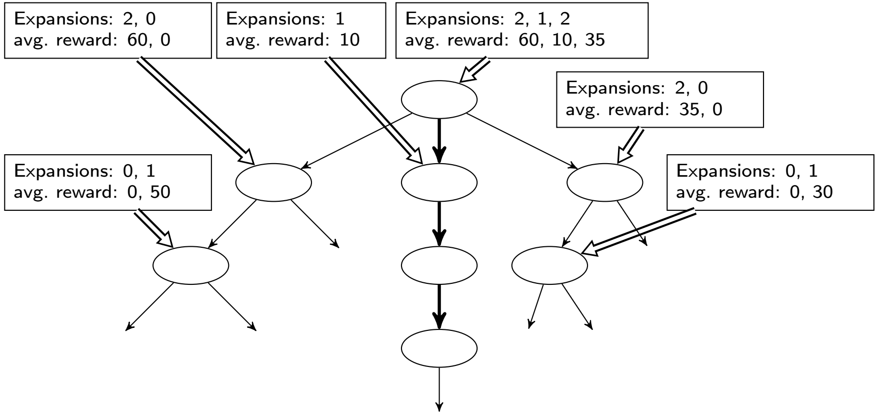
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.64 (Redoing the previous example).**



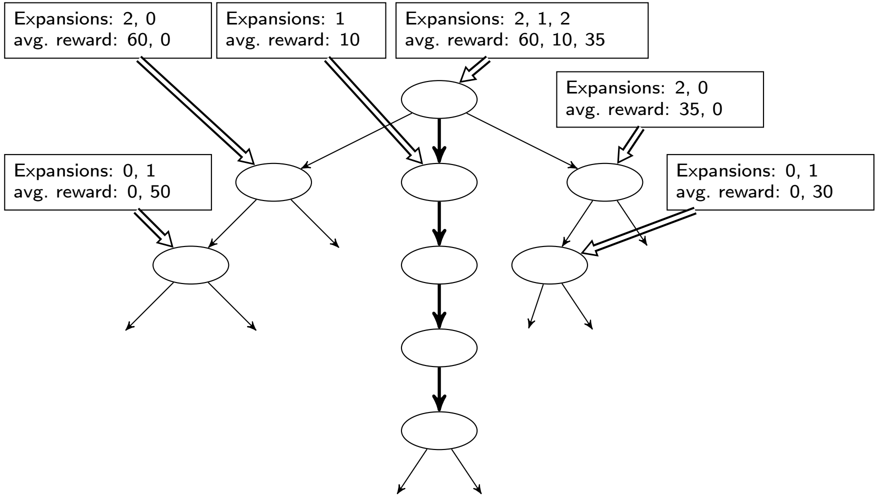
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.65 (Redoing the previous example).**



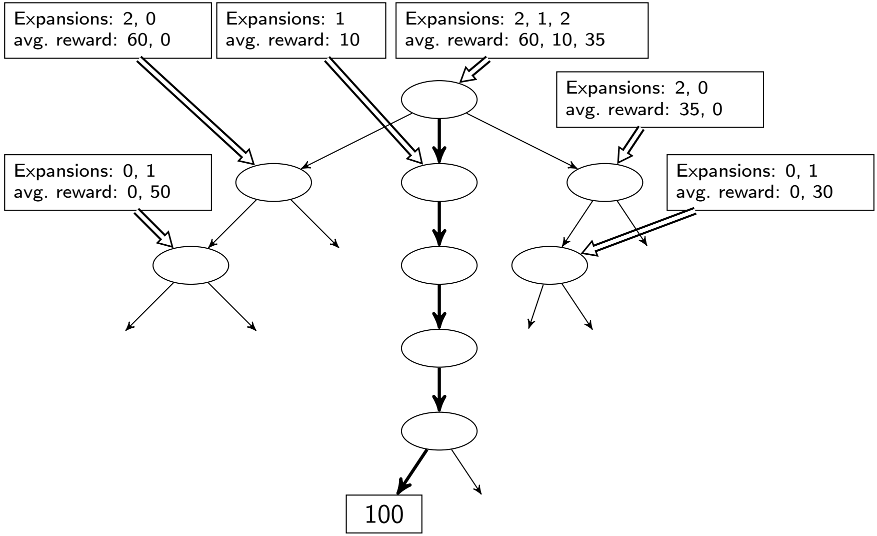
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.66 (Redoing the previous example).**



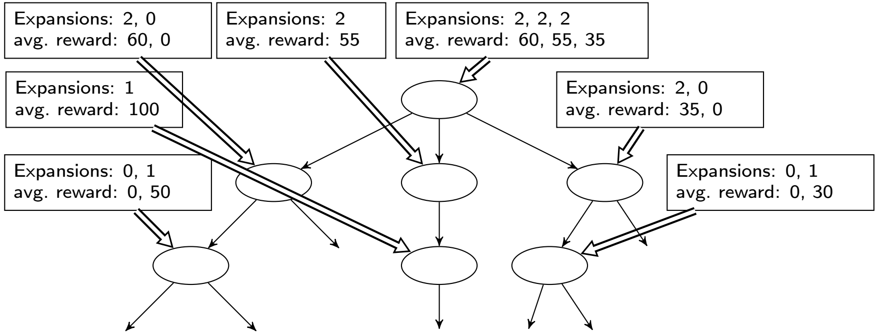
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.67 (Redoing the previous example).**



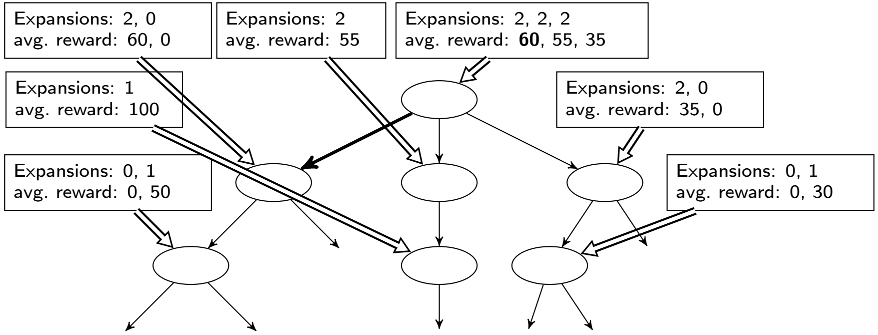
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.68 (Redoing the previous example).**



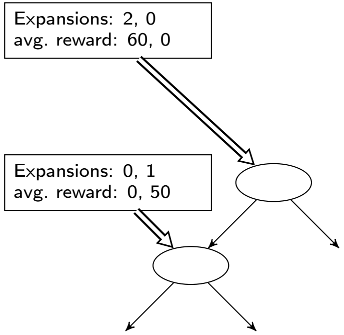
Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.69 (Redoing the previous example).**



Monte-Carlo Tree Search: Building the Tree

- **Idea:** We can save work by building the tree as we go along.
- **Example 5.70 (Redoing the previous example).**



How to Guide the Search in MCTS?

- **How to sample?:** What exactly is "random"?
- **Classical formulation:** balance exploitation vs. exploration.
  - Exploitation: Prefer moves that have high average already (interesting regions of state space)
  - Exploration: Prefer moves that have not been tried a lot yet (don't overlook other, possibly better, options)
- UCT: "Upper Confidence bounds applied to Trees" [KS06].
  - Inspired by Multi-Armed Bandit (as in: Casino) problems.
  - Basically a formula defining the balance. Very popular (buzzword).
  - Recent critics (e.g. [FD14]): Exploitation in search is very different from the Casino, as the "accumulated rewards" are fictitious (we're only thinking about the game, not actually playing and winning/losing all the time).

AlphaGo: Overview

- **Definition 5.71 (Neural Networks in AlphaGo).**
  - Policy networks: Given a state s, output a probability distribution over the actions applicable in s.
  - Value networks: Given a state s, output a number estimating the game value of s.
- **Combination with MCTS:**
  - Policy networks bias the action choices within the MCTS tree (and hence the leaf state selection), and bias the random samples.
  - Value networks are an additional source of state values in the MCTS tree, along with the random samples.
- And now in a little more detail

Neural Networks in AlphaGo

▶ **Neural network training pipeline and architecture:**

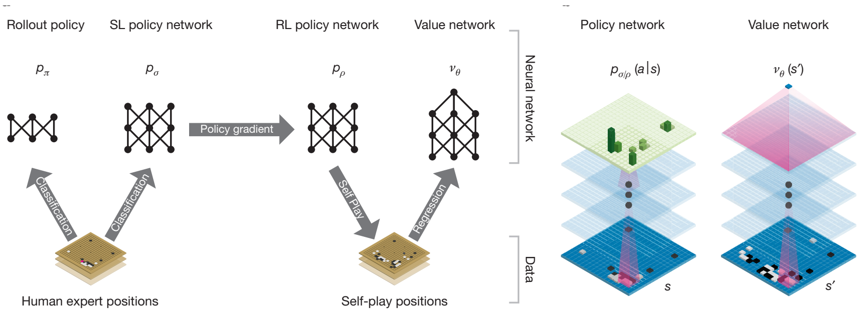


Illustration taken from [Sil+16] .

- Rollout policy p_π: Simple but fast, ≈ prior work on Go.
- SL policy network p_σ: Supervised learning, human-expert data ("learn to choose an expert action").
- RL policy network p_ρ: Reinforcement learning, self-play ("learn to win").
- Value network v_θ: Use self-play games with p_ρ as training data for game-position evaluation v_θ ("predict which player will win in this state").

Neural Networks + MCTS in AlphaGo

- **Monte Carlo tree search in AlphaGo:**

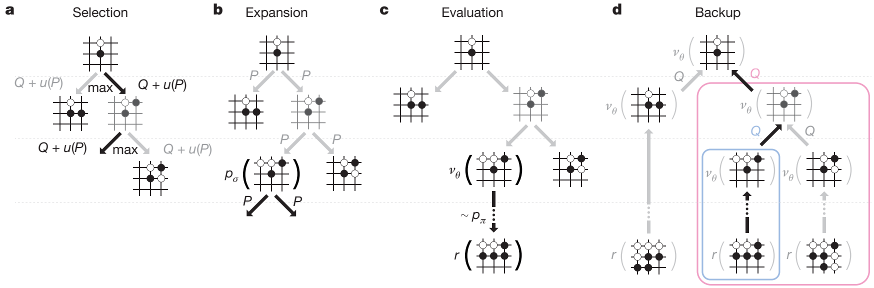


Illustration taken from [Sil+16]

- *Rollout policy* p_π: Action choice in random samples.
- *SL policy network* p_σ: Action choice bias within the UCTS tree (stored as "P", gets smaller to "$u(P)$" with number of visits); along with quality Q.
- *RL policy network* p_ρ: Not used here (used only to learn v_θ).
- *Value network* v_θ: Used to evaluate leaf states s, in linear sum with the value returned by a random sample on s.

7.6 State of the Art

State of the Art

- **Some well-known board games:**
  - **Chess**: Up next.
  - **Othello (Reversi)**: In 1997, “Logistello” beat the human world champion. Best computer players now are clearly better than best human players.
  - **Checkers (Dame)**: Since 1994, “Chinook” is the offical world champion. In 2007, it was shown to be *unbeatable*: Checkers is *solved*. (We know the exact value of, and optimal strategy for, the initial state.)
  - **Go**: In 2016, AlphaGo beat the Grandmaster Lee Sedol, cracking the “holy grail” of board games. In 2017, “AlphaZero” – a variant of AlphaGo with zero prior knowledge beat all reigning champion systems in all board games (including AlphaGo) 100/0 after 24h of self-play.
  - **Intuition:** Board Games are considered a “solved problem” from the AI perspective.

Computer Chess: "Deep Blue" beat Garry Kasparov in 1997

Duell Kasparow gegen Deep Blue (1997): *Demütigende Niederlage*

- 6 games, final score 3.5 : 2.5.
- Specialized chess hardware, 30 nodes with 16 processors each.
- Alphabeta search plus human knowledge. (more details in a moment)
- Nowadays, standard PC hardware plays at world champion level.

Computer Chess: Famous Quotes

- The chess machine is an ideal one to start with, since(Claude Shannon (1949))
  1. the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate),
  2. it is neither so simple as to be trivial nor too difficult for satisfactory solution,
  3. chess is generally considered to require "thinking" for skilful play, [. . .]
  4. the discrete structure of chess fits well into the digital nature of modern computers.
- Chess is the drosophila of Artificial Intelligence. (Alexander Kronrod (1965))

Computer Chess: Another Famous Quote

- In 1965, the Russian mathematician Alexander Kronrod said, "Chess is the Drosophila of artificial intelligence."
  However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing Drosophilae. We would have some science, but mainly we would have very fast fruit flies. (John McCarthy (1997))

7.7 Conclusion

Summary

- Games (2-player turn-taking zero-sum discrete and finite games) can be understood as a simple extension of classical search problems.
- Each player tries to reach a terminal state with the best possible utility (maximal vs. minimal).
- Minimax searches the game depth-first, max'ing and min'ing at the respective turns of each player. It yields perfect play, but takes time $\mathcal{O}(b^d)$ where b is the branching factor and d the search depth.
- Except in trivial games (Tic-Tac-Toe), minimax needs a depth limit and apply an evaluation function to estimate the value of the cut-off states.
- Alpha-beta search remembers the best values achieved for each player elsewhere in the tree already, and prunes out sub-trees that won't be reached in the game.
- Monte Carlo tree search (MCTS) samples game branches, and averages the findings. AlphaGo controls this using neural networks: evaluation function ("value network"), and action filter ("policy network").

Chapter 8
Constraint Satisfaction Problems

8.1 Constraint Satisfaction Problems: Motivation

A (Constraint Satisfaction) Problem

- **Example 1.1 (Tournament Schedule).** Who's going to play against who, when and where?

