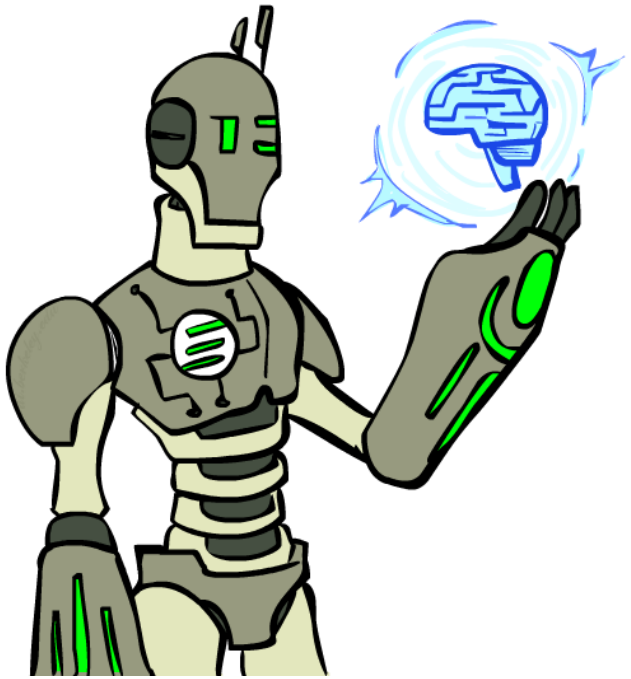
Chapter 2

Solving Problems by Searching

Chapter Outline

- Problem Solving in AI
- Example Problems
- Basic Search Strategies
  - Uninformed Search
  - Informed Search
- Beyond Classical Search
  - Local Search
  - Swarm Intelligence
- Adversarial Search
  - Games
  - Alpha-Beta Pruning
  - Stochastic Games



Problem Solving in AI

What is Problem Solving?

- **Solution**
  - Is a sequence of actions to reach the goal.
- **Process**
  - Look for the sequence of actions, which is called **search**.
- **Problem formulation**
  - Given a goal, decide what actions and states to consider.
- **Why search**
  - Some NP-complete or NP-hard problems can only be solved by search.
- **Problem-solving agent**
  - Is a kind of goal-based agent to solve problems through search.

Simple Problem Solving Agents

function PROBLEM-SOLVING-AGENT(*percept*) **returns** *action*

static: *seq*, an action sequence, initially empty
static: *state*, a description of the current world state
static: *goal*, a goal, initially null
static: *problem*, a problem formulation

state <- UPDATE-STATE(*state*, *percept*)
if *seq* is empty **then**
goal <- FORMULATE-GOAL(*state*)
problem <- FORMULATE-PROBLEM(*state*,*goal*)
seq <- SEARCH(*problem*)

action <- RECOMMENDATION(*seq*, *state*)
seq <- REMAINDER(*seq*)
return *action*

Formulate a goal for a state

Formulate a problem for a state and a goal

After formulation, the agent calls a search procedure to solve it (Finds a sequence of actions)

Update the sequence of actions

Returns an action for percept *p* in state *state*

Related Terms

- **State space**

  The state space of the problem is formally defined by: Initial state, actions, and transition model

- **Graph**

  State space forms a graph, in which nodes are states, and links are actions

- **Path**

  A path in the state space is a sequence of states connected by a sequence of actions

What's a problem?

- **Initial state**: Where the agent starts.
- **Actions**: What the agent can do (in a given state).
- **Transition model**: A mapping from actions to states.
  - A successor is a state reachable from a given state by one action.
  - The state space is the set of all states reachable from the initial state by any sequence of actions.
  - A path is a sequence of states connected by actions.
- **Goal test**: A test whether a given state is a goal state.
- **Path cost**: An assignment of cost to a path.
  - Typically the sum of step costs for actions on a path.

Example Problems:

Romania

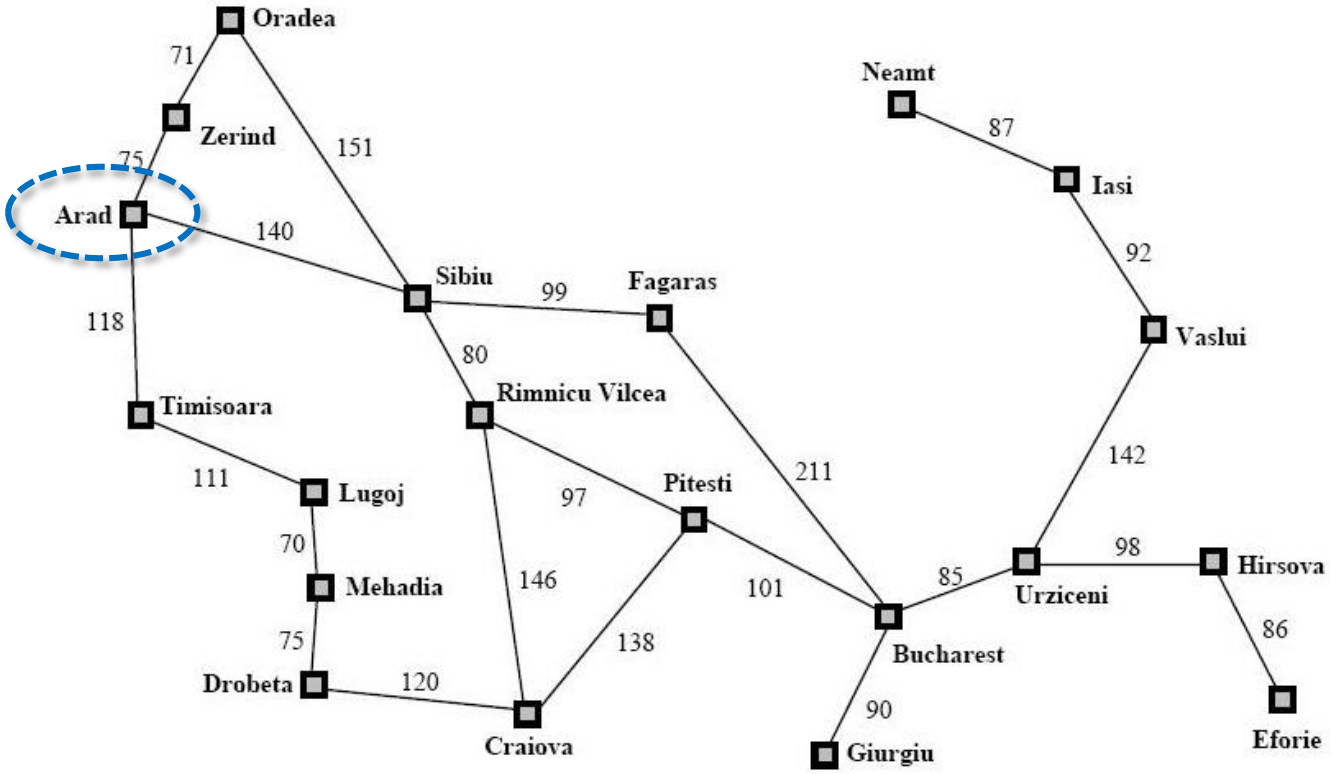
Romania

Say you're on a driving holiday in Romania, and you're currently in Arad. Your flight leaves tomorrow from Bucharest. How would you formulate this problem?

1. Initial state: You're in Arad.
2. Actions: Driving from your current city on the roads leading to other cities.
3. Transition model: Where you end up having chosen one action in a given city.
4. Goal test: Are you in Bucharest?
5. Step cost: Time or distance.

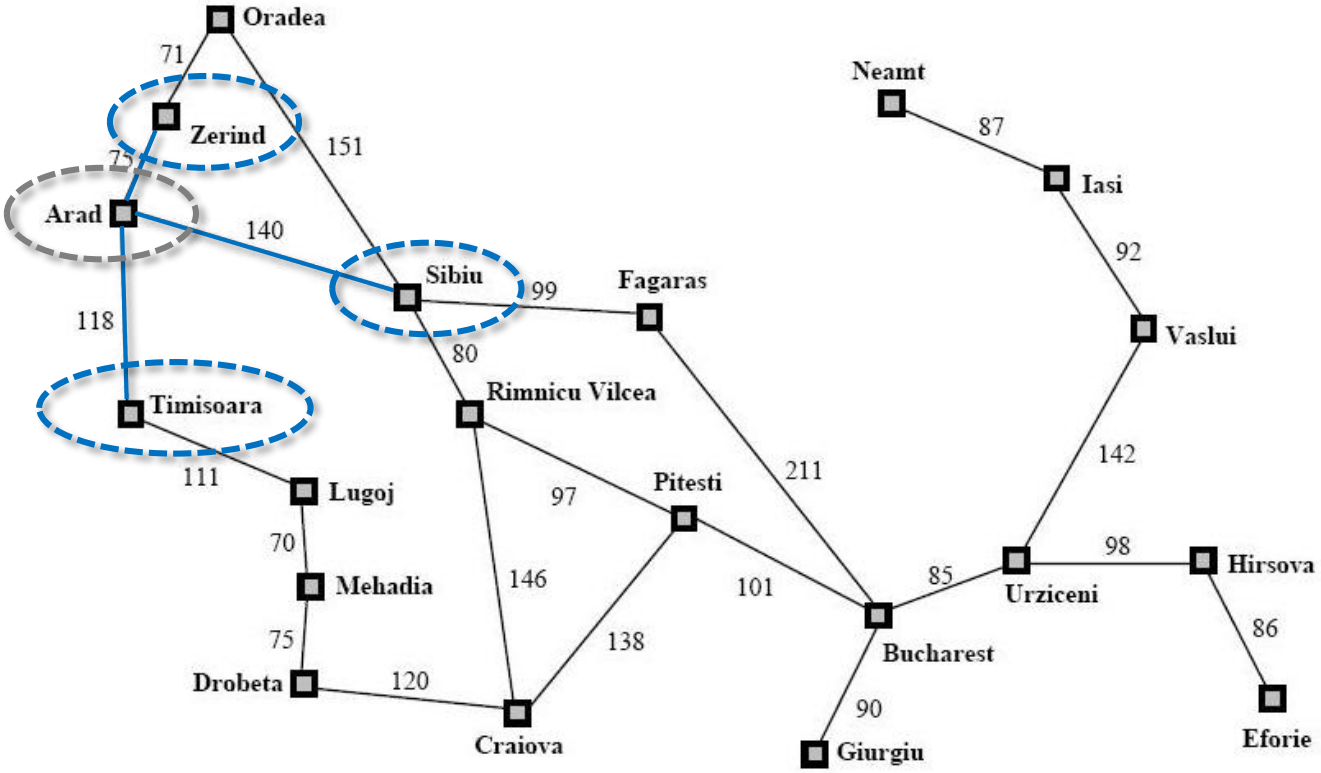
Romania

- 1. **Initial state**
  - The agent starts in.
  - E.g., the initial state for the agent in Arab: ***In*(*Arab*)**



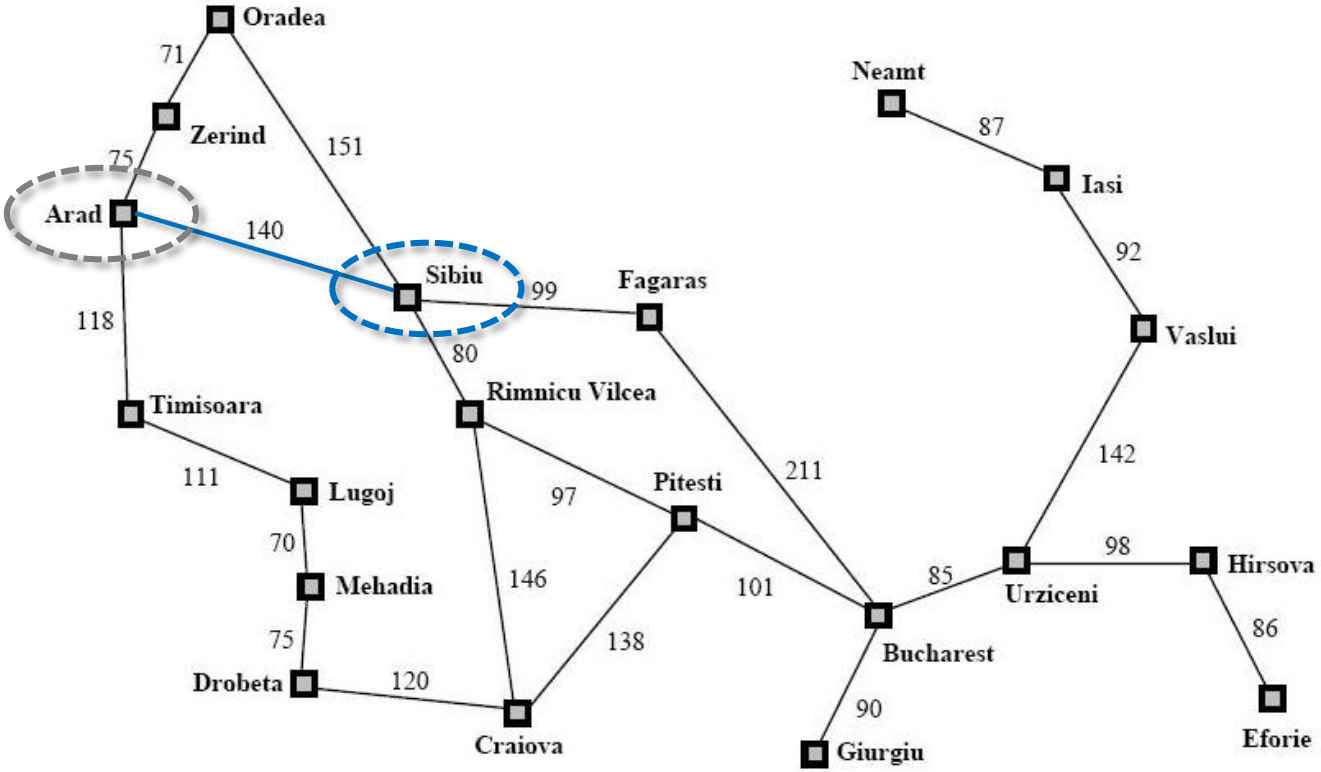
Romania

- 2. **Actions**
  - A description of the possible actions available to the agent.
  - *ACTION*(*s*) returns the actions that can be executed in *s*.
  - E.g., ***{Go(Zerind), Go(Sibiu), Go(Timisoara)}***



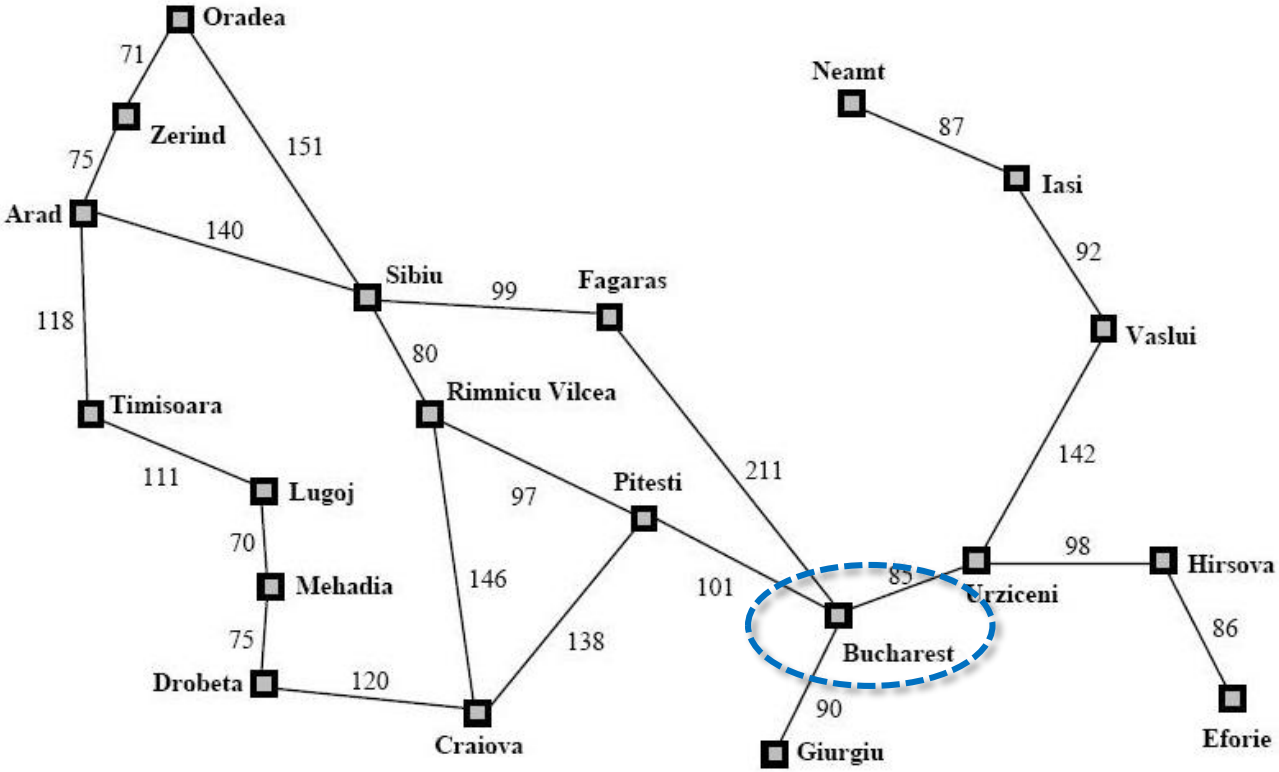
Romania

- 3. **Transition model**
  - A description of what each action does.
  - *RESULT*(*s*,*a*) returns the state from doing action *a* in *s*.
  - E.g., ***RESULT(In(Arad),Go(Sibiu)) = In(Sibiu)***



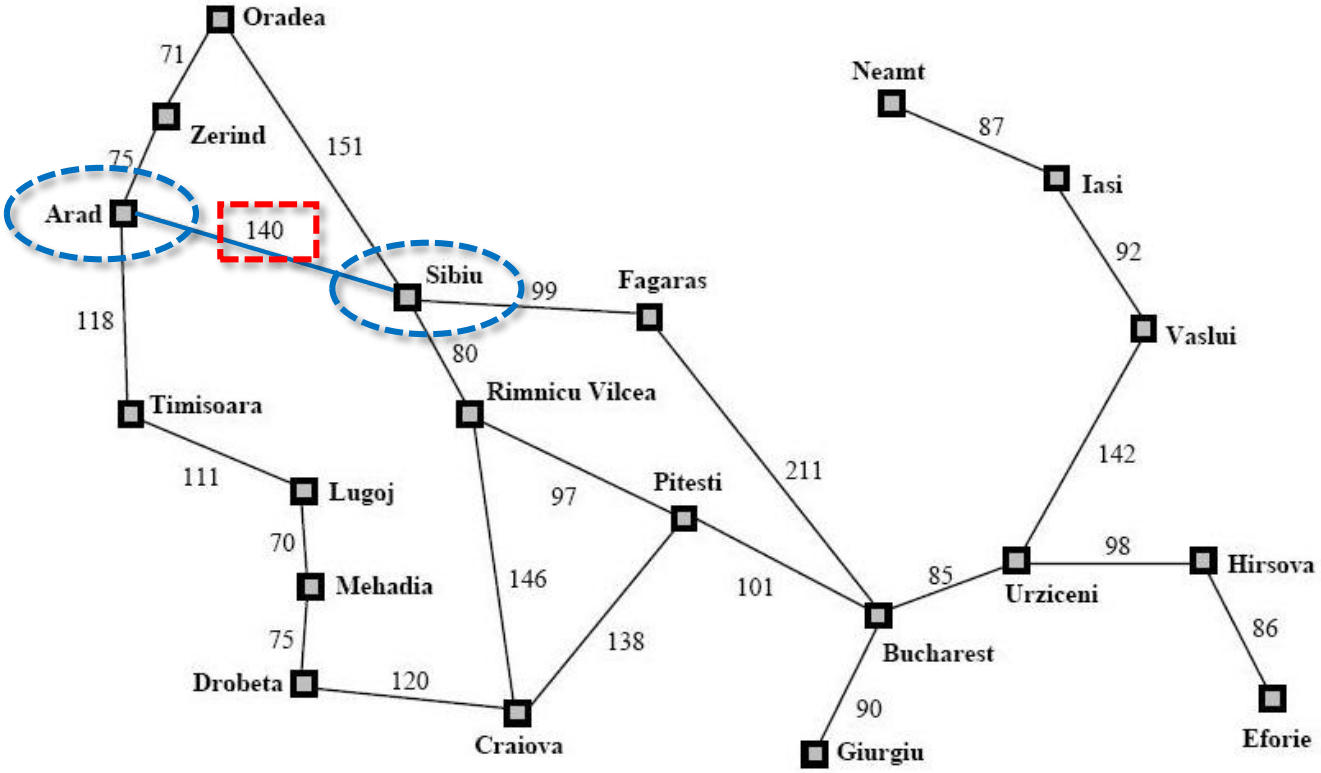
Romania

- 4. **Goal test**
  - To determine whether a given state is a goal state.
  - E.g., the agent's goal in Bucharest is the singleton set: ***{In(Bucharest)}***



Romania

- 4. **Path cost**
  - To assign a numeric cost to each path.
  - Cost of taking action *a* in state *s* to reach state *s'*: ***c(s,a,s')***
  - E.g., ***c(In(Arad), Go(Sibiu), In(Sibiu)) = 140***

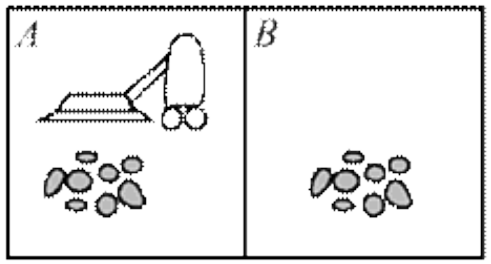


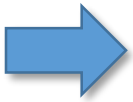
Example Problems:

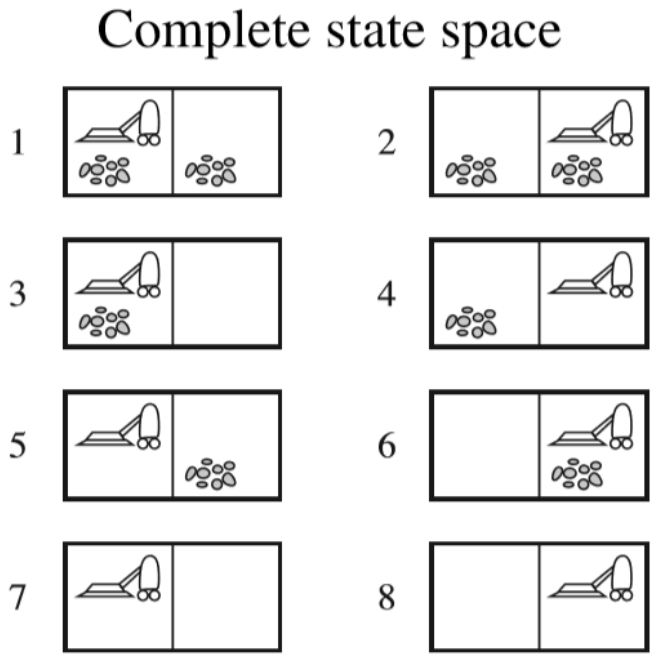
Vacuum-cleaner World

Vacuum-cleaner World

- This particular world has just two locations: **squares A** and **B**.
- The vacuum agent perceives **which square it is in** and **whether there is dirt in the square**.
- It can choose to **move left**, **move right**, **suck up the dirt**, or **do nothing**.
- One very simple agent function is the following: **if the current square is dirty, then suck; otherwise, move to the other square.**





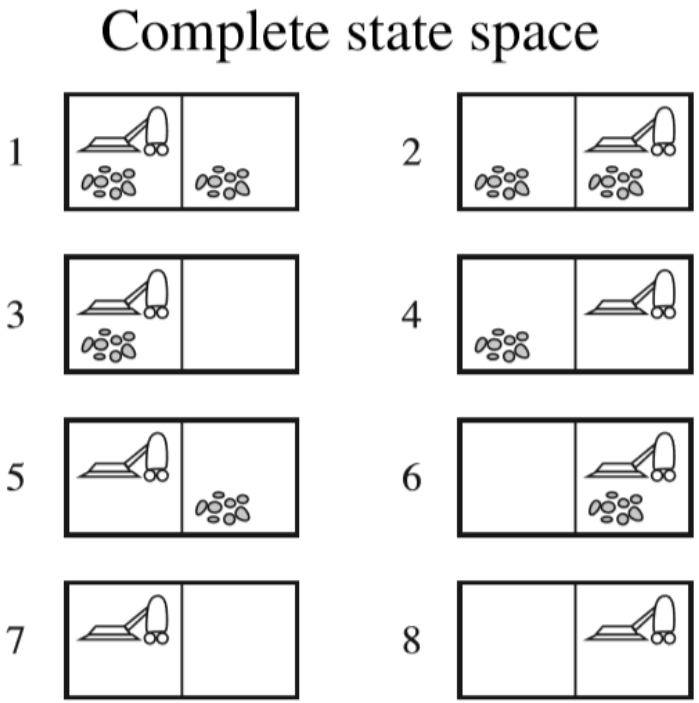


Formulate The Problem

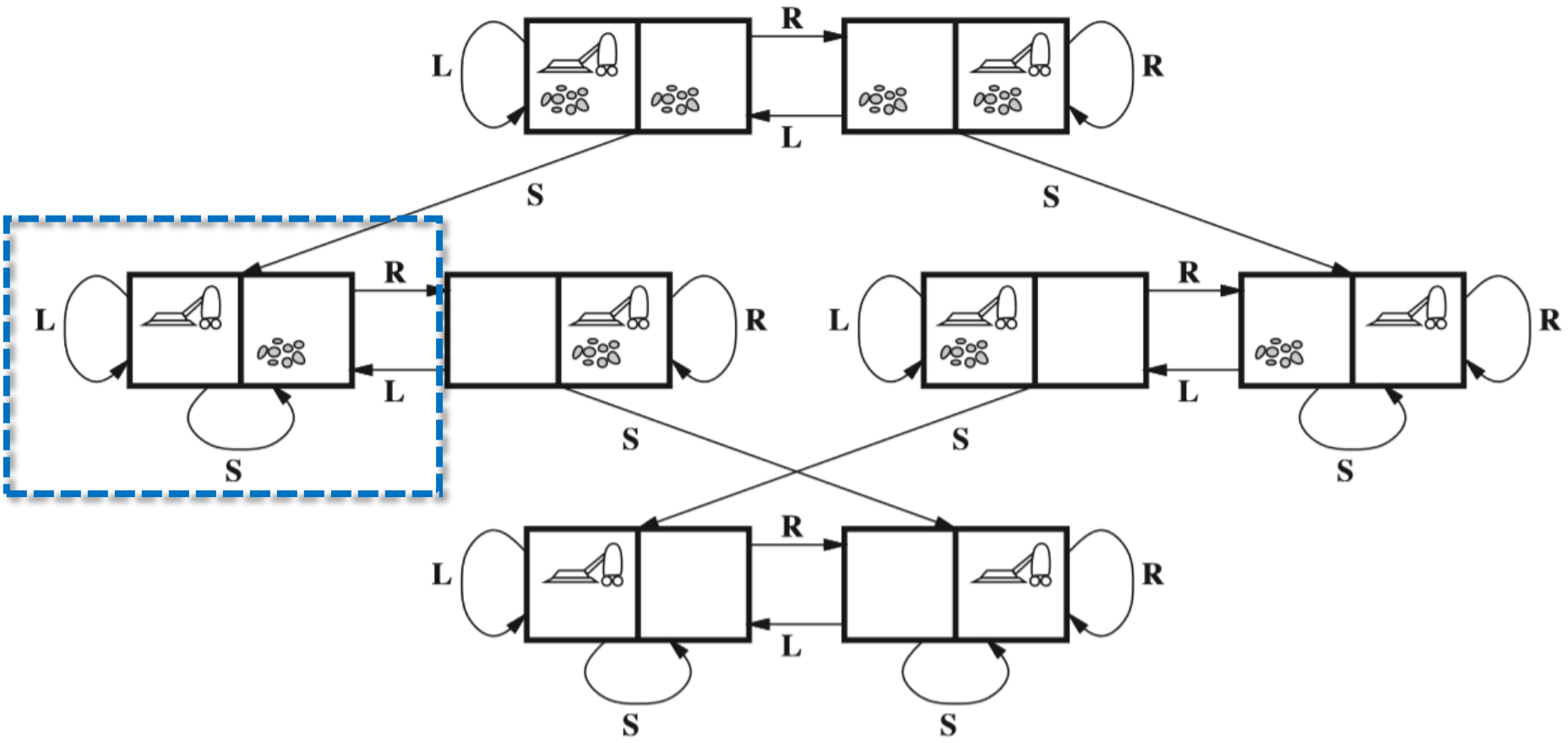
- **States** - Location of robot, presence of dirt
  - $2 \times 2^2 = 8$ total states
- **Initial State** - Any
- **Actions**
  - LEFT, RIGHT - Move the robot (if possible)
  - SUCK - Remove dirt from current location (if present)
  - All actions are available in all states
- **Transition Model**: All actions have their expected effect.
- **Goal Test** - All locations are clean.
- **Path Cost** - one/step.

Finding solutions

- Single-state: Start in 5.
- What's the solution?
- Solution: **[Right, Suck]**



Graph Representation



Finding solutions

- What if the problem were sensorless? (Where do we start? What do the actions do?)
- Solution: **[Right, Suck, Left, Suck]**

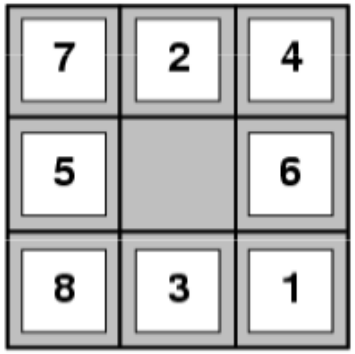
- What if the problem were contingent?
  - **Non-deterministic actions**: Action Suck may fail, or may dirty a clean carpet.
  - **Partially observable**: Dirty or clean, at current location (but no others).
- Solution: **[Right; if dirty, then Suck]**

Example Problems:

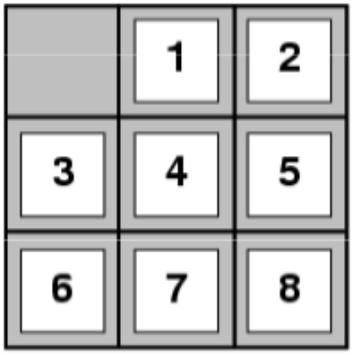
8-Puzzle

8-Puzzle

- **States**: Location of the tiles
- **Initial States**: Any
- **Actions**: move blank left, right, up, down
- **Transition Model**: returns the resulting state
- **Goal Test**: Goal state given
- **Path Cost**: 1 per move

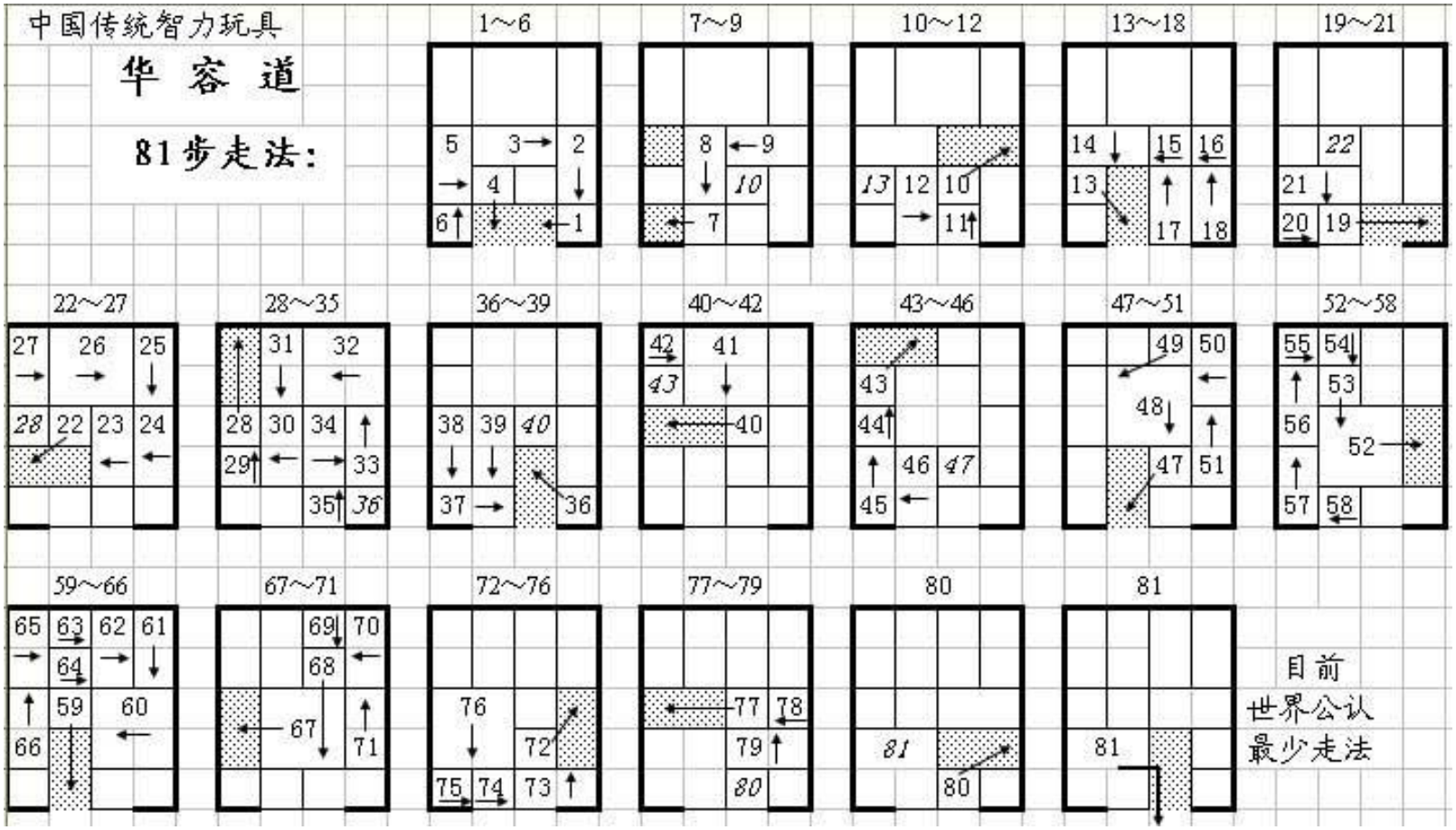


Start State



Goal State

Huarong Dao

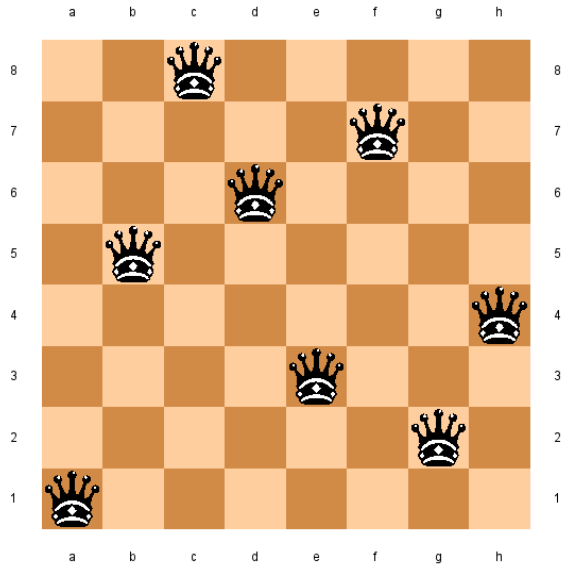

中国传统智力玩具
华 容 道
81步走法:
1~6
7~9
10~12
13~18
19~21
22~27
28~35
36~39
40~42
43~46
47~51
52~58
59~66
67~71
72~76
77~79
80
81
目前
世界公认
最少走法

Example Problems:

8-Queens Problem

8-Queens Problem

- The goal is to place 8 queens on a chessboard such that no queen attacks any other.
  - A queen attacks any piece in the same row, column or diagonal.
- Two types of formulation:
  1. **Incremental formulation**: starts with an empty state, then each action adds a queen to change the state
  2. **Complete-state formulation**: starts with all 8 queens on the board, and moves them around



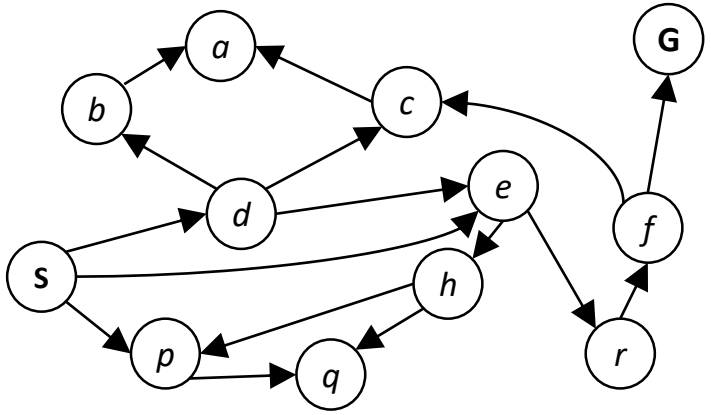
8-Queens Problem

- The incremental formulation:
  - **States**: any arrangement of 1 to 8 queens on the board
  - **Initial State**: No queens on the board
  - **Actions**: Add a queen to any empty square
  - **Transition Model**: Returns the board with a queen added to the specified square
  - **Goal Test**: 8 queens are on the board, none attacked
  - **Path Cost**: The number of steps (1 per step)

Basic Search Strategies

State Space Graphs

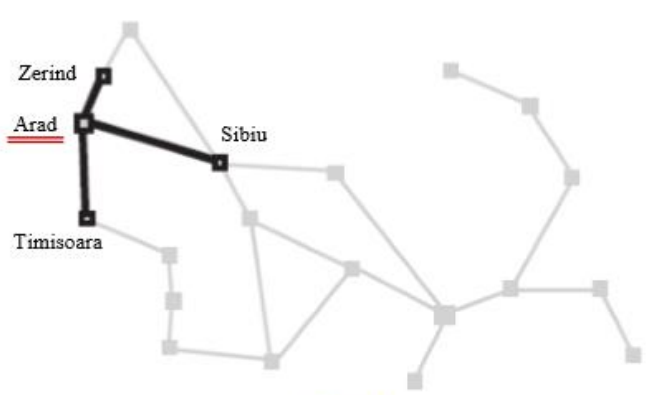
- **State space graph**: A mathematical representation of a search problem
  - **Nodes** are (abstracted) world configurations
  - **Arcs** represent successors (action results)
  - The **goal test** is a set of goal nodes (maybe only one)
- In a search graph, **each state occurs only once**!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



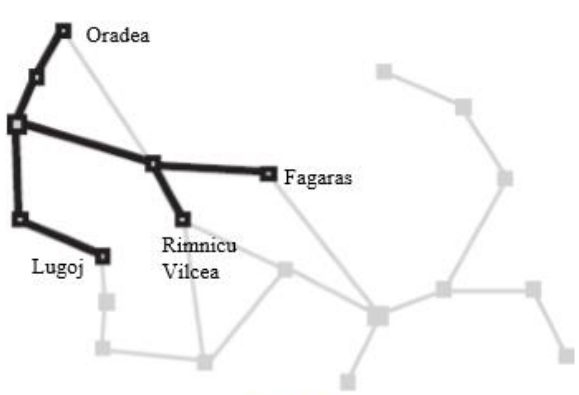
Tiny search graph for a tiny search problem

Shortest Path by Graph Search

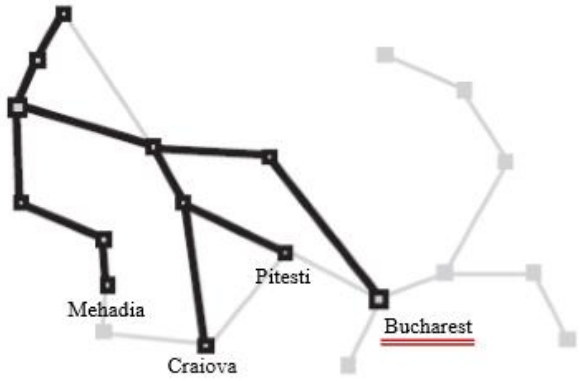
- A sequence of search paths generated by a **graph search**.
- Each path is extended at each stage by one step.
- Notice that at 3rd stage, the northernmost city (Oradea) has become a dead end.



