CS2109S: Introduction to AI and Machine Learning

Lecture 3:
Informed, Local, and Adversarial Search

25 August 2023

Recap

- Problem-solving agents
- Search algorithms
- Uninformed search algorithms
  - Breadth-first Search (BFS)
  - Uniform-cost search
  - Depth-first Search (DFS)
- Variants of uninformed search algorithms
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
- Dealing with repeated states

Questions from Last Week

<u>Uniform-cost search</u>

- **Why the name?**
  - Exploring states of the same path cost uniformly
- **Why O(b^C*/e) is the upperbound depth although there are other nodes explored?**
  - The depth of the other nodes explored are less than the bound
- **Why the condition that step cost >= e?**
  - This is not a condition, but simply an assumption for analysis

Outline

- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
  - Variants of A*
- Local search
  - Hill climbing
  - Simulated annealing
  - Beam search
  - Genetic algorithms
- Adversarial search
  - Games
  - Minimax
  - Alpha-beta pruning
  - Handling large/infinite game trees
  - Optimizing the search

Recap!

Outline

- **Informed search algorithms**
  - Greedy best-first search
  - A* search
  - Heuristics
  - Variants of A*
- Local search
  - Hill climbing
  - Simulated annealing
  - Beam search
  - Genetic algorithms
- Adversarial search
  - Games
  - Minimax
  - Alpha-beta pruning
  - Handling large/infinite game trees
  - Optimizing the search

Recap!