Constraint Satisfaction Problems (CSPs)

- Standard search problem: state is a “black box” any old data structure that supports goal test, eval, successor state, . . .
- **Definition 1.2.** A constraint satisfaction problem (CSP) is a triple $\langle V, D, C\rangle$ where
  1. V is a finite set V of variables,
  2. an V-indexed family $(D_v)_{v\in V}$ of domains, and
  3. for some subsets $\{v_1, \ldots, v_k\} \subseteq V$ a constraint $C_{\{v_1,\ldots,v_k\}} \subset D_{v_1} \times \ldots \times D_{v_k}$.

  A variable assignment $\varphi \in (v\in V) \to D_v$ is a solution for C, iff $\langle\varphi(v_1), \ldots, \varphi(v_k)\rangle \in C_{\{v_1,\ldots,v_k\}}$ for all $\{v_1, \ldots, v_k\} \subseteq V$.
- **Definition 1.3.** A CSP γ is called satisfiable, iff it has a solution: a total variable assignment φ that satisfies all constraints.
- **Definition 1.4.** The process of finding solutions to CSPs is called constraint solving.
- *Remark 1.5.* We are using factored representation for world states now!
- Allows useful *general-purpose* algorithms with more power than standard tree search algorithm.

Another Constraint Satisfaction Problem

- **Example 1.6 (SuDoKu).** Fill the cells with row/column/block-unique digits

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

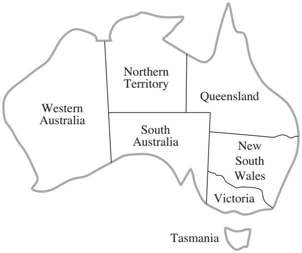
$\leadsto$

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

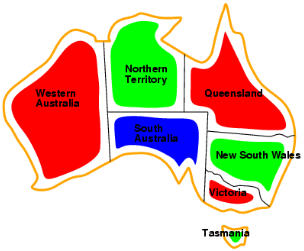
  - Variables: The 81 cells.
  - Domains: Numbers $1, \ldots, 9$.
  - Constraints: Each number only once in each row, column, block.

CSP Example: Map-Coloring

- **Definition 1.7.** Given a map M, the map coloring problem is to assign colors to regions in a map so that no adjoining regions have the same color.
- **Example 1.8 (Map coloring in Australia).**



- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i = \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$
- Constraints: adjacent regions must have different colors e.g., $\mathrm{WA} \neq \mathrm{NT}$ (if the language allows this), or $\langle \mathrm{WA}, \mathrm{NT}\rangle \in \{\langle \mathrm{red}, \mathrm{green}\rangle, \langle \mathrm{red}, \mathrm{blue}\rangle, \langle \mathrm{green}, \mathrm{red}\rangle, \ldots\}$



- **Intuition**: solutions map variables to domain values satisfying all constraints,
- e.g., $\{\mathrm{WA} = \mathrm{red}, \mathrm{NT} = \mathrm{green}, \ldots\}$

Bundesliga Constraints

- Variables: $v_{A\text{vs.}B}$ where A and B are teams, with domains $\{1,\ldots,34\}$: For each match, the index of the weekend where it is scheduled.
- (Some) constraints:

- If $\{A,B\}\cap\{C,D\}\neq\emptyset$: $v_{A\text{vs.}B}\neq v_{C\text{vs.}D}$ (each team only one match per day).
- If $\{A,B\}=\{C,D\}$: $v_{A\text{vs.}B}\leq 17<v_{C\text{vs.}D}$ or $v_{C\text{vs.}D}\leq 17<v_{A\text{vs.}B}$ (each pairing exactly once in each half-season).
- If $A=C$: $v_{A\text{vs.}B}+1\neq v_{C\text{vs.}D}$ (each team alternates between home matches and away matches).
- Leading teams of last season meet near the end of each half-season.
- ...

How to Solve the Bundesliga Constraints?

- 306 nested for-loops (for each of the 306 matches), each ranging from 1 to 306. Within the innermost loop, test whether the current values are (a) a permutation and, if so, (b) a legal Bundesliga schedule.
  - **Estimated running time**: End of this universe, and the next couple billion ones after it . . .
- Directly enumerate all permutations of the numbers $1, \ldots, 306$, test for each whether it's a legal Bundesliga schedule.
  - **Estimated running time**: Maybe only the time span of a few thousand universes.
- View this as variables/constraints and use backtracking (this chapter)
  - **Executed running time**: About 1 minute.
- **How do they actually do it?:** Modern computers and CSP methods: fractions of a second. 19th (20th/21st?) century: Combinatorics and manual work.
- **Try it yourself:** with an off-the shelf CSP solver, e.g. Minion [Min]

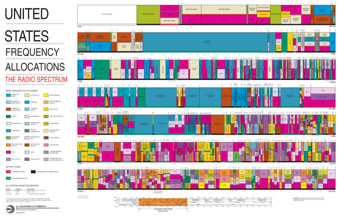
More Constraint Satisfaction Problems

Traveling Tournament Problem

Scheduling

Timetabling

Radio Frequency Assignment



Our Agenda for This Topic

- Our treatment of the topic "Constraint Satisfaction Problems" consists of Chapters 7 and 8. in [RN03]
- **This Chapter**: Basic definitions and concepts; naïve backtracking search.
  - Sets up the framework. Backtracking underlies many successful algorithms for solving constraint satisfaction problems (and, naturally, we start with the simplest version thereof).
- **Next Chapter**: Constraint propagation and decomposition methods.
  - Constraint propagation reduces the search space of backtracking. Decomposition methods break the problem into smaller pieces. Both are crucial for efficiency in practice.

Our Agenda for This Chapter

- How are constraint networks, and assignments, consistency, solutions: How are constraint satisfaction problems defined? What is a solution?
  - Get ourselves on firm ground.

Our Agenda for This Chapter

- How are constraint networks, and assignments, consistency, solutions: How are constraint satisfaction problems defined? What is a solution?
  - Get ourselves on firm ground.
- **Naïve Backtracking**: How does backtracking work? What are its main weaknesses?
  - Serves to understand the basic workings of this wide-spread algorithm, and to motivate its enhancements.

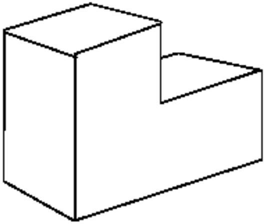
Our Agenda for This Chapter

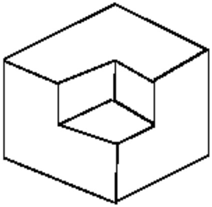
- How are constraint networks, and assignments, consistency, solutions: How are constraint satisfaction problems defined? What is a solution?
  - Get ourselves on firm ground.
- **Naïve Backtracking**: How does backtracking work? What are its main weaknesses?
  - Serves to understand the basic workings of this wide-spread algorithm, and to motivate its enhancements.
- **Variable- and Value Ordering**: How should we guide backtracking searches?
  - Simple methods for making backtracking aware of the structure of the problem, and thereby reduce search.

8.2 The Waltz Algorithm

The Waltz Algorithm

- **Remark:** One of the earliest examples of applied CSPs.
- **Motivation:** Interpret line drawings of polyhedra.

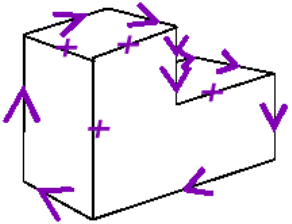


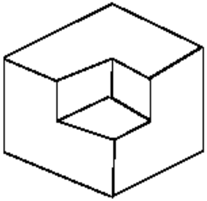


- **Problem:** Are intersections convex or concave? (interpret $\widehat{=}$ label as such)
- **Idea:** Adjacent intersections impose constraints on each other. Use CSP to find a unique set of labelings.

Waltz Algorithm on Simple Scenes

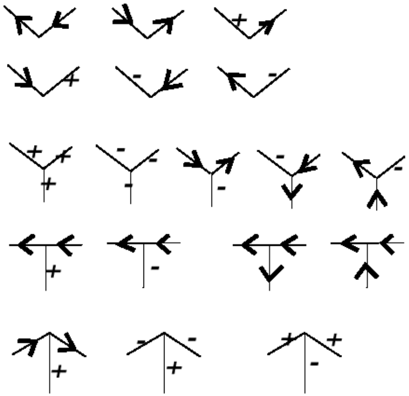
- **Assumptions:** All objects
  - have no shadows or cracks,
  - have only three-faced vertices,
  - are in "general position", i.e. no junctions change with small movements of the eye.
- **Observation 2.1.** *Then each line on the images is one of the following:*
  - *a boundary line (edge of an object) (<) with right hand of arrow denoting "solid" and left hand denoting "space"*
  - *an interior convex edge* (label with "+")
  - *an interior concave edge* (label with "-")





18 Legal Kinds of Junctions

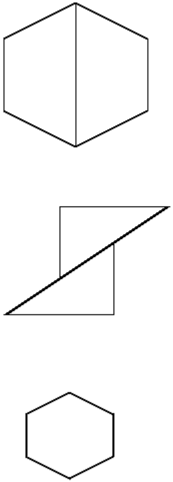
- **Observation 2.2.** *There are only 18 "legal" kinds of junctions:*

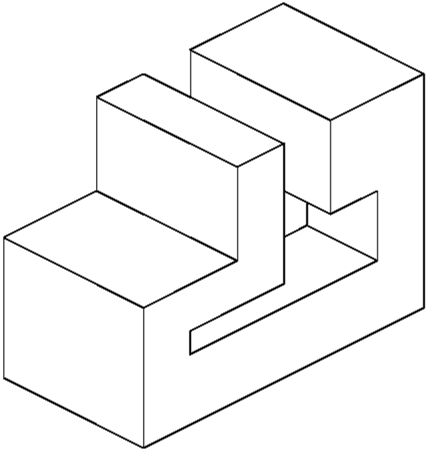


- **Idea:** given a representation of a diagram
  - label each junction in one of these manners (lots of possible ways)
  - junctions must be labeled, so that lines are labeled consistently
- **Fun Fact:** CSP always works perfectly! (early success story for CSP [Wal75])

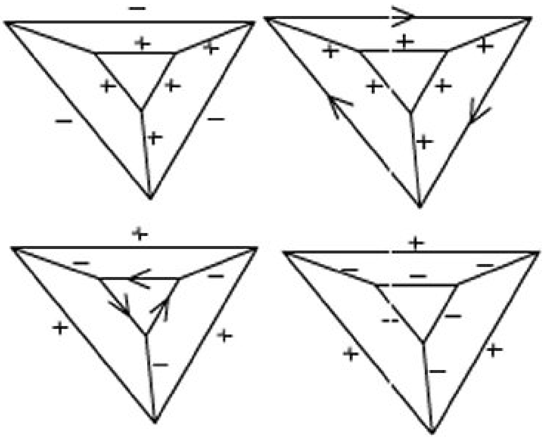
Waltz's Examples

- In his dissertation 1972 [Wal75] David Waltz used the following examples

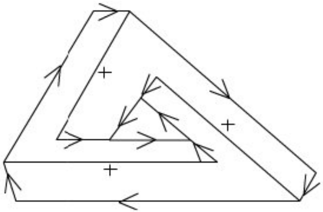




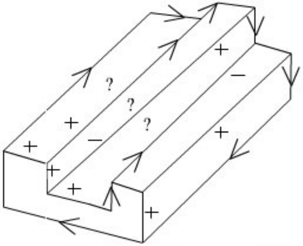
Waltz Algorithm (More Examples): Ambiguous Figures



Waltz Algorithm (More Examples): Impossible Figures



Consistent labelling for impossible figure



No consistent labelling possible

8.3 CSP: Towards a Formal Definition

Types of CSPs

- **Definition 3.1.** We call a CSP discrete, iff all of the variables have countable domains; we have two kinds:
  - finite domains (size $d \rightsquigarrow \mathcal{O}(d^n)$ solutions)
    - e.g., Boolean CSPs (solvability $\widehat{=}$ Boolean satisfiability $\rightsquigarrow$ NP complete)
  - infinite domains (e.g. integers, strings, etc.)
    - e.g., job scheduling, variables are start/end days for each job
    - need a "constraint language", e.g., $\mathit{StartJob}_1 + 5 \leq \mathit{StartJob}_3$
    - linear constraints decidable, nonlinear ones undecidable
- **Definition 3.2.** We call a CSP continuous, iff one domain is uncountable.
- **Example 3.3.** Start/end times for Hubble Telescope observations form a continuous CSP.
- **Theorem 3.4.** *Linear constraints solvable in poly time by linear programming methods.*
- **Theorem 3.5.** *There cannot be optimal algorithms for nonlinear constraint systems.*

Types of Constraints

- We classify the constraints by the number of variables they involve.
- **Definition 3.6.** Unary constraints involve a single variable, e.g., $\mathrm{SA} \neq \mathit{green}$.
- **Definition 3.7.** Binary constraints involve pairs of variables, e.g., $\mathrm{SA} \neq \mathrm{WA}$.
- **Definition 3.8.** Higher-order constraints involve $n = 3$ or more variables, e.g., cryptarithmetic column constraints.
  The number n of variables is called the order of the constraint.
- **Definition 3.9.** Preferences (soft constraint) (e.g., red is better than green) are often representable by a cost for each variable assignment $\leadsto$ constrained optimization problems.

Non-Binary Constraints, e.g. "Send More Money"

- **Example 3.10 (Send More Money).** A student writes home:

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Puzzle: letters stand for digits, addition should work out (parents send MONEY€)

  - Variables: S, E, N, D, M, O, R, Y, each with domain $\{0, \ldots, 9\}$.
  - Constraints:
    1. all variables should have different values: $S \neq E$, $S \neq N$, ...
    2. first digits are non-zero: $S \neq 0$, $M \neq 0$.
    3. the addition scheme should work out: i.e.
       $1000 \cdot S + 100 \cdot E + 10 \cdot N + D + 1000 \cdot M + 100 \cdot O + 10 \cdot R + E = 10000 \cdot M + 1000 \cdot 0 + 100 \cdot N + 10 \cdot E + Y$.

  BTW: The solution is $S \mapsto 9, E \mapsto 5, N \mapsto 6, D \mapsto 7, M \mapsto 1, O \mapsto 0, R \mapsto 8, Y \mapsto 2 \leadsto$ parents send 10652€

- **Definition 3.11.** Problems like the one in **??** are called crypto-arithmetic puzzles.