Stage 1



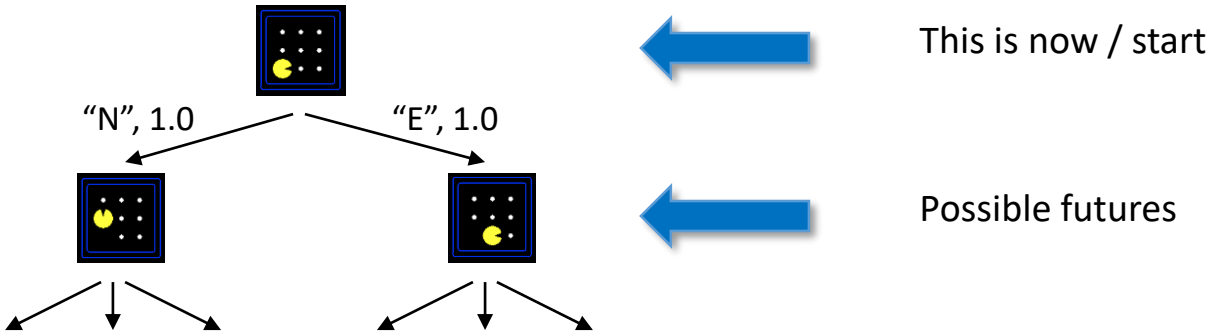
Stage 2



Stage 3

Search Trees

- A **search tree**:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children nodes correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree



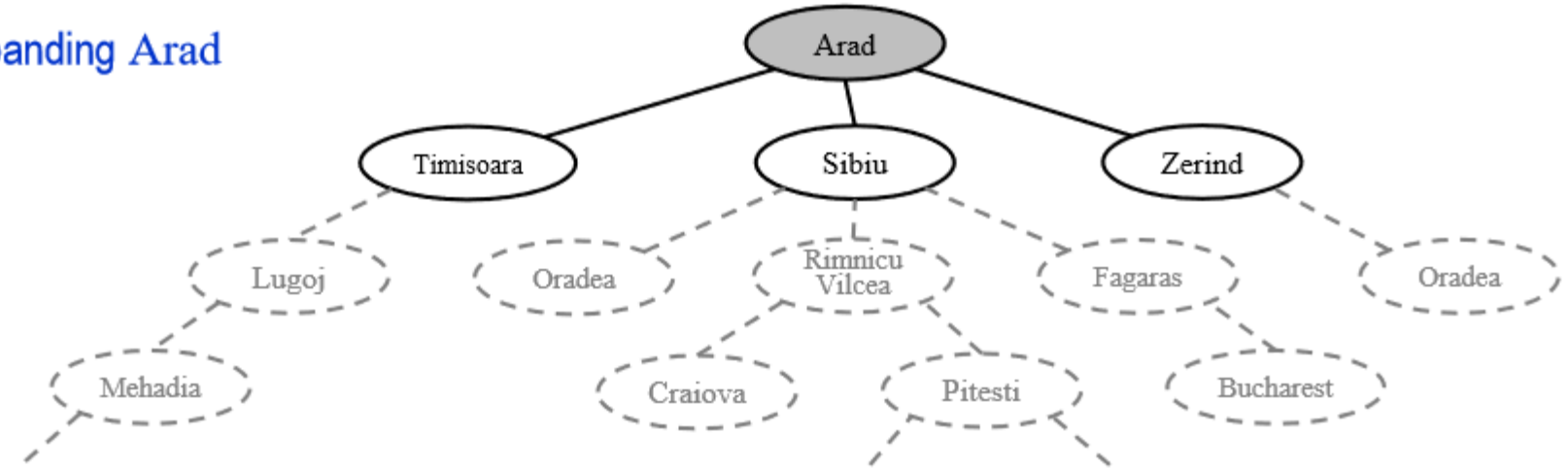
Shortest Path by Tree Search

- Use **search trees** to find a route ***Arad*** to ***Bucharest***.
  - Shaded: the nodes that have been expanded.
  - Outlined: the nodes that have been generated but not yet expanded.
  - Faint dashed lines: the nodes that have not been generated.

Shortest Path by Tree Search

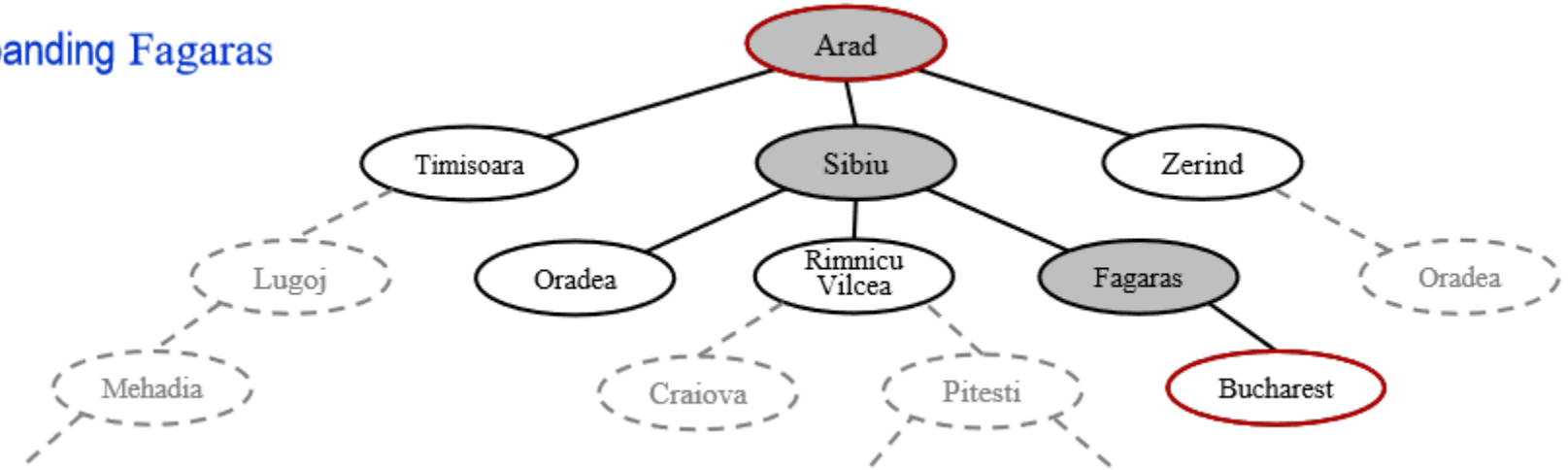
(a) The initial state

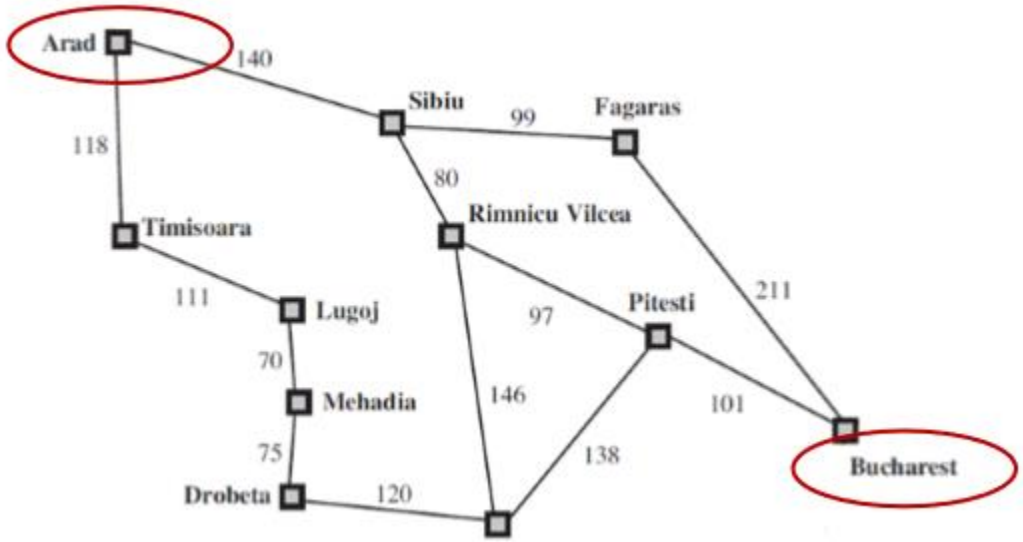
(b) After expanding Arad



Shortest Path by Tree Search

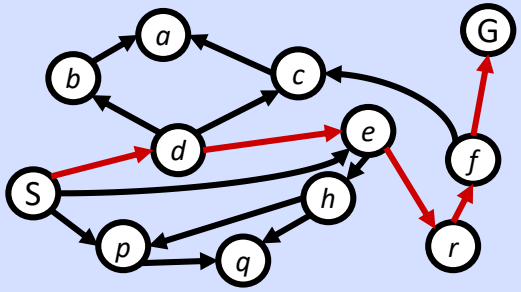
(d) After expanding Fagaras





State Space Graphs vs. Search Trees

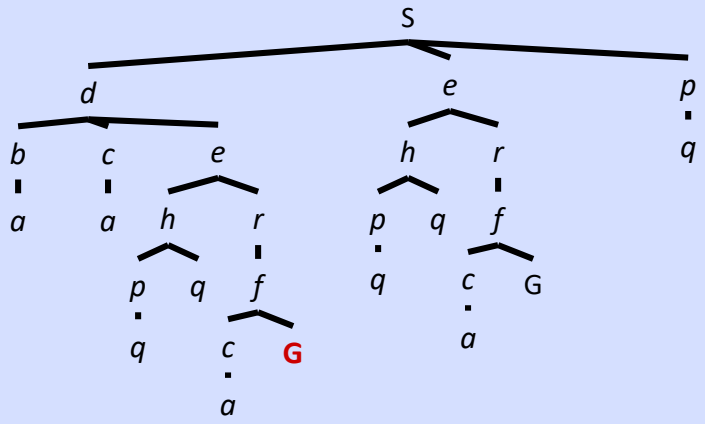
State Space Graph



Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.

Search Tree



General Tree Search Algorithm

function TREE-SEARCH(*problem*) **returns** a *solution*, or *failure*

initialize the *frontier* using the *initial state* of *problem*

loop do

if the *frontier* is empty **then return** *failure*

choose a *leaf node* and remove it from the *frontier*

if the node contains a *goal state* **then return** the corresponding *solution*

expand the chosen *node*, adding the resulting *nodes* to the *frontier*

The *frontier* (also known as open list): an data structure, to store the set of all *leaf nodes*

The process of expanding nodes on the *frontier* continues until either a solution is found or there are no more *states* to expand.

General Graph Search Algorithm

function GRAPH-SEARCH (*problem*) **returns** a *solution*, or *failure*

initialize the *frontier* using the *initial state* of *problem*

initialize the *explored* to be empty

loop do

if the *frontier* is empty **then return** *failure*

choose a *leaf node* and remove it from the *frontier*

if the node contains a *goal state* **then return** the corresponding *solution*

add the *node* to the *explored*

expand the chosen *node*, adding the resulting *nodes* to the *frontier*

only if not in the *frontier* or *explored*

The nodes in the *explored* or the *frontier* can be discarded.

The *explored* (aka closed list) is an data structure to remember every expanded node.

Uninformed Search

What is Uninformed Search?

- The **uninformed search** is also called blind search.
- The term (uninformed, or blind) means that the search strategies **have no additional information about states beyond that provided in the problem definition**.
- All they can do is to
  1. **generate successors**
  2. **distinguish a goal state from a non-goal state**

Uninformed Search

- All search strategies are distinguished by **the order in which nodes are expanded**:
  - **Breadth-first Search**
  - **Depth-first Search**
  - **Uniform-cost Search**
  - **Iterative Deepening Search**

Strategy Evaluation

- **Completeness**: Does it always find a solution if one exists?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed?
- **Optimality**: Does it always find the optimal solution?

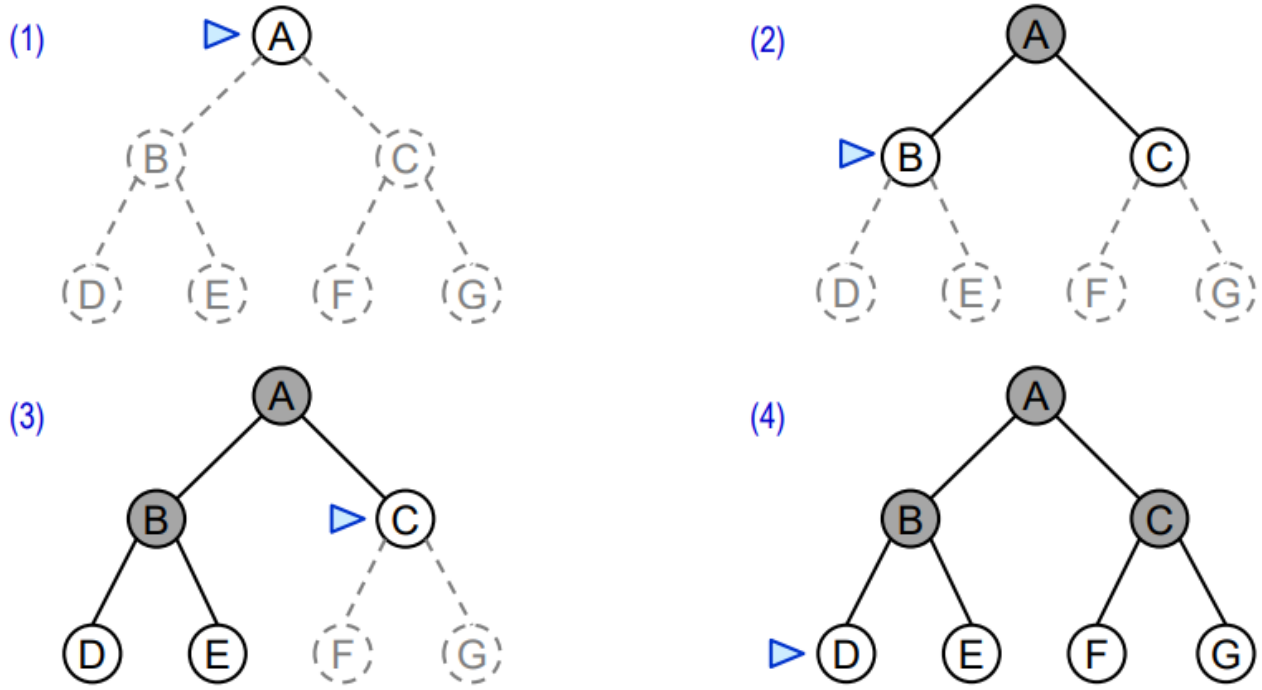
- **Time complexity** and **space complexity** are measured in following terms:
  - *b* -- maximum branching factor of the search tree.
  - *d* -- depth of the shallowest solution.
  - *m* -- maximum depth of the search tree.

Uninformed Search:

Breadth-first Search

Breadth-first Search

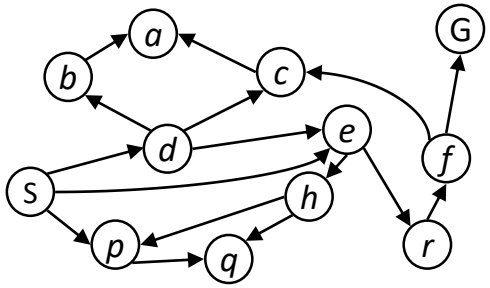
- **Search Strategy**:
  - Expand shallowest unexpanded node.
- **Implementation**:
  - Use FIFO (First-In First-Out) queue, i.e., new successors go at end.

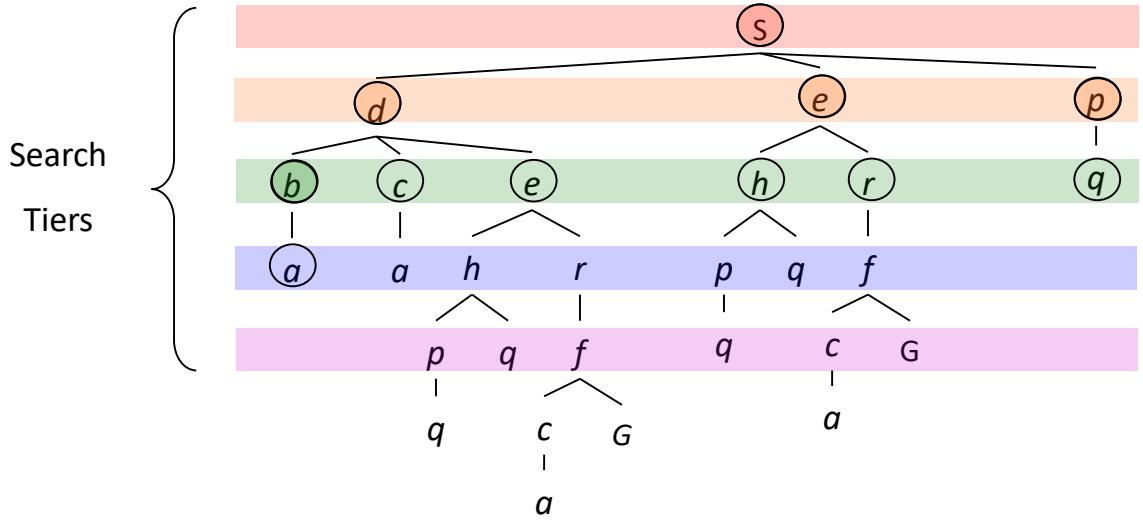


Breadth-First Search

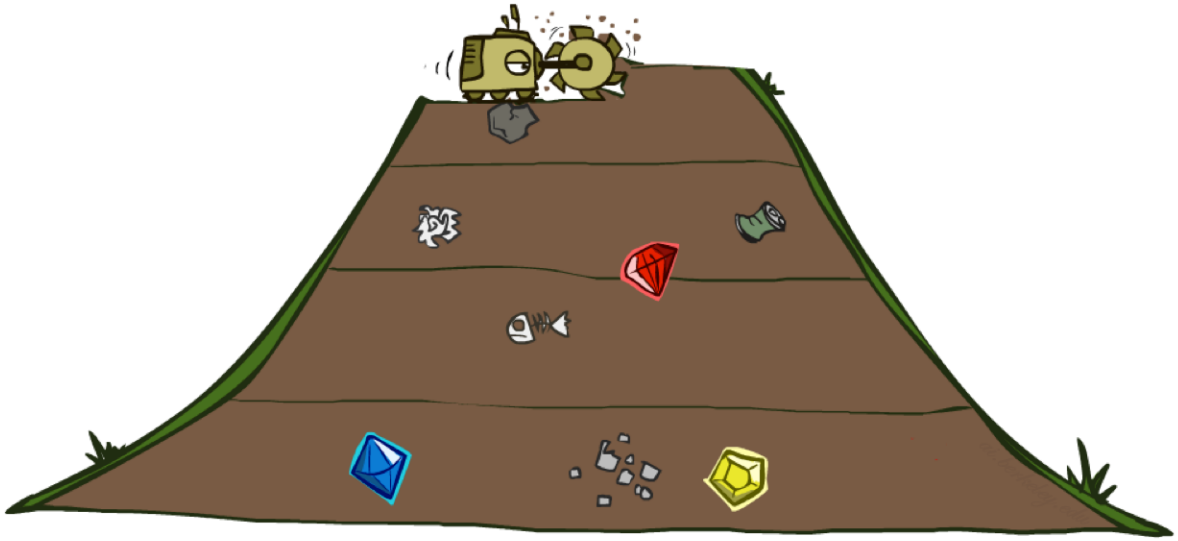
Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue



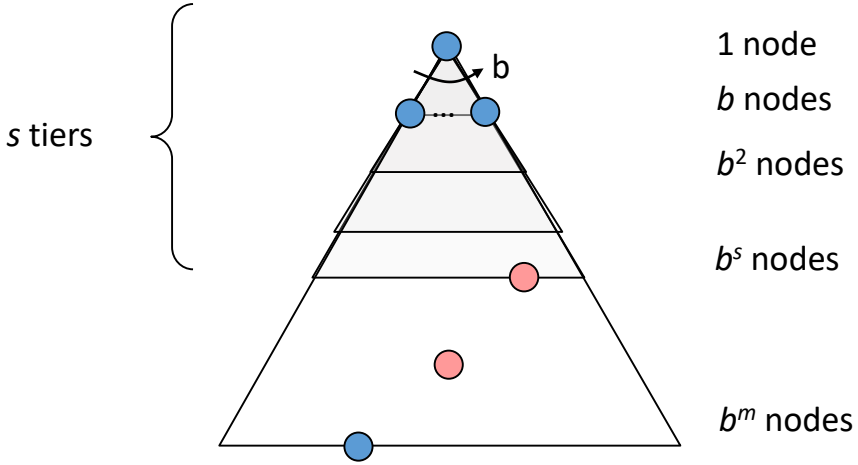


Breadth-First Search



Breadth-First Search

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be *s*
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



Breadth-first Search

- Properties of Breadth-first Search
  - Time complexity: $1 + b + b^2 + \ldots. b^m = O(b^m)$
  - Space complexity: $O(b^m)$
  - where
    - b -- the branching factor
    - s -- the depth of the shallowest solution
    - m -- the depth of the tree space

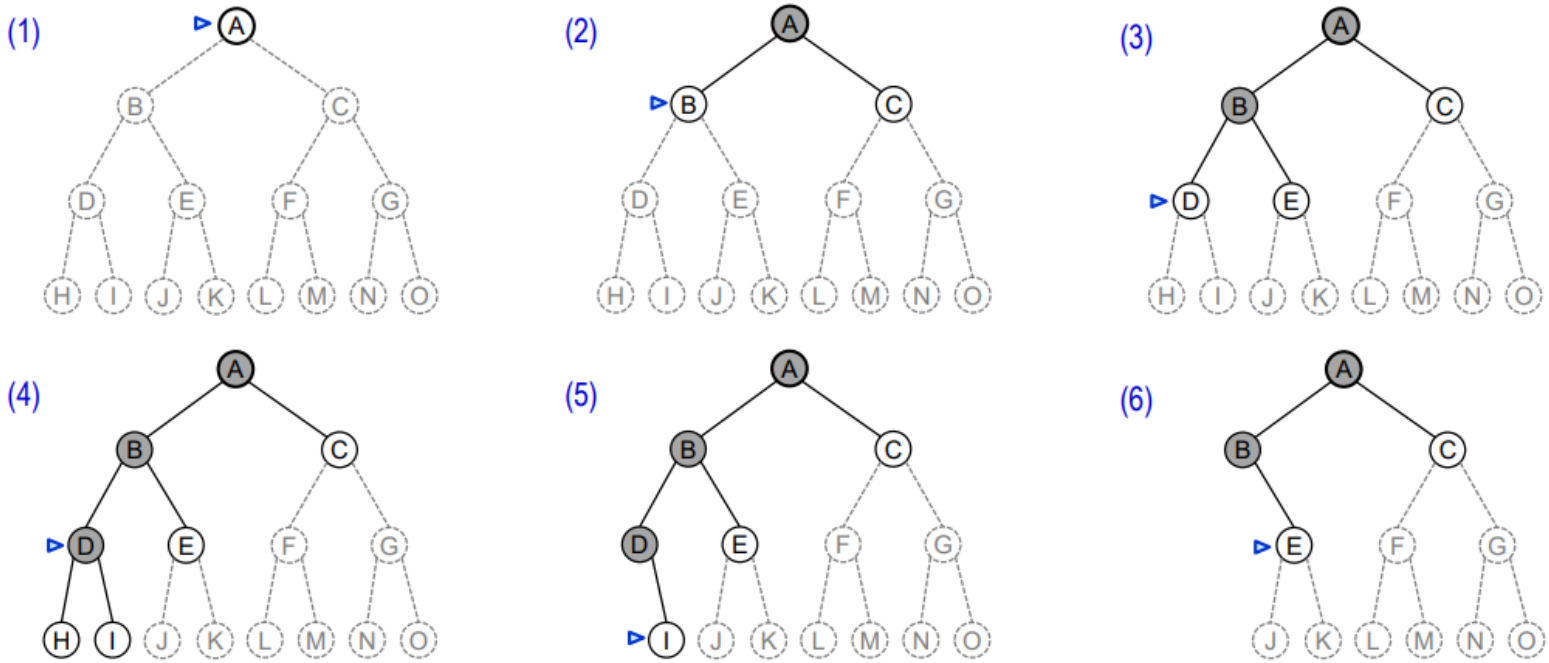
- **Memory requirements** are a bigger problem, execution time is still a major factor.
- Breadth-first search cannot solve exponential complexity problems but small branching factor.

Uninformed Search:

Depth-first Search

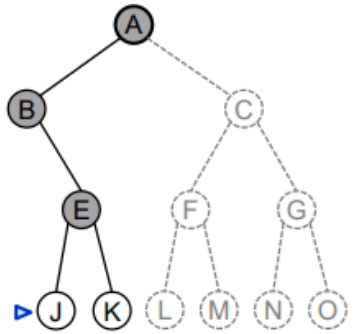
Depth-first Search

- **Search Strategy**
  - Expand deepest unexpanded node.
  - Note: breadth-first-search expands shallowest unexpanded node.
- **Implementation**
  - Use LIFO queue, put successors at front
  - Note: breadth-first-search uses a FIFO queue

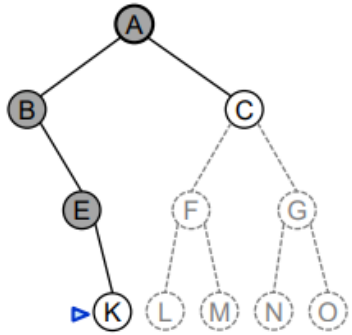


Depth-first Search

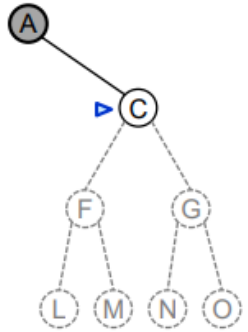
(7)



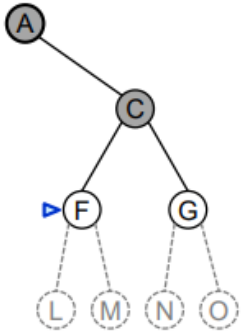
(8)



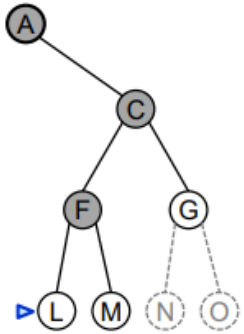
(9)



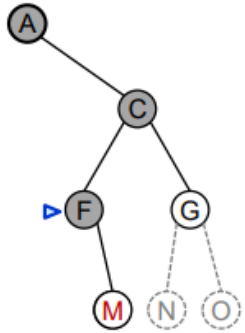
(10)



(11)



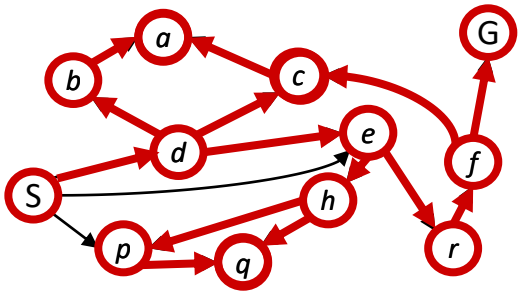
(12)

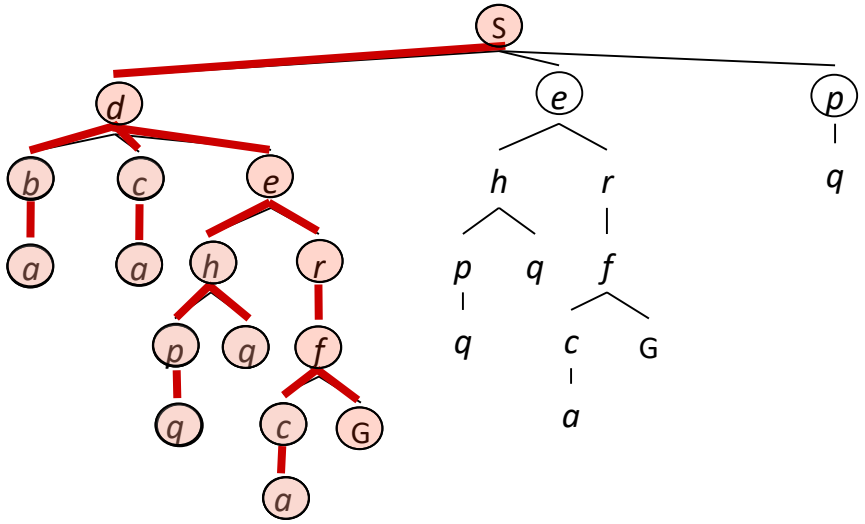


Depth-first Search

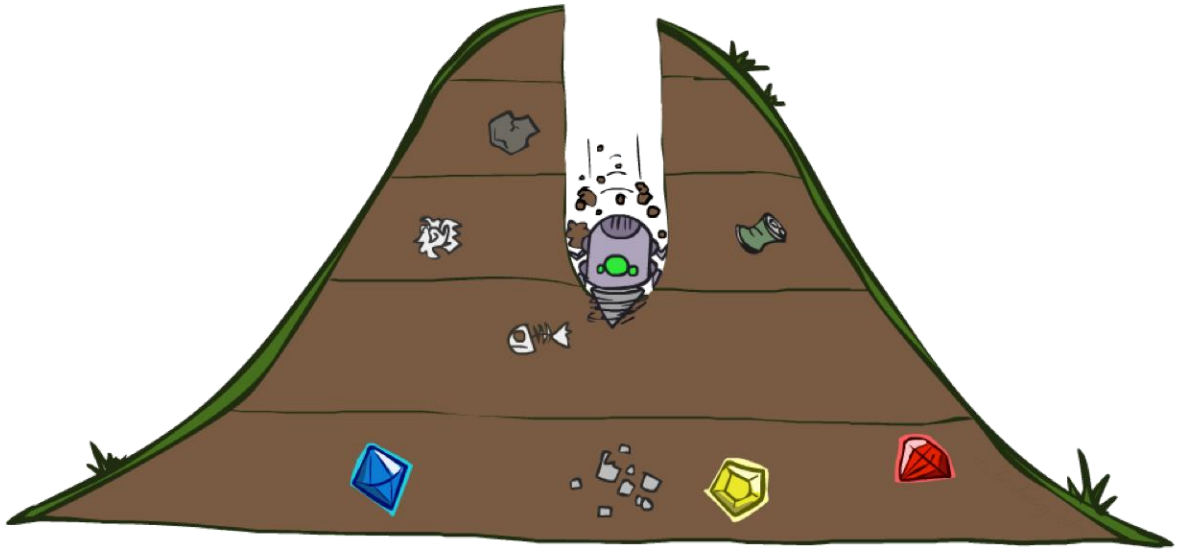
Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack





Depth-first Search

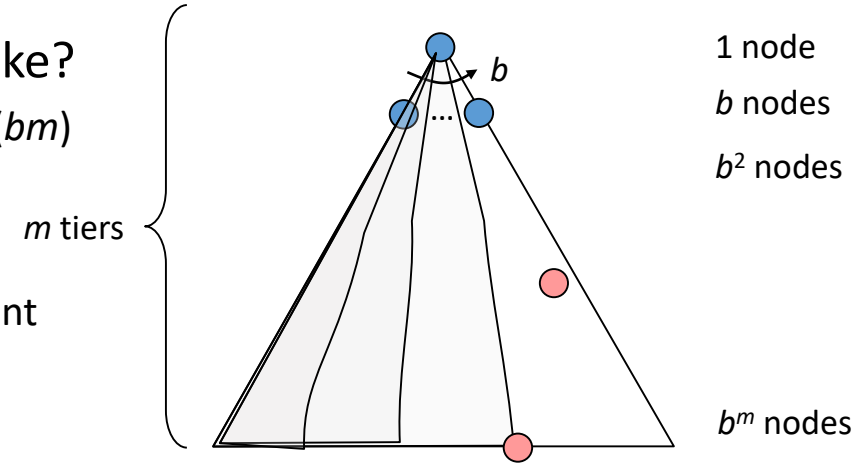


Depth-First Search

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost

b -- the branching factor

m -- the maximum depth of any node

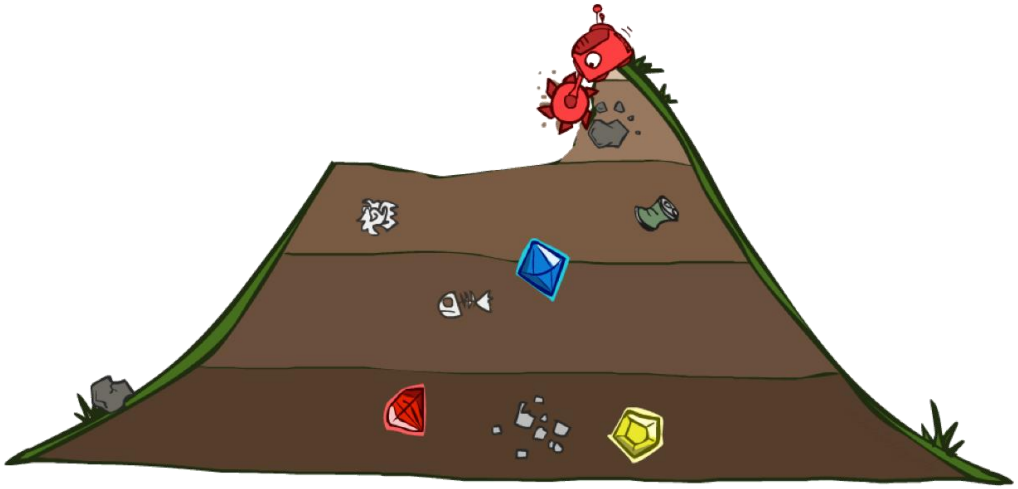


Uninformed Search:

Uniform-cost Search

Uniform Cost Search

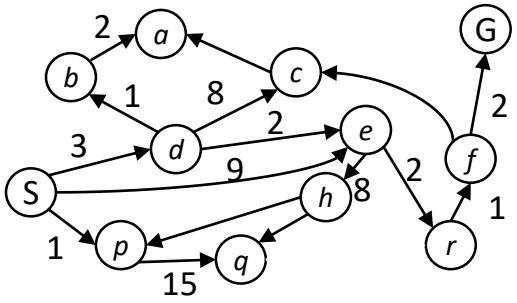
- **Search Strategy**
  - Expand lowest-cost unexpanded node.
- **Implementation**
  - Queue ordered by path cost, lowest first.

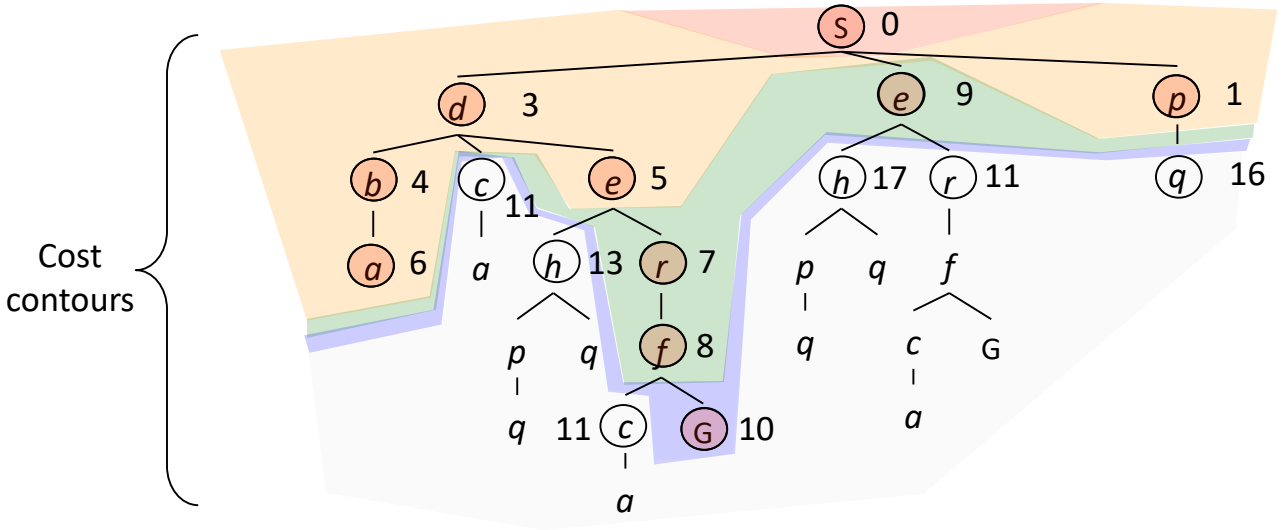


Uniform Cost Search

Strategy: expand a cheapest node first:

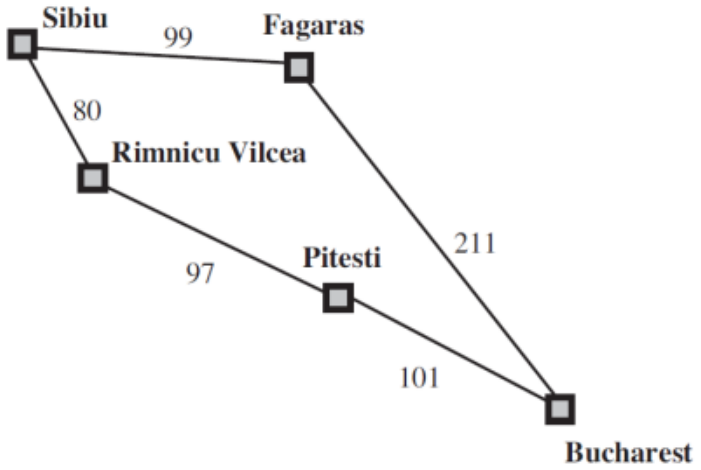
Fringe is a priority queue (priority: cumulative cost)





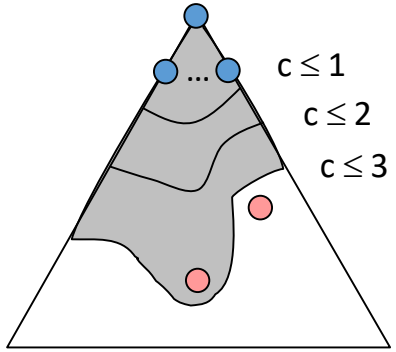
Uniform Cost Search

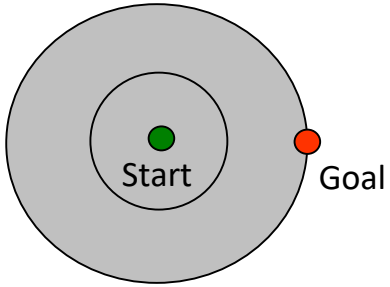
- Example: From Sibiu to Bucharest
  - From Sibiu to Bucharest, least-cost node, Rimnicu Vilcea, is expanded, next, adding Pitesti with cost 80 + 97 = 177.
  - The least-cost node is now Fagaras, and adding goal node Bucharest with cost 99 + 211 = 310.
  - Choosing Pitesti and adding a second path to Bucharest with cost 177 + 101 = 278.
  - This new path is better, so **lowest path cost** is 278.