Encoding Higher-Order Constraints as Binary ones

- **Problem:** The last constraint is of order 8. ($n = 8$ variables involved)
- **Observation 3.12.** *We can write the addition scheme constraint column wise using auxiliary variables, i.e. variables that do not "occur" in the original problem.*

$$\begin{aligned} D + E &= Y + 10 \cdot X_1 \\ X_1 + N + R &= E + 10 \cdot X_2 \\ X_2 + E + O &= N + 10 \cdot X_3 \\ X_3 + S + M &= O + 10 \cdot M \end{aligned}$$

$$\begin{array}{ccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

These constraints are of order ≤ 5.

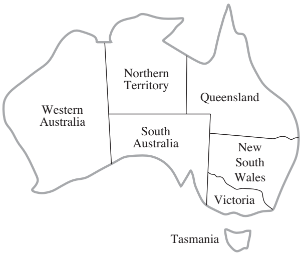
- **General Recipe:** For $n \geq 3$, encode $C(v_1, \ldots, v_{n-1}, v_n)$ as

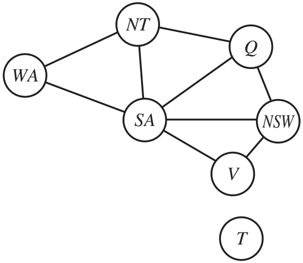
$$C(p_1(x), \ldots, p_{n-1}(x), v_n) \wedge v_1 = p_1(x) \wedge \ldots \wedge v_{n-1} = p_{n-1}(x)$$

- **Problem:** The problem structure gets hidden. (search algorithms can get confused)

Constraint Graph

- **Definition 3.13.** A binary CSP is a CSP where each constraint is unary or binary.
- **Observation 3.14.** *A binary CSP forms a graph called the constraint graph whose nodes are variables, and whose edges represent the constraints.*
- **Example 3.15.** Australia as a binary CSP





- **Intuition:** General-purpose CSP algorithms use the graph structure to speed up search. (E.g., Tasmania is an independent subproblem!)

Real-world CSPs

- **Example 3.16 (Assignment problems).** e.g., who teaches what class
- **Example 3.17 (Timetabling problems).** e.g., which class is offered when and where?
- **Example 3.18 (Hardware configuration).**
- **Example 3.19 (Spreadsheets).**
- **Example 3.20 (Transportation scheduling).**
- **Example 3.21 (Factory scheduling).**
- **Example 3.22 (Floorplanning).**
- **Note:** many real-world problems involve real-valued variables $\leadsto$ continuous CSPs.

8.4 Constraint Networks: Formalizing Binary CSPs

Constraint Networks (Formalizing binary CSPs)

- **Definition 4.1.** A constraint network is a triple $\gamma := \langle V, D, C\rangle$, where
  - V is a finite set of variables,
  - $D := \{D_v \mid v \in V\}$ the set of their domains, and
  - $C := \{C_{uv} \subseteq D_u \times D_v \mid u, v \in V \text{ and } u \neq v\}$ is a set of constraints with $C_{uv} = C_{vu}^{-1}$.

  We call the undirected graph $\langle V, \{(u,v) \in V^2 \mid C_{uv} \neq D_u \times D_v\}\rangle$, the constraint graph of γ.
- We will talk of CSPs and mean constraint networks.
- **Remarks:** The mathematical formulation gives us a lot of leverage:
  - $C_{uv} \subseteq D_u \times D_v \mathrel{\widehat{=}}$ possible assignments to variables u and v
  - Relations are the most general formalization, generally we use symbolic formulations, e.g. "$u = v$" for the relation $C_{uv} = \{(a,b) \mid a = b\}$ or "$u \neq v$".
  - We can express unary constraints C_u by restricting the domain of v: $D_v := C_v$.

Example: SuDoKu as a Constraint Network

- **Example 4.2 (Formalize SuDoKu).** We use the added formality to encode SuDoKu as a constraint network, not just as a CSP as **??**.

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

  - Variables:

Note that the ideas are still the same as **??**, but in constraint networks we have a language to formulate things precisely.

Example: SuDoKu as a Constraint Network

- **Example 4.3 (Formalize SuDoKu).** We use the added formality to encode SuDoKu as a constraint network, not just as a CSP as **??**.

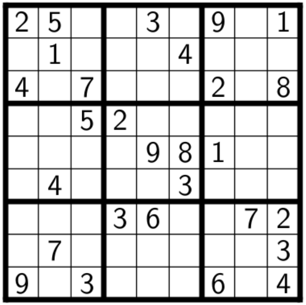
2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

  - Variables: $V = \{v_{ij} \mid 1 \leq i, j \leq 9\}$: v_{ij} =cell in row i column j.
  - Domains

Note that the ideas are still the same as **??**, but in constraint networks we have a language to formulate things precisely.

Example: SuDoKu as a Constraint Network

- **Example 4.4 (Formalize SuDoKu).** We use the added formality to encode SuDoKu as a constraint network, not just as a CSP as **??**.



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

  - Variables: $V = \{v_{ij} \mid 1 \leq i,j \leq 9\}$: v_{ij} =cell in row i column j.
  - Domains For all $v \in V$: $D_v = D = \{1,\ldots,9\}$.
  - Unary constraint:

Note that the ideas are still the same as **??**, but in constraint networks we have a language to formulate things precisely.

Example: SuDoKu as a Constraint Network

- **Example 4.5 (Formalize SuDoKu).** We use the added formality to encode SuDoKu as a constraint network, not just as a CSP as **??**.

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

  - Variables: $V = \{v_{ij} \mid 1 \leq i, j \leq 9\}$: v_{ij} =cell in row i column j.
  - Domains
  - Unary constraint: $C_{v_{ij}} = \{d\}$ if cell i, j is pre-filled with d.
  - (Binary) constraint:

  Note that the ideas are still the same as **??**, but in constraint networks we have a language to formulate things precisely.

Example: SuDoKu as a Constraint Network

- **Example 4.6 (Formalize SuDoKu).** We use the added formality to encode SuDoKu as a constraint network, not just as a CSP as **??**.

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

  - Variables: $V = \{v_{ij} \mid 1 \leq i,j \leq 9\}$: v_{ij} =cell in row i column j.
  - Domains
  - Unary constraint:
  - (Binary) constraint: $C_{v_{ij}v_{i'j'}} \mathrel{\widehat{=}}$ "$v_{ij} \neq v_{i'j'}$", i.e. $C_{v_{ij}v_{i'j'}} = \{(d,d') \in D \times D \mid d \neq d'\}$, for: $i = i'$ (same row), or $j = j'$ (same column), or $(\lceil \frac{i}{3} \rceil, \lceil \frac{j}{3} \rceil) = (\lceil \frac{i'}{3} \rceil, \lceil \frac{j'}{3} \rceil)$ (same block).

Note that the ideas are still the same as **??**, but in constraint networks we have a language to formulate things precisely.

Constraint Networks (Solutions)

- Let $\gamma := \langle V, D, C \rangle$ be a constraint network.
- **Definition 4.7.** We call a partial function $a : V \rightharpoonup \bigcup_{u \in V} D_u$ a variable assignment if $a(u) \in D_u$ for all $u \in \mathrm{dom}(a)$.
- **Definition 4.8.** Let $\mathcal{C} := \langle V, D, C \rangle$ be a constraint network and $a : V \rightharpoonup \bigcup_{v \in V} D_v$ a variable assignment. We say that a satisfies (otherwise violates) a constraint C_{uv}, iff $u, v \in \mathrm{dom}(a)$ and $(a(u),a(v)) \in C_{uv}$. a is called consistent in $\mathcal{C}$, iff it satisfies all constraints in $\mathcal{C}$. A value $w \in D_u$ is legal for a variable u in $\mathcal{C}$, iff $\{(u,w)\}$ is a consistent assignment in $\mathcal{C}$. A variable with illegal value under a is called conflicted.
- **Example 4.9.** The empty assignment ϵ is (trivially) consistent in any constraint network.
- **Definition 4.10.** Let f and g be variable assignments, then we say that f extends (or is an extension of) g, iff $\mathrm{dom}(g) \subset \mathrm{dom}(f)$ and $f|_{\mathrm{dom}(g)} = g$.
- **Definition 4.11.** We call a consistent (total) assignment a solution for γ and γ itself solvable or satisfiable.

How it all fits together

- **Lemma 4.12.** *Higher-order constraints can be transformed into equi-satisfiable binary constraints using auxiliary variables.*
- **Corollary 4.13.** *Any CSP can be represented by a constraint network.*
- **In other words** The notion of a constraint network is a refinement of a CSP.
- So we will stick to constraint networks in this course.
- **Observation 4.14.** *We can view a constraint network as a search problem, if we take the states as the variable assignments, the actions as assignment extensions, and the goal states as consistent assignments.*
- **Idea:** We will explore that idea for algorithms that solve constraint networks.

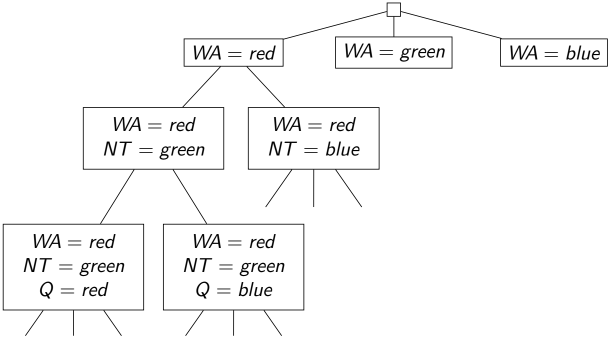
8.5 CSP as Search

Standard search formulation (incremental)

- **Idea:** Every constraint network induces a single state problem.
- **Definition 5.1 (Let's do the math).** Given a constraint network $\gamma := \langle V, D, C \rangle$, then $\Pi_\gamma := \langle \mathcal{S}_\gamma, \mathcal{A}_\gamma, \mathcal{T}_\gamma, \mathcal{I}_\gamma, \mathcal{G}_\gamma \rangle$ is called the search problem induced by γ, iff
  - State $\mathcal{S}_\gamma$ are variable assignments
  - Action $\mathcal{A}_\gamma$: extend $\varphi \in \mathcal{S}_\gamma$ by a pair $x \mapsto v$ not conflicted with φ.
  - Transition model $\mathcal{T}_\gamma(a, \varphi) = \varphi, x \mapsto v$ (extended assignment)
  - Initial state $\mathcal{I}_\gamma$: the empty assignment ϵ.
  - Goal states $\mathcal{G}_\gamma$: the total, consistent assignments
- **What has just happened?:** We interpret a constraint network γ as a search problem Π_γ. A solution to Π_γ induces a solution to γ.
- **Idea:** We have algorithms for that: e.g. tree search.
- **Remark:** This is the same for all CSPs! ☺
  $\leadsto$ fail if no consistent assignments exist (not fixable!)

Standard search formulation (incremental)

- **Example 5.2.** A search tree for $\Pi_{Australia}$:



- **Observation:** Every solution appears at depth n with n variables.
- **Idea:** Use depth first search!
- **Observation:** Path is irrelevant $\leadsto$ can use local search algorithms.
- Branching factor $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!! ☹

Backtracking Search

- Assignments for different variables are independent!
  - e.g. first $\mathrm{WA} = \mathrm{red}$ then $\mathrm{NT} = \mathrm{green}$ vs. first $\mathrm{NT} = \mathrm{green}$ then $\mathrm{WA} = \mathrm{red}$
  - $\rightsquigarrow$ we only need to consider assignments to a single variable at each node
  - $\rightsquigarrow$ $b = d$ and there are d^n leaves.
- **Definition 5.3.** Depth first search for CSPs with single-variable assignment extensions actions is called backtracking search.
- Backtracking search is the basic uninformed algorithm for CSPs.
- It can solve the n-queens problem for $\approxeq n, 25$.

Backtracking Search (Implementation)

- **Definition 5.4.** The generic backtracking search algorithm:

```
procedure Backtracking–Search(csp ) returns solution/failure
     return Recursive–Backtracking ($\emptyset$, csp)

procedure Recursive–Backtracking (assignment) returns soln/failure
  if assignment is complete then return assignment
  var := Select–Unassigned–Variable(Variables[csp], assignment, csp)
  foreach value in Order–Domain–Values(var, assignment, csp) do
      if value is consistent with assignment given Constraints[csp] then
          add {var = value} to assignment
          result := Recursive–Backtracking(assignment,csp)
          if result $\neq$ failure then return result
          remove {var= value} from assignment
  return failure
```

Backtracking in Australia

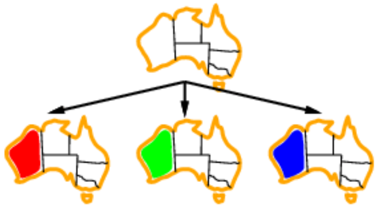
- **Example 5.5.** We apply backtracking search for a map coloring problem:

Step 1:

Backtracking in Australia

- **Example 5.6.** We apply backtracking search for a map coloring problem:

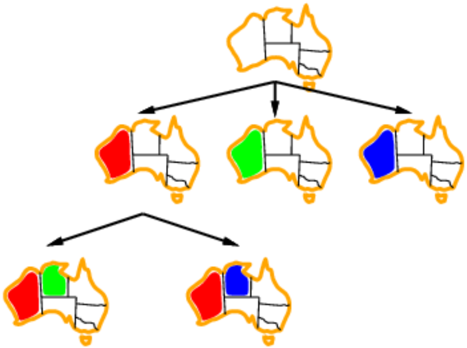
Step 2:



Backtracking in Australia

- **Example 5.7.** We apply backtracking search for a map coloring problem:

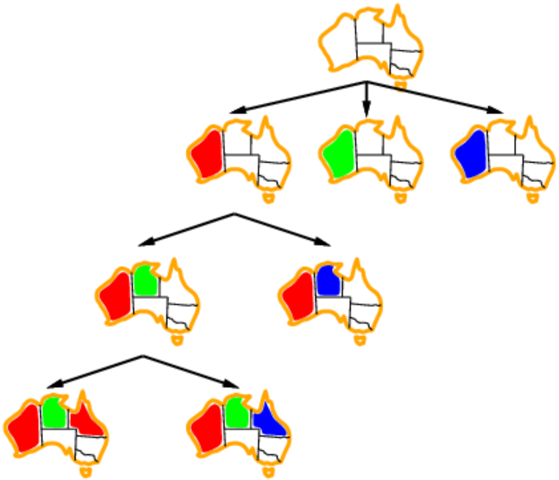
Step 3:



Backtracking in Australia

- **Example 5.8.** We apply backtracking search for a map coloring problem:

Step 4:

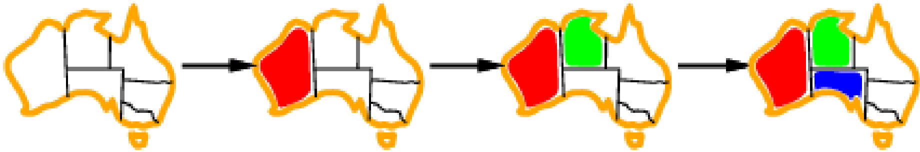


Improving Backtracking Efficiency

- General-purpose methods can give huge gains in speed for backtracking search.
- Answering the following questions well helps find powerful heuristics:
  1. Which variable should be assigned next? (i.e. a variable ordering heuristic)
  2. In what order should its values be tried? (i.e. a value ordering heuristic)
  3. Can we detect inevitable failure early? (for pruning strategies)
  4. Can we take advantage of problem structure? ($\leadsto$ inference)
- **Observation:** Questions 1/2 correspond to the missing subroutines Select–Unassigned–Variable and Order–Domain–Values from **??**.

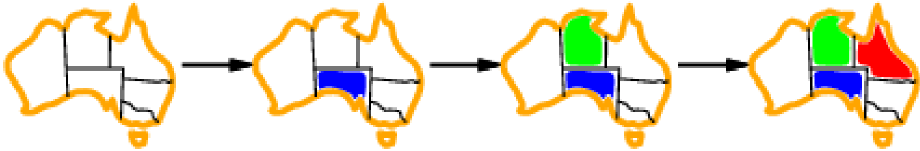
Heuristic: Minimum Remaining Values (Which Variable)

- **Definition 5.9.** The minimum remaining values (MRV) heuristic for backtracking search always chooses the variable with the fewest legal values, i.e. a variable v that given an initial assignment a minimizes $\#(\{d \in D_v \mid a \cup \{v \mapsto d\} \text{ is consistent}\})$.
- **Intuition:** By choosing a most constrained variable v first, we reduce the branching factor (number of sub trees generated for v) and thus reduce the size of our search tree.
- **Extreme case:** If $\#(\{d \in D_v \mid a \cup \{v \mapsto d\} \text{ is consistent}\}) = 1$, then the value assignment to v is forced by our previous choices.
- **Example 5.10.** In step 3 of **??**, there is only one remaining value for SA!



Degree Heuristic (Variable Order Tie Breaker)

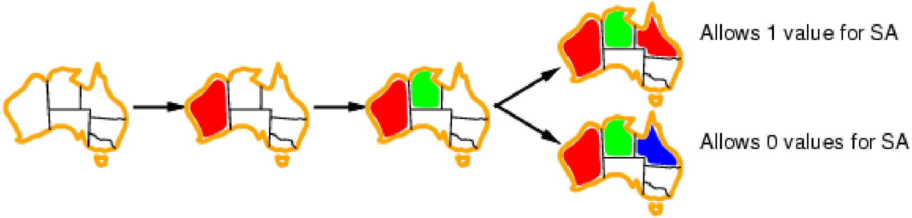
- **Problem:** Need a tie-breaker among MRV variables! (there was no preference in step 1,2)
- **Definition 5.11.** The degree heuristic in backtracking search always chooses a most constraining variable, i.e. given an initial assignment a always pick a variable v with $\#(\{v \in (V \backslash \mathrm{dom}(a)) \mid C_{uv} \in C\})$ maximal.
- By choosing a most constraining variable first, we detect inconsistencies earlier on and thus reduce the size of our search tree.
- **Commonly used strategy combination:** From the set of most constrained variable, pick a most constraining variable.
- **Example 5.12.**



Degree heuristic: $\mathrm{SA} = 5$, $\mathrm{T} = 0$, all others 2 or 3.

Least Constraining Value Heuristic (Value Ordering)

- **Definition 5.13.** Given a variable v, the least constraining value heuristic chooses the least constraining value for v: the one that rules out the fewest values in the remaining variables, i.e. for a given initial assignment a and a chosen variable v pick a value $d \in D_v$ that minimizes $\#(\{e \in D_u \mid u \notin \mathrm{dom}(a), C_{uv} \in C, \text{ and } (e,d) \notin C_{uv}\})$
- By choosing the least constraining value first, we increase the chances to not rule out the solutions below the current node.
- **Example 5.14.**



- Combining these heuristics makes 1000 queens feasible.

8.6 Conclusion & Preview

Summary & Preview

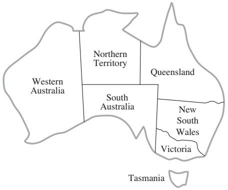
- Summary of "CSP as Search":
  - Constraint networks γ consist of variables, associated with finite domains, and constraints which are binary relations specifying permissible value pairs.
  - A variable assignment a maps some variables to values. a is consistent if it complies with all constraints. A consistent total assignment is a solution.
  - The constraint satisfaction problem (CSP) consists in finding a solution for a constraint network. This has numerous applications including, e.g., scheduling and timetabling.
  - Backtracking search assigns variable one by one, pruning inconsistent variable assignments.
  - Variable orderings in backtracking can dramatically reduce the size of the search tree. Value orderings have this potential (only) in solvable sub trees.
- **Up next:** Inference and decomposition, for improved efficiency.

Chapter 9
Constraint Propagation

9.1 Introduction

Illustration: Constraint Propagation

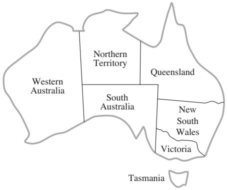
- **Example 1.1.** A constraint network γ:



- **Question:** Can we add a constraint without losing any solutions?
- **Example 1.2.** $C_{\mathrm{WAQ}} :=$ "$=$". If WA and Q are assigned different colors, then NT must be assigned the 3rd color, leaving no color for SA.
- **Intuition:** Adding constraints without losing solutions
  $\widehat{=}$ obtaining an equivalent network with a "tighter description"
  $\rightsquigarrow$ a smaller number of consistent (partial) variable assignments
  $\rightsquigarrow$ more efficient search!

Illustration: Decomposition

- **Example 1.3.** Constraint network γ:



- We can separate this into two independent constraint networks.
- Tasmania is not adjacent to any other state. Thus we can color Australia first, and assign an arbitrary color to Tasmania afterwards.
- Decomposition methods exploit the structure of the constraint network. They identify separate parts (sub-networks) whose inter-dependencies are "simple" and can be handled efficiently.
- **Example 1.4 (Extreme case).** No inter-dependencies at all, as for Tasmania above.

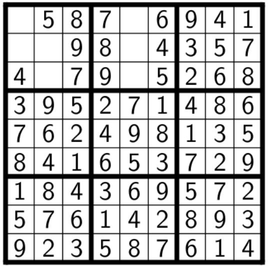
Our Agenda for This Chapter

- Constraint propagation: How does inference work in principle? What are relevant practical aspects?
  - Fundamental concepts underlying inference, basic facts about its use.
- Forward checking: What is the simplest instance of inference?
  - Gets us started on this subject.
- Arc consistency: How to make inferences between variables whose value is not fixed yet?
  - Details a state of the art inference method.
- Decomposition: Constraint graphs, and two simple cases
  - How to capture dependencies in a constraint network? What are "simple cases"?
  - Basic results on this subject.
- Cutset conditioning: What if we're not in a simple case?
  - Outlines the most easily understandable technique for decomposition in the general case.

9.2 Constraint Propagation/Inference

Constraint Propagation/Inference: Basic Facts

- **Definition 2.1.** Constraint propagation (i.e inference in constraint networks) consists in deducing additional constraints, that follow from the already known constraints, i.e. that are satisfied in all solutions.
- **Example 2.2.** It's what you do all the time when playing SuDoKu:



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

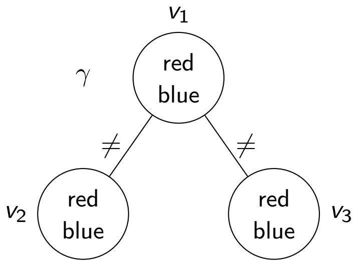
- **Formally:** Replace γ by an equivalent and strictly tighter constraint network γ'.

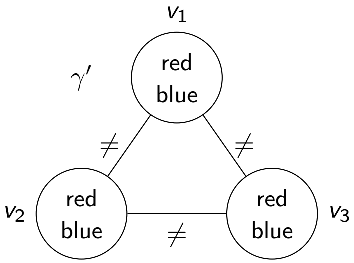
Equivalent Constraint Networks

- **Definition 2.3.** We say that two constraint networks $\gamma := \langle V, D, C\rangle$ and $\gamma' := \langle V, D', C'\rangle$ sharing the same set of variables are equivalent, (write $\gamma' \equiv \gamma$), if they have the same solutions.

Equivalent Constraint Networks

- **Definition 2.5.** We say that two constraint networks $\gamma := \langle V, D, C \rangle$ and $\gamma' := \langle V, D', C' \rangle$ sharing the same set of variables are equivalent, (write $\gamma' \equiv \gamma$), if they have the same solutions.
- **Example 2.6.**

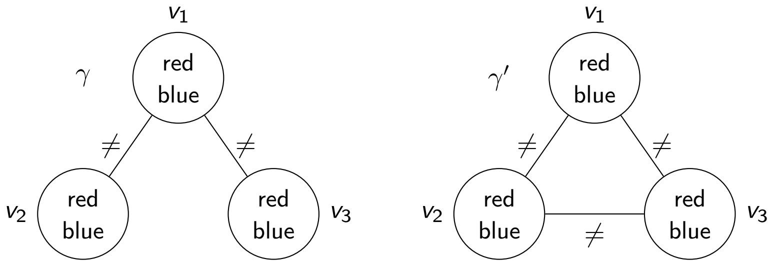




Are these constraint networks equivalent?

Equivalent Constraint Networks

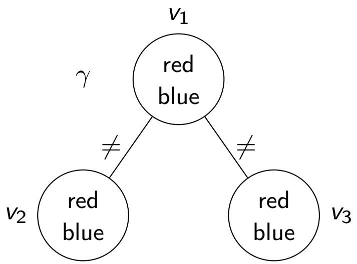
- **Definition 2.7.** We say that two constraint networks $\gamma := \langle V, D, C\rangle$ and $\gamma' := \langle V, D', C'\rangle$ sharing the same set of variables are equivalent, (write $\gamma' \equiv \gamma$), if they have the same solutions.
- **Example 2.8.**

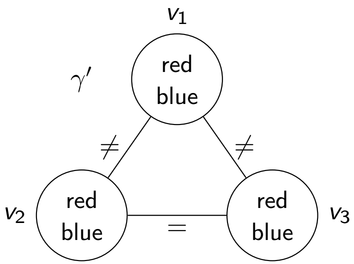


Are these constraint networks equivalent? No.

Equivalent Constraint Networks

- **Definition 2.9.** We say that two constraint networks $\gamma := \langle V, D, C\rangle$ and $\gamma' := \langle V, D', C'\rangle$ sharing the same set of variables are equivalent, (write $\gamma' \equiv \gamma$), if they have the same solutions.
- **Example 2.10.**

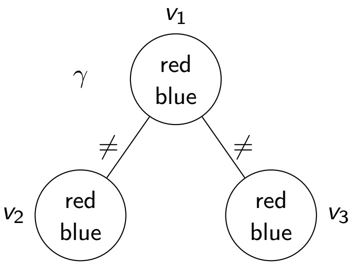


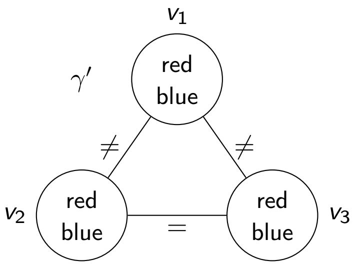


Are these constraint networks equivalent?

Equivalent Constraint Networks

- **Definition 2.11.** We say that two constraint networks $\gamma := \langle V, D, C\rangle$ and $\gamma' := \langle V, D', C'\rangle$ sharing the same set of variables are equivalent, (write $\gamma' \equiv \gamma$), if they have the same solutions.
- **Example 2.12.**





Are these constraint networks equivalent? Yes.

Tightness

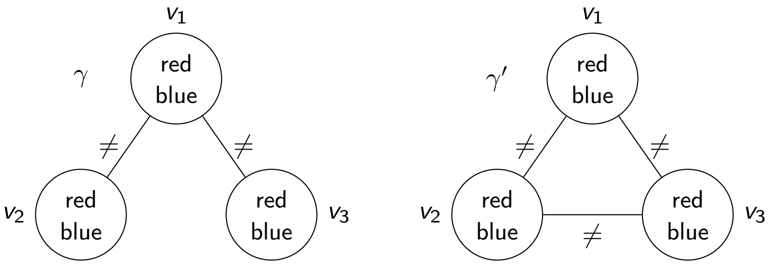
- **Definition 2.13 (Tightness).** Let $\gamma := \langle V, D, C \rangle$ and $\gamma' = \langle V, D', C' \rangle$ be constraint networks sharing the same set of variables, then γ' is tighter than γ, (write $\gamma' \sqsubseteq \gamma$), if:
  (i) For all $v \in V$: $D'_v \subseteq D_v$.
  (ii) For all $u \neq v \in V$ and $C'_{uv} \in C'$: either $C'_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.

  γ' is strictly tighter than γ, (written $\gamma' \sqsubset \gamma$), if at least one of these inclusions is proper.