Uniform Cost Issues

- Remember: UCS explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location
- We'll fix that soon!





Uninformed Search:

Depth-limited Search & Iterative Deepening Search

Depth-limited Search

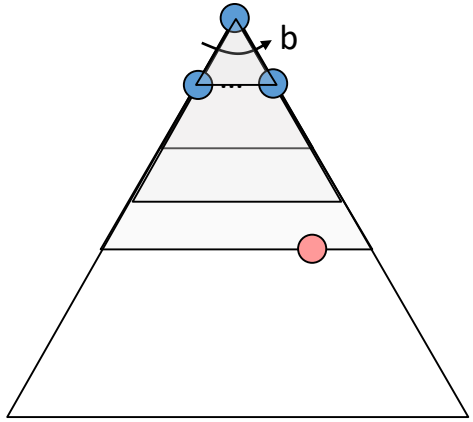
- The failure of depth-first search will be happened **if in infinite state spaces**.
- This problem can be solved with a **predetermined depth limit** *l*, i.e. nodes at depth *l* are treated as if they have no successors.
- Disadvantages
  - It will introduces an additional source of **incompleteness** if we choose $l < d$, that is, the shallowest goal is beyond the depth limit.
  - Depth-limited search will also be **non-optimal** if we choose $l > d$.

Iterative Deepening Search

- It **combines the benefits** of depth-first and breadth-first search, running repeatedly with gradually increasing depth limits until the goal is found.
- It visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.

Iterative Deepening Search

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!



Informed Search

What is Informed Search

- Also known as **Heuristic Search**.
- The strategies use problem-specific knowledge **beyond the definition of the problem itself,** so that can find solutions more efficiently than can an uninformed strategy.
- The general approaches use one or both of following functions:
  - An **evaluation function**, denoted $f(n)$, used to select a node for expansion.
  - A **heuristic function**, denoted $h(n)$, as a component of f.
- Strategies:
  - **Best-first Search**
  - **Greedy Search**
  - **A* Search**
  - **Iterative Deepening A* Search**

Best-first Search

- **Search Strategy**
  - A node is selected for expansion based on an evaluation function, $f(n)$.
  - Most best-first algorithms also include a heuristic function, $h(n)$.
- **Implementation**
  - Identical to that for uniform-cost search.
  - **However** best-first search uses of $f(n)$ instead of $g(n)$ to order the priority queue.

- **Heuristic function**
  - $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
- **Special cases**
  - **Greedy Search**
  - **A* search**

Informed Search:

Greedy Search

Greedy Search

- **Search Strategy**
  - Try to expand the node that is closest to the goal.
- **Evaluation function**

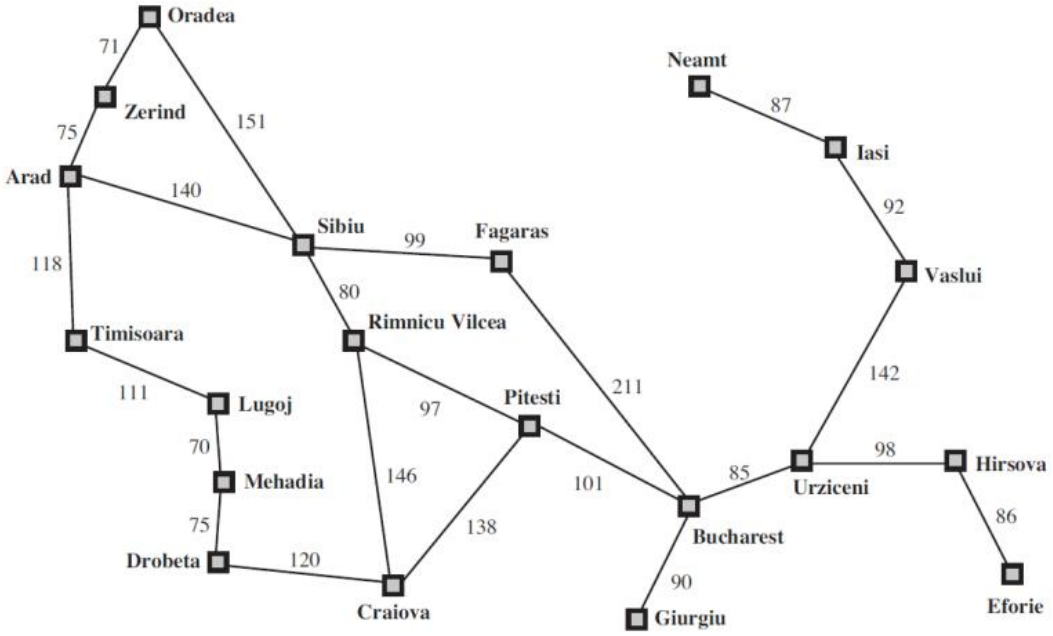
$$f(n) = h(n)$$

  - It evaluates nodes by using just the heuristic function.
  - $h(n)$ -- estimated cost from n to the closest goal.

- Why call “greedy”
  - at each step it tries to get as close to the goal as it can.

Greedy Search

- Example: from Arad to Bucharest
  - h_{SLD} : straight-line distance



h_{SLD} Values

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

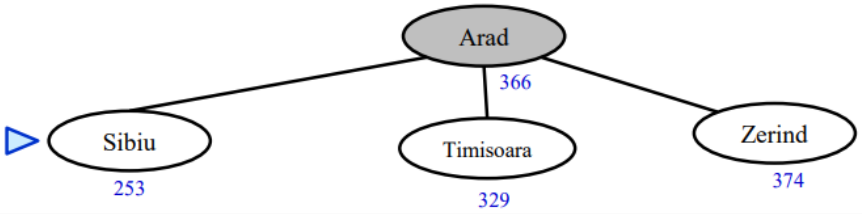
Notice: the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and therefore is a useful heuristic.

Greedy Search

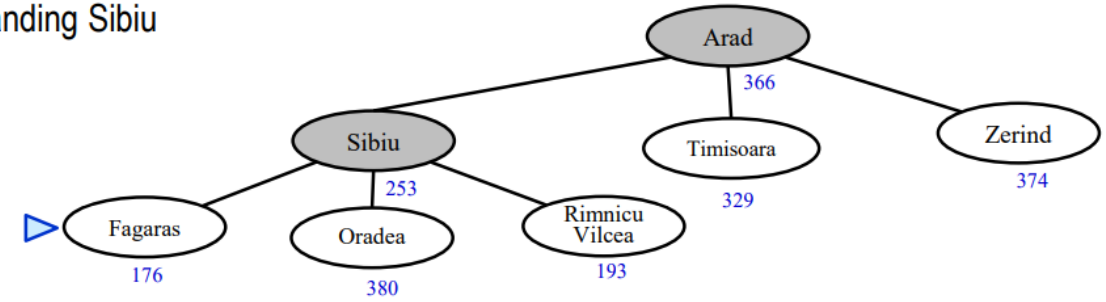
- Example: from Arad to Bucharest

(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu

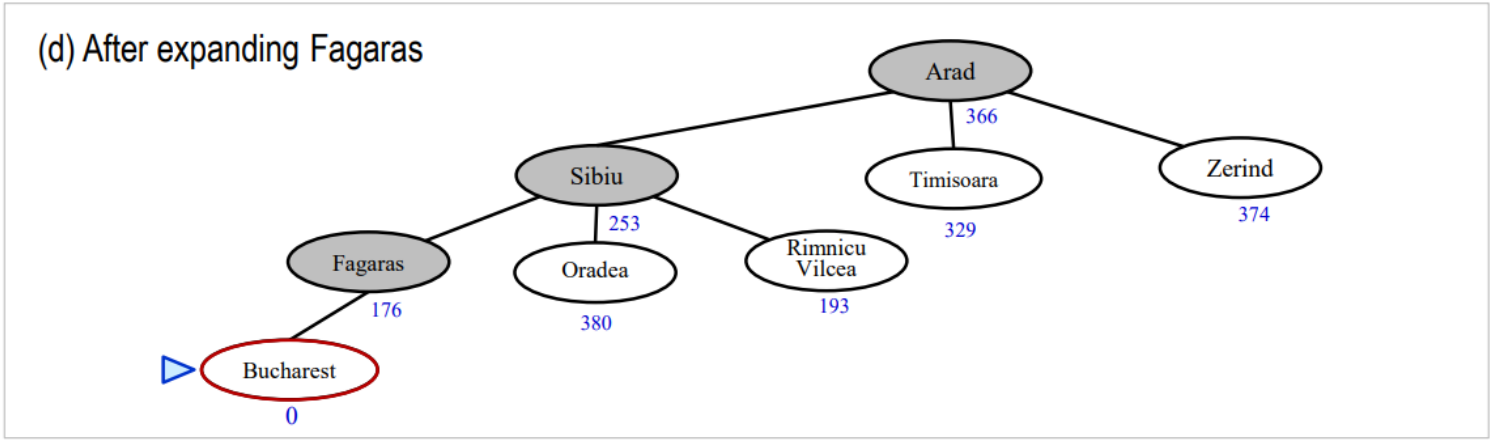


h_{SLD} Values

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

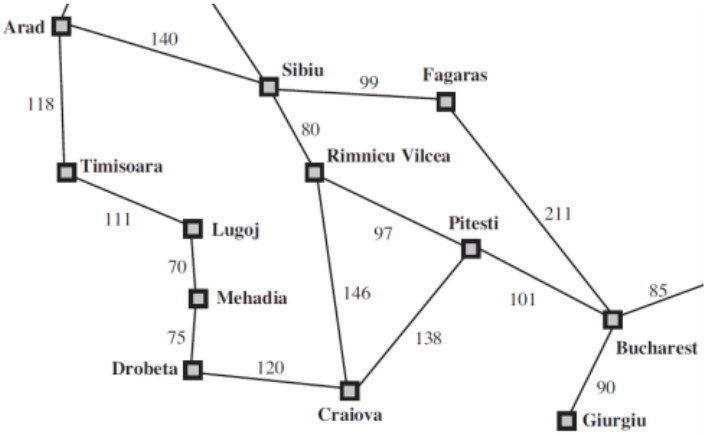
Greedy Search

- Example: from Arad to Bucharest



h_{SLD} Values

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



Notice: For this particular problem, it uses h_{SLD} to find a solution, hence its search cost is minimal. However it is **not optimal**: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

(140+99+211) – (140+80+97+101) = 32

Informed Search:

A* Search

A* Search

- **Search Strategy**
  - avoid expanding expensive paths, **minimizing the total estimated solution cost.**
- **Evaluation function**

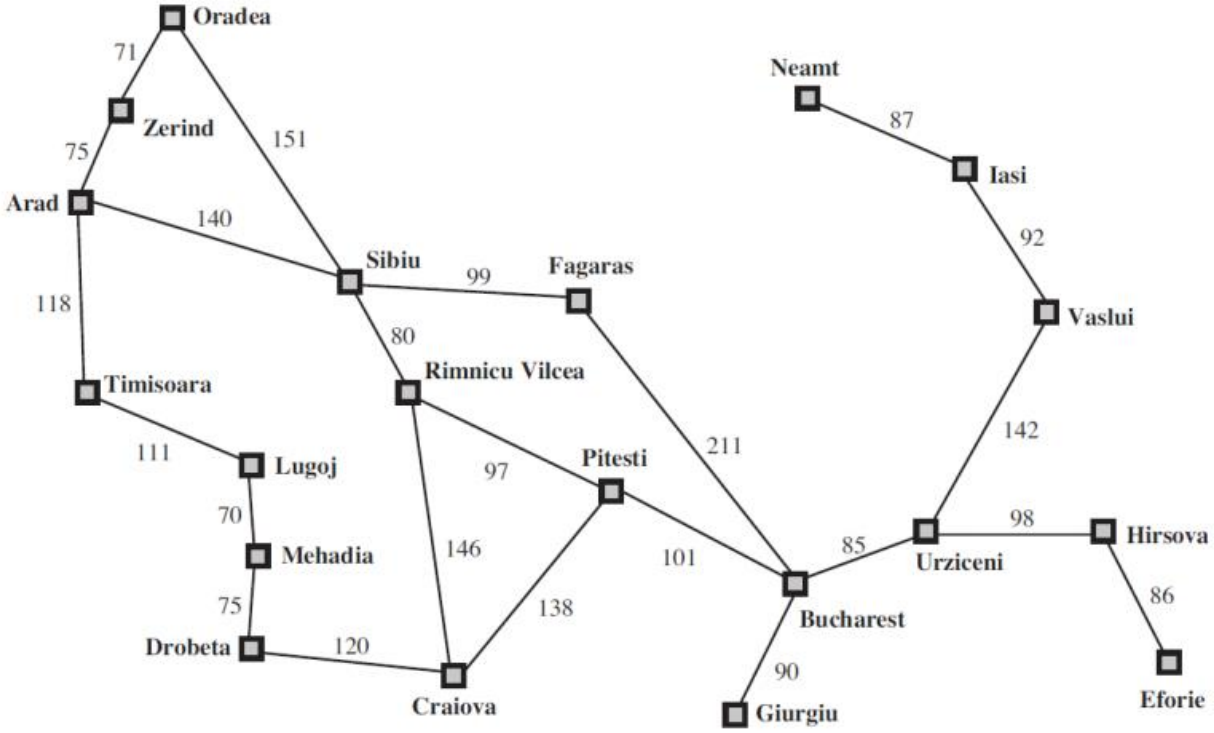
$$f(n) = g(n) + h(n)$$

  - $g(n)$ -- cost to reach the node
  - $h(n)$ -- estimated cost to get from the node to the goal

- **Theorem**: **A* search is optimal**

A* Search

- Example: from Arad to Bucharest



$f(n) = g(n) + h(n)$, which $g(n)$ = path cost, $h(n) = h^{SLD}$

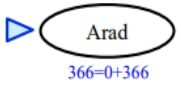
h_{SLD} Values

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

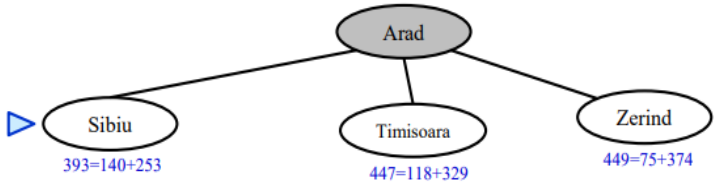
A* Search

- Example: from Arad to Bucharest

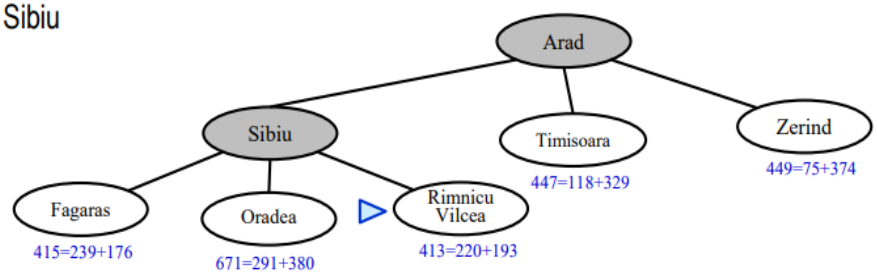
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



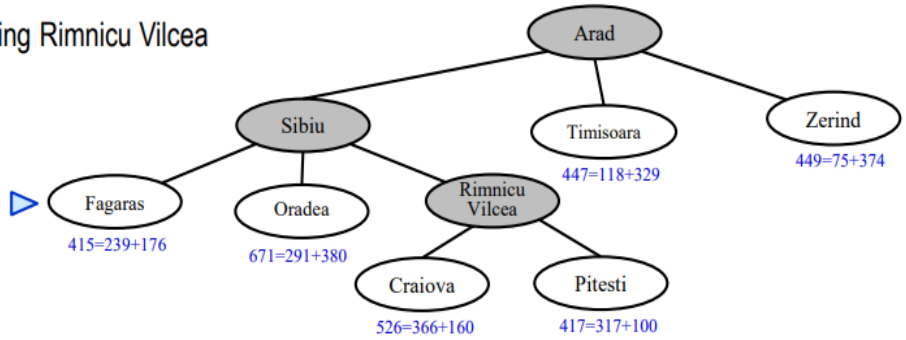
h_{SLD} Values

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

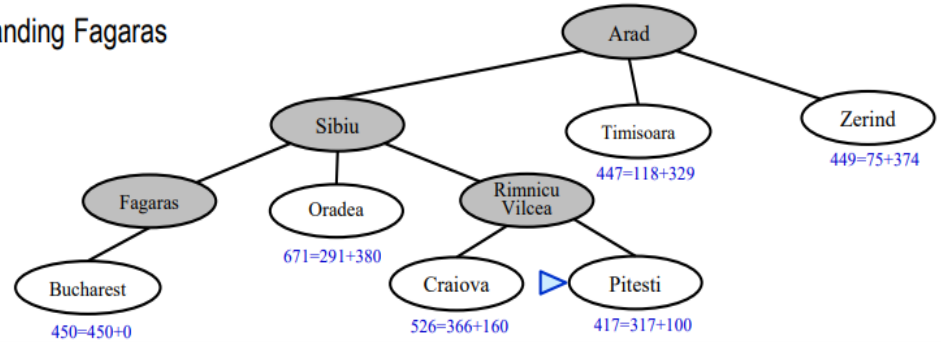
A* Search

- Example: from Arad to Bucharest

(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



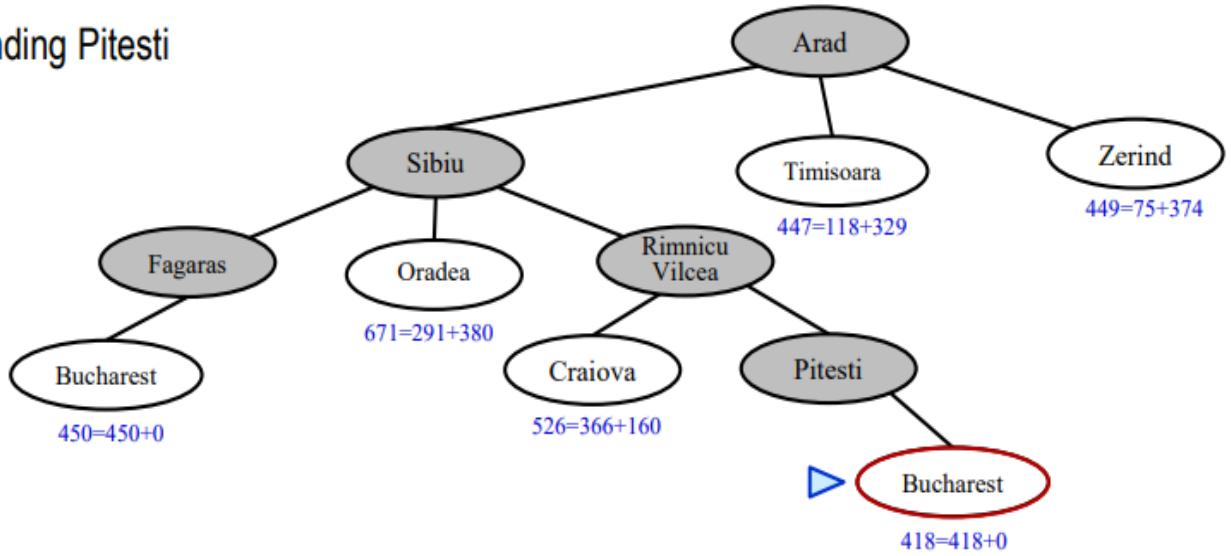
h_{SLD} Values

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

A* Search

- Example: from Arad to Bucharest

(f) After expanding Pitesti



h_{SLD} Values

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

Informed Search:

Heuristic Functions

Heuristics for 8-puzzle

- To find shortest solutions by using A* , need a heuristic function that are two commonly used candidates.
- *h*1 = number of misplaced tiles = 8
- *h*2 = total Manhattan distance (tiles from desired locations) = 3+1+2+2+2+3+3+2 = 18

7	2	4
5		6
8	3	1

Start state

	1	2
3	4	5
6	7	8

Goal state

Heuristics for 8-puzzle

- Search Cost (nodes generated)

d (depth)	Iterative Deepening Search	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

If $h2(n) \geq h1(n)$ for all n, then $h2$ dominates $h1$ and is better for search.

Search Cost

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

Search Cost (nodes generated)

d (depth)	Iterative Deepening Search	A*(h_1)	A*(h_2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

Beyond Classical Search

Beyond Classical Search

- In previous chapter, we addressed a single category of problems, where the solution is a sequence of actions with following features:
  - ***Observable***
  - ***Deterministic***
  - ***Known environments***
- The classical search algorithms are designed to explore search spaces systematically:
  - Keeping one or more paths in memory, and recording which alternatives have been explored at each point along the path
  - When a goal is found, the **path** also constitutes **a solution to the problem**
- However, in many problems, the path to the goal is **irrelevant**
  - **Local search**
  - **Swarm intelligence**

Beyond Classical Search:

Local Search
Swarm Intelligence

What is Local Search?

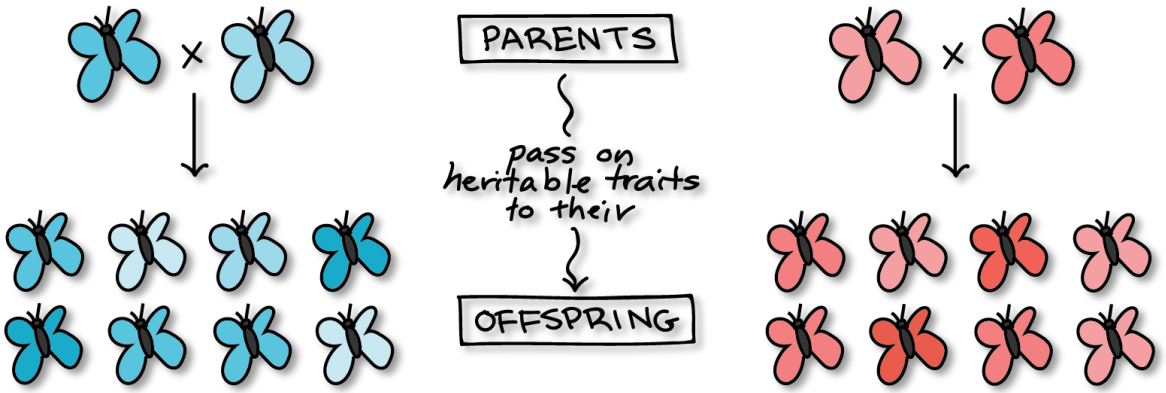
- Local search is a different class of algorithms that **do not worry about paths**.
- Local search algorithms operate using **a single current node** (rather than multiple paths), and generally move only to **neighbors** of that node.
- Typically, the paths followed by the search are not retained.
- Local search algorithms have two **key advantages**:
  - **Use very little memory**
  - **Can find reasonable solutions in large or infinite (continuous) state spaces**

Applications of Local Search

- In many application problems, the path is irrelevant, the goal state itself is the solution, such as:
  - *integrated-circuit design* (集成电路设计)
  - *factory-floor layout* (工厂车间布局)
  - *job-shop scheduling* (车间作业调度)
  - *automatic programming* (自动规划)
  - *telecommunications* (通讯)
  - *network optimization* (网络优化)
  - *vehicle routing* (车辆路由)
  - *portfolio management* (投资组合管理)

Optimization Problem

- In addition to finding goals, local search algorithms are useful for **solving pure optimization problems**.
- The aim in optimization is to **find the best state according to an objective function**.
- But many optimization problems **do not** fit using the previously introduced classical search algorithms.
- E.g., Darwinian evolution could be seen as attempting to optimize, but for this problem there is no "**goal test**", and no "**path cost**".



Local Search:

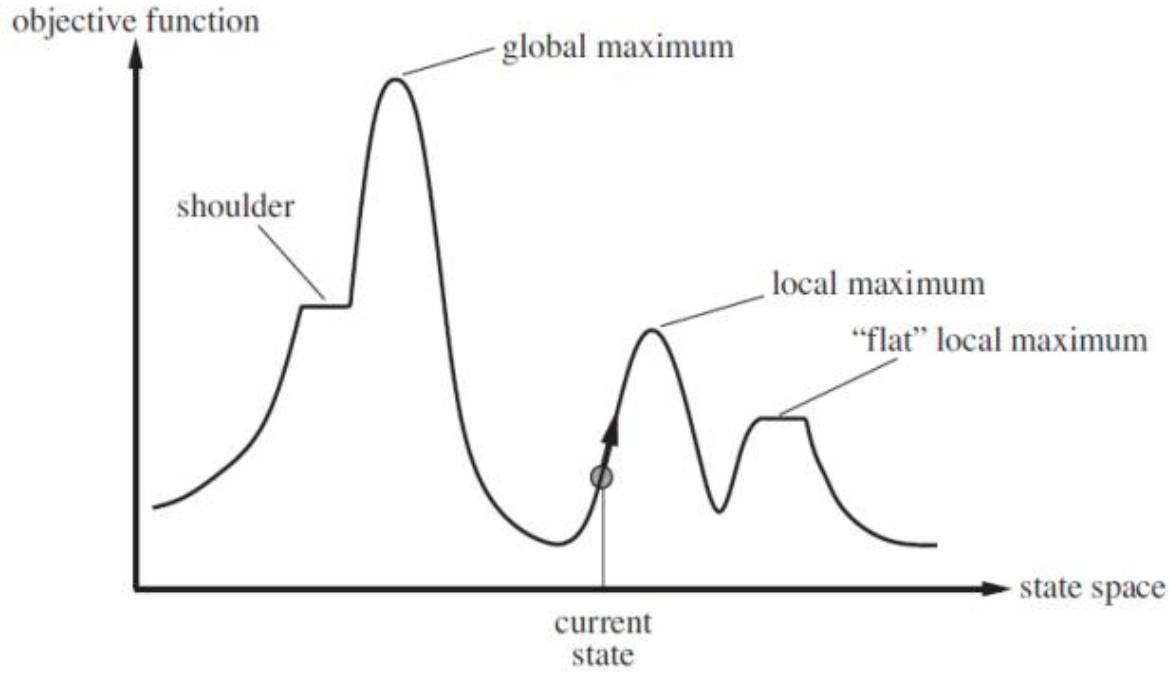
Hill-Climbing Search
Local Beam Search
Tabu Search
Simulated Annealing

Hill-Climbing Search

- A mathematical optimization technique which belongs to the family of local search.
- It is an **iterative** algorithm:
  - *Starts with an arbitrary solution to a problem*
  - *Then incrementally changes a single element of the solution*
  - *If it's a better solution, the change is made to the new solution*
  - *Repeating until no further improvements can be found*
- The **most basic local search** algorithm without maintaining a search tree

Hill-Climbing Search

- State-Space Landscape
  - It can be explored by one of local search algorithms.
  - A **complete local search algorithm** always finds **a goal** if one exists.
  - An **optimal local search algorithm** always finds **a global minimum or maximum**.



Hill-Climbing Search Algorithm

function HILL-CLIMBING(*problem*) **returns** a *state* that is a local maximum

persistent: *current*, a node

neighbor, a node

current ← MAKE-NODE(*problem*.INITIAL-STATE)

loop do

neighbor ← a successor of *current*

if *neighbor*.VALUE ≤ *current*.VALUE **then return** *current*.STATE

current ← *neighbor*

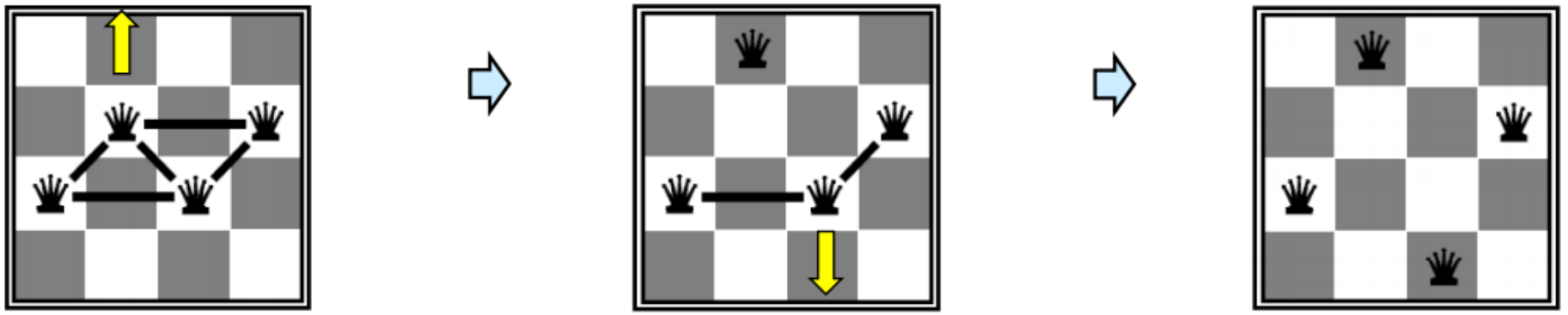
Steepest-ascent version: at each step, current node is replaced by the best neighbor (the neighbor with highest value), else it reaches a "peak".

Hill-Climbing Search Algorithm

- It often makes **rapid** progress toward a solution, because it is usually quite **easy to improve a bad state**.
- It is sometimes called **greedy local search**, because it grabs a good neighbor state **without thinking ahead about where to go next**.
- Although greed is considered one of the "*seven deadly sins*", it turns out that greedy algorithms often perform quite well.

Hill-Climbing Search

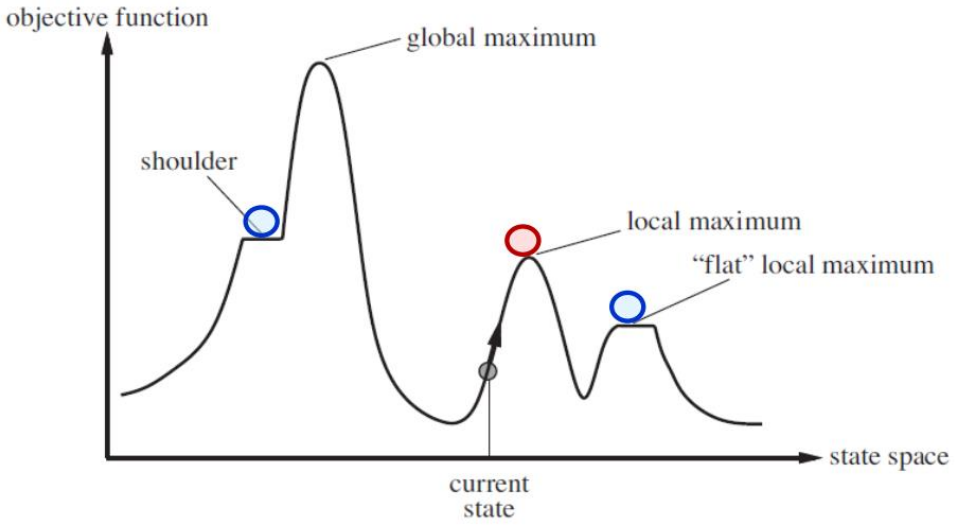
- Example: *n*-queens problems
  - To illustrate hill climbing, we will use the *n*-queens problem. Local search typically use a **complete-state formulation**.
  - Put all *n* queens on an *n* x *n* board. Each time move a queen to reduce number of conflicts, to be with no two queens on the same row, column, or diagonal.

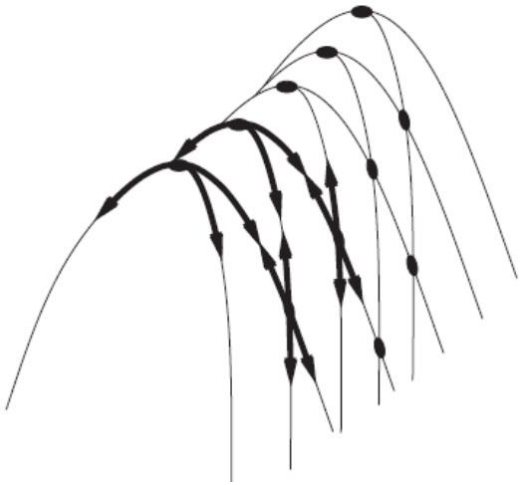


(a) $h = 5$ (b) $h = 2$ (c) $h = 0$

Weaknesses of Hill-Climbing

- It often **gets stuck** for the three reasons:
  - **Local maxima** 局部最大值
    - higher than its neighbors but lower than global maximum
  - **Plateaux** 高原
    - can be a flat local maximum, or a shoulder
  - **Ridges** 山岭
    - a sequence of local maxima that is very difficult to navigate





Variants of Hill-Climbing

- **Stochastic hill-climbing** 随机爬山法
  - It chooses at random among uphill moves; the probability of selection can **vary with the steepness** of uphill move.
  - This usually **converges more slowly** than steepest ascent.
- **First-choice hill-climbing** 首选爬山法
  - It implements stochastic hill climbing by **generating successors randomly** until one is generated that is better than the current state.
  - This is a good strategy when a state has many of successors
- **Random-restart hill-climbing** 随机重启爬山法
  - It conducts a series of hill-climbing searches from **randomly generated initial states**, until a goal is found.
  - It is trivially complete with probability approaching **1**, because it will eventually generate a goal state as the initial state.
  - If each hill-climbing search has a probability ***p*** of success, then the expected number of restarts required is **1/*p***.
  - It adopts the well-known adage: "*If at first you don't succeed, try, try again.*"

Local Search:

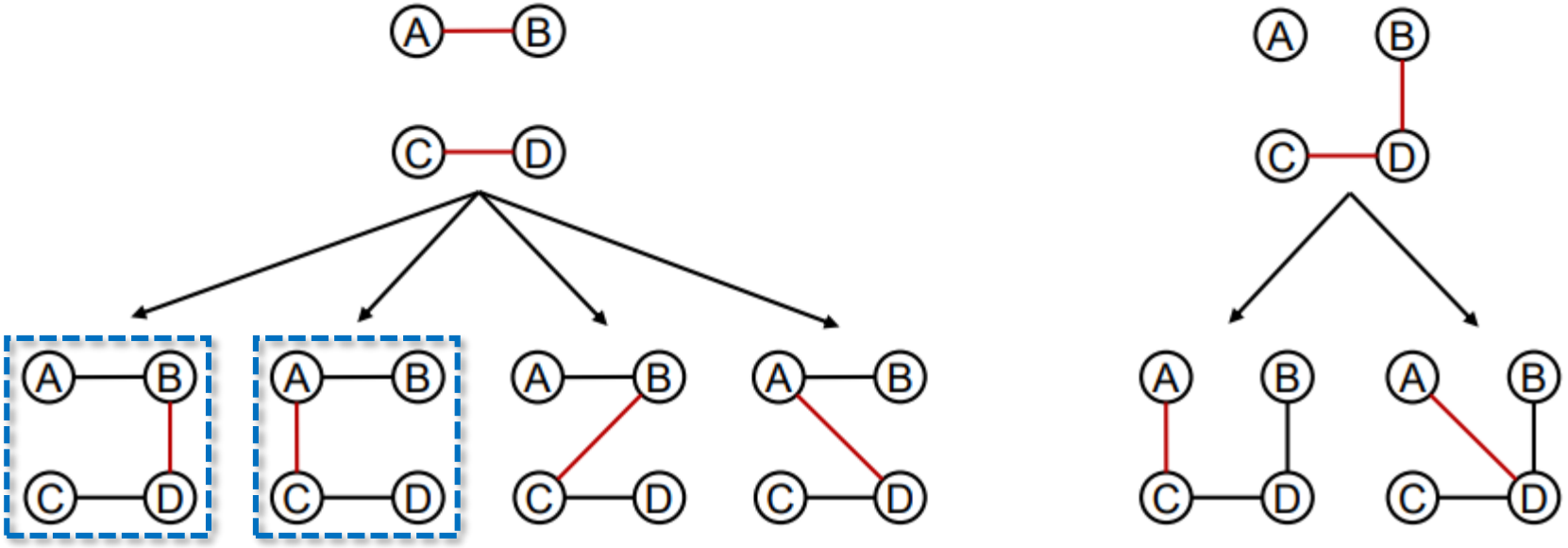
Hill-Climbing Search
Local Beam Search
Tabu Search
Simulated Annealing

Local Beam Search

- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.
- Local beam search keeps track of ***k*** states rather than just 1.
  - It begins with *k* randomly generated states.
  - At each step, all the successors of all *k* states are generated.
  - If any one is a goal, the algorithm halts, else it selects the *k* best successors from the complete list, and repeats.
- In a local beam search, useful information can be passed among the parallel search threads.

Local Beam Search

- Example: Travelling Salesperson Problem (TSP)
  - Keeps track of *k* states rather than just 1. Start with *k* randomly generated states. ***k*=2** in this example.
  - Generate **all successors** of all the *k* states. None of these is a goal state so we continue.
  - Select the **best *k* successors** from the complete list. Repeat the process until goal found.