Tightness

- **Definition 2.15 (Tightness).** Let $\gamma := \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ be constraint networks sharing the same set of variables, then γ' is tighter than γ, (write $\gamma' \sqsubseteq \gamma$), if:
  (i) For all $v \in V$: $D'_v \subseteq D_v$.
  (ii) For all $u \neq v \in V$ and $C'_{uv} \in C'$: either $C'_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.

  γ' is strictly tighter than γ, (written $\gamma' \sqsubset \gamma$), if at least one of these inclusions is proper.
- **Example 2.16.**

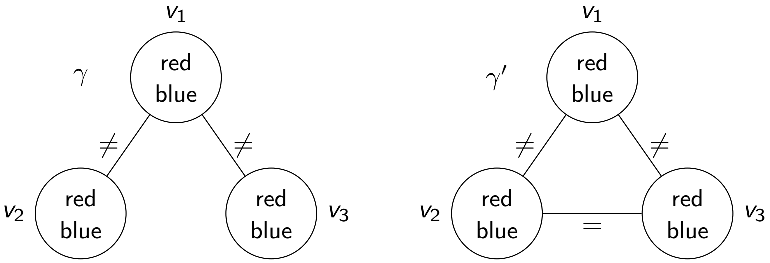


Here, we do have $\gamma' \sqsubseteq \gamma$.

Tightness

- **Definition 2.17 (Tightness).** Let $\gamma := \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ be constraint networks sharing the same set of variables, then γ' is tighter than γ, (write $\gamma' \sqsubseteq \gamma$), if:
  (i) For all $v \in V$: $D'_v \subseteq D_v$.
  (ii) For all $u \neq v \in V$ and $C'_{uv} \in C'$: either $C'_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.

  γ' is strictly tighter than γ, (written $\gamma' \sqsubset \gamma$), if at least one of these inclusions is proper.
- **Example 2.18.**

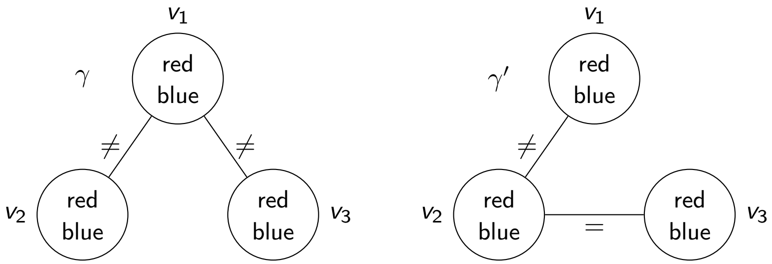


Here, we do have $\gamma' \sqsubseteq \gamma$.

Tightness

- **Definition 2.19 (Tightness).** Let $\gamma := \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ be constraint networks sharing the same set of variables, then γ' is tighter than γ, (write $\gamma' \sqsubseteq \gamma$), if:
  (i) For all $v \in V$: $D'_v \subseteq D_v$.
  (ii) For all $u \neq v \in V$ and $C'_{uv} \in C'$: either $C'_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.

  γ' is strictly tighter than γ, (written $\gamma' \sqsubset \gamma$), if at least one of these inclusions is proper.
- **Example 2.20.**

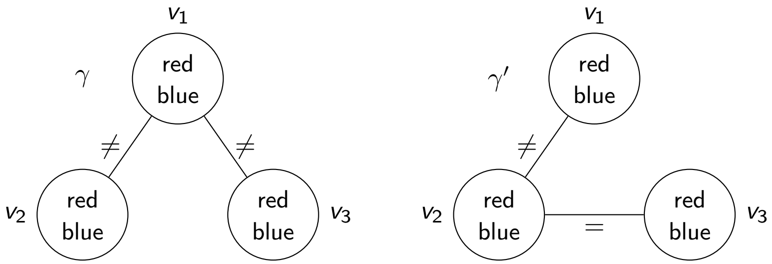


Here, we do not have $\gamma' \sqsubseteq \gamma$!.

Tightness

- **Definition 2.21 (Tightness).** Let $\gamma := \langle V, D, C\rangle$ and $\gamma' = \langle V, D', C'\rangle$ be constraint networks sharing the same set of variables, then γ' is tighter than γ, (write $\gamma' \sqsubseteq \gamma$), if:
  (i) For all $v \in V$: $D'_v \subseteq D_v$.
  (ii) For all $u \neq v \in V$ and $C'_{uv} \in C'$: either $C'_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.

  γ' is strictly tighter than γ, (written $\gamma' \sqsubset \gamma$), if at least one of these inclusions is proper.
- **Example 2.22.**

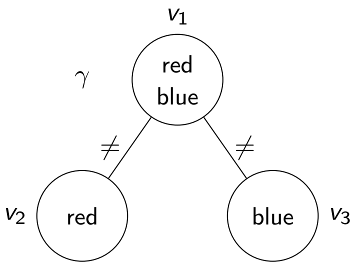


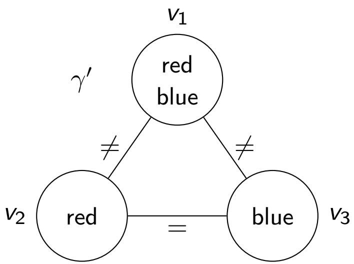
Here, we do not have $\gamma' \sqsubseteq \gamma$!.

- **Intuition:** Strict tightness $\hat{=}$ γ' *has the same constraints as* γ*, plus some*.

Equivalence + Tightness = Inference

- **Theorem 2.23.** *Let γ and γ' be constraint networks such that $\gamma' \equiv \gamma$ and $\gamma' \sqsubseteq \gamma$. Then γ' has the same solutions as, but fewer consistent assignments than, γ.*
- $\leadsto$ γ' is a better encoding of the underlying problem.
- **Example 2.24.** Two equivalent constraint networks (one obviously unsolvable)





ϵ cannot be extended to a solution (neither in γ nor in γ' because they're equivalent); this is obvious (red $\neq$ blue) in γ', but not in γ.

How to Use Constraint Propagation in CSP Solvers?

- **Simple:** Constraint propagation as a pre-process:
  - **When**: Just once before search starts.
  - **Effect**: Little running time overhead, little pruning power. (not considered here)
- **More Advanced:** Constraint propagation during search:
  - **When**: At every recursive call of backtracking.
  - **Effect**: Strong pruning power, may have large running time overhead.
- **Search vs. Inference:** The more complex the inference, the *smaller* the number of search nodes, but the *larger* the running time needed at each node.
- **Idea:** Encode variable assignments as unary constraints (i.e., for $a(v) = d$, set the unary constraint $D_v = \{d\}$), so that inference reasons about *the network restricted to the commitments already made in the search*.

Backtracking With Inference

- **Definition 2.25.** The general algorithm for backtracking with inference is

```
function BacktrackingWithInference($\gamma$,$a$) returns a solution, or ``inconsistent''
  if $a$ is inconsistent then return ``inconsistent''
  if $a$ is a total assignment then return $a$
  $\gamma'$ := a copy of $\gamma$ /* $\gamma' = (V_{\gamma'}, D_{\gamma'}, C_{\gamma'})$ */
  $\gamma'$ := Inference($\gamma'$)
  if exists $v$ with $D_{\gamma'}{}_v = \emptyset$ then return ``inconsistent''
  select some variable $v$ for which $a$ is not defined
  for each $d \in$ copy of $D_{\gamma'}{}_v$ in some order do
    $a' := a \cup \{v = d\}$; $D_{\gamma'}{}_v := \{d\}$ /* makes $a$ explicit as a constraint */
    $a''$ := BacktrackingWithInference($\gamma'$,$a'$)
    if $a'' \neq$ "inconsistent" then return $a''$
  return ``inconsistent''
```

- Exactly the same as **??**, only line 5 new!
- **Inference**(): Any procedure delivering a (tighter) equivalent network.
- **Inference**() typically prunes domains; indicate unsolvability by $D_{\gamma'}{}_v = \emptyset$.
- When backtracking out of a search branch, retract the inferred constraints: these were dependent on a, the search commitments so far.

9.3 Forward Checking

Forward Checking

- **Definition 3.1.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.

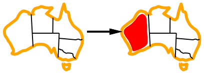
Forward Checking

- **Definition 3.4.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.
- **Example 3.5.** Forward checking in Australia

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

Forward Checking

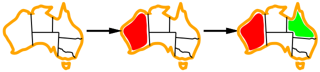
- **Definition 3.7.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.
- **Example 3.8.** Forward checking in Australia



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

Forward Checking

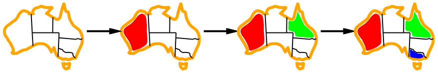
- **Definition 3.10.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.
- **Example 3.11.** Forward checking in Australia



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

Forward Checking

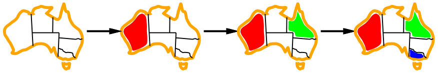
- **Definition 3.13.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.
- **Example 3.14.** Forward checking in Australia



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

Forward Checking

- **Definition 3.16.** Forward checking propagates information about illegal values: Whenever a variable u is assigned by a, delete all values inconsistent with $a(u)$ from every D_v for all variables v connected with u by a constraint.
- **Example 3.17.** Forward checking in Australia



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

- **Definition 3.18 (Inference, Version 1).** Forward checking implemented

```
function ForwardChecking($\gamma$,$a$) returns modified $\gamma$
  for each $v$ where $a(v) = d'$ is defined do
    for each $u$ where $a(u)$ is undefined and $C_{uv} \in C$ do
      $D_u := \{d \in D_u \mid (d,d') \in C_{uv}\}$
  return $\gamma$
```

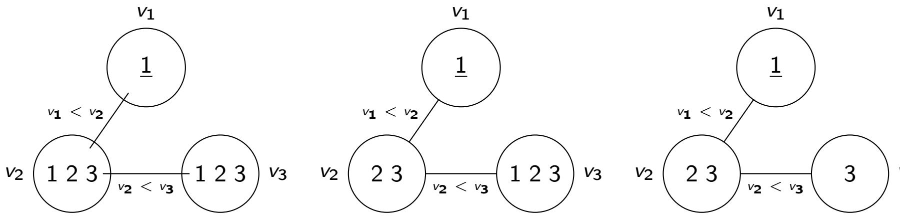
Forward Checking: Discussion

- **Definition 3.19.** An inference procedure is called sound, iff for any input γ the output γ' have the same solutions.
- **Lemma 3.20.** *Forward checking is sound.*
  Proof sketch: Recall here that the assignment a is represented as unary constraints inside γ.
- **Corollary 3.21.** *γ and γ' are equivalent.*
- Incremental computation: Instead of the first for-loop in **??**, use only the inner one every time a new assignment $a(v) = d'$ is added.
- **Practical Properties:**
  - Cheap but useful inference method.
  - Rarely a good idea to not use forward checking (or a stronger inference method subsuming it).
- **Up next:** A stronger inference method (subsuming forward checking).
- **Definition 3.22.** Let p and q be inference procedures, then p subsumes q, if $p(\gamma) \sqsubseteq q(\gamma)$ for any input γ.

9.4 Arc Consistency

When Forward Checking is Not Good Enough

- **Problem:** Forward checking makes inferences only from assigned to unassigned variables.
- **Example 4.1.**



We could do better here: value 3 for v_2 is not consistent with any remaining value for v_3 $\rightsquigarrow$ it can be removed!
But forward checking does not catch this.

Arc Consistency: Definition

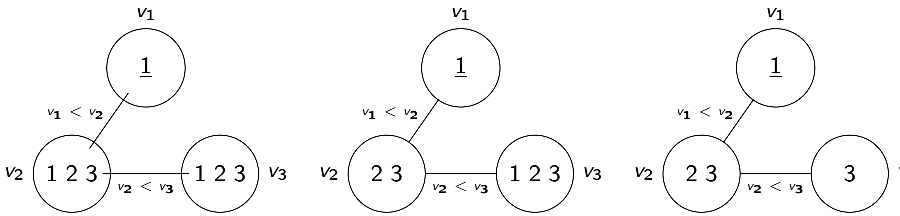
- **Definition 4.2 (Arc Consistency).** Let $\gamma := \langle V, D, C\rangle$ be a constraint network.
  1. A variable $u \in V$ is arc consistent relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d,d') \in C_{uv}$.
  2. The constraint network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

  The concept of arc consistency concerns both levels.
- **Intuition:** Arc consistency $\widehat{=}$ for every domain value and constraint, at least one value on the other side of the constraint "works".
- **Note** the asymmetry between u and v: arc consistency is directed.

Arc Consistency: Example

- **Definition 4.3 (Arc Consistency).** Let $\gamma := \langle V, D, C\rangle$ be a constraint network.
  1. A variable $u \in V$ is arc consistent relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d,d') \in C_{uv}$.
  2. The constraint network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

  The concept of arc consistency concerns both levels.
- **Example 4.4 (Arc Consistency).**

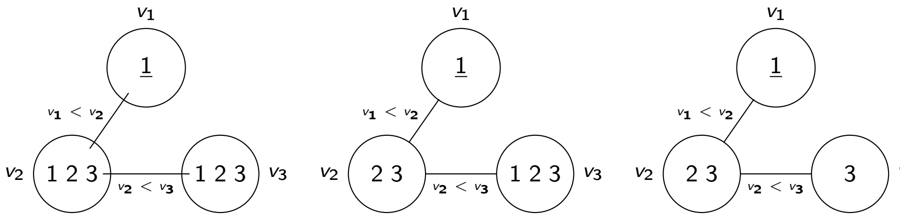


  - **Question**: On top, middle, is v_3 arc consistent relative to v_2?

Arc Consistency: Example

- **Definition 4.5 (Arc Consistency).** Let $\gamma := \langle V, D, C\rangle$ be a constraint network.
  1. A variable $u \in V$ is arc consistent relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d,d') \in C_{uv}$.
  2. The constraint network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

  The concept of arc consistency concerns both levels.
- **Example 4.6 (Arc Consistency).**

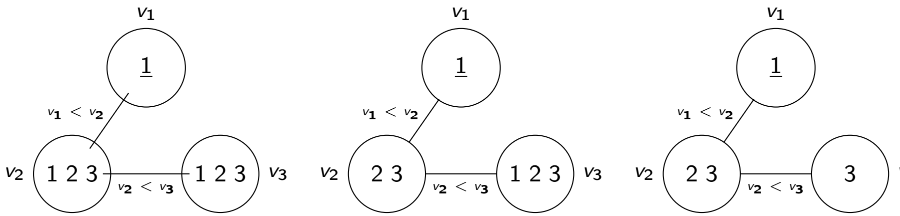


  - **Question**: On top, middle, is v_3 arc consistent relative to v_2?
  - **Answer**: No. For values 1 and 2, D_{v_2} does not have a value that works.
  - **Note**: Enforcing arc consistency for one variable may lead to further reductions on another variable!
  - **Question**: And on the right?

Arc Consistency: Example

- **Definition 4.7 (Arc Consistency).** Let $\gamma := \langle V, D, C \rangle$ be a constraint network.
  1. A variable $u \in V$ is arc consistent relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d,d') \in C_{uv}$.
  2. The constraint network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

  The concept of arc consistency concerns both levels.
- **Example 4.8 (Arc Consistency).**

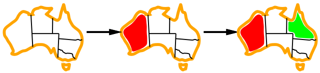


  - **Question**: On top, middle, is v_3 arc consistent relative to v_2?
  - **Answer**: No. For values 1 and 2, D_{v_2} does not have a value that works.
  - **Note**: Enforcing arc consistency for one variable may lead to further reductions on another variable!
  - **Question**: And on the right?

Arc Consistency: Example

- **Definition 4.9 (Arc Consistency).** Let $\gamma := \langle V, D, C \rangle$ be a constraint network.
  1. A variable $u \in V$ is arc consistent relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d,d') \in C_{uv}$.
  2. The constraint network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

  The concept of arc consistency concerns both levels.
- **Example 4.10.**



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

$\rightsquigarrow ?$

🟥	🟦	🟩	🟥🟦	🟥🟩🟦		🟥🟩🟦

  - **Note**: SA is not arc consistent relative to NT in 3rd row.

Enforcing Arc Consistency: General Remarks

- **Inference, version 2:** "Enforcing Arc Consistency" = removing domain values until γ is arc consistent. (Up next)
- **Note:** Assuming such an inference method $\mathrm{AC}(\gamma)$.
- **Lemma 4.11.** $\mathrm{AC}(\gamma)$ *is sound: guarantees to deliver an equivalent network.*
- *Proof sketch:* If, for $d \in D_u$, there does not exist a value $d' \in D_v$ such that $(d,d') \in C_{uv}$, then $u = d$ cannot be part of any solution.
- **Observation 4.12.** $\mathrm{AC}(\gamma)$ *subsumes forward checking:* $\mathrm{AC}(\gamma) \sqsubseteq \mathrm{ForwardChecking}(\gamma)$.
- *Proof:* Recall from slide 283 that $\gamma' \sqsubseteq \gamma$ means γ' is tighter than γ.
  1. Forward checking removes d from D_u only if there is a constraint C_{uv} such that $D_v = \{d'\}$ (i.e. when v was assigned the value d'), and $(d,d') \notin C_{uv}$.
  2. Clearly, enforcing arc consistency of u relative to v removes d from D_u as well.

Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.13 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

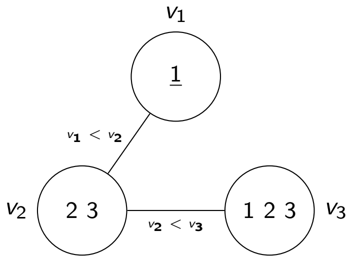
- **Lemma 4.14.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma,u,v)$ is $\mathcal{O}(d^2)$.*

Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.16 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.17.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.18.** $\mathrm{Revise}(\gamma, v_3, v_2)$

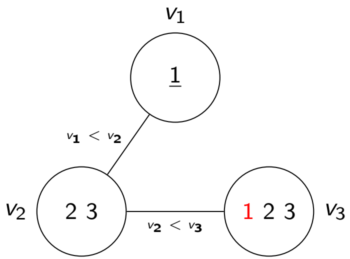


Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.19 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.20.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.21.** $\mathrm{Revise}(\gamma, v_3, v_2)$

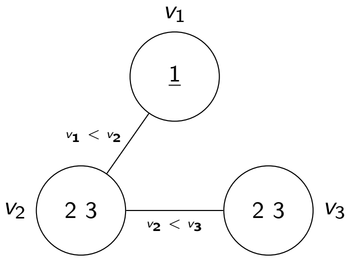


Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.22 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.23.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.24.** $\mathrm{Revise}(\gamma, v_3, v_2)$

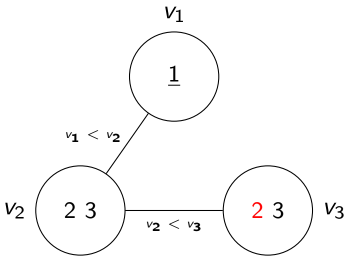


Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.25 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.26.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.27.** $\mathrm{Revise}(\gamma, v_3, v_2)$

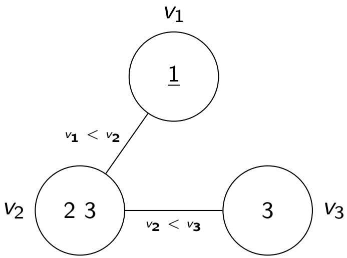


Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.28 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.29.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.30.** $\mathrm{Revise}(\gamma, v_3, v_2)$

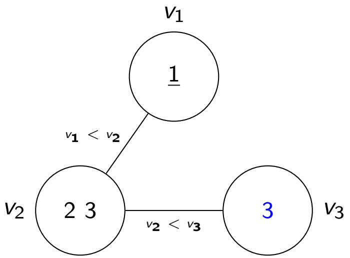


Enforcing Arc Consistency for *One* Pair of Variables

- **Definition 4.31 (Revise).** Revise is an algorithm enforcing arc consistency of u relative to v

```
function Revise($\gamma$,$u$,$v$) returns modified $\gamma$
  for each $d \in D_u$ do
    if there is no $d' \in D_v$ with $(d,d') \in C_{uv}$ then $D_u := D_u \backslash \{d\}$
  return $\gamma$
```

- **Lemma 4.32.** *If d is maximal domain size in γ and the test "$(d,d') \in C_{uv}$?" has time complexity $\mathcal{O}(1)$, then the running time of $\mathrm{Revise}(\gamma, u, v)$ is $\mathcal{O}(d^2)$.*
- **Example 4.33.** $\mathrm{Revise}(\gamma, v_3, v_2)$



AC-1: Enforcing Arc Consistency (Version 1)

- **Idea:** Apply Revise pairwise up to a fixed point.
- **Definition 4.34.** AC-1 enforces arc consistency in constraint networks:

```
function AC−1($\gamma$) returns modified $\gamma$
  repeat
    changesMade := False
    for each constraint $C_{uv}$ do
      Revise($\gamma$,$u$,$v$) /* if $D_u$ reduces, set changesMade := True */
      Revise($\gamma$,$v$,$u$) /* if $D_v$ reduces, set changesMade := True */
  until changesMade = False
  return $\gamma$
```

AC-1: Enforcing Arc Consistency (Version 1)

- **Idea:** Apply Revise pairwise up to a fixed point.
- **Definition 4.36.** AC-1 enforces arc consistency in constraint networks:

```
function AC−1($\gamma$) returns modified $\gamma$
  repeat
    changesMade := False
    for each constraint $C_{uv}$ do
      Revise($\gamma$,$u$,$v$) /∗ if $D_u$ reduces, set changesMade := True ∗/
      Revise($\gamma$,$v$,$u$) /∗ if $D_v$ reduces, set changesMade := True ∗/
  until changesMade = False
  return $\gamma$
```

- **Observation:** Obviously, this does indeed enforce arc consistency for γ.
- **Lemma 4.37.** *If γ has n variables, m constraints, and maximal domain size d, then the time complexity of $\mathrm{AC1}(\gamma)$ is $\mathcal{O}(md^2nd)$.*
- *Proof sketch:* $\mathcal{O}(md^2)$ for each inner loop, fixed point reached at the latest once all nd variable values have been removed.

AC-1: Enforcing Arc Consistency (Version 1)

- **Idea:** Apply Revise pairwise up to a fixed point.
- **Definition 4.38.** AC-1 enforces arc consistency in constraint networks:

```
function AC−1($\gamma$) returns modified $\gamma$
  repeat
    changesMade := False
    for each constraint $C_{uv}$ do
      Revise($\gamma$,$u$,$v$) /∗ if $D_u$ reduces, set changesMade := True ∗/
      Revise($\gamma$,$v$,$u$) /∗ if $D_v$ reduces, set changesMade := True ∗/
  until changesMade = False
  return $\gamma$
```

- **Observation:** Obviously, this does indeed enforce arc consistency for γ.
- **Lemma 4.39.** *If γ has n variables, m constraints, and maximal domain size d, then the time complexity of $\mathrm{AC1}(\gamma)$ is $\mathcal{O}(md^2nd)$.*
- *Proof sketch:* $\mathcal{O}(md^2)$ for each inner loop, fixed point reached at the latest once all nd variable values have been removed.
- **Problem:** There are redundant computations.
- **Question:** Do you see what these redundant computations are?

AC-1: Enforcing Arc Consistency (Version 1)

- **Idea:** Apply Revise pairwise up to a fixed point.
- **Definition 4.40.** AC-1 enforces arc consistency in constraint networks:

```
function AC−1($\gamma$) returns modified $\gamma$
  repeat
    changesMade := False
    for each constraint $C_{uv}$ do
      Revise($\gamma$,$u$,$v$) /* if $D_u$ reduces, set changesMade := True */
      Revise($\gamma$,$v$,$u$) /* if $D_v$ reduces, set changesMade := True */
  until changesMade = False
  return $\gamma$
```

- **Observation:** Obviously, this does indeed enforce arc consistency for γ.
- **Lemma 4.41.** *If γ has n variables, m constraints, and maximal domain size d, then the time complexity of $\mathrm{AC1}(\gamma)$ is $\mathcal{O}(md^2nd)$.*
- *Proof sketch:* $\mathcal{O}(md^2)$ for each inner loop, fixed point reached at the latest once all nd variable values have been removed.
- **Problem:** There are redundant computations.
- **Question:** Do you see what these redundant computations are?
- **Redundant computations:** u and v are revised even if theirdomains haven't changed since the last time.
- Better algorithm avoiding this: AC 3 (coming up)

AC-3: Enforcing Arc Consistency (Version 3)

- **Idea:** Remember the potentially inconsistent variable pairs.
- **Definition 4.42.** AC-3 optimizes AC-1 for enforcing arc consistency.

```
function AC−3($\gamma$) returns modified $\gamma$
  $M := \emptyset$
  for each constraint $C_{uv} \in C$ do
    $M := M \cup \{(u,v),(v,u)\}$
  while $M \neq \emptyset$ do
    remove any element $(u,v)$ from $M$
    Revise($\gamma, u, v$)
    if $D_u$ has changed in the call to Revise then
      for each constraint $C_{wu} \in C$ where $w \neq v$ do
        $M := M \cup \{(w,u)\}$
  return $\gamma$
```

- **Question:** $\mathrm{AC}-3(\gamma)$ enforces arc consistency because?

AC-3: Enforcing Arc Consistency (Version 3)

- **Idea:** Remember the potentially inconsistent variable pairs.
- **Definition 4.43.** AC-3 optimizes AC-1 for enforcing arc consistency.

```
function AC−3($\gamma$) returns modified $\gamma$
  $M := \emptyset$
  for each constraint $C_{uv} \in C$ do
    $M := M \cup \{(u,v),(v,u)\}$
  while $M \neq \emptyset$ do
    remove any element $(u,v)$ from $M$
    Revise($\gamma, u, v$)
    if $D_u$ has changed in the call to Revise then
      for each constraint $C_{wu} \in C$ where $w \neq v$ do
        $M := M \cup \{(w,u)\}$
  return $\gamma$
```

- **Question:** $\mathrm{AC}-3(\gamma)$ enforces arc consistency because?
- **Answer:** At any time during the while-loop, if $(u,v) \notin M$ then u is arc consistent relative to v.
- **Question:** Why only "where $w \neq v$"?

AC-3: Enforcing Arc Consistency (Version 3)

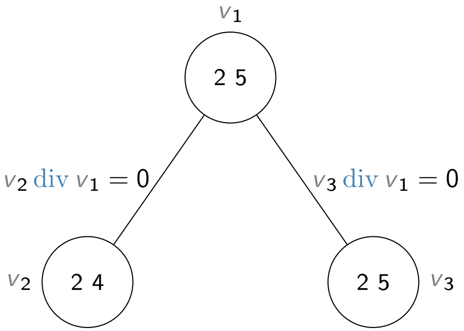
- **Idea:** Remember the potentially inconsistent variable pairs.
- **Definition 4.44.** AC-3 optimizes AC-1 for enforcing arc consistency.

```
function AC−3($\gamma$) returns modified $\gamma$
  $M := \emptyset$
  for each constraint $C_{uv} \in C$ do
    $M := M \cup \{(u,v),(v,u)\}$
  while $M \neq \emptyset$ do
    remove any element $(u,v)$ from $M$
    Revise($\gamma, u, v$)
    if $D_u$ has changed in the call to Revise then
      for each constraint $C_{wu} \in C$ where $w \neq v$ do
        $M := M \cup \{(w,u)\}$
  return $\gamma$
```

- **Question:** $\mathrm{AC}-3(\gamma)$ enforces arc consistency because?
- **Answer:** At any time during the while-loop, if $(u,v) \notin M$ then u is arc consistent relative to v.
- **Question:** Why only "where $w \neq v$"?
- **Answer:** If $w = v$ is the reason why D_u changed, then w is still arc consistent relative to u: the values just removed from D_u did not match any values from D_w anyway.