Stochastic Beam Search

- **Local beam search** may quickly become **concentrated in a small region of state space**, making the search little more than an **expensive version of hill climbing**.
- It analogous to **stochastic hill climbing**, helps alleviate this problem.
  - Instead of choosing best *k* successors, it **chooses *k* successors randomly**, with the probability of choosing a successor being an **increasing** function of its value.
  - Stochastic beam search bears some similar to the process of **natural selection**.

Local Search:

Hill-Climbing Search
Local Beam Search
Tabu Search
Simulated Annealing

Tabu Search

- Tabu, indicates things that cannot be touched.
- Tabu search is created by Fred Glover in 1986 and formalized in 1989.
- It is a meta-heuristic algorithm, used for solving **combinatorial optimization problems**.
- It uses a local or neighborhood search procedure, to **iteratively move from one potential solution x to an improved neighborhood solution x'**, until some stopping condition has been satisfied.
- The memory structure to determine the solutions is called **tabu list**.

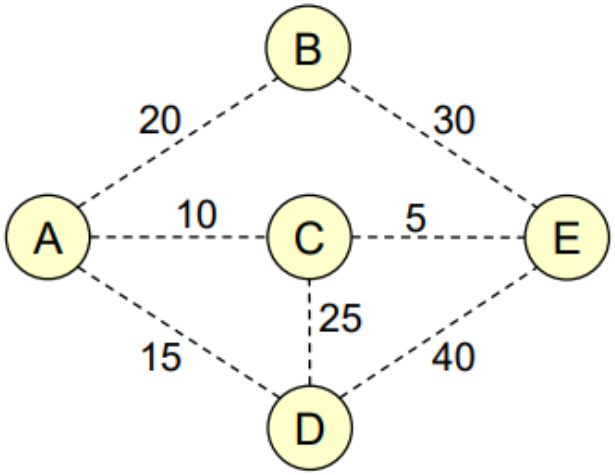
Tabu Search Strategies

- **Forbidding strategy**
  - Control what enters the tabu list.
- **Freeing strategy**
  - Control what exits the tabu list and when.
- **Short-term strategy**
  - Manage interplay between the forbidding strategy and freeing strategy to select trial solutions.

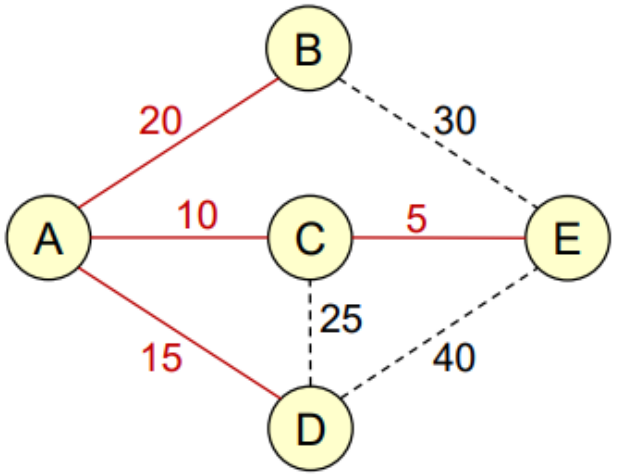
Tabu Search Problem

- Minimum Spanning Tree Problem

Objects:



Connects all nodes with minimum cost



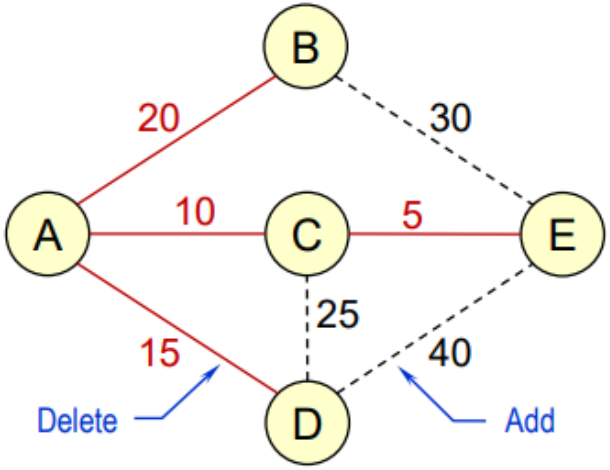
An optimal solution without constraints

- **Constraints 1**: Link AD can be included only if link DE also is included. (Penalty:100)
- **Constraints 2**: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)

Tabu Search Problem

- Minimum Spanning Tree Problem

Iteration 1:



Cost = 50 + 200 (constraint penalty)

Local optimum

局部最优

Add	Delete	Cost
BE	CE	75 + 200 = 275
BE	AC	70 + 200 = 270
BE	AB	60 + 100 = 160
CD	AD	60 + 100 = 160
CD	AC	65 + 300 = 365
DE	CE	85 + 100 = 185
DE	AC	80 + 100 = 180
DE	AD	75 + 0 = 75

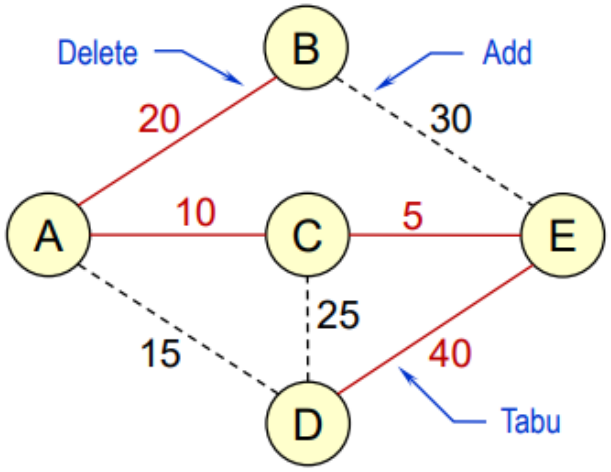
New Cost = 75

- **Constraints 1**: Link AD can be included only if link DE also is included. (Penalty:100)
- **Constraints 2**: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)

Tabu Search Problem

- Minimum Spanning Tree Problem

Iteration 2:



Cost = 75, Tabu list: DE

Escape local optimum

溢出局部最优

Add	Delete	Cost
AD	DE*	Tabu move
AD	CE	85 + 100 = 185
AD	AC	80 + 100 = 180
BE	CE	100 + 0 = 100
BE	AC	95 + 0 = 95
BE	AB	85 + 0 = 85
CD	DE*	Tabu move
CD	CE	95 + 100 = 195

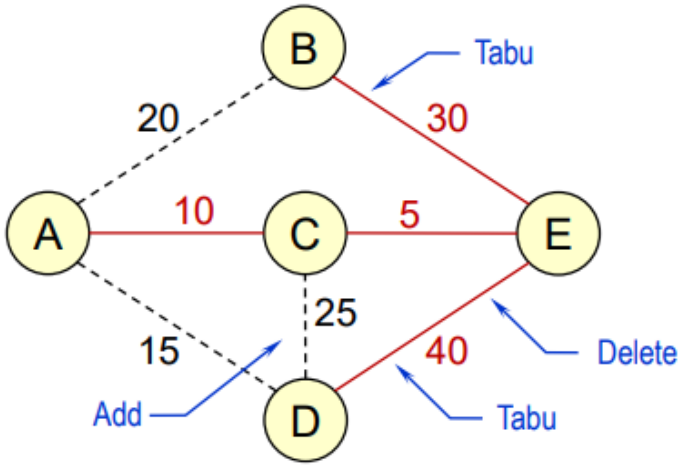
New Cost = 85

- **Constraints 1**: Link AD can be included only if link DE also is included. (Penalty:100)
- **Constraints 2**: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)

Tabu Search Problem

- Minimum Spanning Tree Problem

Iteration 3:



Cost = 85, Tabu list: DE & BE

Override tabu status

覆盖禁忌状态

Add	Delete	Cost
AB	BE*	Tabu move
AB	CE	100 + 0 = 100
AB	AC	95 + 0 = 95
AD	DE*	60 + 100 = 160
AD	CE	95 + 0 = 95
AD	AC	90 + 0 = 90
CD	DE*	70 + 0 = 70
CD	CE	105 + 0 = 105

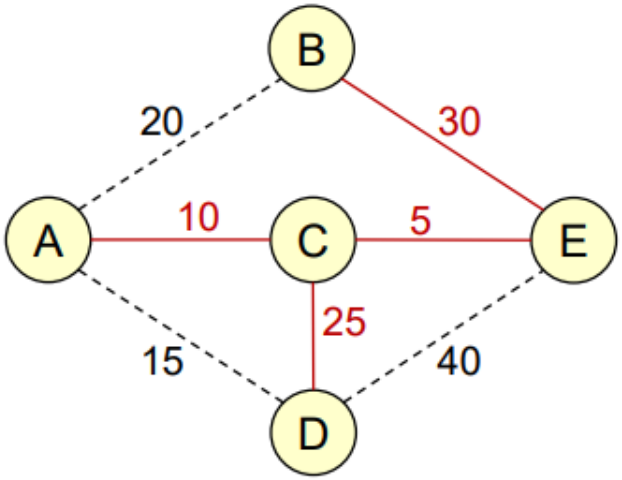
New Cost = 70

- **Constraints 1**: Link AD can be included only if link DE also is included. (Penalty:100)
- **Constraints 2**: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)

Tabu Search Problem

- Minimum Spanning Tree Problem

Iteration 4:



Optimal Solution

Additional iterations only find inferior solutions

- **Constraints 1**: Link AD can be included only if link DE also is included. (Penalty:100)
- **Constraints 2**: At most one of the three links (AD, CD, and AB) can be included. (Penalty: 100 if selected two of the three, 200 if selected all three.)

Tabu Search Applications

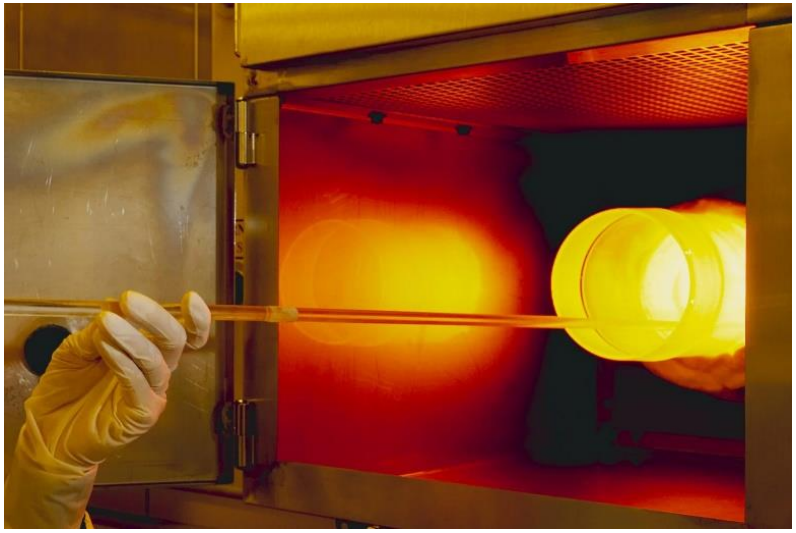
- Resource planning (资源规划)
- Telecommunications (通讯)
- VLSI design (VLSI设计)
- Financial analysis (金融分析)
- Scheduling (调度)
- Space planning (空间规划)
- Energy distribution (能源分配)
- Molecular engineering (分子工程)
- Logistics (物流)
- Flexible manufacturing (柔性生产)
- Waste management (废物管理)
- Mineral exploration (矿产勘探)
- Biomedical analysis (生物医药分析)
- Environmental conservation (环境保护)

Local Search:

Hill-Climbing Search
Local Beam Search
Tabu Search
Simulated Annealing

Annealing

- In metallurgy, annealing is used to temper or harden metals and glass.
- A solid is heated in a hot bath, increasing the temperature up to a maximum value.
- At this temperature, all material is **in liquid state** and the **particles arrange themselves randomly**.
- As the temperature of the hot bath is **cooled gradually**, all the particles of this structure will be arranged in the state of **lower energy**.



Simulated Annealing

- Simulated annealing is a **probabilistic technique** for **approximating the global optimum of a given function**. Proposed in 1953.
- Specifically, it is a **meta-heuristic** to approximate global optimization in a large search space.
- **Optimization** and **Thermodynamics** 优化与热力学

目标函数 **Objective function**	⇔	**Energy level** 能量极位
可接受解 **Admissible solution**	⇔	**System state** 系统状态
相邻解 **Neighbor solution**	⇔	**Change of state** 状态变化
控制参数 **Control parameter**	⇔	**Temperature** 温度
更优解 **Better solution**	⇔	**Solidification state** 凝固状态

Simulated Annealing Algorithm

- **Initial Solution**
  - Generated using an **heuristic**. Chosen at **random**.
- **Neighborhood**
  - Generated randomly. **Mutating** the current solution.
- **Acceptance**
  - Neighbor has lower cost value, higher cost value is accepted with the probability *p*.
- **Stopping Criteria**
  - Solution with a lower value than threshold.
  - Maximum total number of iterations.

Simulated Annealing Algorithm

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution *state*

inputs: *problem*, a problem

schedule, a mapping from time to "temperature"

current ← MAKE-NODE(*problem*.INITIAL-STATE)

for $t = 1$ **to** ∞ **do**

$T \leftarrow$ schedule(t)

if $T = 0$ **then return** *current*

next ← a randomly selected successor of *current*

$\Delta E \leftarrow$ *next*.VALUE – *current*.VALUE

if $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with probability $e^{\Delta E/T}$

A version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on.

Beyond Classical Search:

Local Search
Swarm Intelligence

What is Swarm Intelligence?

- Study of computational systems is inspired by the ***collective intelligence*** (集群智能)
  - which emerges through the cooperation of large numbers of homogeneous agents in the environment.
  - Examples: schools of **fish**, flocks of **birds**, and colonies of **ants**.
- Such intelligence is **decentralized**, **self-organizing** and **distributed** through out an environment.
- In nature, such systems are commonly used to: effective foraging for food (有效觅食), prey evading (猎物躲避), colony relocation (群体搬迁), *etc*.

Swarm Intelligence

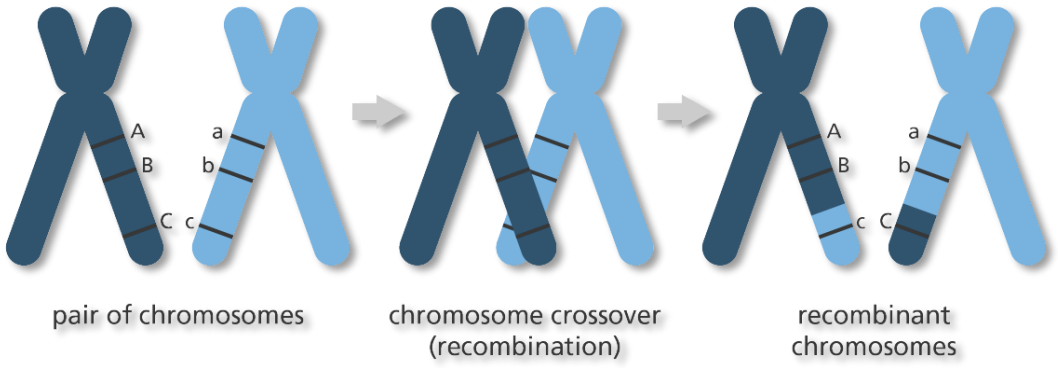
- Altruism algorithm (利他算法)
- Ant colony optimization (蚁群优化)
- Bee colony algorithm (蜂群算法)
- Artificial immune systems (人工免疫系统)
- Bat algorithm (蝙蝠算法)
- Differential evolution (差分进化)
- Multi-swarm optimization (多群体优化)
- Gravitational search algorithm (引力搜索算法)
- Glowworm swarm optimization (萤火虫群优化)
- Particle swarm optimization (粒子群优化)
- Bacterial colony optimization (细菌群优化)
- River formation dynamics (河流形成动力学)
- Self-propelled particles (自行式粒子系统)
- Stochastic diffusion search (随机扩散搜索)

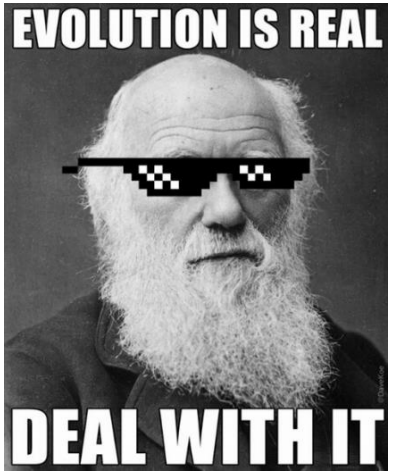
Swarm Intelligence:

Genetic Algorithms
Ant Colony Optimization
Particle Swarm Optimization

Genetic Algorithms

- The elements of genetic algorithms was introduced in 1960s. Became popular through the work of John Holland in early 1970s, and particularly his book *Adaptation in Natural and Artificial Systems* (1975).
- A **search heuristic** that mimics the process of natural selection.
- Belongs to the larger class of **evolutionary algorithms**.
- The algorithm is **a variant of stochastic beam search**, in which successor states are generated by **combining two parent states** rather than by modifying a single state. It is dealing with **sexual reproduction** rather than asexual reproduction.



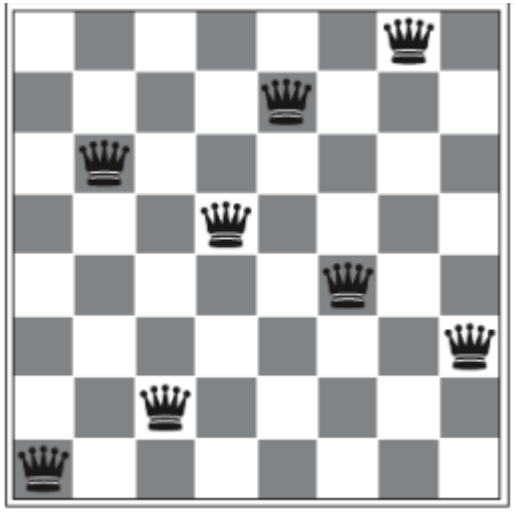


Genetic Algorithms

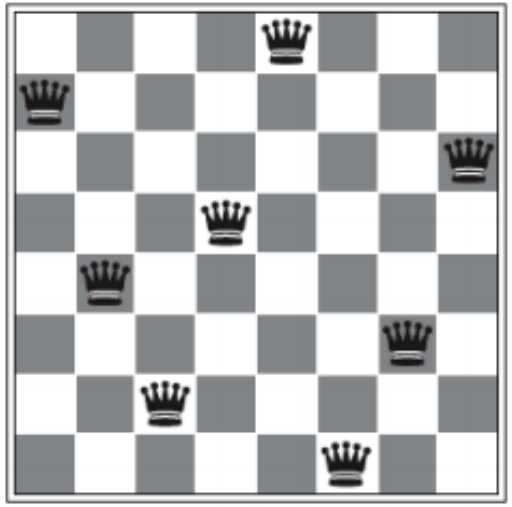
- The algorithms generate solutions to optimization problems using techniques inspired by natural evolution, such as
  - **inheritance**, **mutation**, **selection**, and **crossover**.
- It begin with a set of *k* randomly generated states, called the **population**.
- Each state (**individual**) is represented as a string over a finite alphabet, most commonly, **a string of 0s and 1s**.

Example: 8-queens problem

- An 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, the state could be represented as 8 digits, each in the range from 1 to 8.

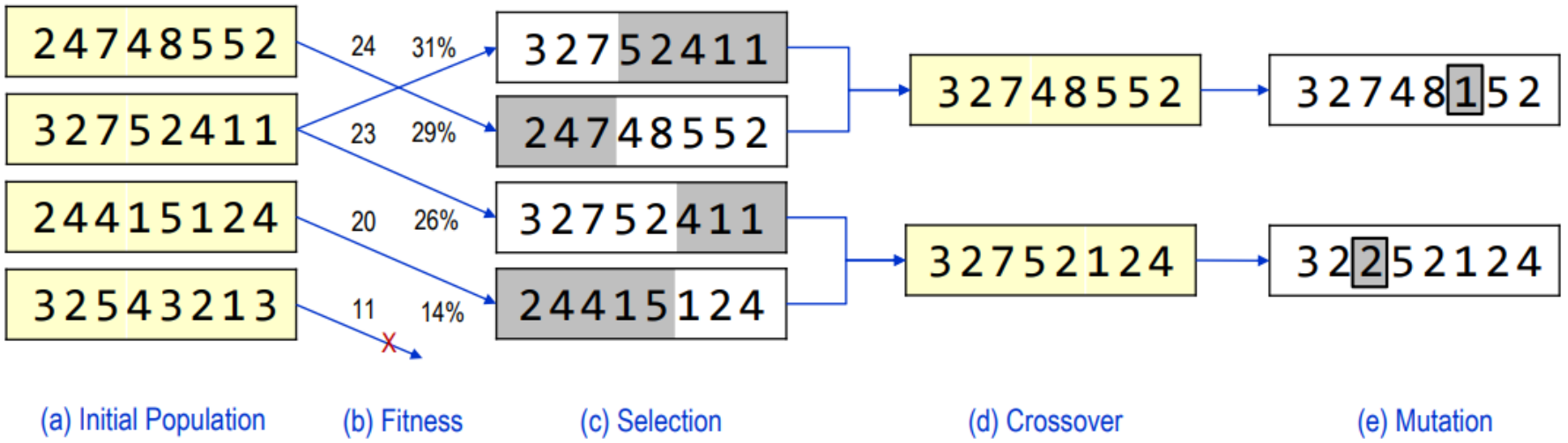


16257483



74258136

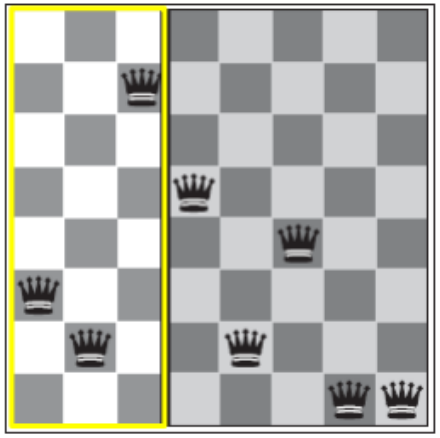
Example: 8-queens problem



Digit strings representing 8-queens states.

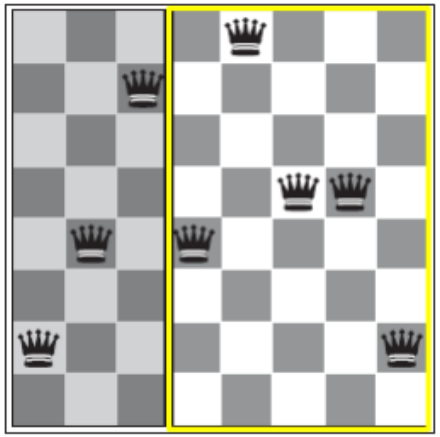
(a) the initial population, (b) ranked by the fitness function, (c) resulting in pairs for mating, (d) reproduce child, (e) subject to mutation.

Example: 8-queens problem



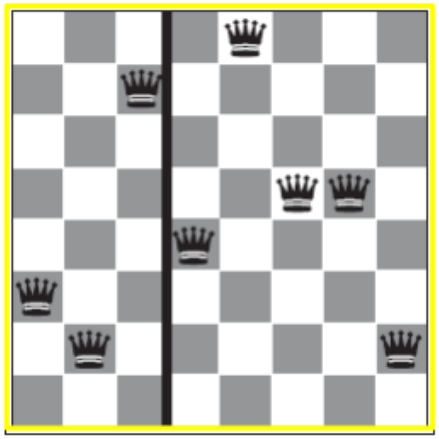
32752411

+



24748552

=



32748552

Those three 8-queens states correspond to the first two parents in "Selection" and their offspring in "Crossover". The shaded columns are lost in the crossover step and the unshaded columns are retained.

The Genetic Algorithm

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an *individual*
  inputs: *population*, a set of *individuals*
    FITNESS-FN, a function that measures the fitness of an *individual*
  repeat
    new_population ← empty set
    for *i* = 1 **to** SIZE(*population*) **do**
      x ← RANDOM-SELECTION(*population*, FITNESS-FN)
      y ← RANDOM-SELECTION(*population*, FITNESS-FN)
      child ← REPRODUCE(*x*, *y*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
      population ← *new_population*
  until some *individual* is fit enough, or enough time has elapsed
  return the best *individual* in *population*, according to FITNESS-FN

Applications

- Bioinformatics 生物信息学
- Computational science 计算科学
- Engineering 工程
- Economics 经济学
- Chemistry 化学
- Manufacturing 制造
- Mathematics 数学
- Physics 物理
- Phylogenetics 种系遗传学
- Pharmacometrics 定量药剂学

Swarm Intelligence:

Genetic Algorithms

Ant Colony Optimization

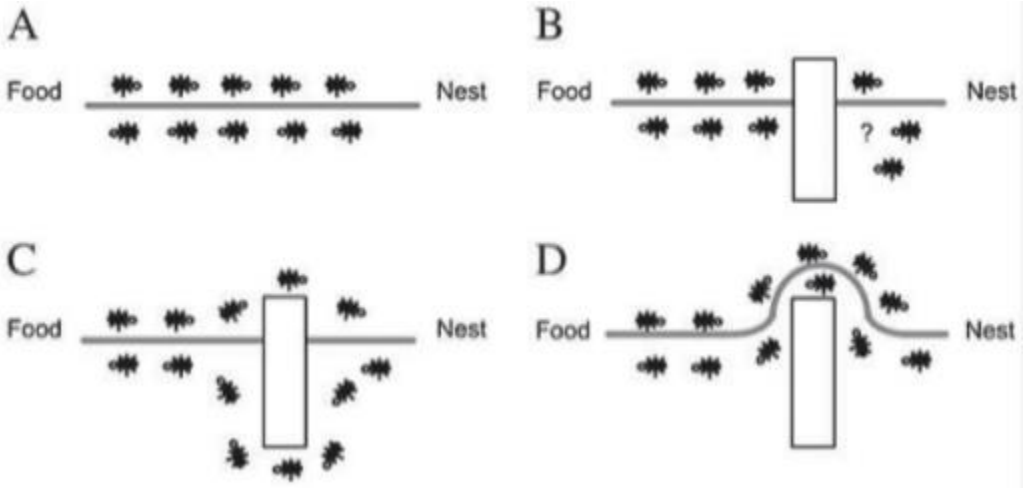
Particle Swarm Optimization

Ant Colony Optimization (ACO)

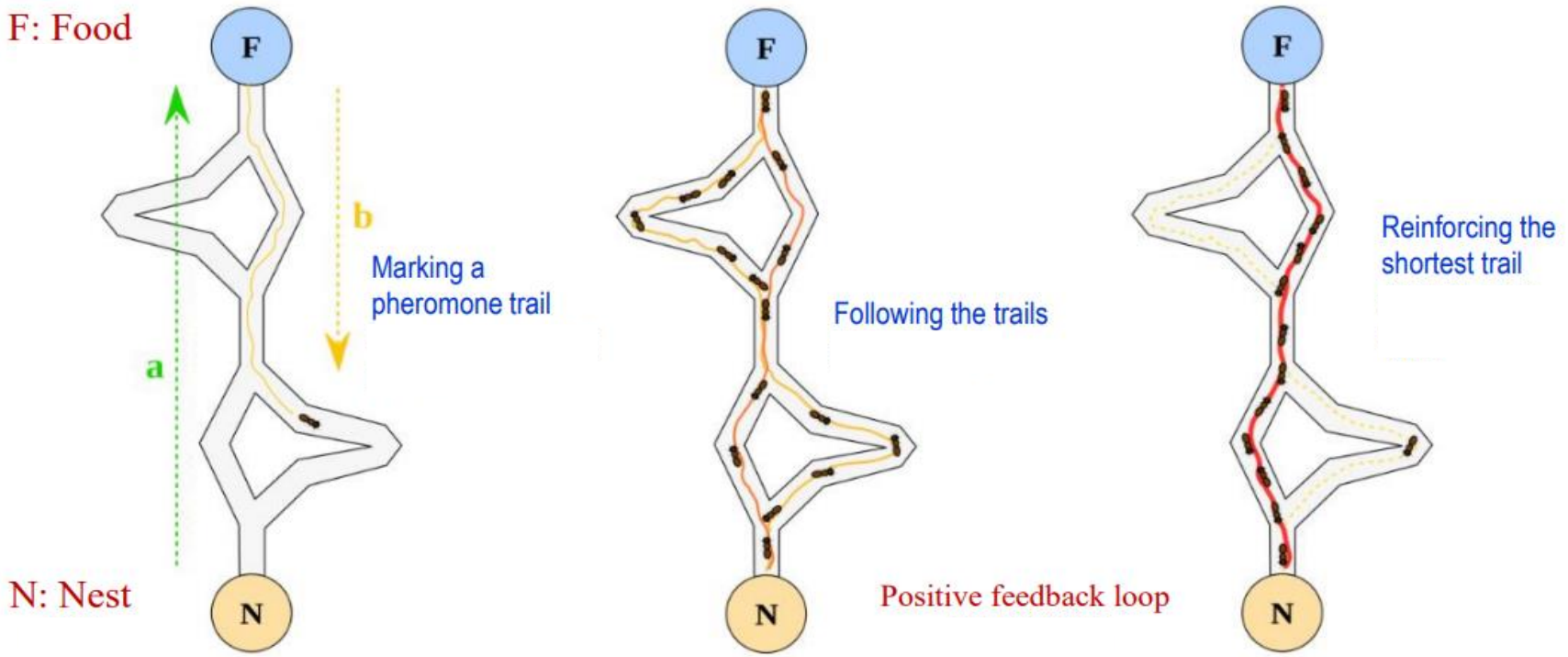
- It is a probabilistic technique for solving computational problems which can be used to finding optimal paths in a graph.
- Initially was proposed by Marco Dorigo in 1992 in his PhD thesis.
- The algorithm was inspired by the behavior of ants seeking a path between their nest and a source of food.

Concepts

- Ants navigate from nest to food source blindly:
  - Shortest path is discovered via pheromone (费洛蒙/信息素) trails
  - Each ant moves at random
  - Pheromone is deposited on path
  - Ants detect lead ant's path, inclined to follow
  - More pheromone on path increases probability of path being followed



Concepts



Demo: http://lab.breezedust.com/aco/

Algorithm of ACO

1. Virtual “trail” accumulated on path segments
2. Starting a node selected at random
3. The path selected at random: based on amount of “trail” present on possible paths from starting node; higher probability for paths with more “trail”
4. Ant reaches next node, selects next path
5. Repeated until most ants select the same path on every cycle

ACO vs. GA

- Commonalities:
  - Can be used in dynamic applications
    - Adapts to changes such as new distances, *etc*.
  - Good choice for constrained discrete problems
  - Theoretical analysis is difficult
    - Due to sequences of random decisions (not independent)
    - Probability distribution changes by iteration
    - Research is experimental rather than theoretical
- Difference:
  - ACO retains memory of entire colony instead of previous generation only
  - Less affected by poor initial solutions (due to combination of random path selection and colony memory)

Applications

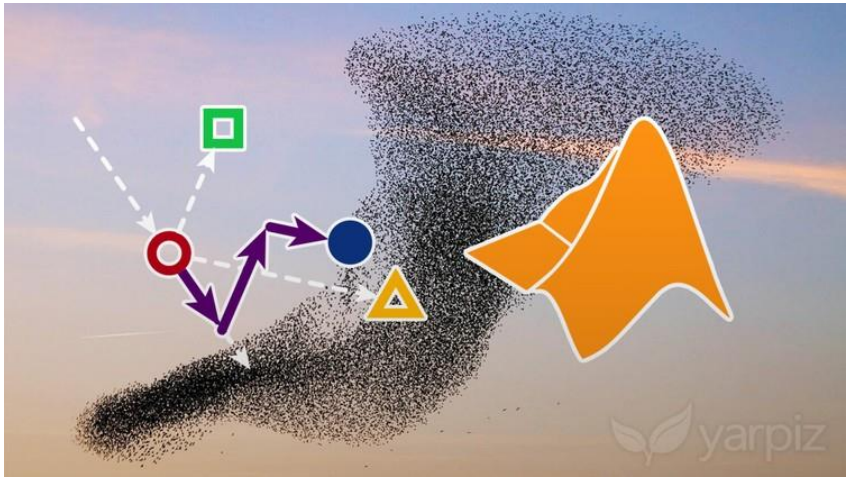
- Have been applied to many combinatorial optimization problems:
  - Scheduling problem 进度安排问题
  - Vehicle routing problem 车辆路径问题
  - Assignment problem 分派问题
  - Device Sizing Problem in Physical Design 物理设计中的设备量尺问题
  - Edge Detection in Image Processing 图像处理中的边缘检测
  - Classification 分类
  - Data mining 数据挖掘

Swarm Intelligence:

Genetic Algorithms
Ant Colony Optimization
Particle Swarm Optimization

Particle Swarm Optimization

- Proposed by James Kennedy & Russell Eberhart in 1995. Inspired by social behavior of birds and fishes.
- Uses a number of particles that constitute a swarm.
- moving around in the search space looking for the best solution.
- Each particle in search space adjusts its “flying” according to its own flying experience as well as the flying experience of other particles.

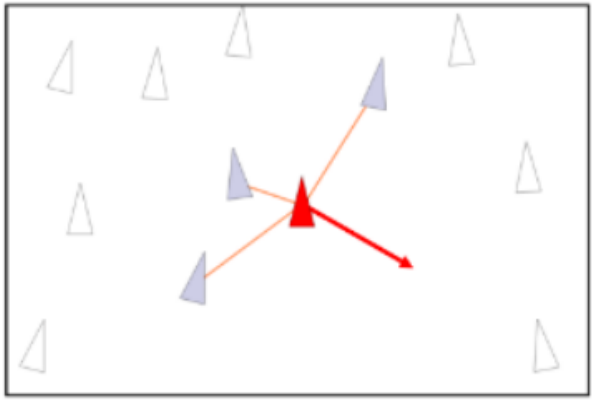


Bird Flocking

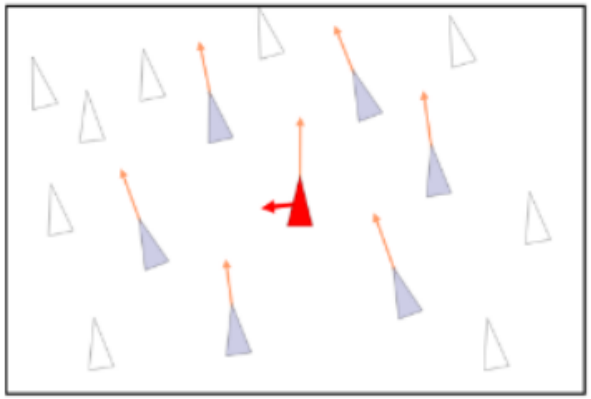
- A group of birds are randomly searching food in an area.
- There is only one piece of food in the area being searched.
- All the birds do not know where the food is.
- But **they know how far the food is in each iteration**.
- So what's the best strategy to find the food?
- The effective one is to **follow the bird which is nearest to the food**.

Bird Flocking

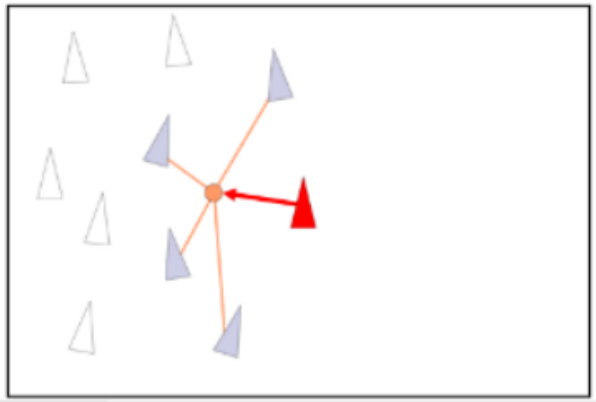
- Only three simple rules:
  a) Avoid collision with neighboring birds;
  b) Match the velocity of neighboring birds;
  c) Stay near neighboring birds.



(a)



(b)



(c)

Algorithm of PSO

For each *particle*
    Initialize *particle*
Do
    For each *particle* **Do**
        Calculate *fitness_value*
        If *fitness_value* > best fitness value (*pBest*) in history
            set *fitness_value* as new *pBest*
    Choose *particle* with best *fitness_value* of all particles as *gBest*
    For each *particle*
        Calculate *particle_velocity*
        Update *particle_position*
While maximum iterations or minimum error criteria is not attained

PSO vs. GA

- Commonalities
  - Population based stochastic optimization
  - Start with a group of a randomly generated population
  - Have fitness values to evaluate the population
  - Update the population and search for the optimum with random techniques
  - Do no guarantee success
- Differences
  - PSO does not have genetic operators like crossover and mutation. Particles update themselves with the internal velocity.
  - Particles in PSO have memory.
    - Not “what” that best solution was, but “where” it was.
  - Particles in PSO do not die. (No selection.)
  - Information sharing mechanism:
    - In PSO, info from best to others
    - In GA, population moves together

PSO & ANN

- An Artificial Neural Network (ANN) is a computing paradigm that is a simple model of the brain, and the backpropagation algorithm is the one of the most popular method to train the ANN.
- There have been significant research efforts to apply evolutionary computation (EC) techniques for the purposes of evolving one or more aspects of ANNs.
- Several papers reported **using PSO to replace the back-propagation learning algorithm** in ANN.
- It showed PSO is a promising method to train ANN. It is faster and gets better results in most cases.

Adversarial Search

Search vs. Adversarial Search

Search	Adversarial Search
Single agent	**Multiple** agents
Solutions is **method for finding goal**	Solution is **strategy** (strategy specifies move for every possible opponent reply)
Heuristics can find **optimal** solution	Time limits force an **approximate** solution
Evaluation function: estimate of **cost** from start to goal through given node	Evaluation function: evaluate **goodness** of game position

Adversarial Search:

Games
Alpha-Beta Pruning
Stochastic Games

Game Theory

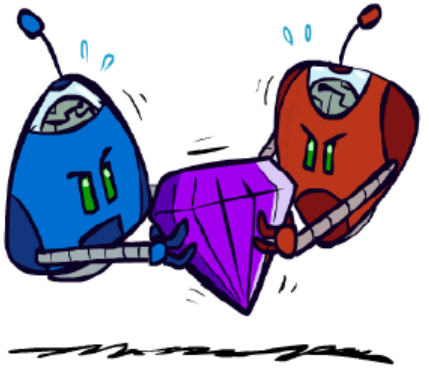
- **Adversarial search** often known as **Games**.
- Game theory:
  - Study of **strategic decision making**. Specifically, study of **mathematical models of conflict and cooperation** between intelligent rational decision-makes.
  - An alternative term is **interactive decision theory**.
- Applications:
  - Economics, political science, psychology, logic, computer science, and biology.
  - Behavioral relations and decision science, including both humans and non-humans (e.g., computers)

Game Theory

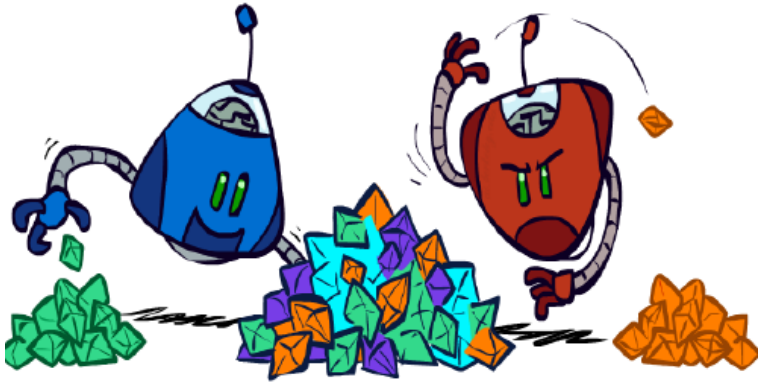
- Games are good problems for AI
  - Two or more players (agents)
  - Turn-taking vs. simultaneous moves
  - Perfect information vs. imperfect information
  - Deterministic vs. stochastic
  - Cooperative vs. competitive
  - Zero-sum vs. non zero-sum

Zero Sum vs. Non-zero Sum

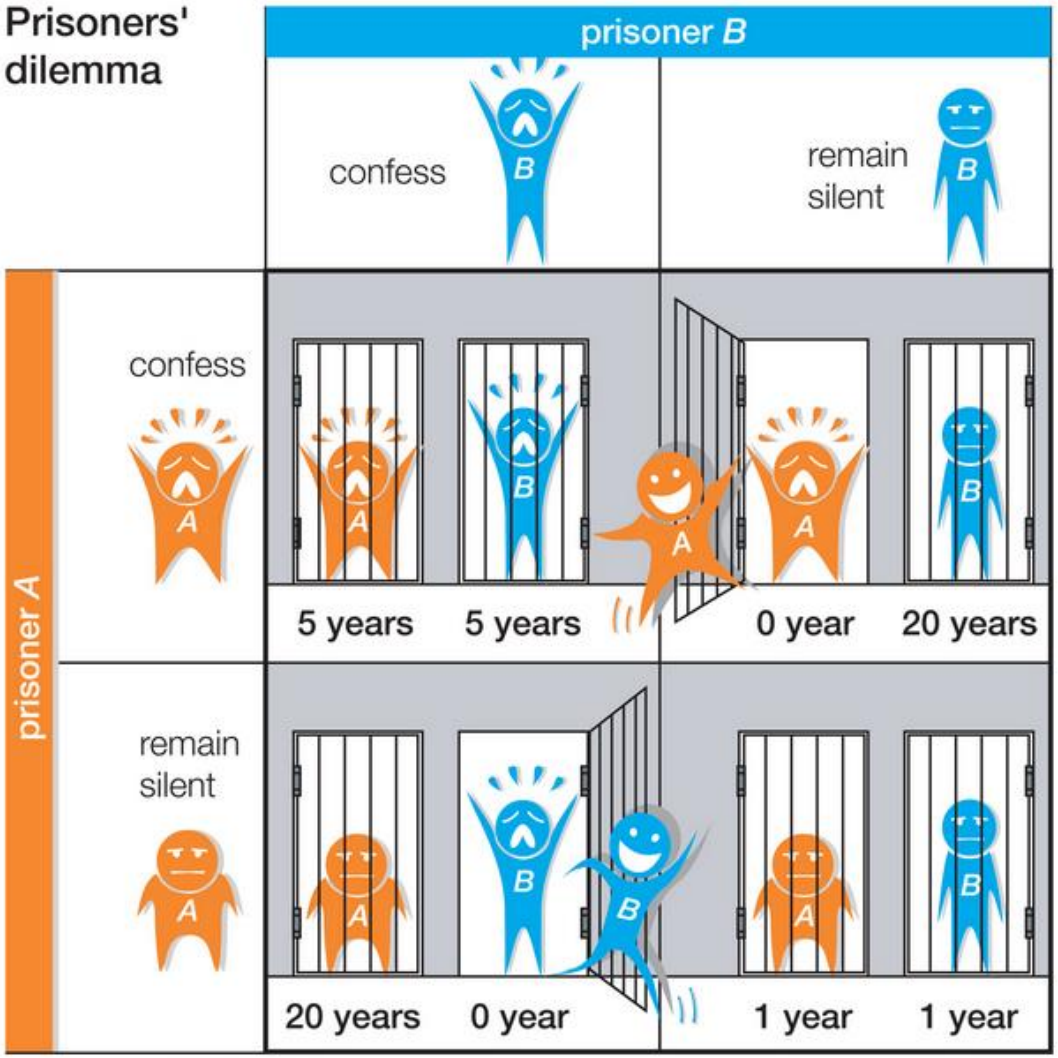
- **Zero sum games**
  - Agents have opposite utilities
  - Pure competition: win-lose, its sum is **zero**



- **Non-zero sum games**
  - Agents have independent utilities
  - Cooperation, indifference, competition, …
  - Win-win, win-lose, or lose-lose, its sum is **not zero**



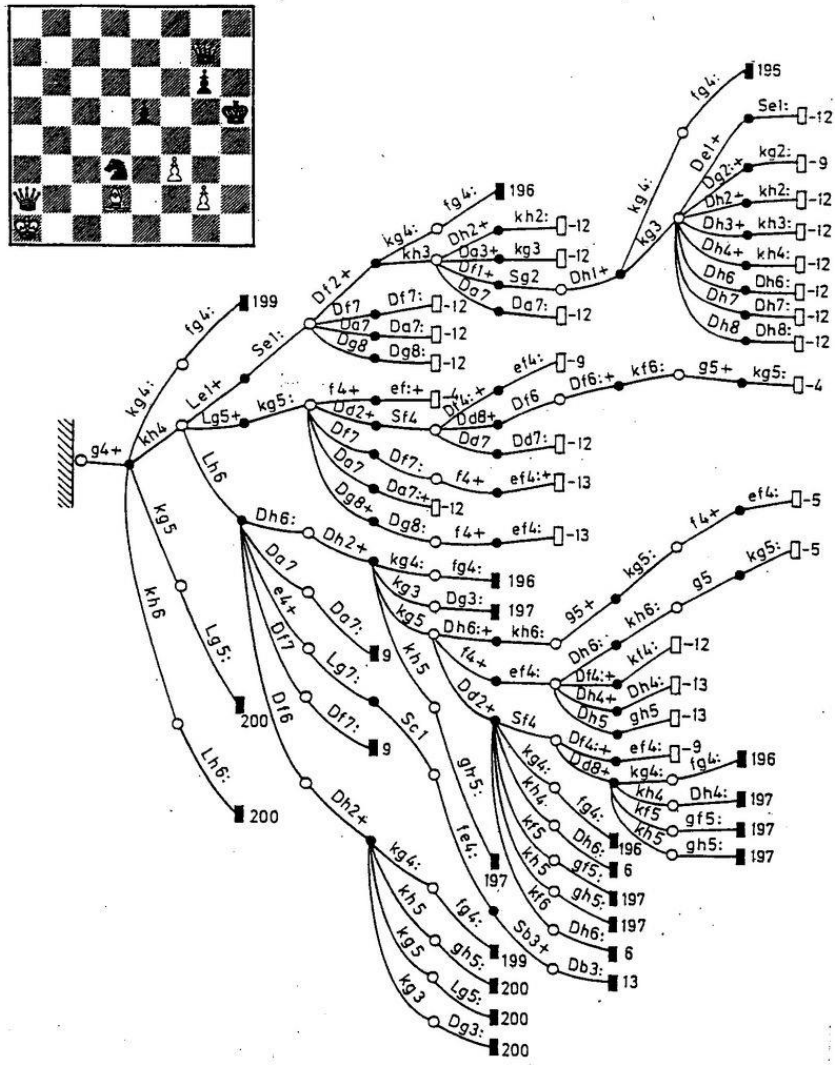
Example: Prisoner's Dilemma



https://beingmarketer.wordpress.com/2015/08/18/how-to-break-prisoners-dilemma-in-marketing/

Games

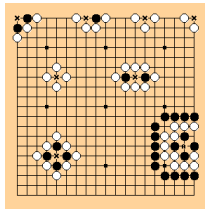
- Interesting,
- But **too hard to solve**:
- E.g., **Chess**: average branching factor ≈ 35, each player often go to 50 moves, so search tree has about 35^{100} or 10^{154} nodes.



Types of Games

	Deterministic	Stochastic
Perfect information (fully observation)	Chess Checkers Go Othello	Backgammon Monopoly
Imperfect information (partially observation)	Stratego Battleships	Bridge Poker Scrabble








Origins of Game Playing

- 1912, **Ernst Zermelo**
  - A German logician and mathematician
  - **Minimax algorithm**
- 1949, **Claude Shannon**
  - An American mathematician, and cryptographer known as "the father of information theory"
  - **Chess playing** with evaluation function, selective search
- 1956, **John McCarthy**
  - An American computer scientist and cognitive scientist, and one of the founders of AI
  - **Alpha-beta search**
- 1956, **Arthur Samuel**
  - An American pioneer of computer gaming, AI, and ML
  - **Checkers program** that learns its own evaluation function

Two Players Games

- **Features**:
  - **Deterministic**
  - **Perfect information**
  - **Turn-taking**
  - **Two players**
  - **Zero-sum**
- Name these two players: **Max** and **Min**
- At game end:
  - **Winner**: award points
  - **Loser**: give penalties

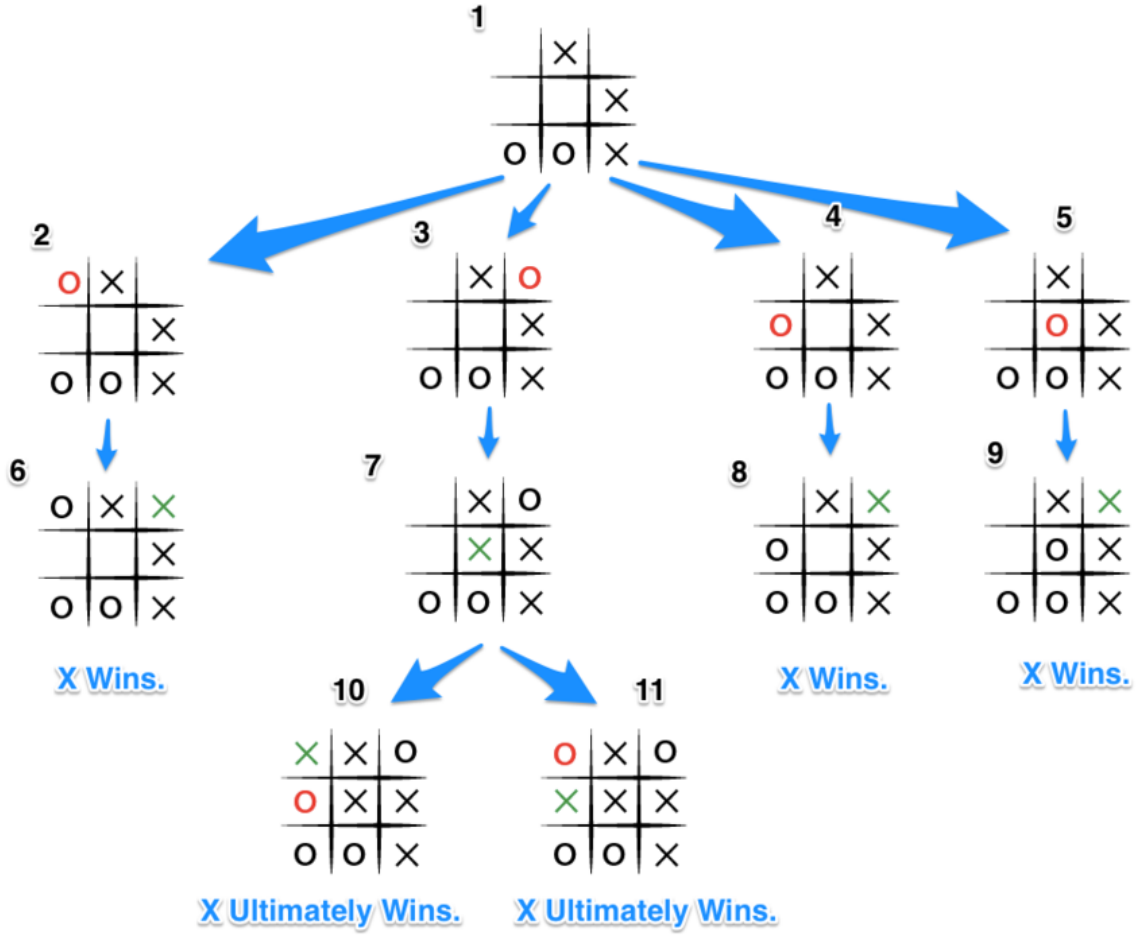
Formal Definition

- Formally defined as a search problem

S_0	**Initial state**, specifies how the game is set up at the start
PLAYER(s)	Defines which **player** has the move in a state
ACTIONS(s)	Returns the set of **legal moves** in a state
RESULTS(s,a)	**Transition model**, defines the result of a move
TERMINAL-TEST(s)	**Terminal test**, true when the game is over, and false otherwise
UTILITY(s,p)	**Utility function**, defines the value in state s for a player p

Example: Tic-tac-toe

- Part of the tree, giving alternating moves by MIN(O) and MAX(X)



Optimal Solution

- In **normal search**, the optimal solution would be a sequence of actions leading to a **goal state** (terminal state) that is a win.

- In **adversarial search**, both MAX and MIN could have an optimal strategy
  - In **initial state**, MAX must find a strategy to specify MAX's move,
  - then MAX's moves in the states resulting from every **possible response by MIN**, and so on.

Minimax Theorem

For every two-player, zero-sum game with finitely many strategies, there exists a value *V* and a mixed strategy for each player, such that

- Given player 2's strategy, the best payoff possible for player 1 is *V*;
- Given player 1's strategy, the best payoff possible for player 2 is –*V*.

For a zero-sum game, the name **minimax** arises because

- each player **minimizes** the **maximum** payoff possible for the other,
- he also **minimizes** his own **maximum** loss.

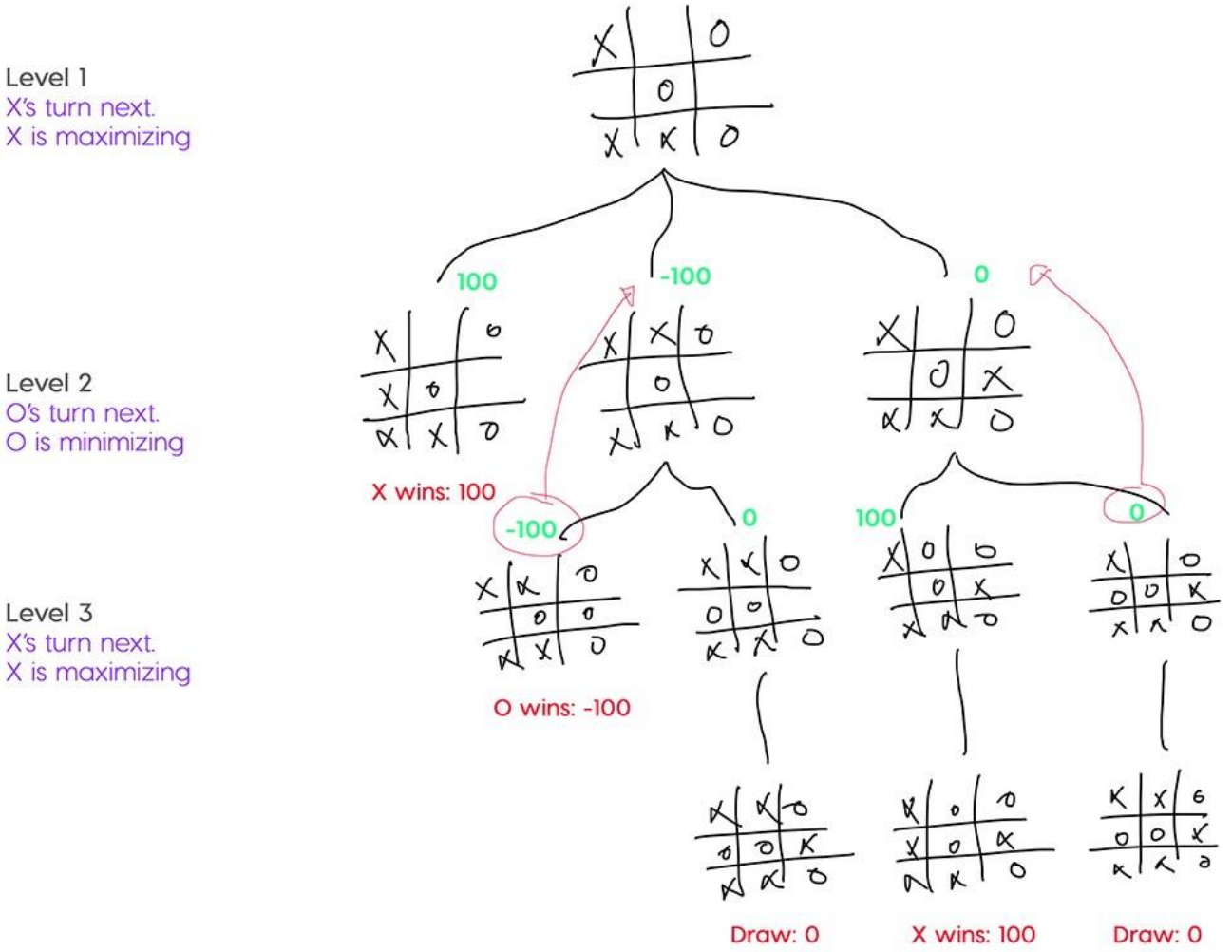
Minimax Algorithm

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, write as MINIMAX(*n*).
- Assume that both players play optimally from there to the end of the game.

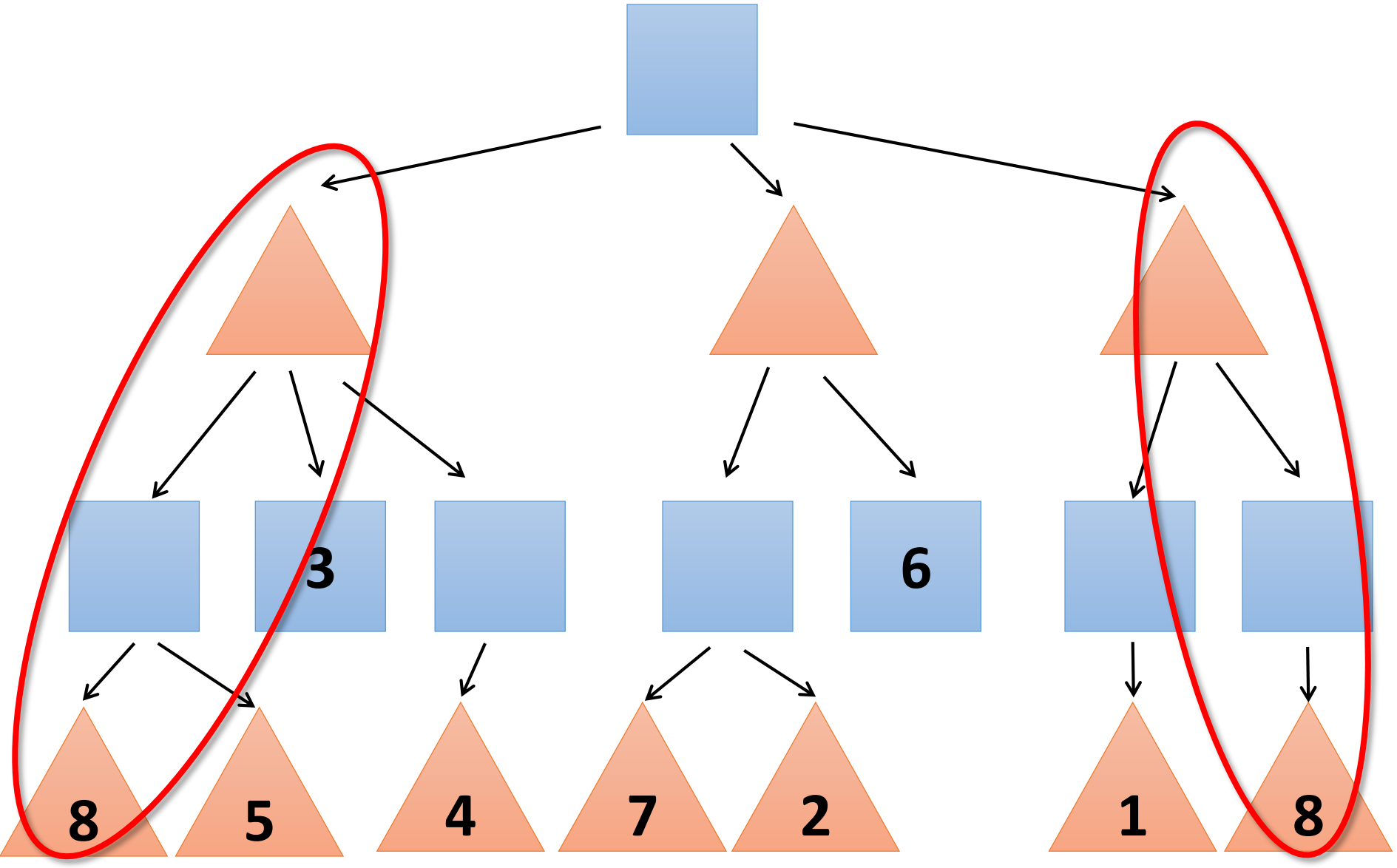
function MINIMAX(*s*) **returns** an action
    if TERMINAL-TEST(*s*) **then return** UTILITY(*s*)
    if PLAYER(*s*)=MAX **then return** $\max_{a \in ACTIONS(s)}$MINIMAX(RESULT(*s*,*a*))
    if PLAYER(*s*)=MIN **then return** $\min_{a \in ACTIONS(s)}$MINIMAX(RESULT(*s*,*a*))

The minimax value of a terminal state is just its utility.
MAX prefers to move to a state of maximum value, MIN prefers a state of minimum value.