AC-3: Example

- **Example 4.45.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x

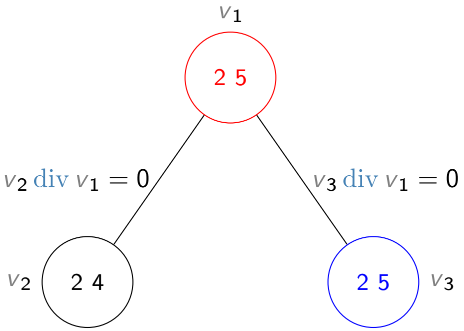


v_1
2 5
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2, v_1)
(v_1, v_2)
(v_3, v_1)
(v_1, v_3)

AC-3: Example

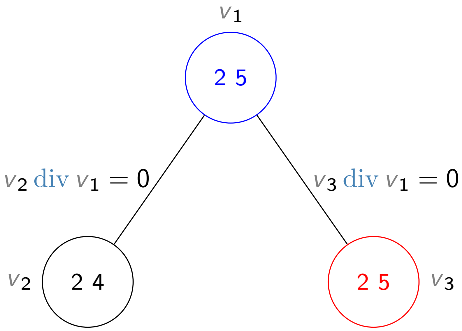
- **Example 4.46.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x


v_1
2 5
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2, v_1)
(v_1, v_2)
(v_3, v_1)
(v_1, v_3)

AC-3: Example

- **Example 4.47.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x

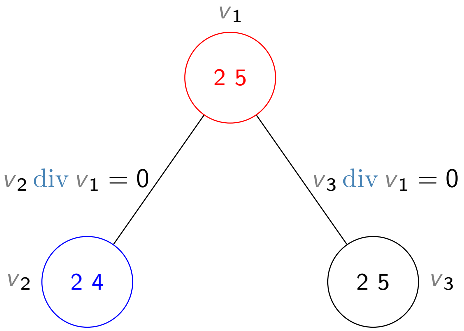


v_1
2 5
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2, v_1)
(v_1, v_2)
(v_3, v_1)

AC-3: Example

▶ **Example 4.48.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x

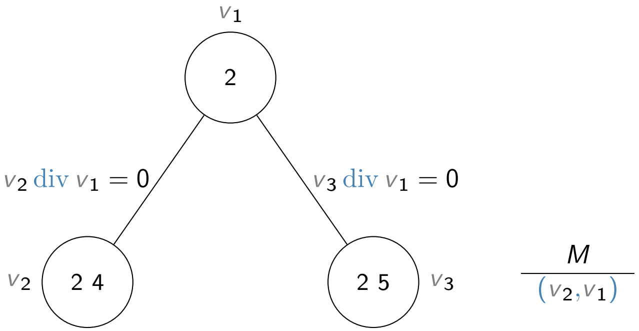


v_1
2 5
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2, v_1)
(v_1, v_2)

AC-3: Example

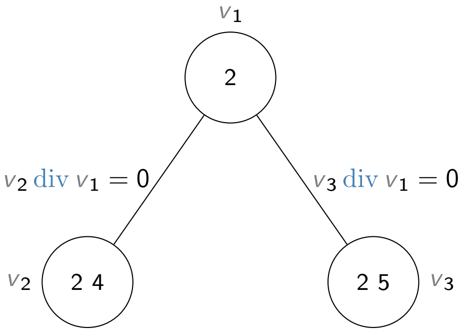
- **Example 4.49.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x



v_1
2
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3
M
(v_2, v_1)

AC-3: Example

- **Example 4.50.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x

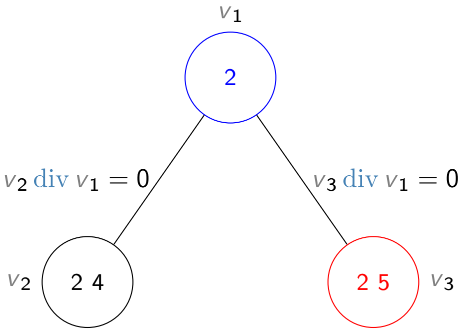


v_1
2
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2,v_1)
(v_3,v_1)

AC-3: Example

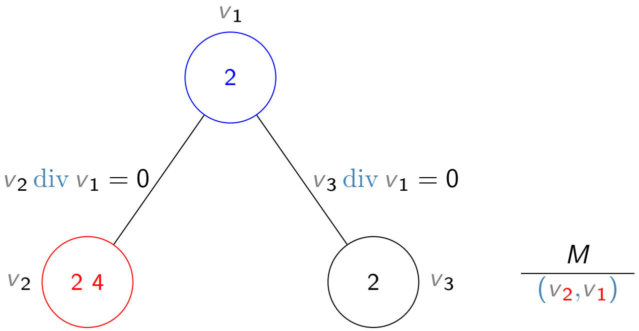
- **Example 4.51.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x


v_1
2
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2 5
v_3

M
(v_2, v_1)
(v_3, v_1)

AC-3: Example

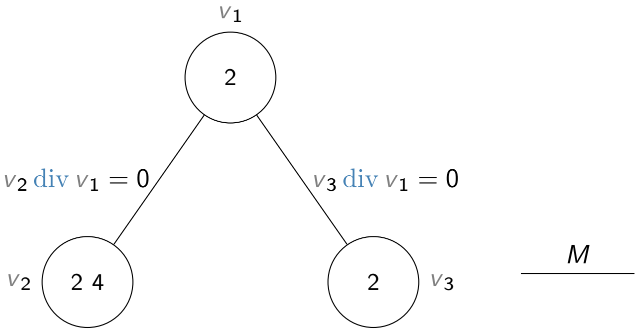
- **Example 4.52.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x



v_1
2
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2
v_3
M
(v_2, v_1)

AC-3: Example

- **Example 4.53.** $y \operatorname{div} x = 0$: y modulo x is 0, i.e., y is divisible by x



v_1
2
$v_2 \operatorname{div} v_1 = 0$
$v_3 \operatorname{div} v_1 = 0$
v_2
2 4
2
v_3
M

AC-3: Runtime

- **Theorem 4.54 (Runtime of AC-3).** *Let $\gamma := \langle V, D, C\rangle$ be a constraint network with m constraints, and maximal domain size d. Then $\mathrm{AC}-3(\gamma)$ runs in time $\mathcal{O}(md^3)$.*
- *Proof:* by counting how often Revise is called.
  1. Each call to $\mathrm{Revise}(\gamma, u, v)$ takes time $\mathcal{O}(d^2)$ so it suffices to prove that at most $\mathcal{O}(md)$ of these calls are made.
  2. The number of calls to $\mathrm{Revise}(\gamma, u, v)$ is the number of iterations of the while-loop, which is at most the number of insertions into M.
  3. Consider any constraint C_{uv}.
  4. Two variable pairs corresponding to C_{uv} are inserted in the for-loop. In the while loop, if a pair corresponding to C_{uv} is inserted into M, then
  5. beforehand the domain of either u or v was reduced, which happens at most $2d$ times.
  6. Thus we have $\mathcal{O}(d)$ insertions per constraint, and $\mathcal{O}(md)$ insertions overall, as desired.

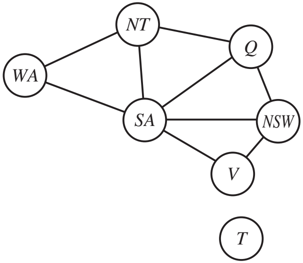
9.5 Decomposition: Constraint Graphs, and Three Simple Cases

Reminder: The Big Picture

- Say γ is a constraint network with n variables and maximal domain size d.
  - d^n total assignments must be tested in the worst case to solve γ.
- **Inference:** One method to try to avoid/ameliorate this combinatorial explosion in practice.
  - Often, from an assignment to some variables, we can easily make inferences regarding other variables.
- **Decomposition:** Another method to avoid/ameliorate this combinatorial explosion in practice.
  - Often, we can exploit the *structure* of a network to *decompose* it into smaller parts that are easier to solve.
  - **Question**: What is “structure”, and how to “decompose”?

Problem Structure

- **Idea:** Tasmania and mainland are "independent subproblems"
- **Definition 5.1.** Independent subproblems are identified as connected components of constraint graphs.
- Suppose each independent subproblem has c variables out of n total. (d is max domain size)
- Worst-case solution cost is $n \operatorname{div} c \cdot d^c$ (linear in n)
- E.g., $n = 80$, $d = 2$, $c = 20$
  - $2^{80} \widehat{=}$ 4 billion years at 10 million nodes/sec
  - $4 \cdot 2^{20} \widehat{=}$ 0.4 seconds at 10 million nodes/sec



"Decomposition" 1.0: Disconnected Constraint Graphs

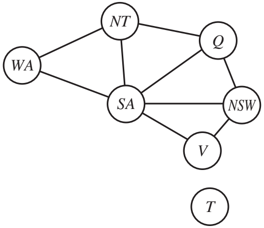
- **Theorem 5.2 (Disconnected Constraint Graphs).** *Let $\gamma := \langle V, D, C\rangle$ be a constraint network. Let a_i be a solution to each connected component γ_i of the constraint graph of γ. Then $a := \bigcup_i a_i$ is a solution to γ.*

"Decomposition" 1.0: Disconnected Constraint Graphs

- **Theorem 5.6 (Disconnected Constraint Graphs).** *Let $\gamma := \langle V, D, C \rangle$ be a constraint network. Let a_i be a solution to each connected component γ_i of the constraint graph of γ. Then $a := \bigcup_i a_i$ is a solution to γ.*
- *Proof:*
  1. a satisfies all C_{uv} where u and v are inside the same connected component.
  2. The latter is the case for all C_{uv}.
  3. If two parts of γ are not connected, then they are independent.

"Decomposition" 1.0: Disconnected Constraint Graphs

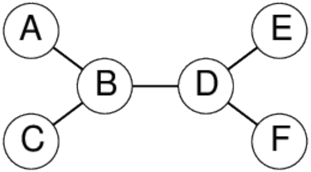
- **Theorem 5.10 (Disconnected Constraint Graphs).** *Let $\gamma := \langle V, D, C\rangle$ be a constraint network. Let a_i be a solution to each connected component γ_i of the constraint graph of γ. Then $a := \bigcup_i a_i$ is a solution to γ.*
- *Proof:*
  1. a satisfies all C_{uv} where u and v are inside the same connected component.
  2. The latter is the case for all C_{uv}.
  3. If two parts of γ are not connected, then they are independent.
- **Example 5.11.** Color Tasmania separately in Australia



"Decomposition" 1.0: Disconnected Constraint Graphs

- **Theorem 5.14 (Disconnected Constraint Graphs).** *Let $\gamma := \langle V, D, C \rangle$ be a constraint network. Let a_i be a solution to each connected component γ_i of the constraint graph of γ. Then $a := \bigcup_i a_i$ is a solution to γ.*
- *Proof:*
  1. a satisfies all C_{uv} where u and v are inside the same connected component.
  2. The latter is the case for all C_{uv}.
  3. If two parts of γ are not connected, then they are independent.
- **Example 5.15.** Color Tasmania separately in Australia
- **Example 5.16 (Doing the Numbers).**
  - γ with $n = 40$ variables, each domain size $k = 2$. Four separate connected components each of size 10.
  - Reduction of worst-case when using decomposition:
    - No decomposition: 2^{40}. With: $4 \cdot 2^{10}$. Gain: $2^{28} \approx 280.000.000$.
- **Definition 5.17.** The process of decomposing a constraint network into components is called decomposition. There are various decomposition algorithms.

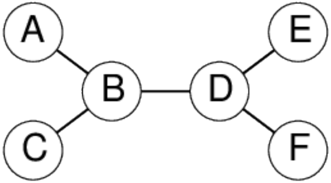
Tree-structured CSPs

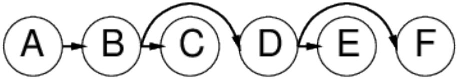


- **Definition 5.18.** We call a CSP tree-structured, iff its constraint graph is acyclic
- **Theorem 5.19.** *Tree-structured CSP can be solved in $\mathcal{O}(nd^2)$ time.*
- Compare to general CSPs, where worst case time is $\mathcal{O}(d^n)$.
- This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering





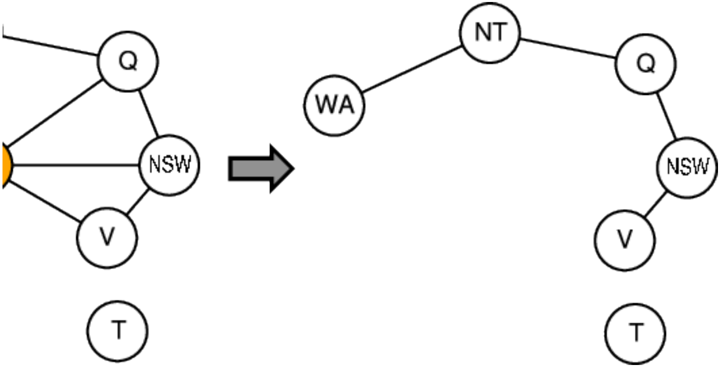
2. For j from n down to 2, apply

   RemoveInconsistent(Parent(X_j,X_j))

3. For j from 1 to n, assign X_j consistently with $Parent(X_j)$

Nearly tree-structured CSPs

- **Definition 5.20.** Conditioning: instantiate a variable, prune its neighbors' domains.
- **Example 5.21.**



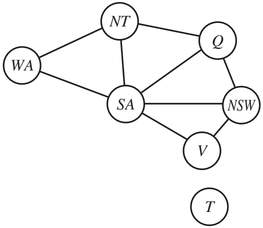
- **Definition 5.22.** Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree.
- Cutset size $c \rightsquigarrow$ running time $\mathcal{O}(d^c(n-c)d^2)$, very fast for small c.

"Decomposition" 2.0: Acyclic Constraint Graphs

- **Theorem 5.23 (Acyclic Constraint Graphs).** *Let $\gamma := \langle V, D, C \rangle$ be a constraint network with n variables and maximal domain size k, whose constraint graph is acyclic. Then we can find a solution for γ, or prove γ to be unsatisfiable, in time $\mathcal{O}(nk^2)$.*
- *Proof sketch:* See the algorithm on the next slide
- Constraint networks with acyclic constraint graphs can be solved in (low order) polynomial time.