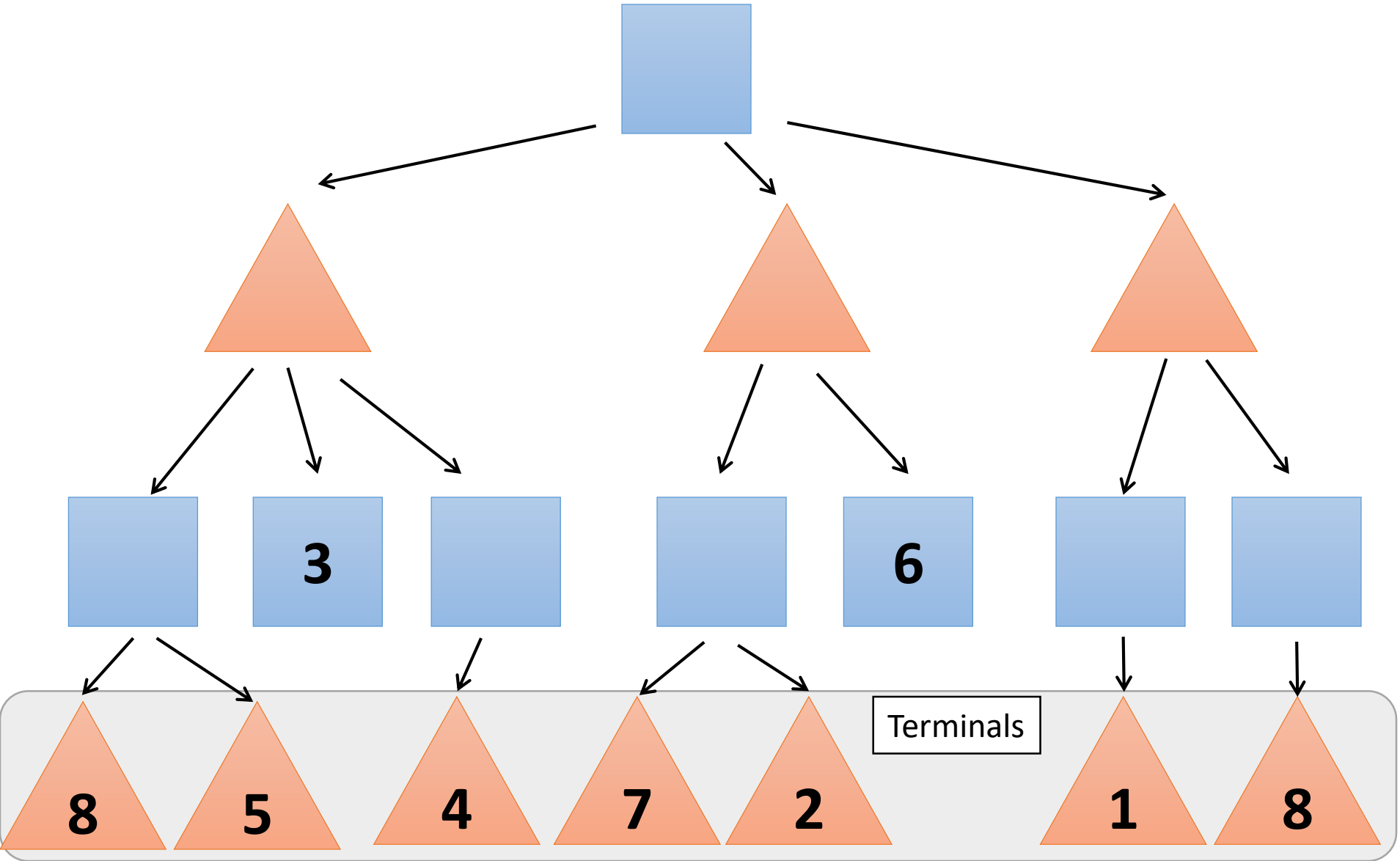
Example: Tic-tac-toe



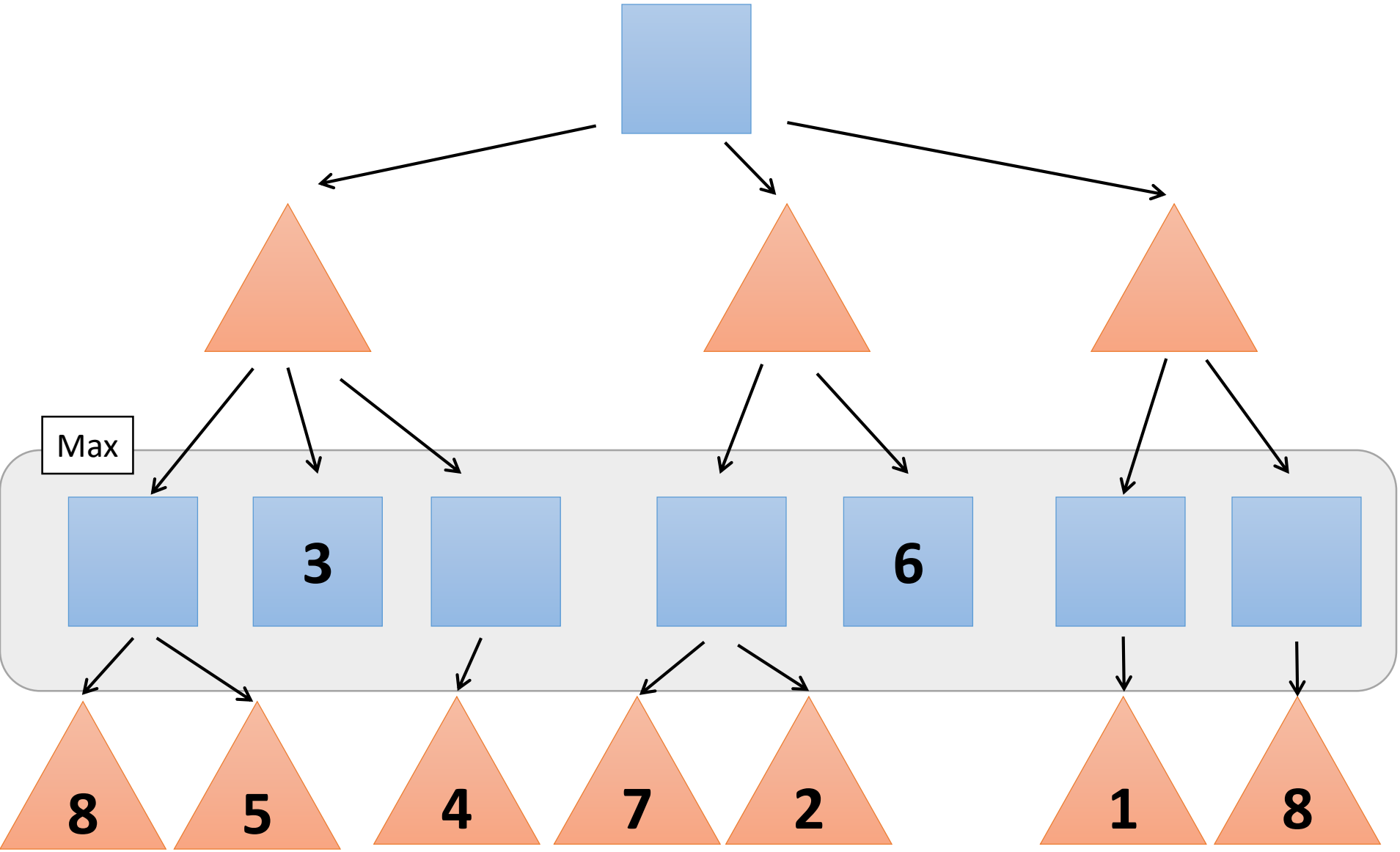
Minimax



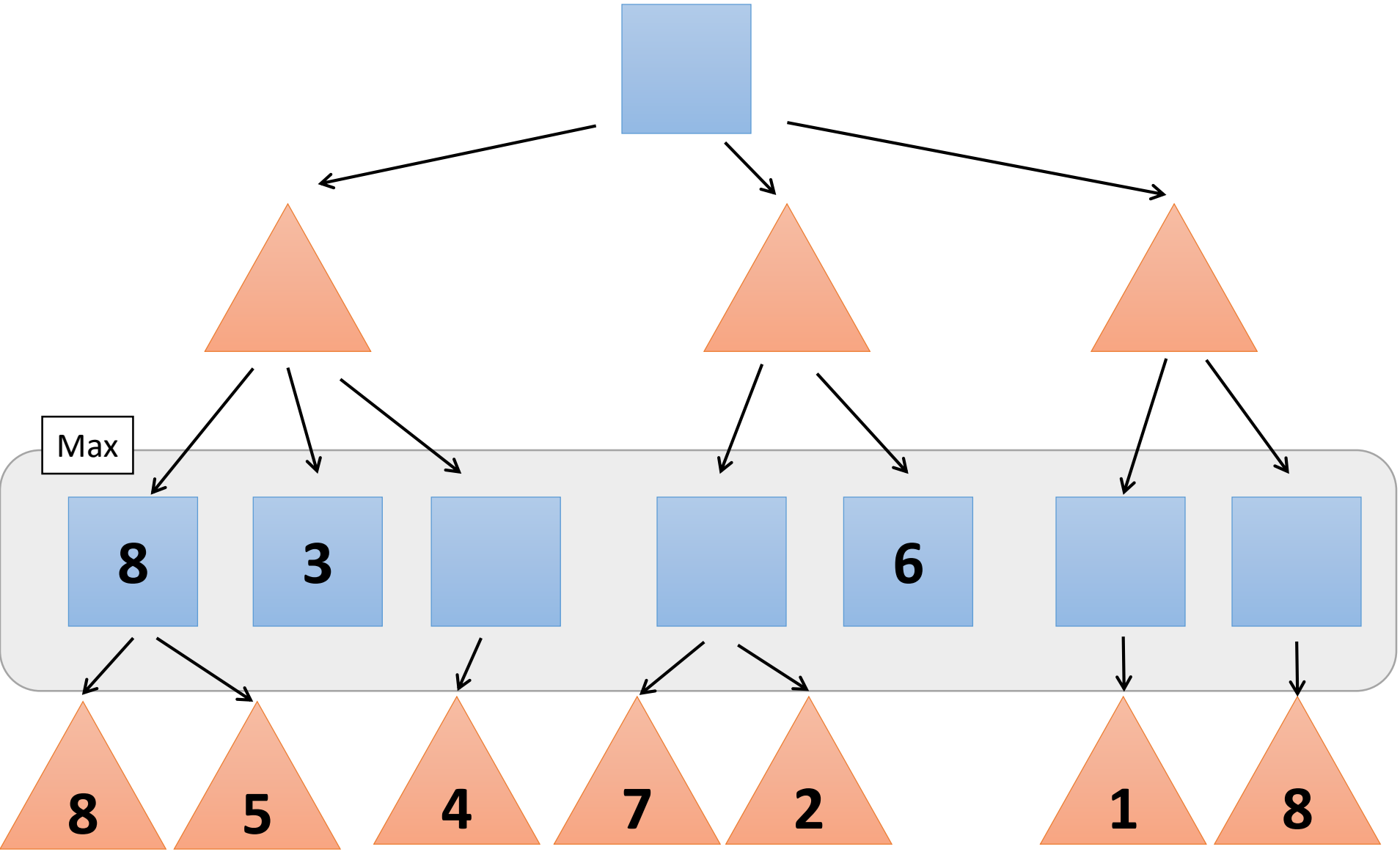
Minimax



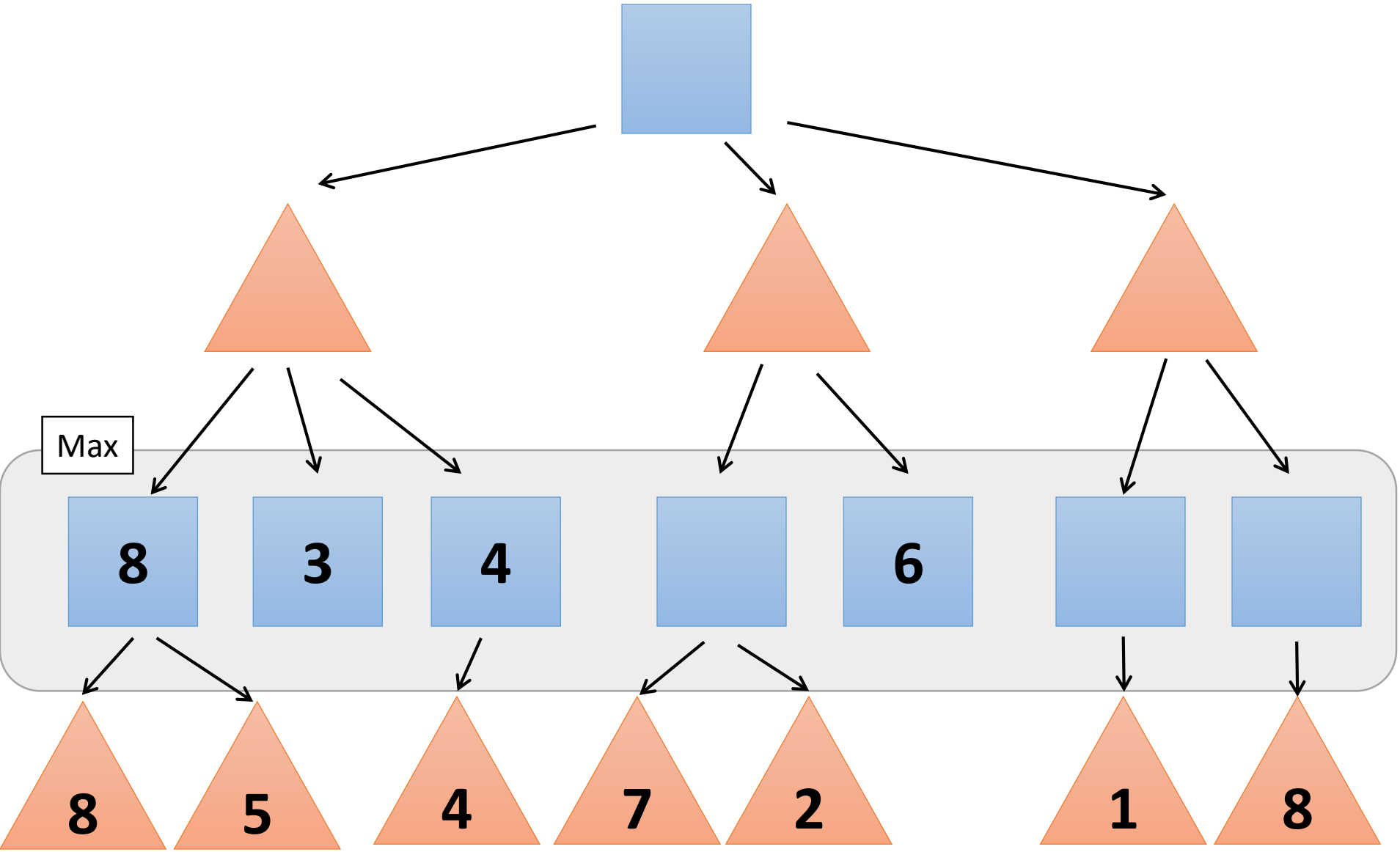
Minimax



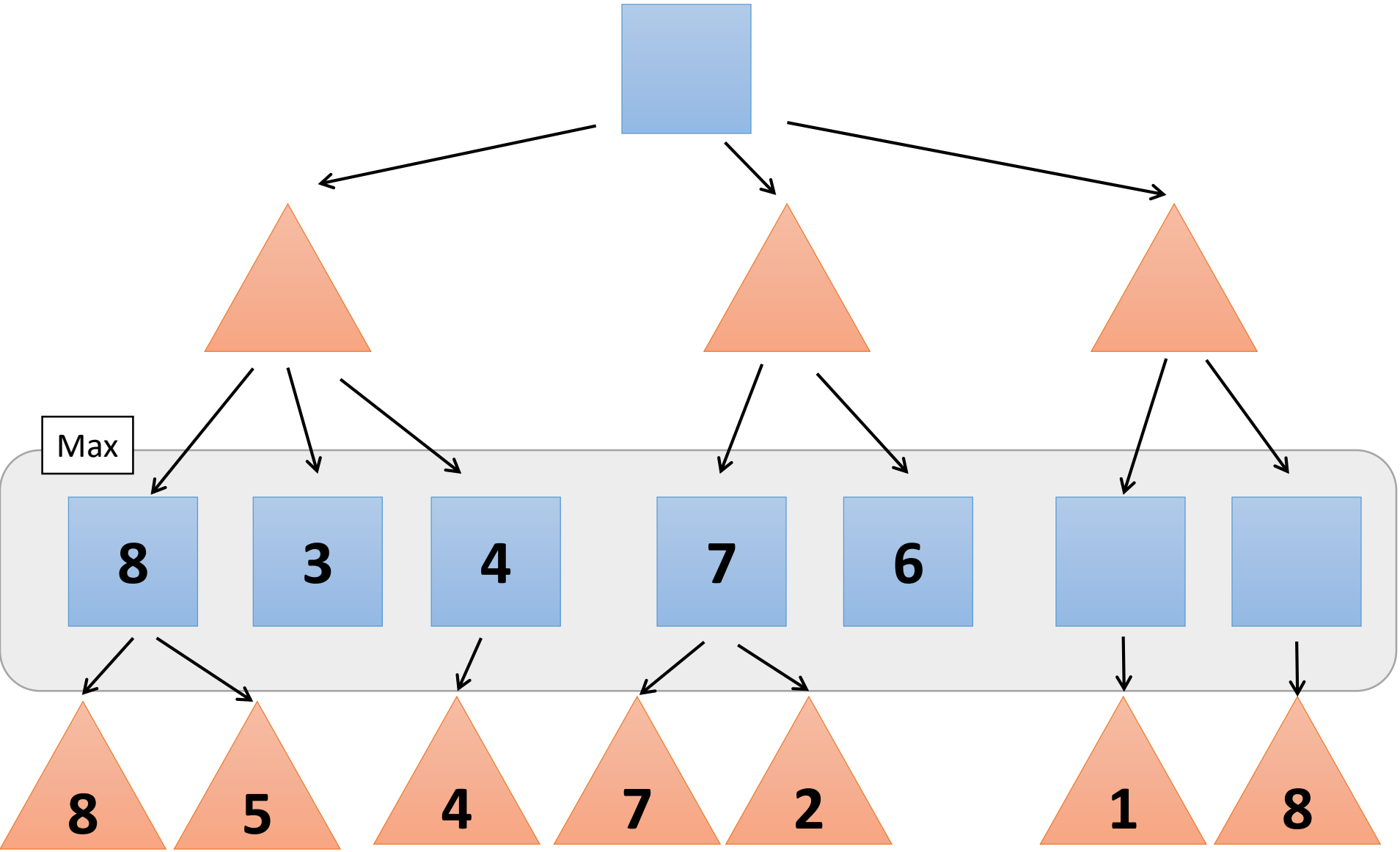
Minimax



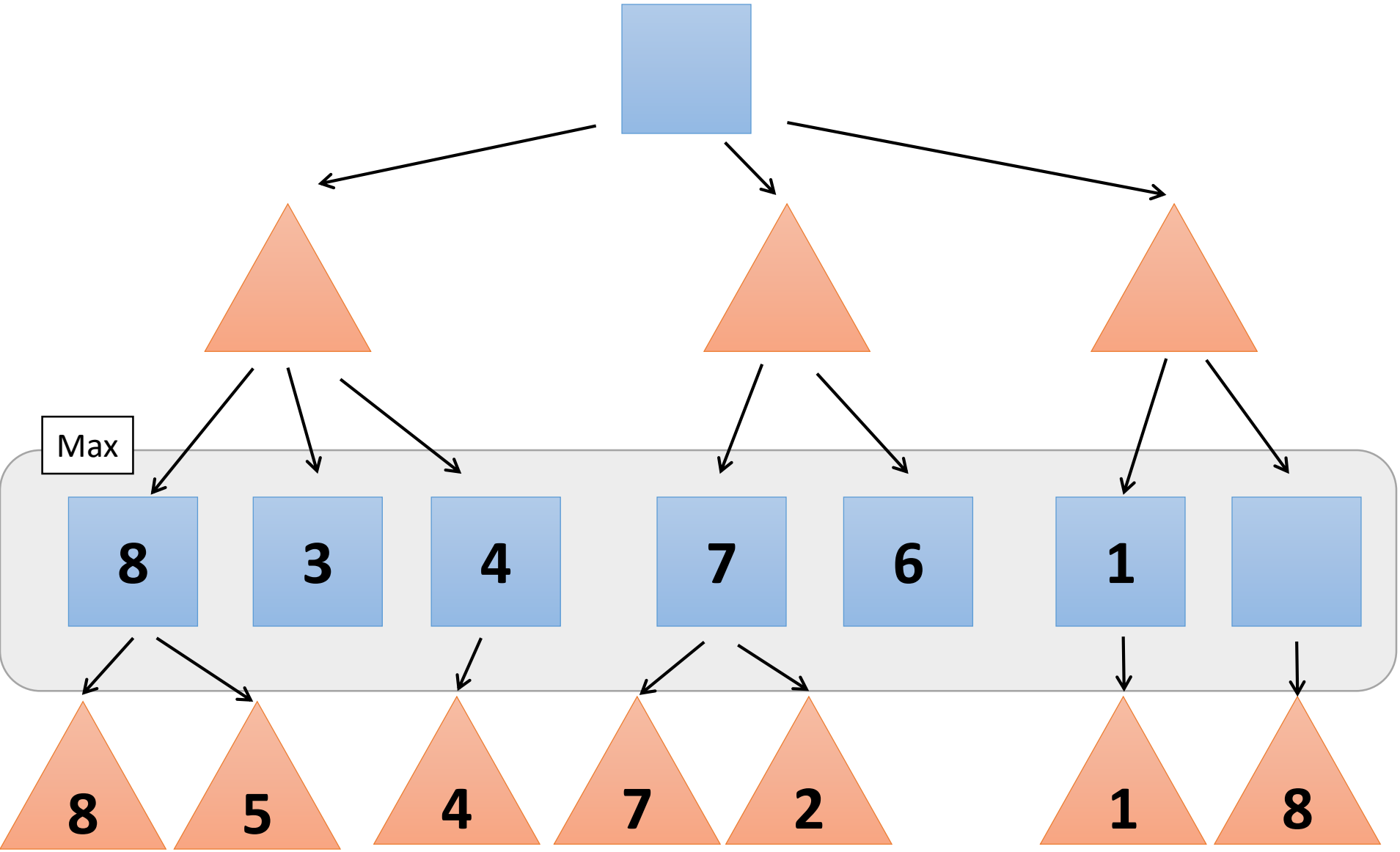
Minimax



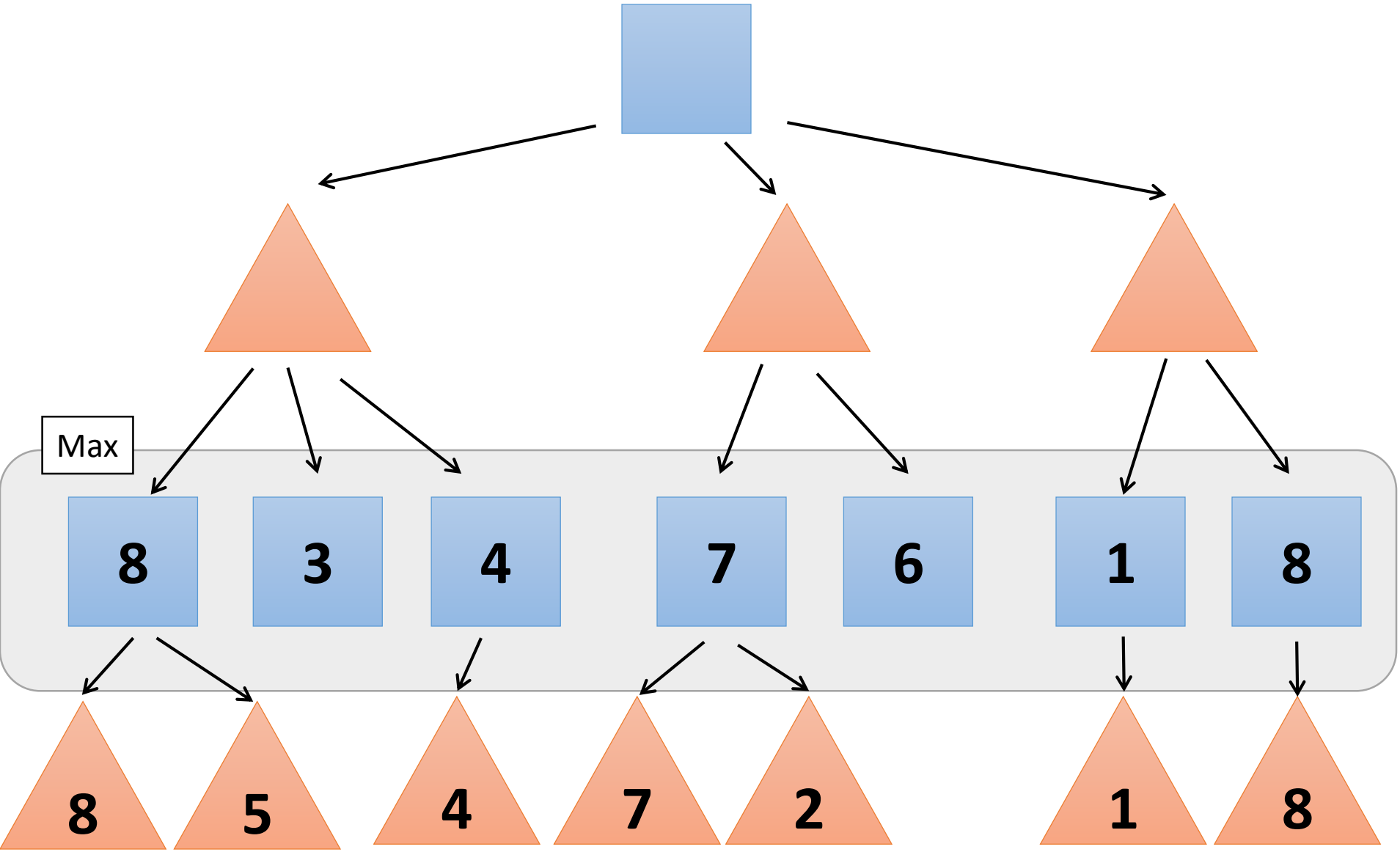
Minimax



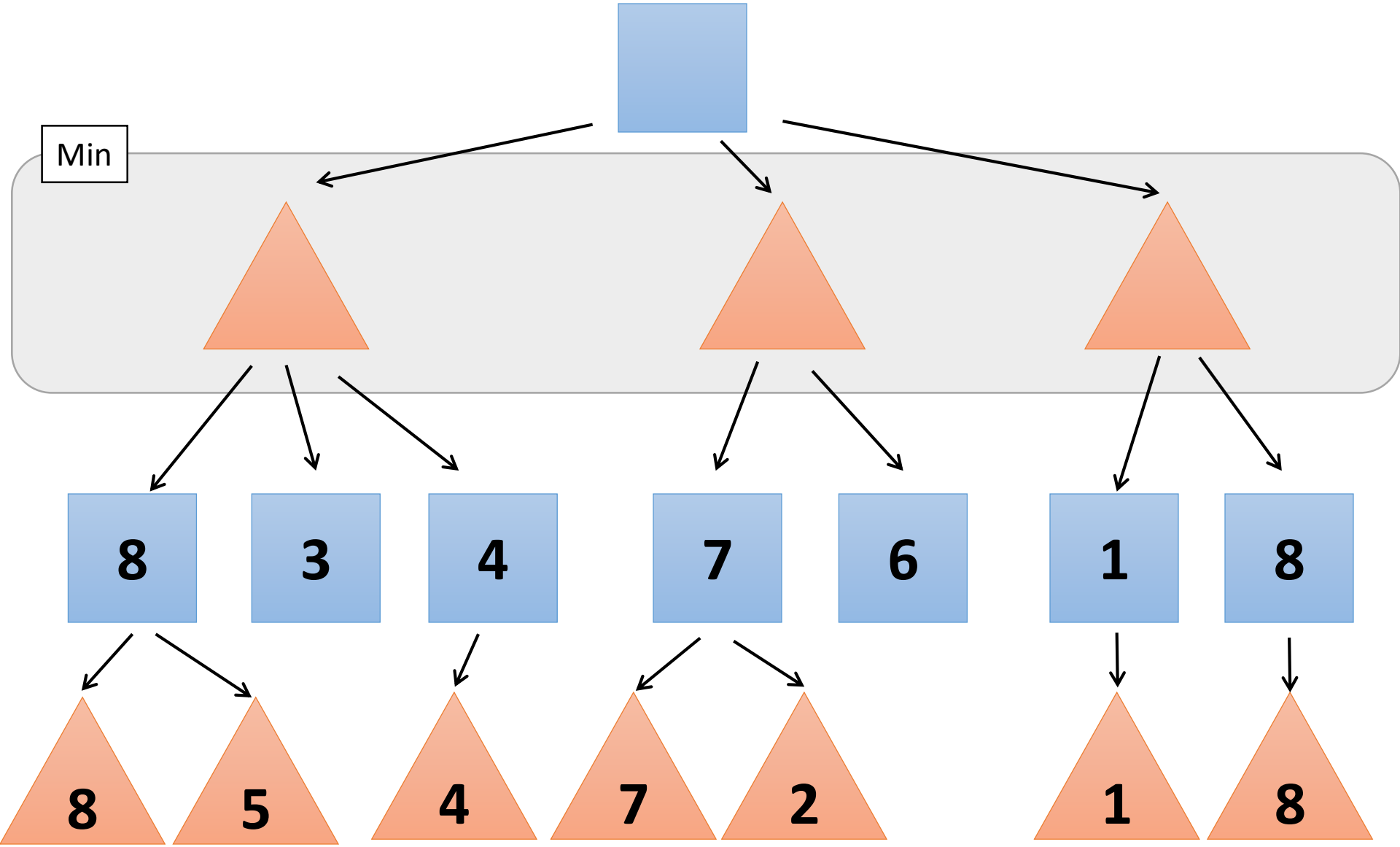
Minimax



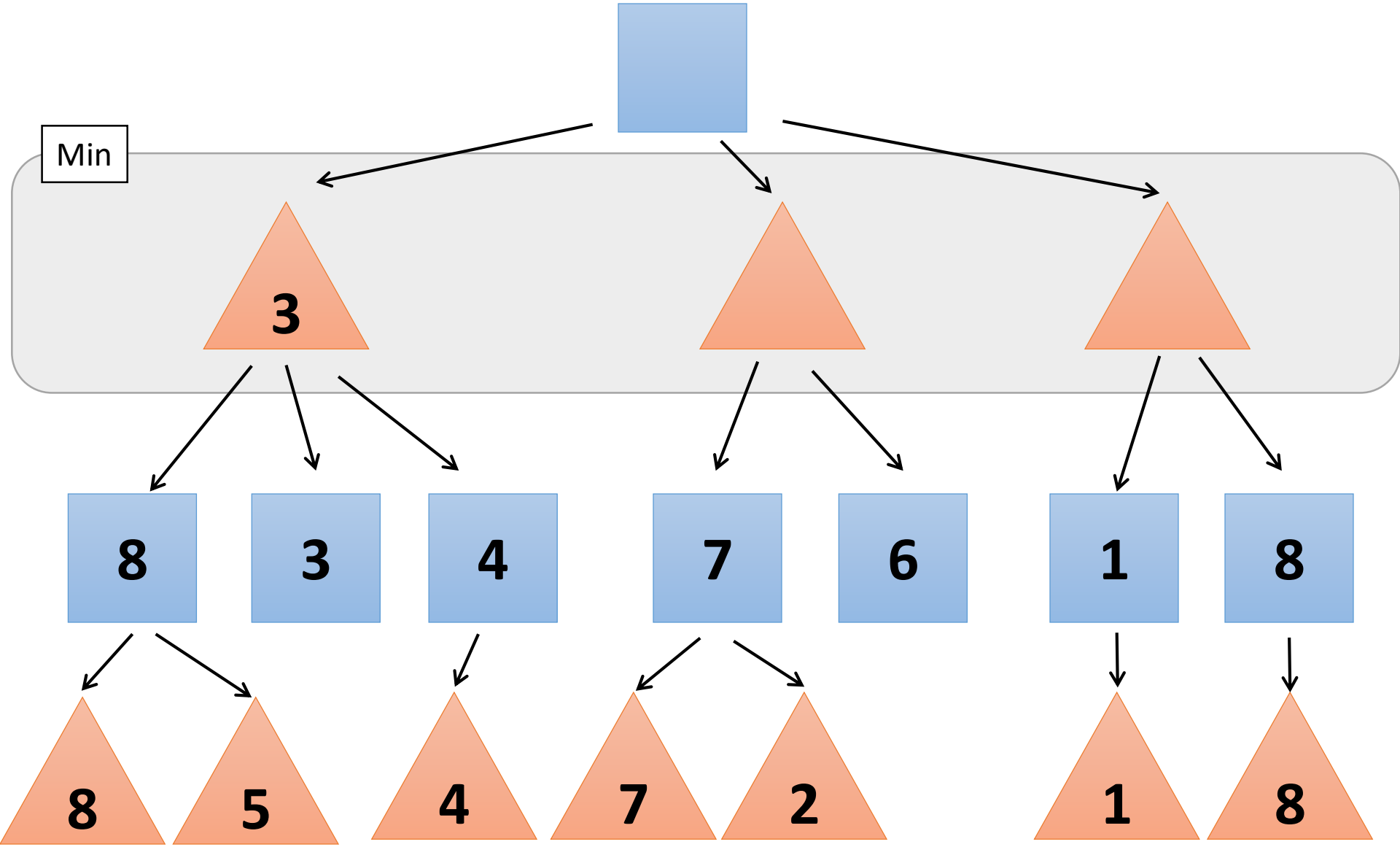
Minimax



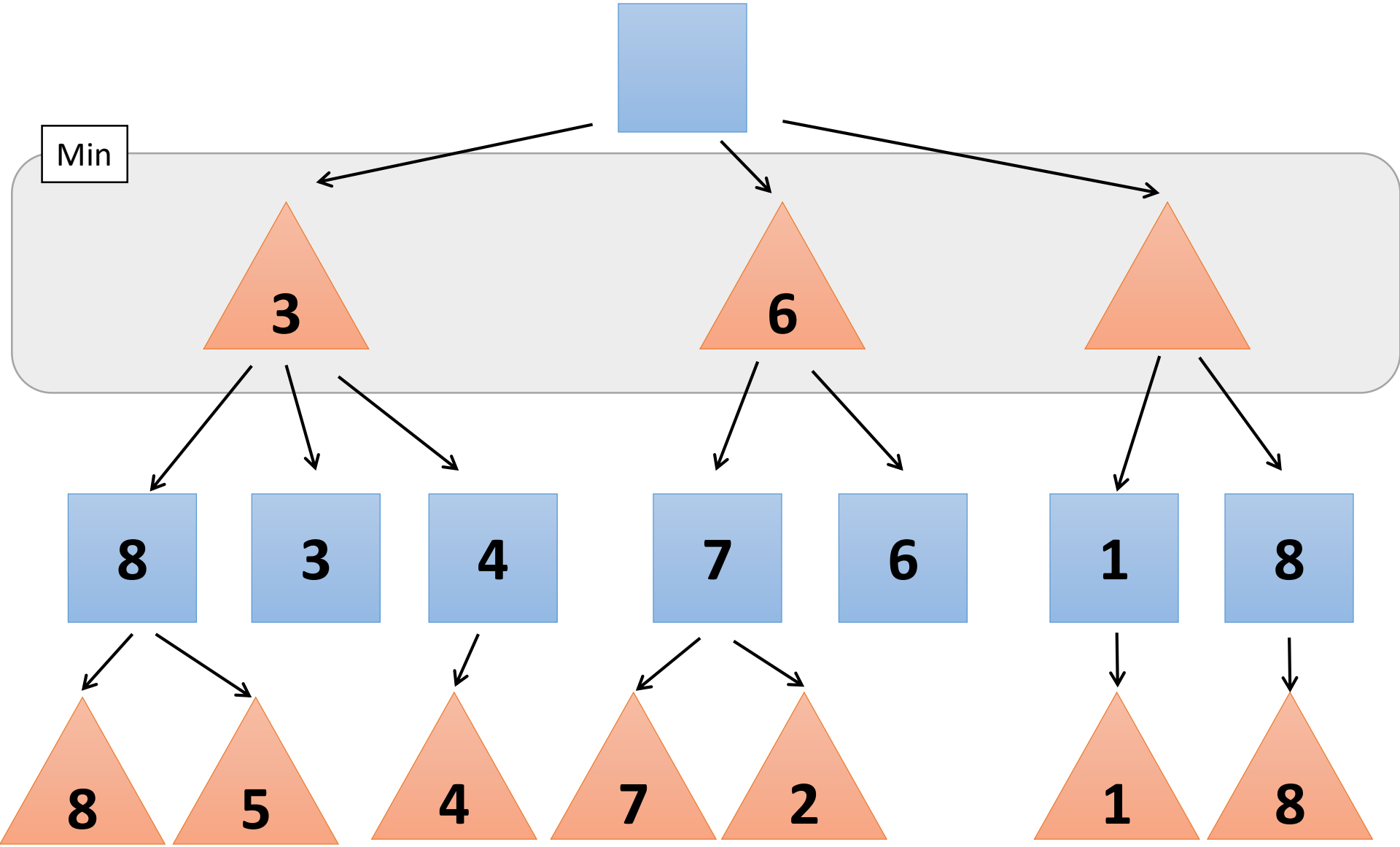
Minimax



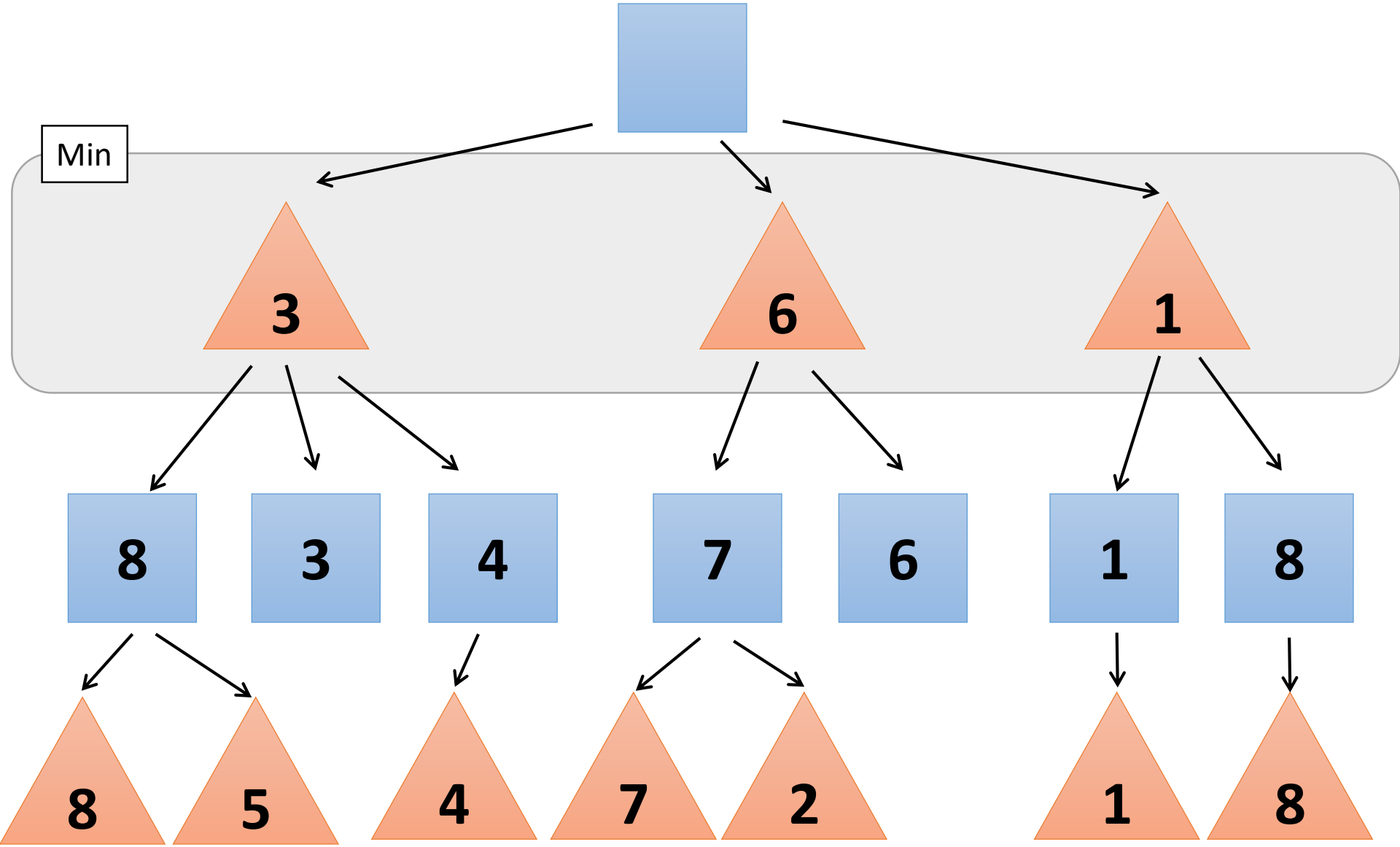
Minimax



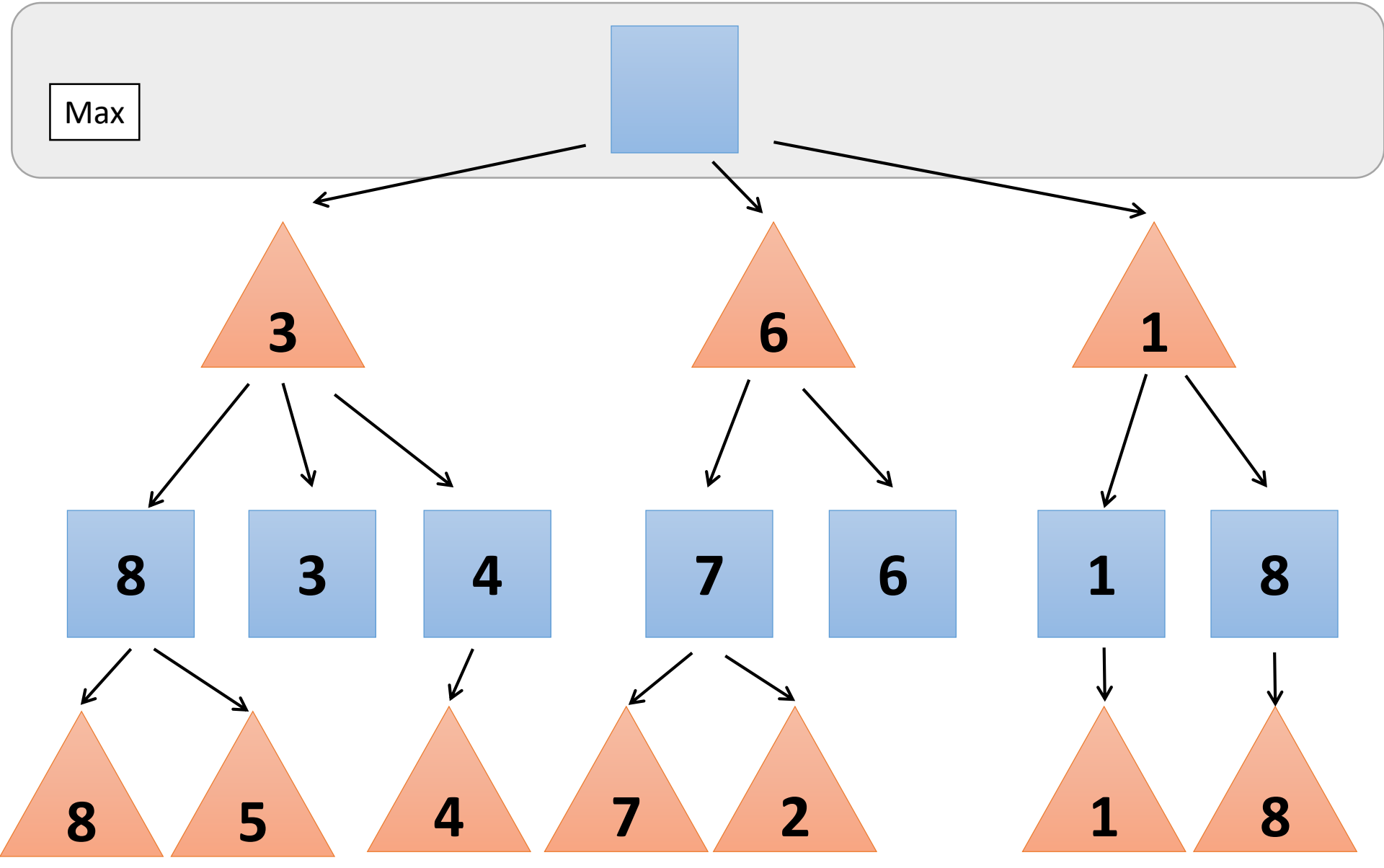
Minimax



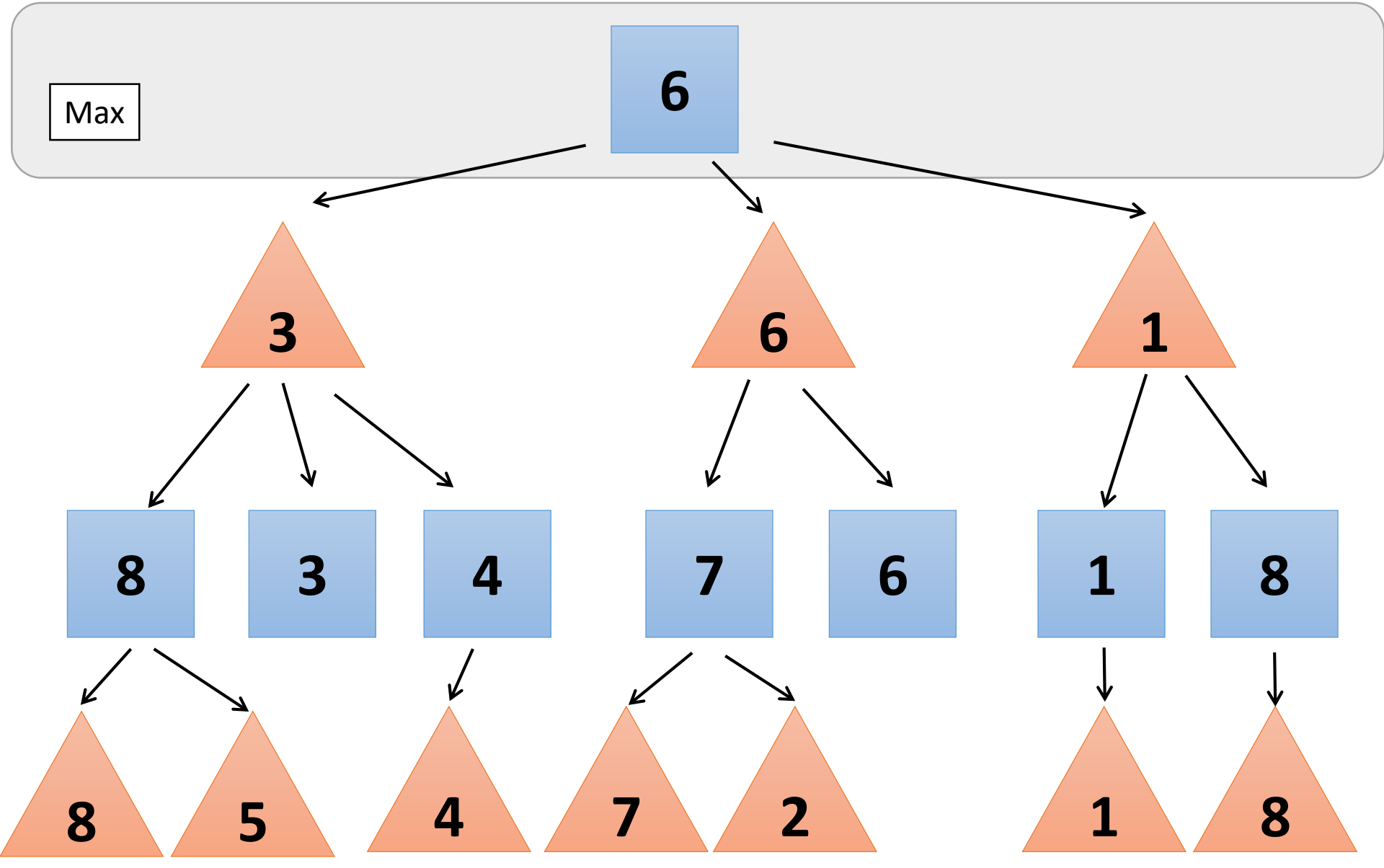
Minimax



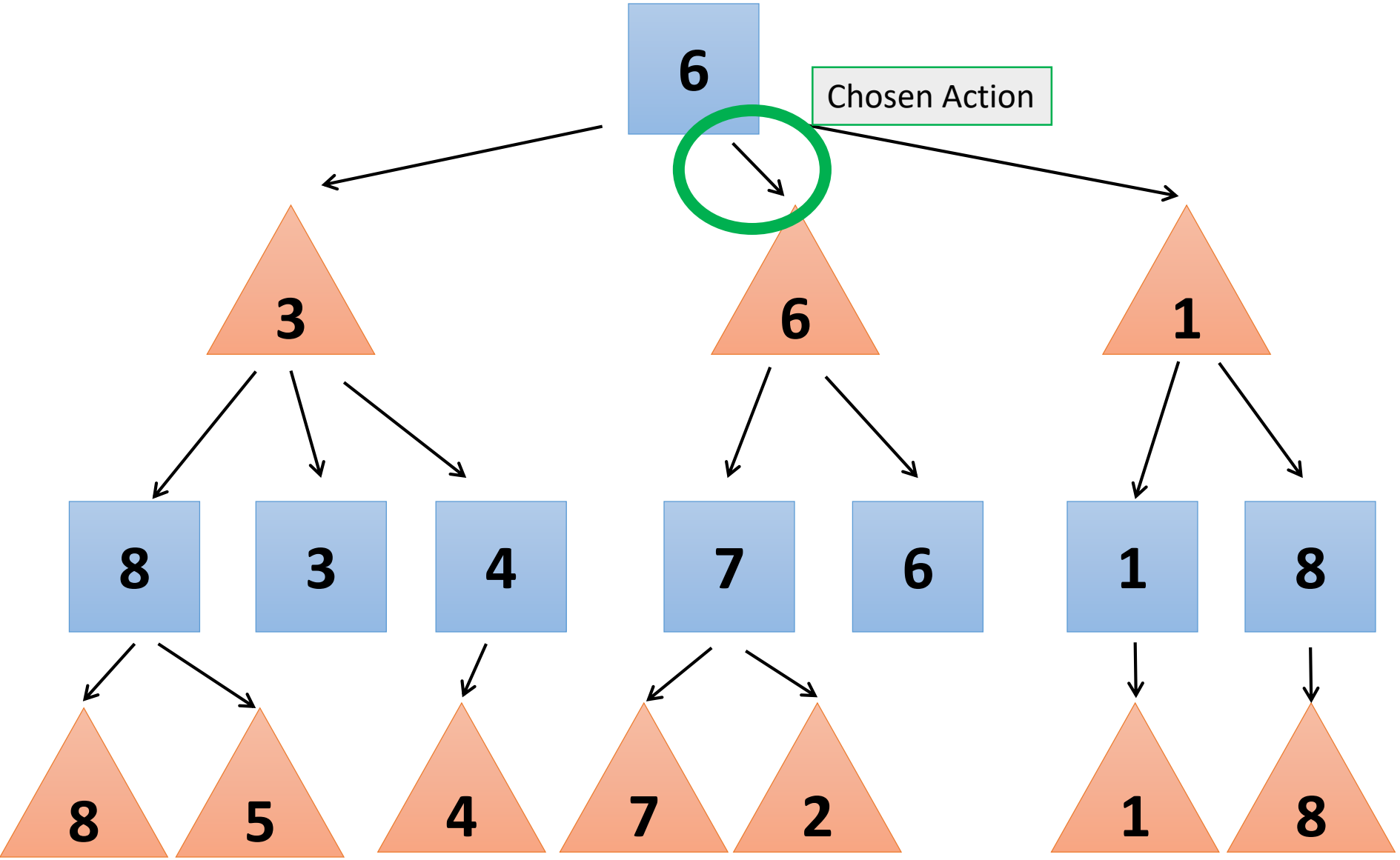
Minimax



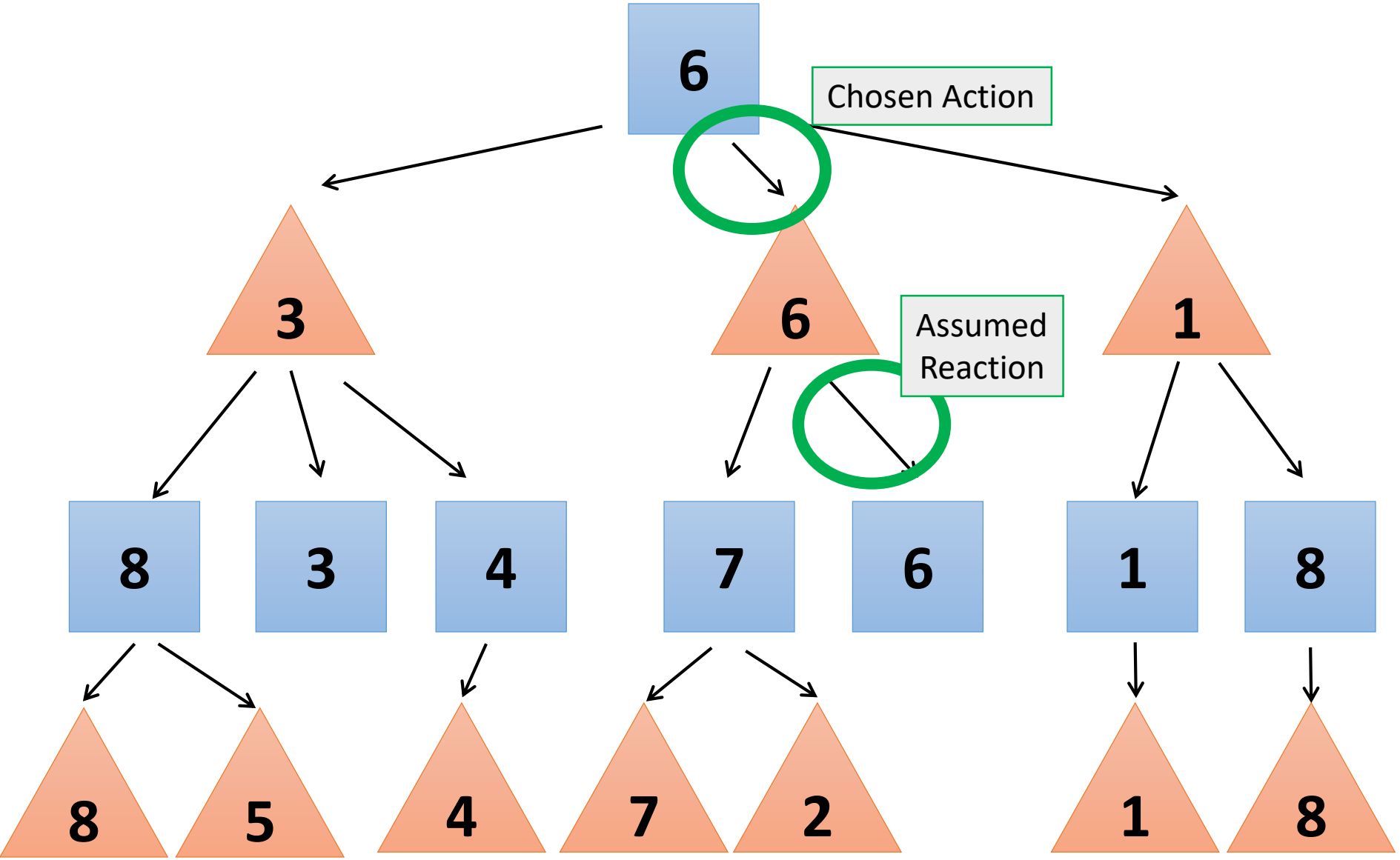
Minimax



Minimax



Minimax



Multi-player Games

- Extend the minimax idea to **multiplayer games**.
- Replace single value for each node with **a vector of values**.
  - E.g., in a three-player game with players *A*, *B*, and *C*, a vector (v_A,v_B,v_C) is associated with each node.
- For terminal states, this vector gives the utility of the state from each player's viewpoint. The simplest way to implement this is to have the UTILITY function return a vector of utilities.

- Multiplayer games usually involve formal or informal **alliances** among the players.
- Alliances are made and broken as the game proceeds.

Adversarial Search:

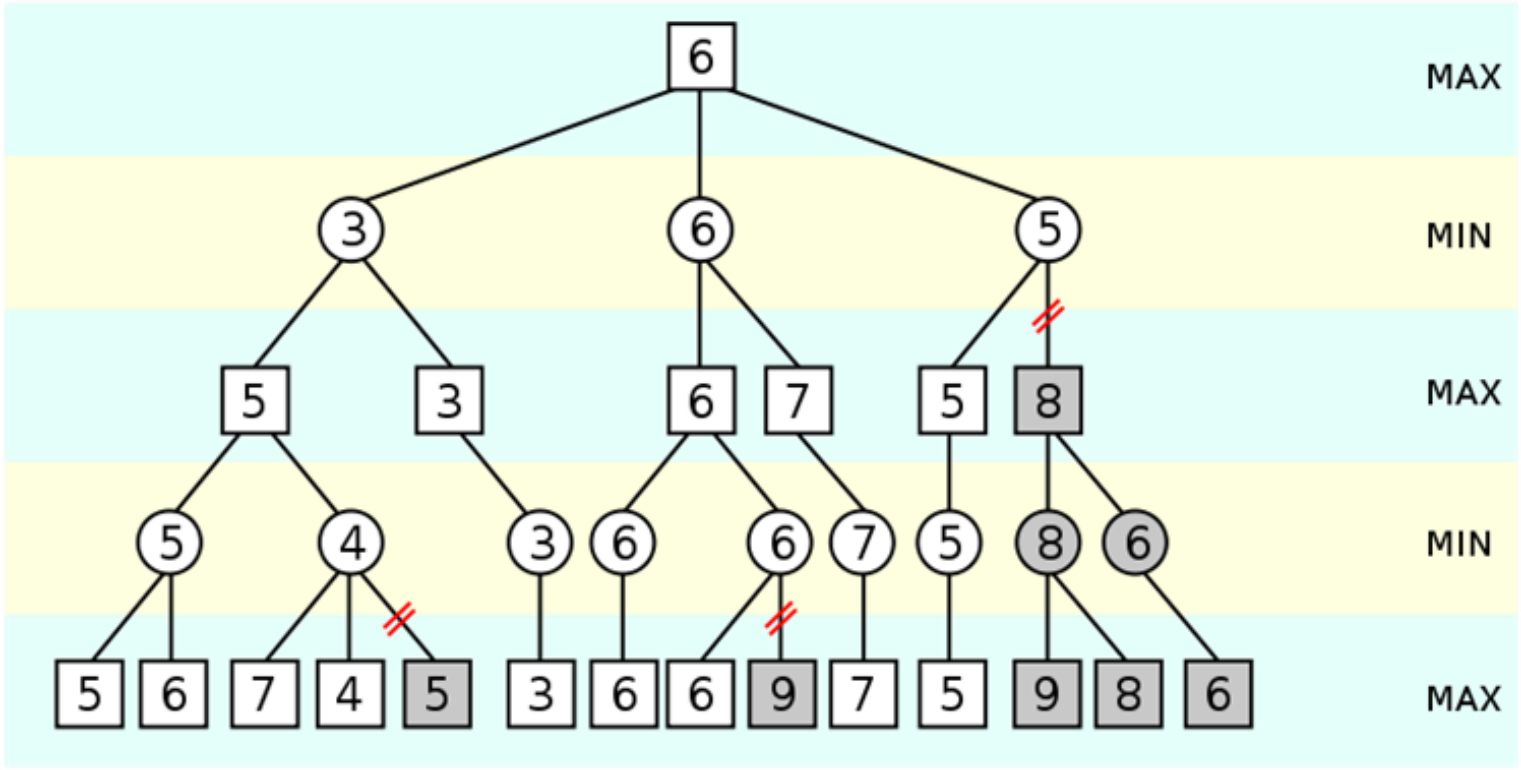
Games

Alpha-Beta Pruning

Stochastic Games

Problems with Minimax

- Number of game states is **exponential** in depth of the tree
- Cannot eliminate the exponent, but we can effectively **cut** it



http://geekswithblogs.net/JoshReuben/archive/2012/10/10/hpc-server-dynamic-job-scheduling-when-jobs-spawn-jobs.aspx

How to Solve It?

- *If you have an idea that is surely bad, don't take the time to see how truly awful it is*. -- Pat Winston (Director, MIT AI Lab, 1972-1997)

- The trick to solve the problem:
  - Compute correct minimax decision without looking at every node in game tree.
  - That is, use "pruning" to eliminate large parts of the tree.

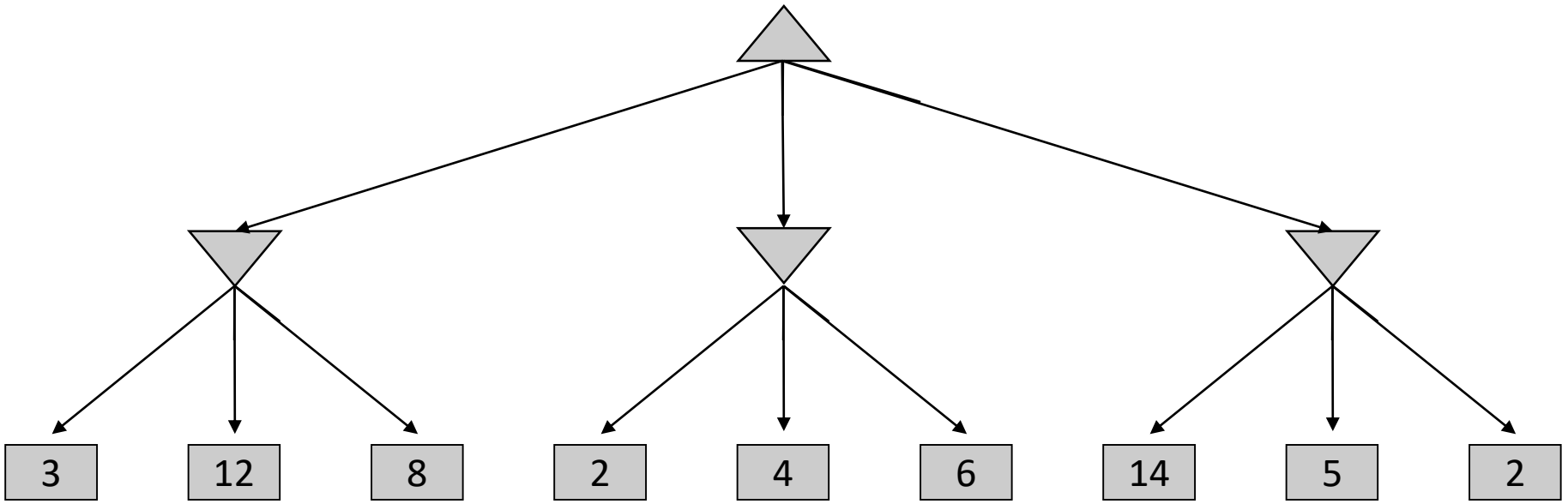
- **Alpha-Beta pruning**
  - A search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm.

Alpha-Beta Pruning

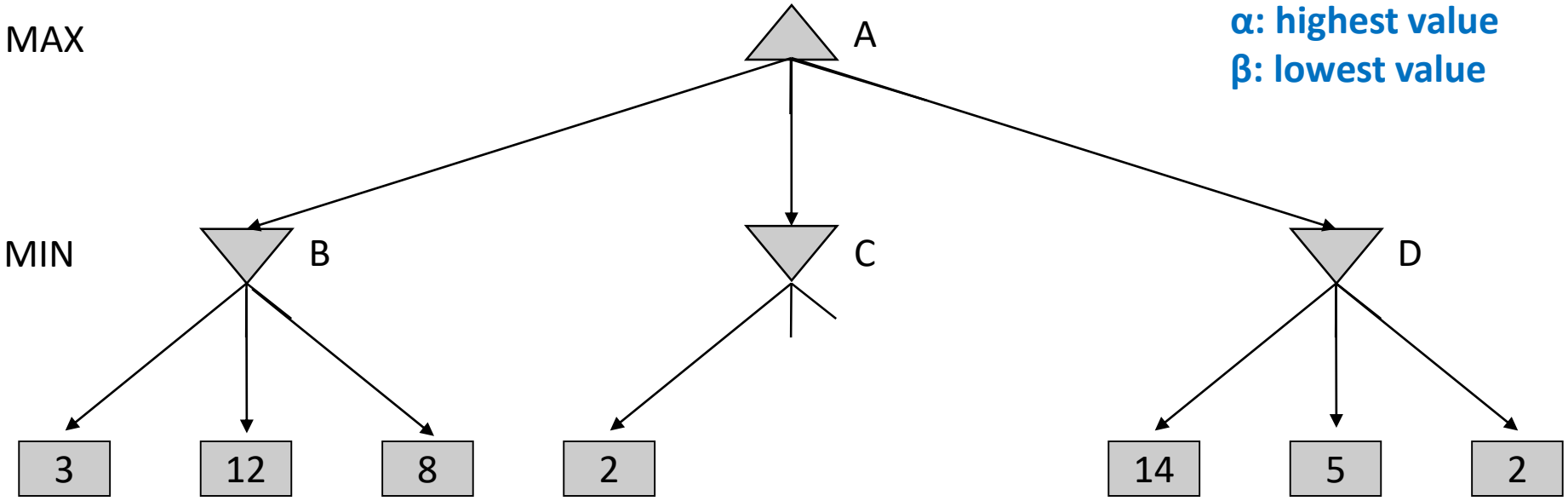
- Alpha-beta pruning gets its name from the following two parameters:
  - α: highest value we have found so far at any point along the path for **MAX**
  - β: lowest value we have found so far at any point along the path for **MIN**

- Alpha-beta search respectively:
  - **Updates** the values of α and β as it goes along
  - **Prunes** the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for **MAX** or **MIN**.

Example: without Pruning



Example: with Pruning



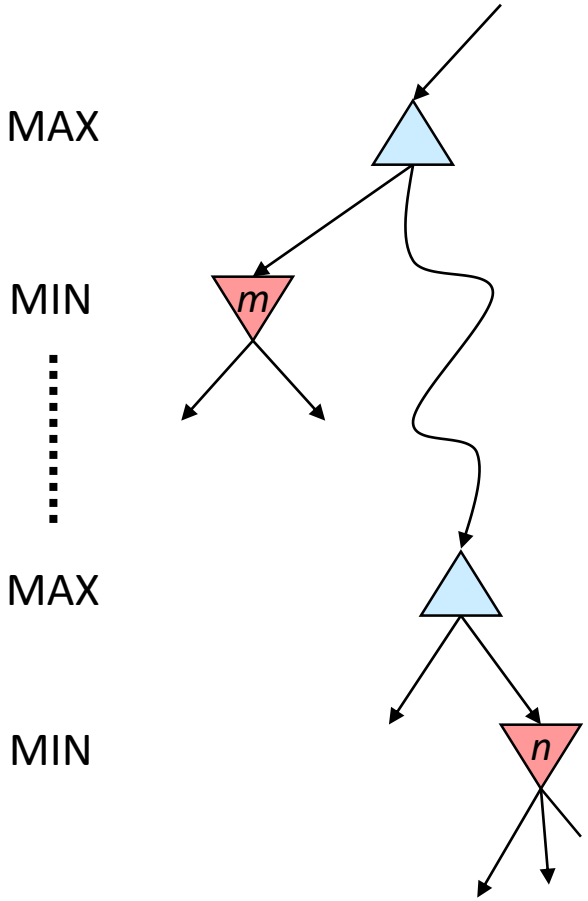
A[α=-∞, β=+∞]
B[α=-∞, β=3]
A[α=3, β=+∞]
B[α=3, β=3]
C[α=-∞, β=2]
A[α=3, β=2]
D[α=-∞, β=14]
D[α=2, β=2]

The value of root node is given by:

MINIMAX(*root*) = max(min(3,12,8), min(2,*x*,*y*), min(14,5,2))
=max(3, min(2,*x*,*y*), 2)
=max(3,*z*,2) where $z = \min(2,x,y) \leq 2$
=3

General Principle

- Alpha-beta pruning can be applied to trees of any depth, and often possible to prune **entire subtrees** rather than just leaves.
- The general principle:
  - Consider a node *n* somewhere in the tree, such that player has a choice of moving to that node.
  - If player has a **better** choice *m* at parent node of *n*, or at any choice point further up, then *n* will **never** be reached in actual play.



Imperfect Real-Time Decision

- Alpha-beta still has to search all the way to terminal states for at least a portion of the search space.
- This depth is usually **not practical**, because moves must be made in a reasonable amount of time.
- Claude Shannon:

  *Programs should cut off the search earlier and apply a **heuristic evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves.*

- The suggestion is in two ways:
  - Use **EVAL** instead of **UTILITY**
  - Use **CUTOFF-TEST** instead of **TERMINAL-TEST**

Evaluation Functions

- An **evaluation function** returns an **estimate of the expected utility** of the game from a given position.

- How to design good evaluation functions?
  - It should order the terminal states in the same way as the true utility function:
    - The states that are **wins** must evaluate better than **draws**
    - The states that are **draws** must evaluate better than **losses**
  - The computation must not take too long
  - Nonterminal states should be strongly correlated with actual chances of wining.

Weighted Linear Function

- A kind of evaluation function

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

where

w_i : a weight

f_i : a feature of the position

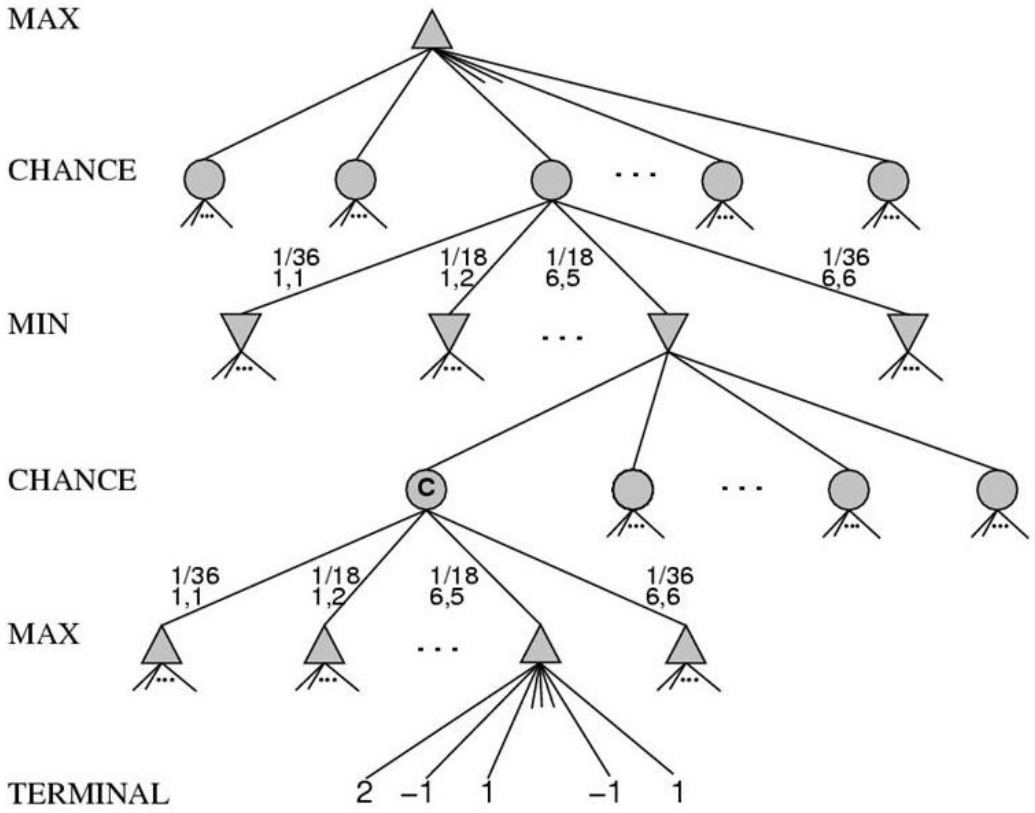
- For Chess:
  - f_i could be the numbers of each kind of piece on the board
  - w_i could be the values of the pieces: 1 for pawn (兵), 3 for bishop (象), *etc*.

Adversarial Search:

Games
Alpha-Beta Pruning
Stochastic Games

Nonopoly & Backgammon

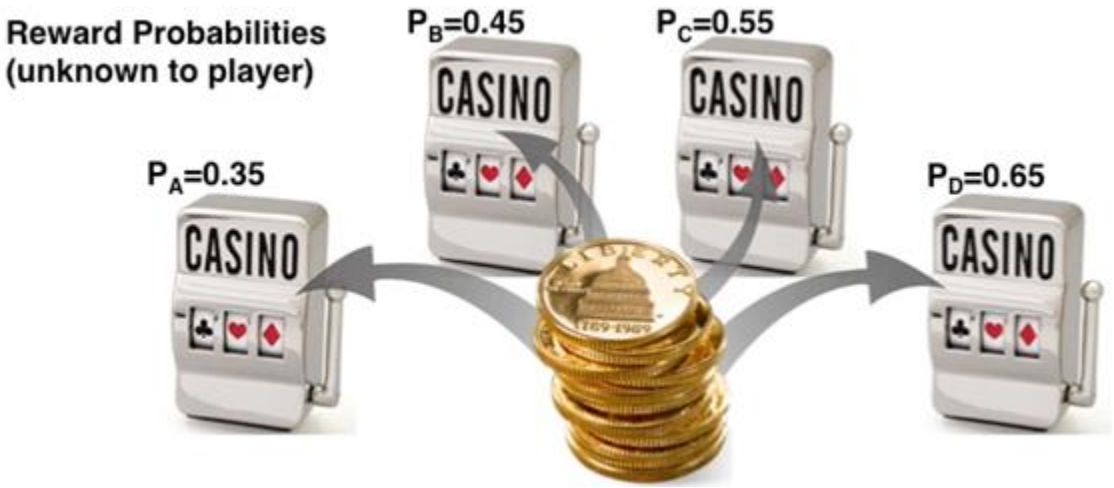
Game Tree of Backgammon



- There are 36 ways to roll two dice, each equally likely; but because a (6,5) is the same as a (5,6), there are only 21 distinct rolls.
- The six doubles, (1,1) through (6,6), each have a probability of **1/36**.
- The other 15 distinct rolls each have a **1/18** probability.

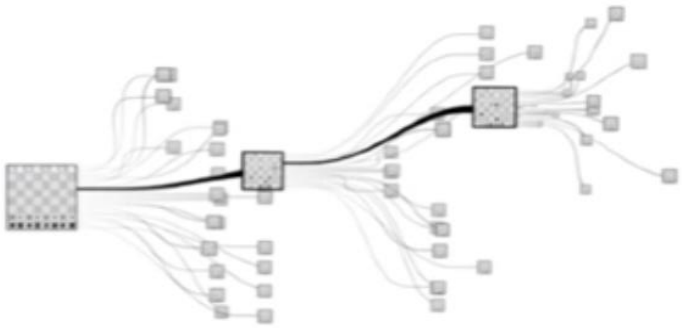
Multi-Armed Bandit

- A gambler faces several slot machines (one-armed bandits), that look identical but produce different expected winnings.
- The issue is a **trade-off** between **acquiring new information** and **capitalizing on the information available so far**.
- One aspect that we are interested in concerns **modeling** and **efficiently using various types of side information** that may be available to the algorithm.



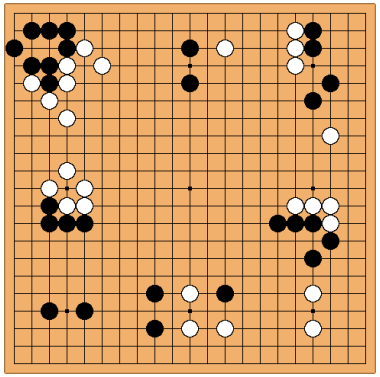
Go vs. Chess

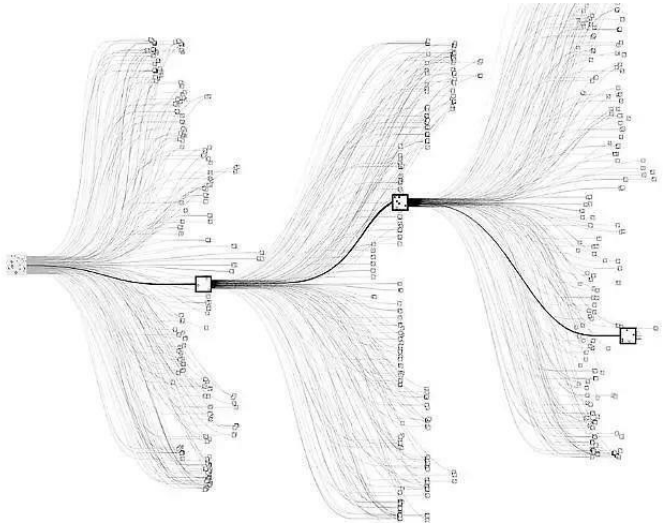
- Go has long been viewed as one of most complex game and most challenging of classic games for AI.



Chess:

- b≈35, d≈80
- 8×8 = 64
- Possible games ≈ 10^{120}





Go:

- b≈250, d≈150
- 19×19 = 361
- Possible games ≈ 10^{170}

Go vs. Chess

- **Deep neural networks**
  - **Value networks**: to evaluate board positions
  - **Policy networks**: to select moves
- **Monte-Carlo Tree Search (MCTS)**
  - Combines Monte-Carlo simulation with value networks and policy networks
- **Reinforcement learning**
  - Used to improve its play

AlphaGo

Mastering the game of Go with deep neural networks and tree search
Nature, Jan. 28, 2016

Go vs. Chess

ALPHAGO ZERO CHEAT SHEET

The training pipeline for AlphaGo Zero consists of three stages, executed in parallel

SELF PLAY

Create a 'training set'

The best current player plays 25,000 games against itself

See MCTS section for understanding how AlphaGo Zero selects each move

At each move, the following information is stored

The game state
(see 'What is a Game State' section)

The search probabilities
(from the MCTS)

The winner
(+1 if the player won, -1 if the player lost - added once the game has finished)

RETRAIN NETWORK

Optimise the network weights

A TRAINING LOOP

Sample a mini-batch of 2048 positions from the last 500,000 games

Retrain the current neural network on these positions
- The game states are the input (see 'Deep Neural Network Architecture')

Loss Function
Compares predictions from the neural network with the search probabilities and actual winner

After every 1,000 training loops, evaluate the network

EVALUATE NETWORK

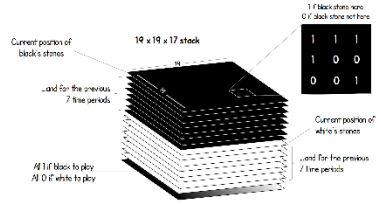
Test to see if the new network is stronger

Play 400 games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

Latest player must win 55% of games to be declared the new best player

WHAT IS A 'GAME STATE'



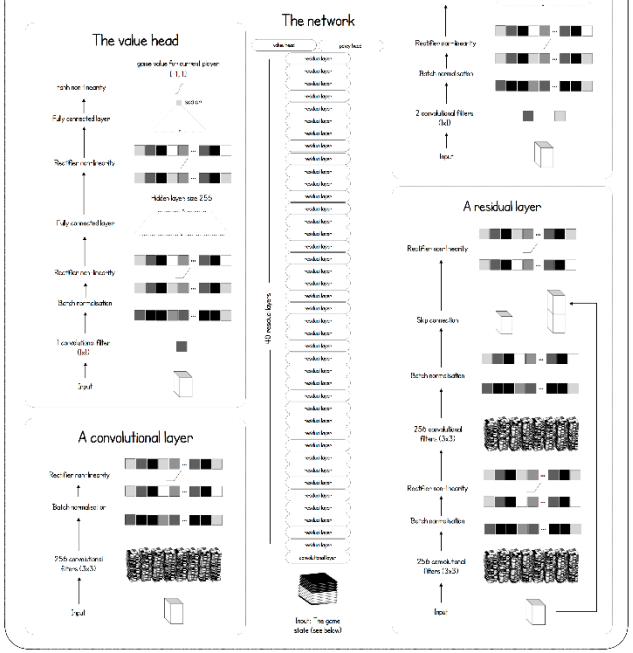
This stack is the input to the deep neural network

THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

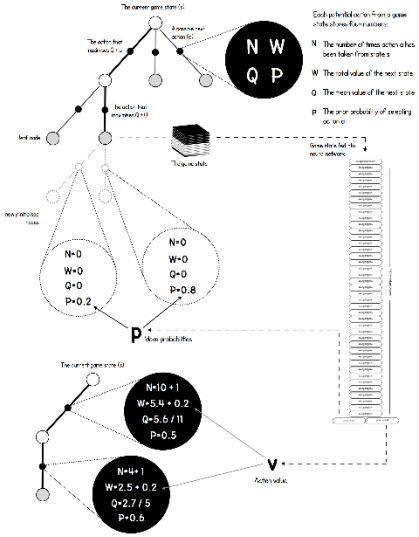
The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves



MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move



First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

1. Choose the action that maximises...

$$Q + U$$

Q: The mean value of the next state

U: A function of P and N that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later Q is more important (less exploration)

2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

p Move probabilities

v Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$$N \rightarrow N + 1$$
$$W \rightarrow W + v$$
$$Q = W / N$$

...then select a move

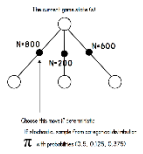
After 1,600 simulations, the move can either be chosen:

Deterministically (for competitive play)
Choose the action from the current state with greatest N

Stochastically (for exploratory play)
Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where τ is a temperature parameter controlling exploration



Other points

- The sub-tree from this chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

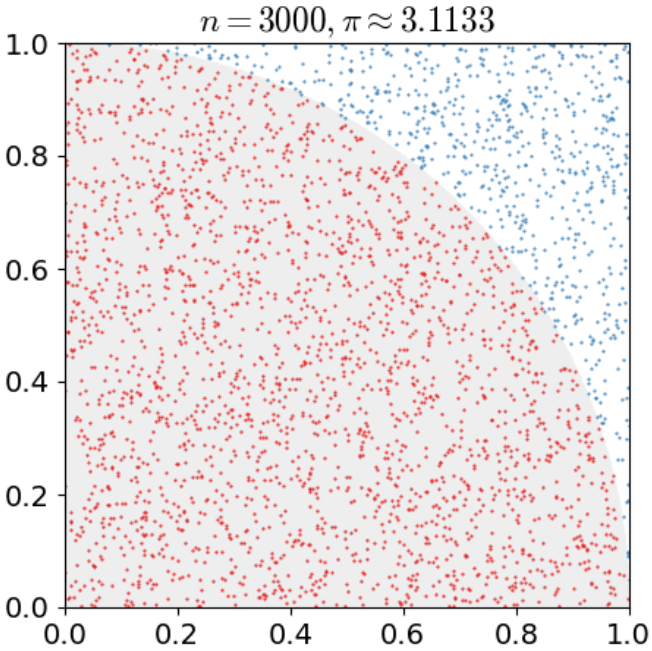
Monte-Carlo Method

- A broad class of computational algorithms that rely on **repeated random sampling** to obtain numerical results.
- Most useful when it is difficult or impossible to use other mathematical methods
- They tend to follow a particular pattern:
  1. Define a domain of possible inputs
  2. Generate inputs randomly from a probability distribution over the domain
  3. Perform a deterministic computation on the inputs
  4. Aggregate the results

Example: Approximating π

- Given a circle and square with a ratio of areas $\pi/4$, the value of π can be approximated by:

1. Draw a square, and then inscribe a circle within in.
2. Uniformly scatter some objects of uniform size over the square.
3. Count the number of objects inside the circle and the square.
4. The ratio of the two counts is an estimate of the ratio of the two areas, which is $\pi/4$.
5. Multiply the result by 4 to estimate π.



Family of Monte-Carlo Methods

- Classical Monte-Carlo 经典蒙特卡罗
  - Samples are drawn from a probability distribution, often the classical Boltzmann distribution
- Quantum Monte-Carlo 量子蒙特卡罗
  - Random walks are used to compute quantum-mechanical energies and wave functions
- Volumetric Monte-Carlo 容积式蒙特卡罗
  - Random number generators are used to generate volumes per atom or to perform other types of geometrical analysis
- Kinetic Monte-Carlo 动力学蒙特卡罗
  - Simulate processes using scaling arguments to establish timescales or by introducing stochastic effects into molecular dynamics

Monte-Carlo Tree Search

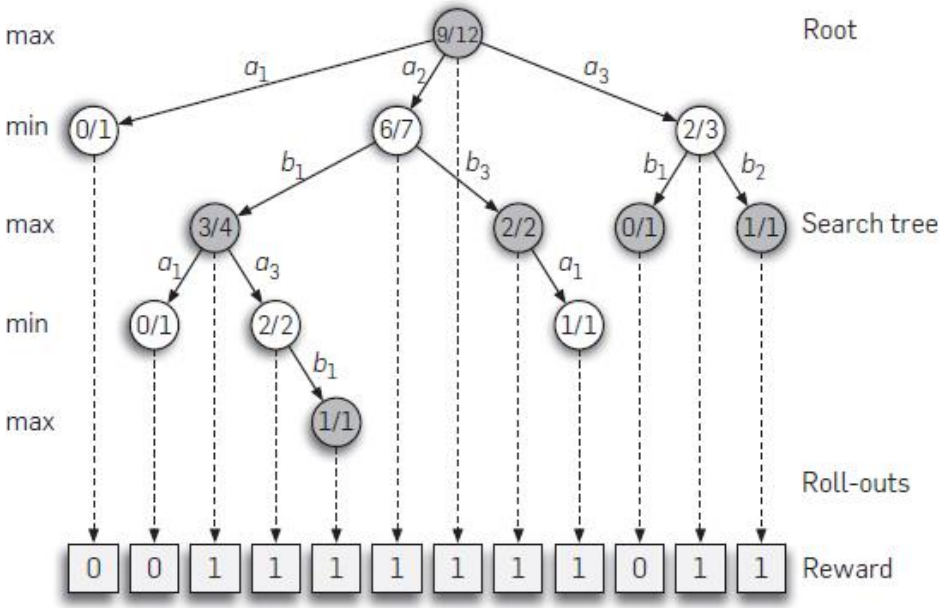
- MCTS combines Monte-Carlo simulation with game tree search.
  - **Like** minimax, each node corresponds to a single state of game.
  - **Unlike** minimax, the values of nodes are estimated by Monte-Carlo simulation.

$W(a)/N(a)$ = the value of action a

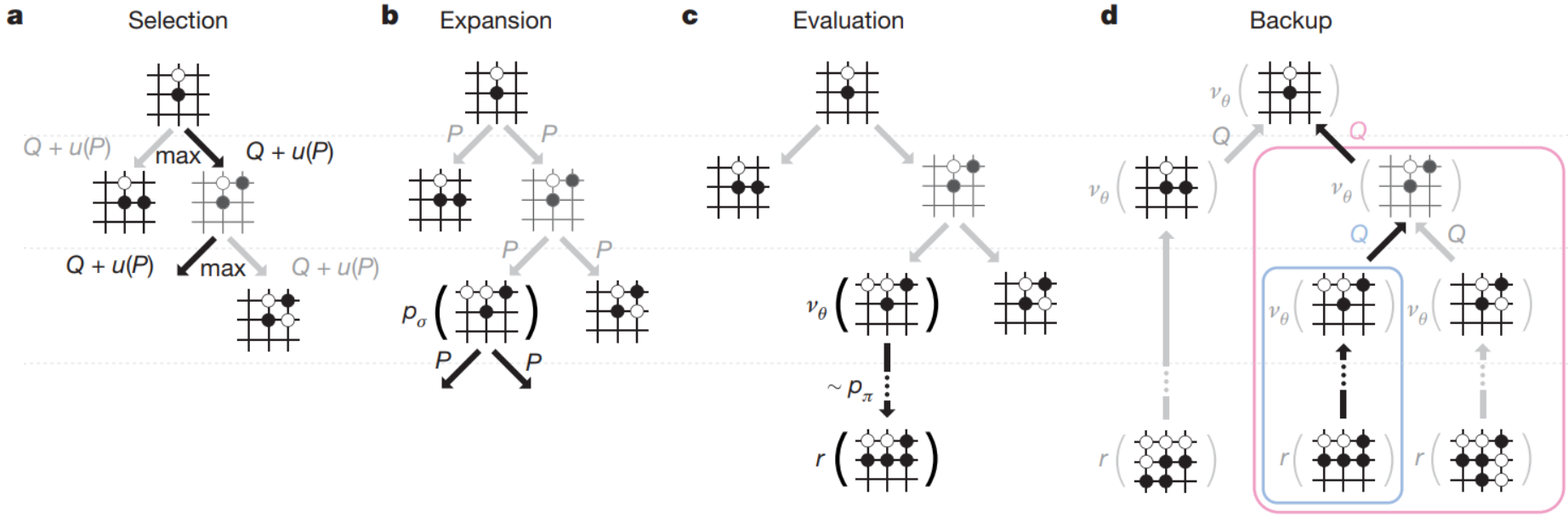
where:

$W(a)$ = the total reward

$N(a)$ = the number of simulations

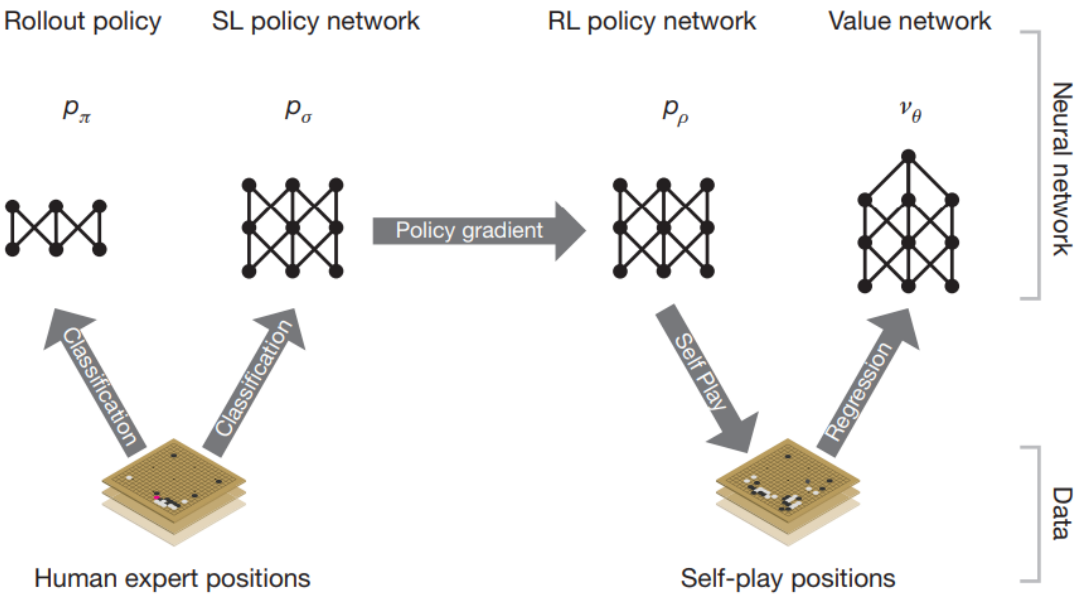


MCTS in AlphaGo



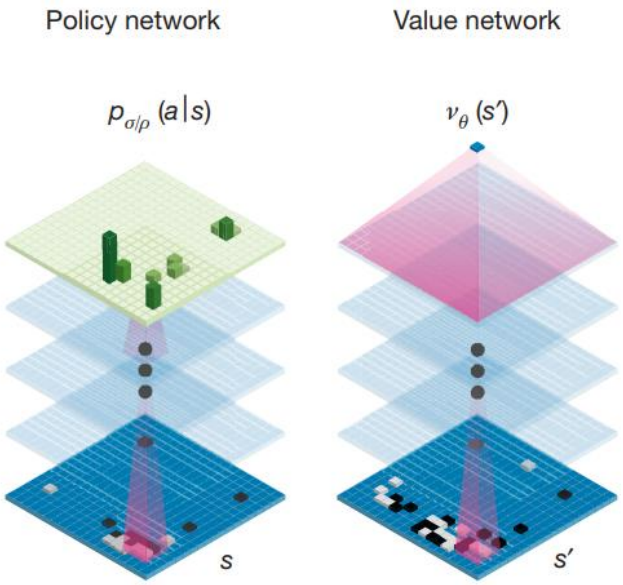
a, Each simulation **traverses** the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge.
b, The leaf node may be **expanded**; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action.
c, At the end of a simulation, the leaf node is **evaluated in two ways**: using the value network v_ϑ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r.
d, Action values Q are **updated** to track the mean value of all evaluations $r(\cdot)$ and $v_\vartheta(\cdot)$ in the subtree below that action.

Training Pipeline in AlphaGo



- A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions.
- A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network.
- A new data set is generated by playing games of self-play with the RL policy network.
- Finally, a value network v_ϑ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set.