"Decomposition" 2.0: Acyclic Constraint Graphs

- **Theorem 5.26 (Acyclic Constraint Graphs).** *Let $\gamma := \langle V, D, C \rangle$ be a constraint network with n variables and maximal domain size k, whose constraint graph is acyclic. Then we can find a solution for γ, or prove γ to be unsatisfiable, in time $\mathcal{O}(nk^2)$.*
- *Proof sketch:* See the algorithm on the next slide
- Constraint networks with acyclic constraint graphs can be solved in (low order) polynomial time.
- **Example 5.27.** Australia is not acyclic. (But see next section)



“Decomposition” 2.0: Acyclic Constraint Graphs

- **Theorem 5.29 (Acyclic Constraint Graphs).** *Let $\gamma := \langle V, D, C\rangle$ be a constraint network with n variables and maximal domain size k, whose constraint graph is acyclic. Then we can find a solution for γ, or prove γ to be unsatisfiable, in time $\mathcal{O}(nk^2)$.*
- *Proof sketch:* See the algorithm on the next slide
- Constraint networks with acyclic constraint graphs can be solved in (low order) polynomial time.
- **Example 5.30.** Australia is not acyclic. (But see next section)
- **Example 5.31 (Doing the Numbers).**
  - γ with $n = 40$ variables, each domain size $k = 2$. Acyclic constraint graph.
  - Reduction of worst-case when using decomposition:
    - No decomposition: 2^{40}.
    - With decomposition: $40 \cdot 2^2$. Gain: 2^{32}.

Acyclic Constraint Graphs: How To

- **Definition 5.32.** Algorithm $\mathrm{AcyclicCG}(\gamma)$:
  1. Obtain a (directed) tree from γ's constraint graph, picking an arbitrary variable v as the root, and directing edges outwards.[1]

[1] We assume here that γ's constraint graph is connected. If it is not, do this and the following for each component separately.

Acyclic Constraint Graphs: How To

- **Definition 5.34.** Algorithm $\mathrm{AcyclicCG}(\gamma)$:
  1. Obtain a (directed) tree from γ's constraint graph, picking an arbitrary variable v as the root, and directing edges outwards.[1]
  2. Order the variables topologically, i.e., such that each node is ordered before its children; denote that order by $v_1, \ldots, v_n$.

[1] We assume here that γ's constraint graph is connected. If it is not, do this and the following for each component separately.

Acyclic Constraint Graphs: How To

- **Definition 5.36.** Algorithm $\mathrm{AcyclicCG}(\gamma)$:
  1. Obtain a (directed) tree from γ's constraint graph, picking an arbitrary variable v as the root, and directing edges outwards.[1]
  2. Order the variables topologically, i.e., such that each node is ordered before its children; denote that order by $v_1, \ldots, v_n$.
  3. **for** $i := n, n-1, \ldots, 2$ **do**:

     3.1 $\mathrm{Revise}(\gamma, v_{parent(i)}, v_i)$.

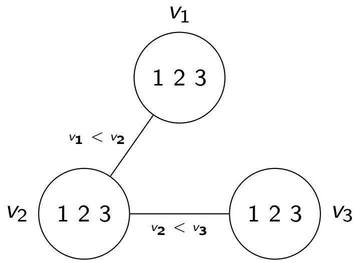
     3.2 **if** $D_{v_{parent(i)}} = \emptyset$ **then return** "inconsistent"

     Now, every variable is arc consistent relative to its children.
  4. Run BacktrackingWithInference with forward checking, using the variable order $v_1, \ldots, v_n$.
- **Lemma 5.37.** *This algorithm will find a solution without ever having to backtrack!*

[1] We assume here that γ's constraint graph is connected. If it is not, do this and the following for each component separately.

AcyclicCG(γ): Example

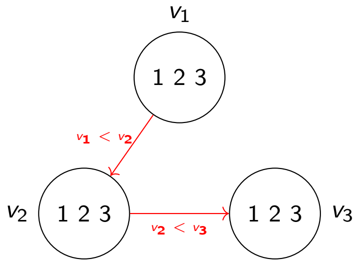
- **Example 5.38 (AcyclicCG() execution).**



Input network γ.

AcyclicCG(γ): Example

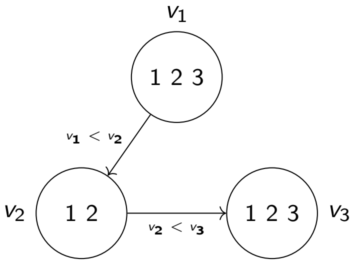
- **Example 5.39 (AcyclicCG() execution).**



Step 1: Directed tree for root v_1.
Step 2: Order v_1, v_2, v_3.

AcyclicCG(γ): Example

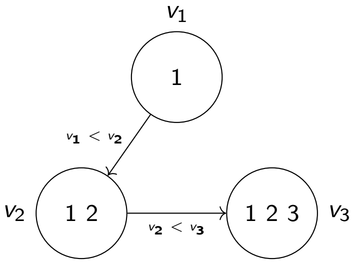
- **Example 5.40 (AcyclicCG() execution).**



Step 3: After Revise(γ, v_2, v_3).

AcyclicCG(γ): Example

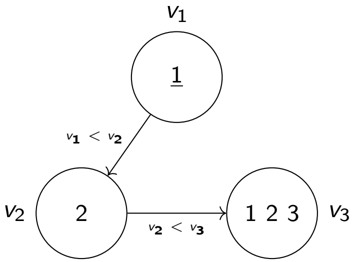
- **Example 5.41 (AcyclicCG() execution).**



Step 3: After Revise(γ, v_1, v_2).

AcyclicCG(γ): Example

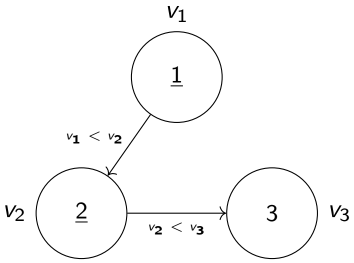
- **Example 5.42 (AcyclicCG() execution).**



Step 4: After $a(v_1) := 1$ and forward checking.

AcyclicCG(γ): Example

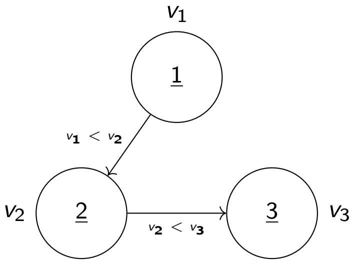
- **Example 5.43 (AcyclicCG() execution).**



Step 4: After $a(v_2) := 2$ and forward checking.

AcyclicCG(γ): Example

- **Example 5.44 (AcyclicCG() execution).**

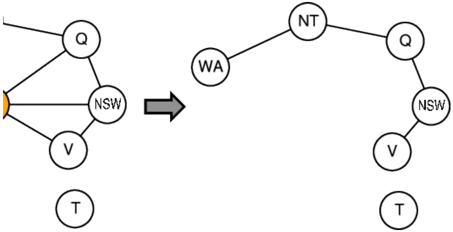


Step 4: After $a(v_3) := 3$ (and forward checking).

9.6 Cutset Conditioning

"Almost" Acyclic Constraint Graphs

- **Example 6.1 (Coloring Australia).**



- **Cutset Conditioning: Idea:**
  1. Recursive call of backtracking search on a s.t. the subgraph of the constraint graph induced by $\{v \in V \mid a(v) \text{ is undefined}\}$ is acyclic.
     - Then we can solve the remaining sub-problem with $\mathrm{AcyclicCG}()$.
  2. Choose the variable ordering so that removing the first d variables renders the constraint graph acyclic.
     - Then with (1) we won't have to search deeper than d . . . !

"Decomposition" 3.0: Cutset Conditioning

- **Definition 6.2 (Cutset).** Let $\gamma := \langle V, D, C \rangle$ be a constraint network, and $V_0 \subseteq V$. Then V_0 is a cutset for γ if the subgraph of γ's constraint graph induced by $V \setminus V_0$ is acyclic. V_0 is called optimal if its size is minimal among all cutsets for γ.
- **Definition 6.3.** The cutset conditioning algorithm, computes an optimal cutset, from γ and an existing cutset V_0.

```
function CutsetConditioning($\gamma$,$V_0$,$a$) returns a solution, or ``inconsistent''
  $\gamma'$ := a copy of $\gamma$; $\gamma'$ := ForwardChecking($\gamma'$,$a$)
  if ex. $v$ with $D_{\gamma'}{}_v = \emptyset$ then return ``inconsistent''
  if ex. $v \in V_0$ s.t. $a(v)$ is undefined then select such $v$
  else $a'$ := AcyclicCG($\gamma'$);
  if $a' \neq$ "inconsistent" then return $a \cup a'$ else return ``inconsistent''
  for each $d \in$ copy of $D_{\gamma'}{}_v$ in some order do
    $a'$ := $a \cup \{v = d\}$; $D_{\gamma'}{}_v$ := $\{d\}$;
    $a''$ := CutsetConditioning($\gamma'$,$V_0$,$a'$)
  if $a'' \neq$ "inconsistent" then return $a''$ else return ``inconsistent''
```

- Forward checking is required so that "$a \cup \mathrm{AcyclicCG}(\gamma')$" is consistent in γ.
- **Observation 6.4.** *Running time is exponential only in $\#(V_0)$, not in $\#(V)$!*
- *Remark 6.5.* Finding optimal cutsets is NP hard, but good approximations exist.

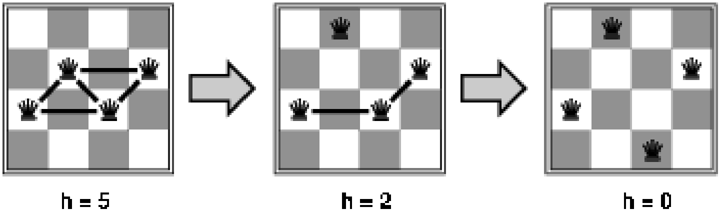
9.7 Constraint Propagation with Local Search

Iterative algorithms for CSPs

- Local search algorithms like hill climbing and simulated annealing typically work with “complete” states, i.e., all variables are assigned
- To apply to CSPs: allow states with unsatisfied constraints, actions reassign variable values.
- **Variable selection:** Randomly select any conflicted variable.
- **Value selection** by min conflicts heuristic: choose value that violates the fewest constraints i.e., hill climb with $h(n)$:=total number of violated constraints.

Example: 4-Queens

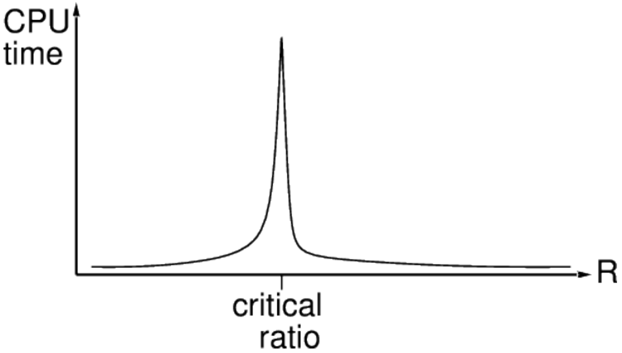
- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Actions: Move queen in column
- Goal state: No conflicts
- Heuristic: $h(n) \widehat{=}$ number of conflict



Performance of min-conflicts

- Given a random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



9.8 Conclusion & Summary

Conclusion & Summary

- γ and γ' are equivalent if they have the same solutions. γ' is tighter than γ if it is more constrained.
- Inference tightens γ without losing equivalence, during backtracking search. This reduces the amount of search needed; that benefit must be traded off against the running time overhead for making the inferences.
- Forward checking removes values conflicting with an assignment already made.
- Arc consistency removes values that do not comply with any value still available at the other end of a constraint. This subsumes forward checking.
- The constraint graph captures the dependencies between variables. Separate connected components can be solved independently. Networks with acyclic constraint graphs can be solved in low order polynomial time.
- A cutset is a subset of variables removing which renders the constraint graph acyclic. Cutset conditioning backtracks only on such a cutset, and solves a sub-problem with acyclic constraint graph at each search leaf.

Topics We Didn't Cover Here

- **Path consistency, k-consistency:** Generalizes arc consistency to size k subsets of variables. Path consistency $\widehat{=}$ 3-consistency.
- **Tree decomposition:** Instead of instantiating variables until the leaf nodes are trees, distribute the variables and constraints over sub-CSPs whose connections form a tree.
- **Backjumping:** Like backtracking search, but with ability to back up *across several levels* (to a previous variable assignment identified to be responsible for failure).
- **No-Good Learning:** Inferring additional constraints based on information gathered during backtracking search.
- **Local search:** In space of total (but not necessarily consistent) assignments. (E.g., 8 queens in **??**)
- **Tractable CSP:** Classes of CSPs that can be solved in P.
- **Global Constraints:** Constraints over many/all variables, with associated specialized inference methods.
- **Constraint Optimization Problems (COP):** Utility function over solutions, need an optimal one.

References I

[FD14] Zohar Feldman and Carmel Domshlak. “Simple Regret Optimization in Online Planning for Markov Decision Processes”. In: *Journal of Artificial Intelligence Research* 51 (2014), pp. 165–205.

[KS06] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. LNCS. Springer-Verlag, 2006, pp. 282–293.

[Met+53] N. Metropolis et al. “Equations of state calculations by fast computing machines”. In: *Journal of Chemical Physics* 21 (1953), pp. 1087–1091.

[Min] *Minion - Constraint Modelling*. System Web page at http://constraintmodelling.org/minion/. URL: http://constraintmodelling.org/minion/.

[Pól73] George Pólya. *How to Solve it. A New Aspect of Mathematical Method*. Princeton University Press, 1973.

References II

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearso n Education, 2003. ISBN: 0137903952.

[Sil+16] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[Wal75] David Waltz. "Understanding Line Drawings of Scenes with Shadows". In: *The Psychology of Computer Vision*. Ed. by P. H. Winston. McGraw-Hill, 1975, pp. 1–19.