Training Pipeline in AlphaGo



- Schematic representation of the neural network architecture used in AlphaGo.
- The **policy network**:
  - **Input**: a representation of the board position *s*.
  - **Process**: passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network).
  - **Output**: a probability distribution over legal moves *a*, represented by a probability map over the board.
- The **value network**:
  - **Input**: a representation of the board position *s*.
  - **Process**: similarly uses many convolutional layers with parameters ϑ.
  - **Output**: a scalar value $v_\vartheta(s')$ that predicts the expected outcome in position *s'*.

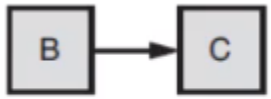
Constraint Satisfaction Problem

What is CSP?

- **Constraint Satisfaction Problem (CSP)** is a mathematical problem, defined as a set of objects whose state must satisfy a number of constraints or limitations.
- CSP represents the entities in a problem as **homogeneous collection of finite constraints over variables**, which is solved by **constraint satisfaction methods**.
- CSP is the research subject in both **artificial intelligence** and **operations** research
  - The regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families.

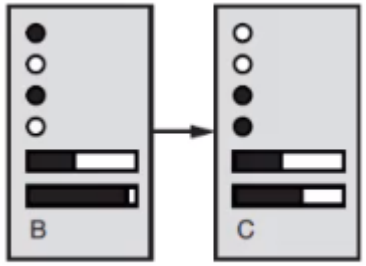
Standard Search vs. CSP

- **Standard Search Problem**
  - The state is **atomic** or indivisible, a **black box** with arbitrary data structure.
  - Goal test can be any function over states
  - Successor function can also be anything



Atomic

- **Constraint Satisfaction Problem**
  - The state is a **factored** representation, a set of variables, each of which has a value.
  - Take advantage of the **structure** of states.
  - Use **general-purpose** rather than **problem-specific heuristics**.



Factored

CSP

- CSPs often exhibit **high complexity**, requiring a combination of **heuristics** and **combinatorial search** methods

- Some certain forms of the CSP:
  - Boolean Satisfiability Problem (SAT) 布尔可满足性问题
  - Satisfiability Modulo Theories (SMT) 可满足性模理论
  - Answer Set Programming (ASP) 答案集编排

- Examples that can be modeled as a CSP:
  - 8-queens puzzle
  - Map coloring problem
  - Cryptarithmetic
  - Sudoku

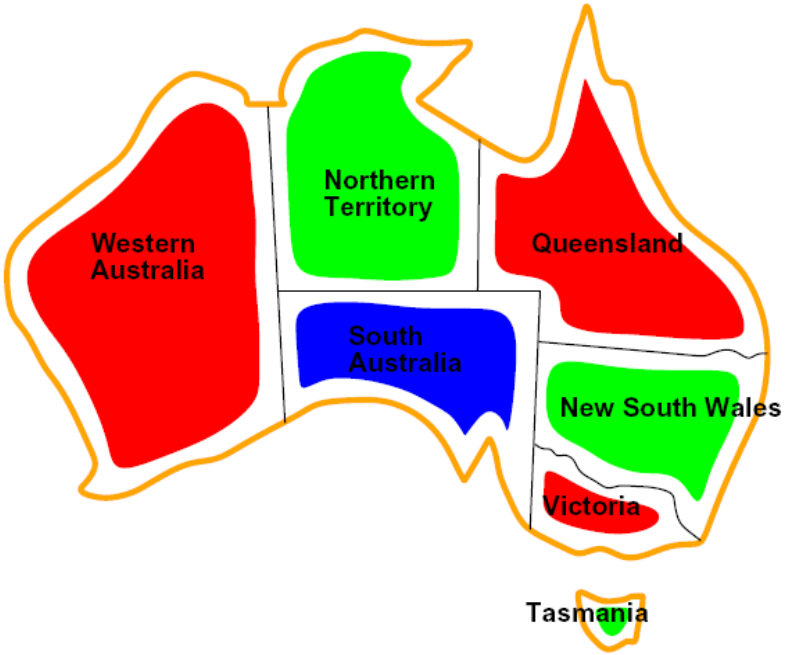
Formal Definition of CSP

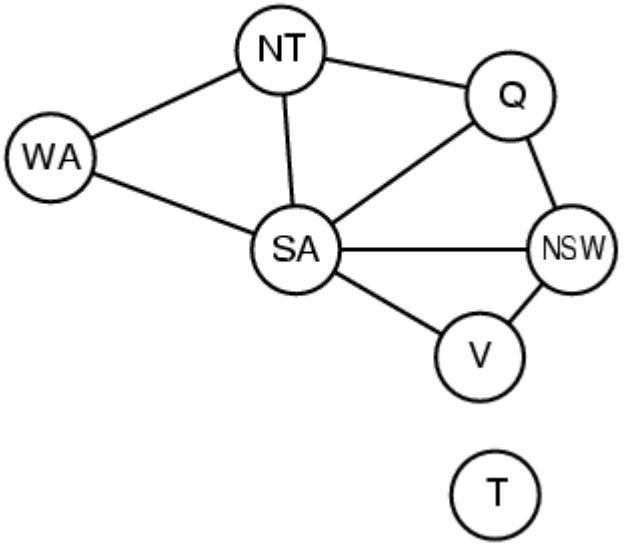
- A CPS is defined as a triple <X,D,C>, where
  - X is a set of **variables** {X_1, …, X_n}
  - D is a set of **domains** {D_1, …, D_n}
    - One D_i for each variable X_i, consisting of a set of values {V_1, …, V_k}
  - C is a set of **constraints** {C_1, …, C_m}
    - Each C_i consists of a pair <*scope*, *rel*>
    - where *scope* is a tuple of variables that participate in the constraint, *rel* is a relation that defines the values that those variables can take on.
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables.

Formal Definition of CSP

- To solve a CSP, we need to define a **state space** and the **notion of a solution**
- Each state is defined by **assignment of values to variables**: $\{X_i = v_i, X_j = v_j, \ldots\}$
  - **Consistent** assignment
    - A legal assignment that does not violate any constraints
  - **Complete** assignment
    - Every variable is assigned, and the assignment is consistent, complete
  - **Partial** assignment
    - Assigns values to only some of the variables

Example: Map Coloring





Variables:

$X = \{WA, NT, Q, NSW, V, SA, T\}$

Domains:

$D_i = \{\text{red, green, blue}\}$

Constraints:

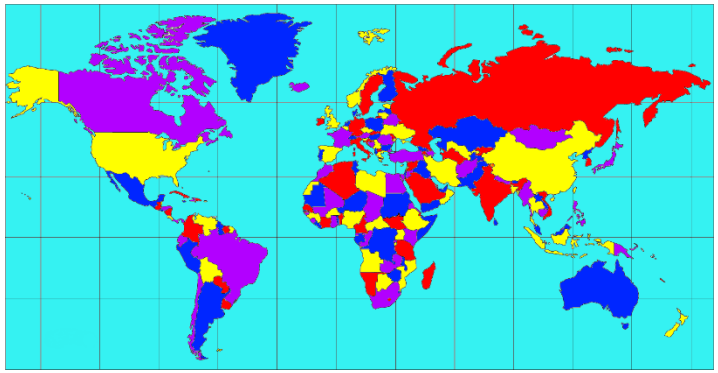
adjacent regions must have different colors.

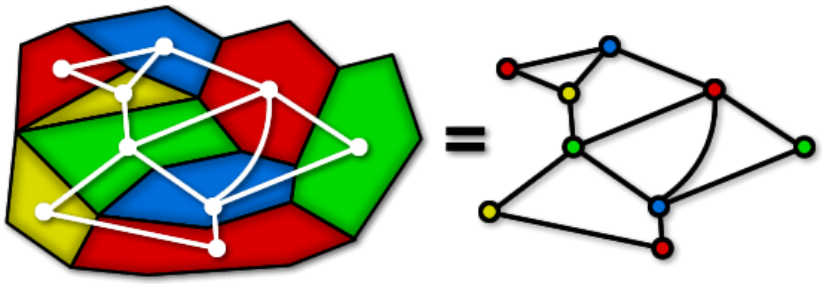
e.g., $WA \neq NT$, short for $<(WA, NT), SA \neq NT>$

or (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

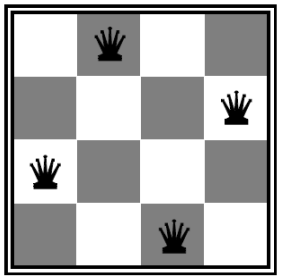
Appendix: Four Color Theorem

- Given any separation of a plane into contiguous regions, called a map, the regions can be colored using **at most four colors** sot that no two adjacent regions have the same color.





Example: N-Queens



- **Formulation 1**
  - Variables: X_{ij}
  - Domains: {0,1}
  - Constraints:

$$\forall i,j,k \;\; (X_{ij}, X_{ik}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \;\; (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \;\; (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i,j,k \;\; (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

- **Formulation 2**
  - Variables: Q_k
  - Domains: {1,2,3,…,*N*}
  - Constraints:
    - Implicit: $\forall i,j \;\; \text{non-threatening}(Q_i, Q_j)$
    - Explicit: $(Q_1, Q_2) \in \{(1,3),(1,4),\ldots\}$

$$\ldots$$

Example: Cryptarithmetic

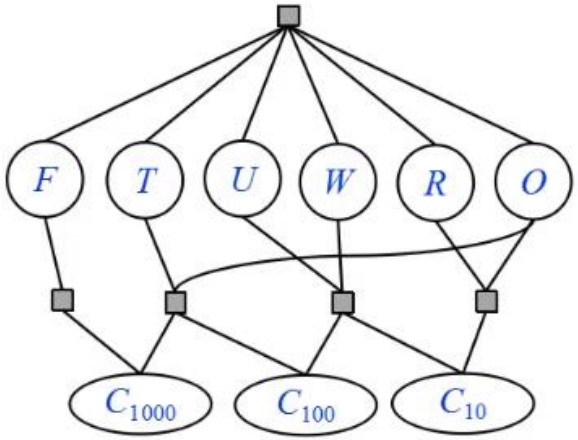
- 算式迷: A type of mathematical game consisting of a mathematical equation among unknown numbers, whose digits are represented by letters.
- The goal is to identify the value of each letter.
- Also known as:
  - Alphametics 字母算数
  - Verbal arithmetic 覆面算
  - Word addition 文字加法
  - Cryptarithm 隐算术

Example: Cryptarithmetic

TWO + TWO = FOUR

- Formatted as a CSP:
  - **Variables**: $X = \{F, T, U, W, R, O, C_{10}, C_{100}, C_{1000}\}$
  - **Domains**: $D_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - **Global Constraints**: *Alldiff*(F, T, U, W, R, O)
  - **Addition Constraints**:

$$O + O = R + 10 \times C_{10}$$
$$C_{10} + W + W = U + 10 \times C_{100}$$
$$C_{100} + T + T = O + 10 \times C_{1000}$$
$$C_{1000} = F$$

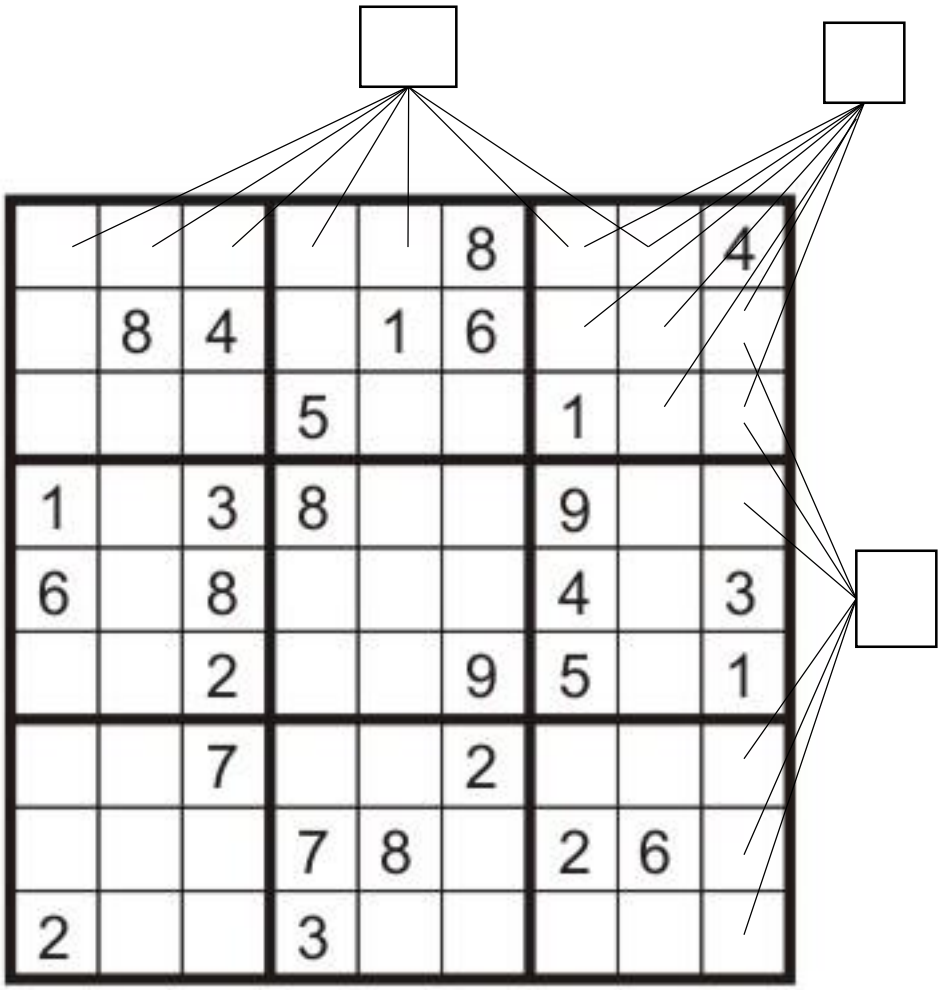


Constraint hypergraph

Where C_{10}, C_{100}, C_{1000} are auxiliary variables, representing the digit carried over into the tens, hundreds, or thousands column.

- A solution: 938 + 938 = 1876

Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

9-way alldiff for each column

9-way alldiff for each row

9-way alldiff for each region

(or can have a bunch of pairwise inequality constraints)

Why CSP?

- Why formulate a problem as a CSP?
  - CSP is a natural representation for a wide variety of problems.
  - CSP-solving system is easier to solve a problem than another search technique.
  - CSP solver can be faster than state-space searchers, since it can quickly **eliminate large swatches of the search space**.

- Example:
  - Once we choose {*SA*=blue} in the map coloring problem, we can conclude that none of the 5 neighboring variables can take on the value blue, therefore:
    - 3^5=243 assignments → 2^5 = 32 (a reduction of 87%)

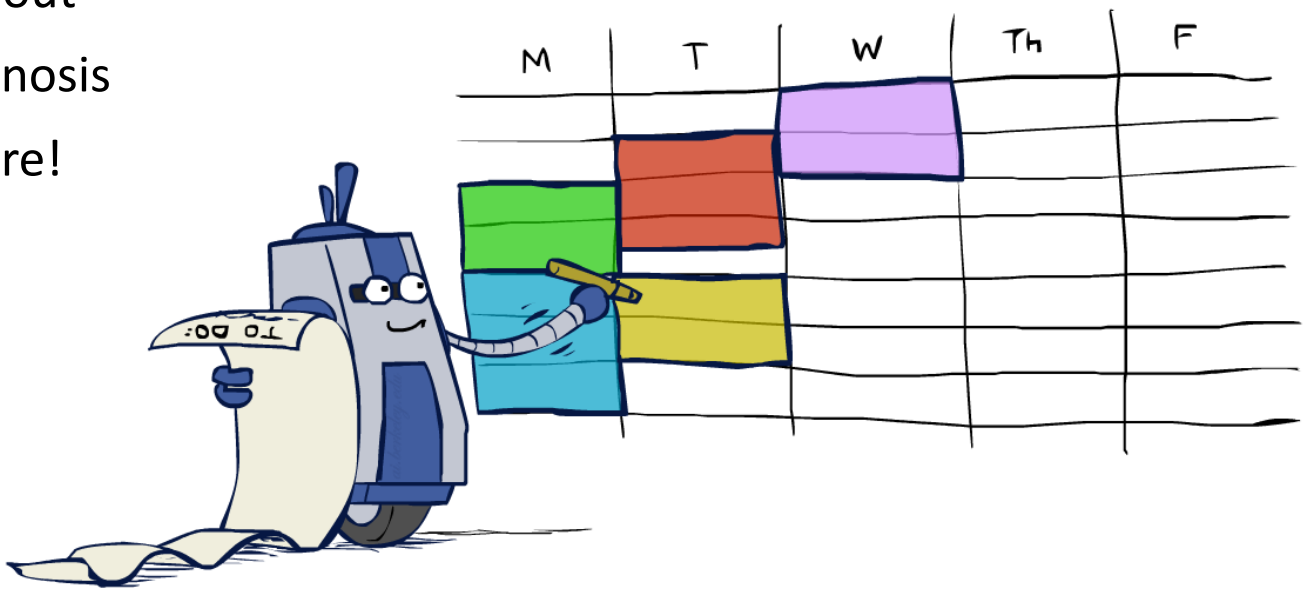
Why CSP?

- In regular state-space search
  - (We can only ask) is this specific state a goal? No? What about this one?

- But with CSP:
  - Once we find out that partial assignment is not a solution, we can immediately discard further refinements.
  - We can see why the assignment is not a solution – we see **which variables violate a constraint**.
  - Many problems can be solved quickly when formulated as a CSP.

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



Standard Search Formulation

- CSPs on finite domains are typically solved using a form of **search**.
- States defined by the values assigned so far (partial assignments)
  - **Initial state**: the empty assignment, {}
  - **Successor function**: assign a value to an unassigned variable
  - **Goal test**: the current assignment is complete and satisfies all constraints

- The most used techniques are variants of
  - **Constraint propagation**
  - **Backtracking**
  - **Local search**

CSP:

Constraint Propagation
Backtracking Search for CSP
Local Search for CSP

Constraint Propagation

- Regular state-space search:
  - can do only one thing -- **search**.
- CSPs:
  - Can do **search**, choose a new variable assignment from several possibilities,
  - Can also do a specific type of **inference**, called **constraint propagation**.
- Constraint Propagation:
  - Uses the constraints to **reduce** the number of legal values for a variable, which in turn can **reduce** the legal values for another variable, and so on.

Constraint Propagation

- Those techniques enforce a form of **local consistency**, which are conditions related to the **consistency of a group of variables and/or constraints**.

- Constraint propagation has various uses:
  - Turns a problem into one that is equivalent but is usually **simpler** to solve.
  - May **prove satisfiability** or **unsatisfiability** of problems.

- Different types of local consistency:
  - **Node consistency**
  - **Arc consistency**
  - **Path consistency**
  - ***k*-consistency**

Node Consistency

- Definition
  - A single variable (node) is node-consistent, if all the values in the variable's domain satisfy the variable's **unary constraints**.

- Example
  - In the variant of the Australia map coloring problem, where South Australians (*SA*) dislike green, i.e.,

    <(*SA*), *SA* ≠ green>

  - leaving SA with the reduced domain {red, blue}

Arc Consistency

- Definition
  - A variable is arc-consistent, if every value in its domain satisfies the variable's **binary constraints**.

- Example
  - In the variant of the Australia map coloring problem, the constraint SA ≠ WA, we can write this constraint explicitly as

<(*SA*, *WA*), {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}>

Path & *k*-Consistency

- Path Consistency
  - A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

- *k*-Consistency
  - A CSP is *k*-consistent if, for any set of *k*-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any k^{th} variable.
  - Features:
    - *k* = 1: equivalent to **node consistency**
    - *k* = 2: equivalent to **arc consistency**
    - *k* = 3: equivalent to **path consistency**

CSP:

Constraint Propagation

Backtracking Search for CSP

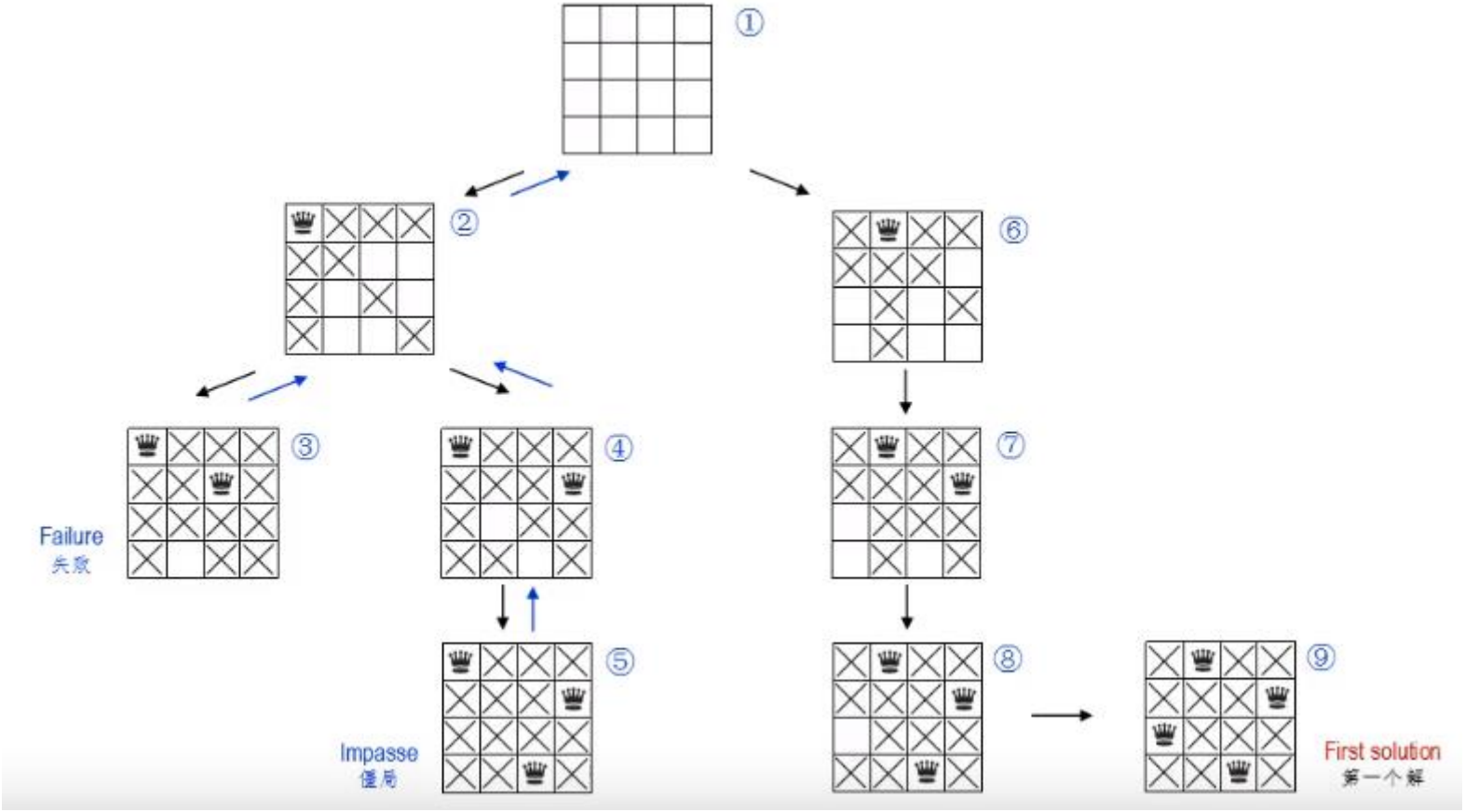
Local Search for CSP

Backtracking Search

- A general algorithm on **depth-first search**, used for finding solutions to some computational problems, notably CSPs.
- **Incrementally** builds candidates to the solutions, and **abandons** each partial candidates *c*(backtracks), as soon as it determines that *c* cannot possibly be completed to a valid solution.
- Backtracking search is the **basic uninformed algorithm** for solving CSPs.

- Example: 8-qeens puzzle
  - In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns.

Example: 4-Queen Problem



Backtracking Search

- **Idea 1: One variable at a time**
  - Variable assignments are commutative, so fix ordering.
  - i.e., [*WA* = red then *NT* = green], same as [*NT* = green then *WA* = red]
  - Only need to consider assignments to a single variable at each step.

- **Idea 2: Check constraints as you go**
  - i.e. consider only values which do not conflict previous assignments.
  - Might have to do some computation to check the constraints.
  - "Incremental goal test".

- Depth-first search with these two improvements is called **backtracking search**.

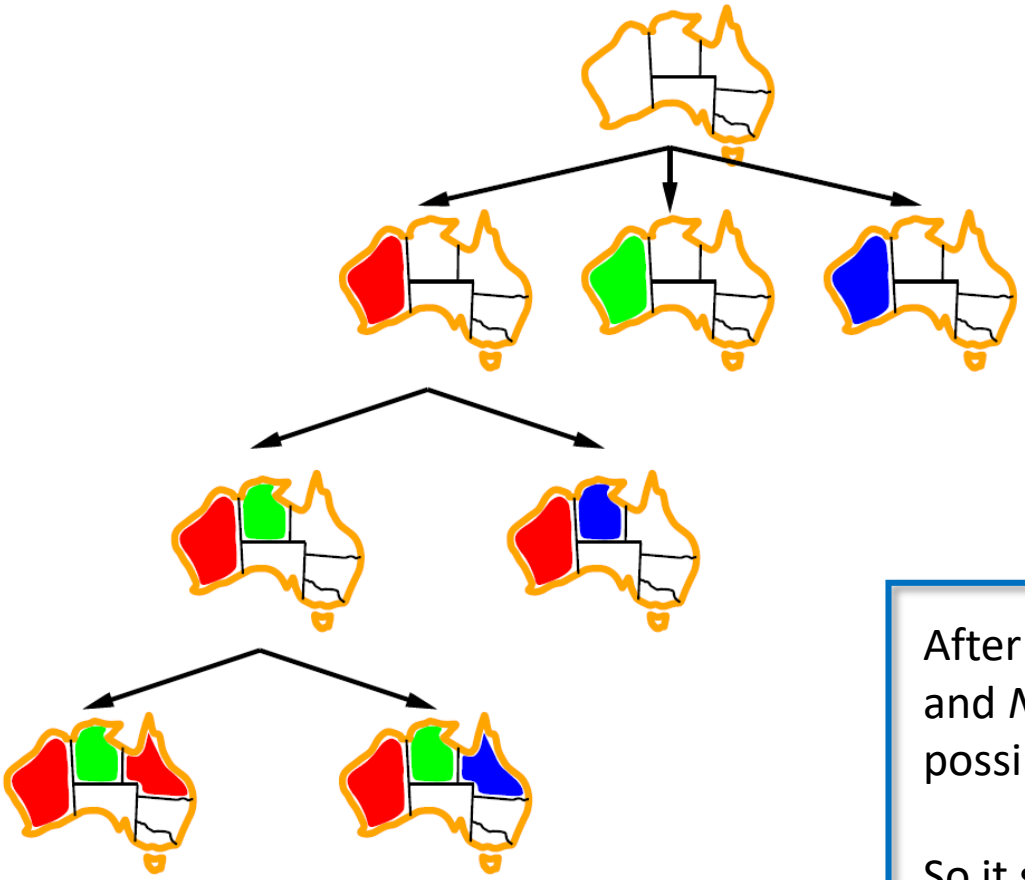
Questions to Improvement

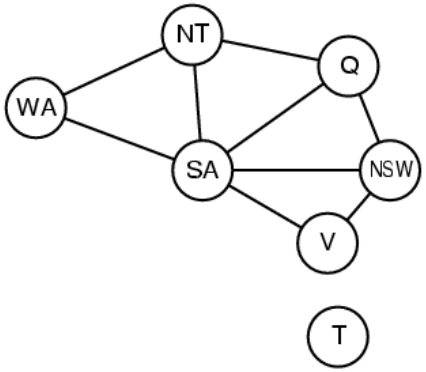
- **Question 1**
  - Which variable should be assigned next?
    - (SELECT-UNASSIGNED-VARIABLE)
  - In what order should its values be tried?
    - (ORDER-DOMAIN-VALUES)

- **Question 2**
  - What inferences should be performed at each step?
    - (INFERENCE)

- **Question 3**
  - When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

Discussion 1: Ordering





After the assignments for *WA* = red and *NT* = green, there is only one possible value for *SA*.

So it should assign *SA* = blue next rather than assigning *Q*.

Discussion 1: Ordering

- The heuristic strategies of ordering
  - **Minimum-remaining-values (MRV)**
    - To choose the variable with the **fewest** "**legal**" values, also has been called the "**most constrained variable**".
  - **Degree heuristic**
    - To **reduce the branching factor** on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.
  - **Least-constraining-value heuristic**
    - To prefer the value that **rules out the fewest** choices for the neighboring variables in the constraint graph.

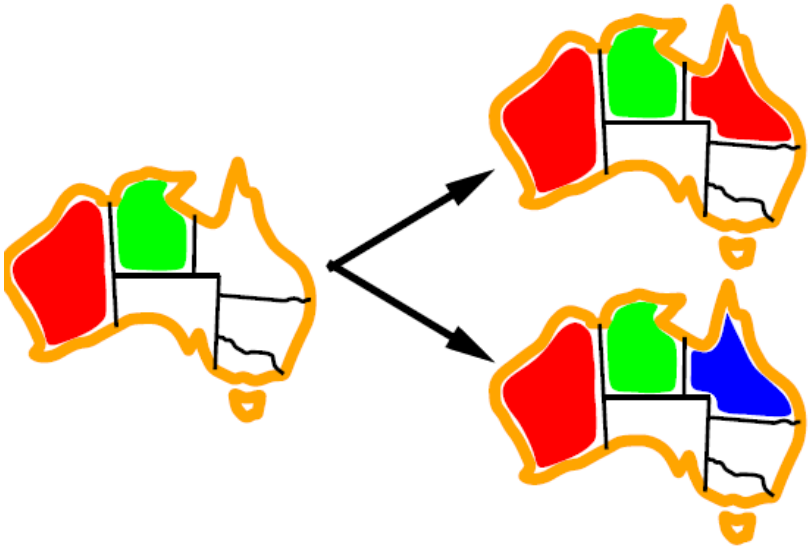
Minimum Remaining Values

- Choose the **variable** with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

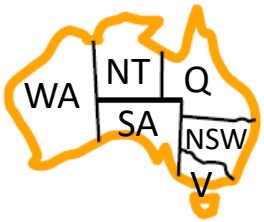
Least Constraining Value

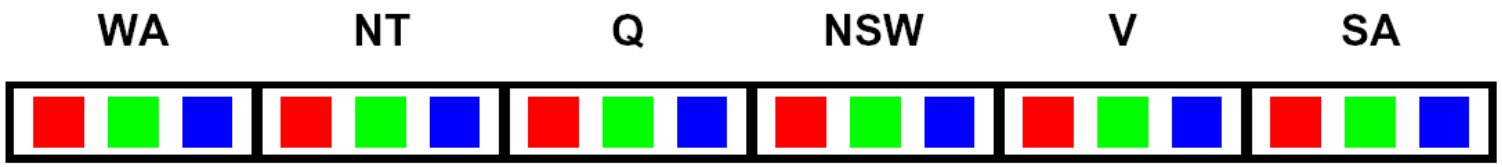
- Least Constraining Value
  - Given a choice of variable, choose the *least constraining **value***
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)
- Why least rather than most?



Discussion 2: Inference

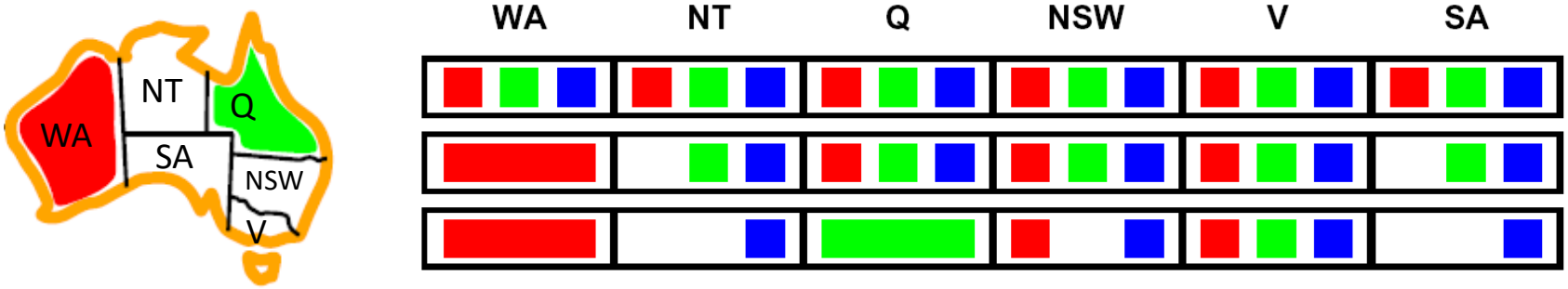
- The inference forward checking can be powerful
- **Filtering**: Keep track of domains for unassigned variables and cross off bad options
- **Forward checking**: Cross off values that violate a constraint when added to the existing assignment





Discussion 2: Inference

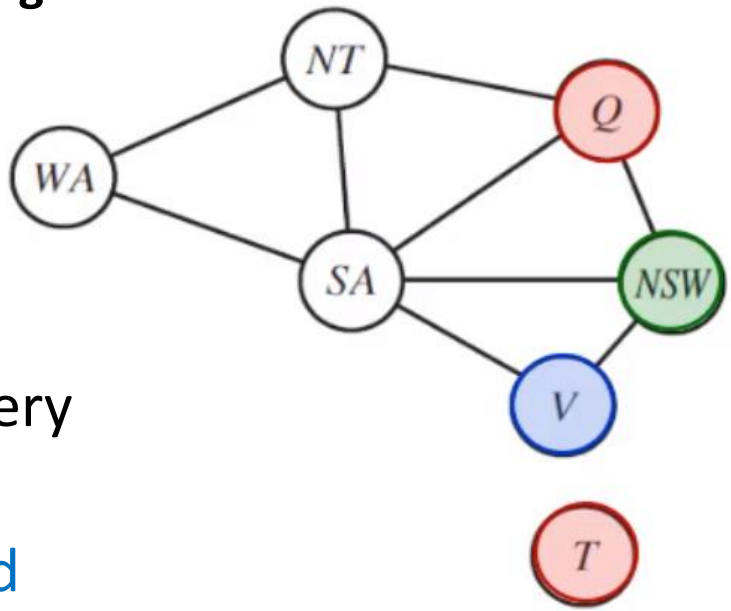
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- Constraint propagation: reason from constraint to constraint

Discussion 3: Intelligent Backtracking

- Backtracking search algorithm has a policy when a search fails: back up to the preceding variable and try a different value.
  - This is called **chronological backtracking**.
- E.g., for a variable ordering
  - {*Q, NSW, V, T, SA, WA, NT*}
- the partial assignment
  - {*Q*=red, *NSW*=green, *V*=blue, *T*=red}
- Try next variable *SA*, we see that every value violates a constraint.
- Backjumping would jump over *T* and try a new value for *V*.



CSP:

Constraint Propagation
Backtracking Search for CSP
Local Search for CSP

Local Search for CSPs

- Local search algorithms are also effective in solving many CSPs, using **complete-state formulation**:
  - **Initial state**: assign a value for each variable
  - **Search**: change the value of one variable at a time

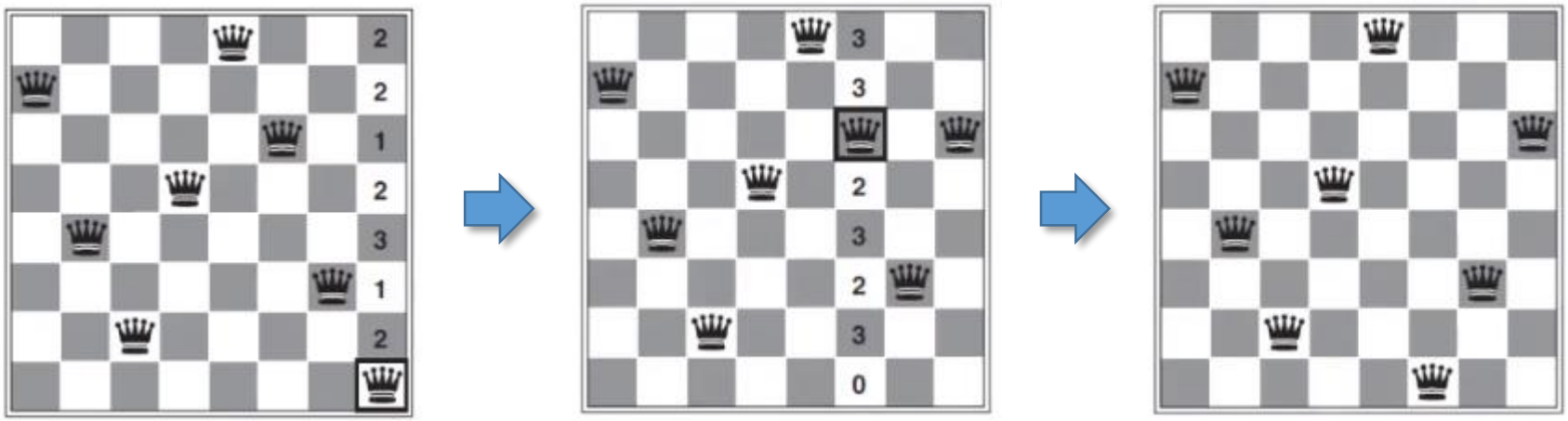
- Example: 8-queens problem
  - Initial state is a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column
  - Typically, the initial guess violates several constraints, we need to eliminate the violated constraints.

Min-conflicts Heuristic

- A most obvious heuristic is to select a new value for a variable.
- Features:
  - **Variable section**: **randomly** select any conflicted variable
  - **Value selection**: select new value that results in a **minimum number of conflicts** with others
- Surprisingly effective for many CSPs
  - On *n*-queens problem, the run time of min-conflicts is roughly independent of problem size.
  - Even on million-queens problem, it solves in an average of 50 steps, after the initial assignment.

Example: 8-Queens problem

- A solution using min-conflicts
  - At each stage, a queen is chosen for reassignment in its column.
  - The algorithm moves the queen to the min-conflicts square, breaking ties randomly.



Constraint Weighting

- Focus on the important constraints
  - Give each constraint a numeric weight, W_i, initially all 1.
  - At each step, choose a variable/value pair to change, that will result in **lowest total weight** of all violated constraints.
  - Then weights are adjusted by **incrementing** the weight of each constraint that is violated by current assignment.

- Two benefits
  - Add topography to plateau, making sure that it is possible to **improve from the current state**.
  - Also add weight to the constraints that are proving **difficult** to solve.

End of Chapter 1