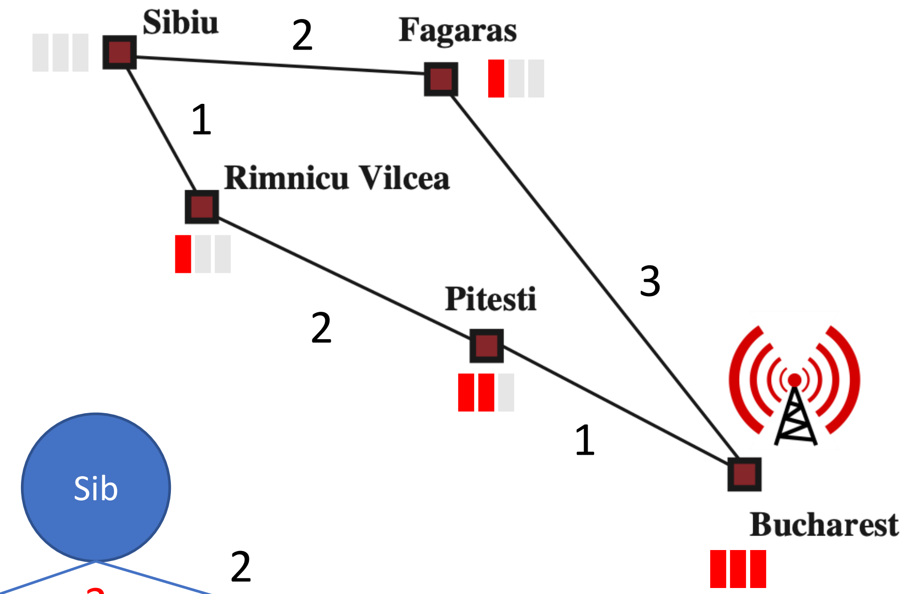
Informed Search Algorithms

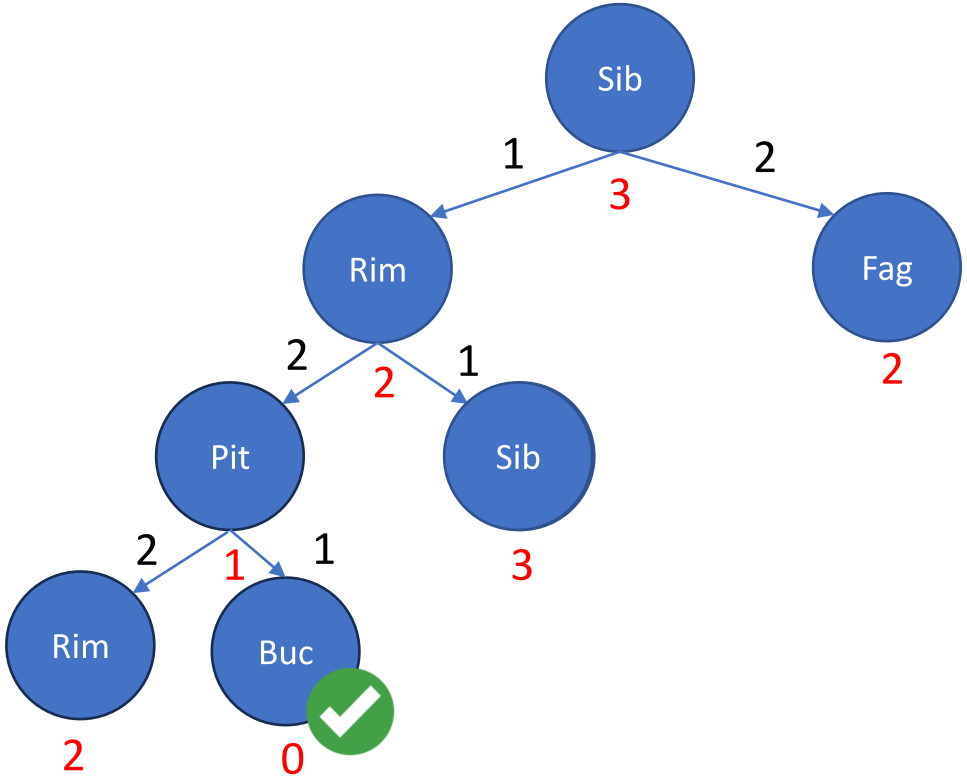
Uninformed search:

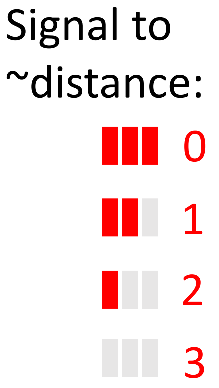
Search blindly

Informed search:

Use domain information to <u>guide</u> the search



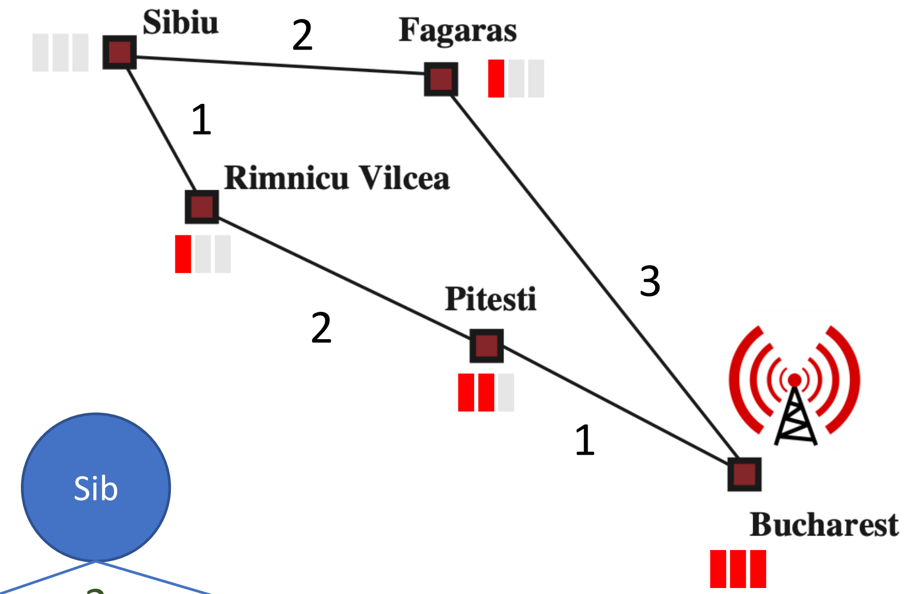


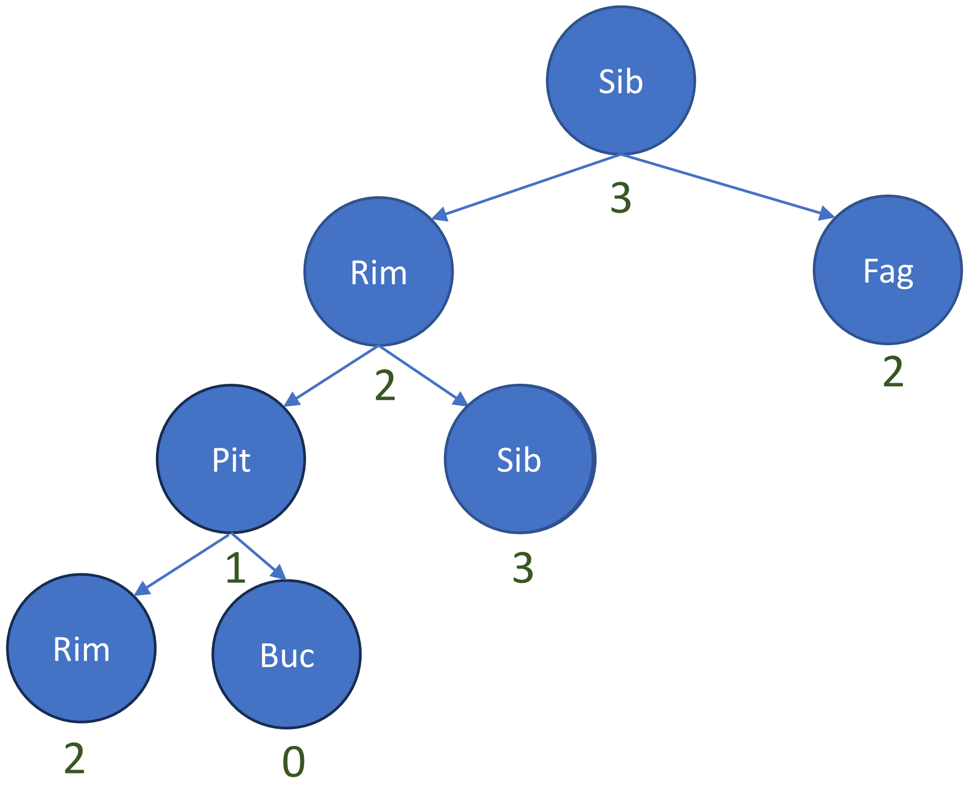


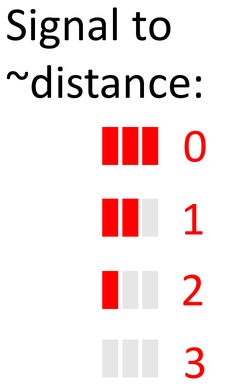
Best-first Search

```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n):
estimate the "goodness" of a state







Special cases:
- Greedy best-first search
- A* search

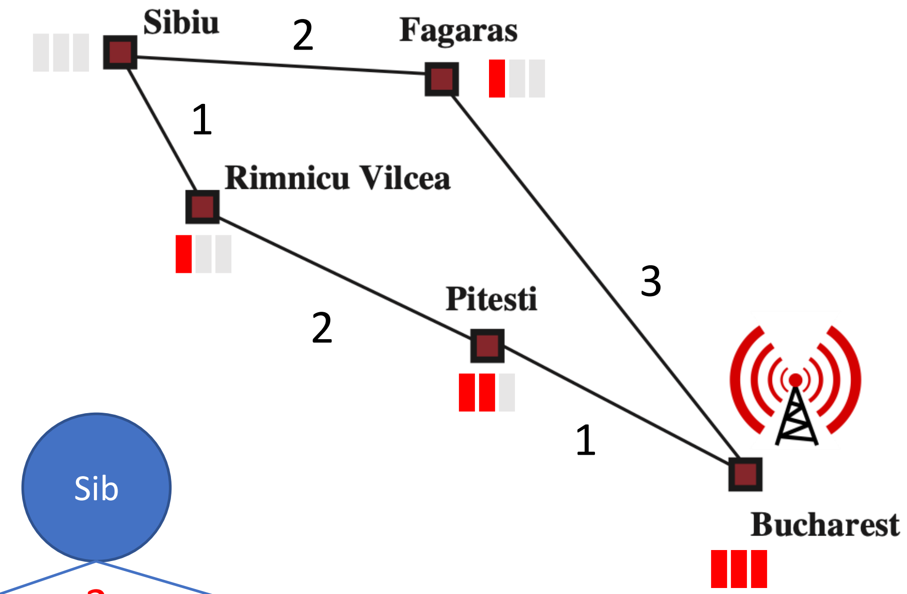
Greedy Best-first Search

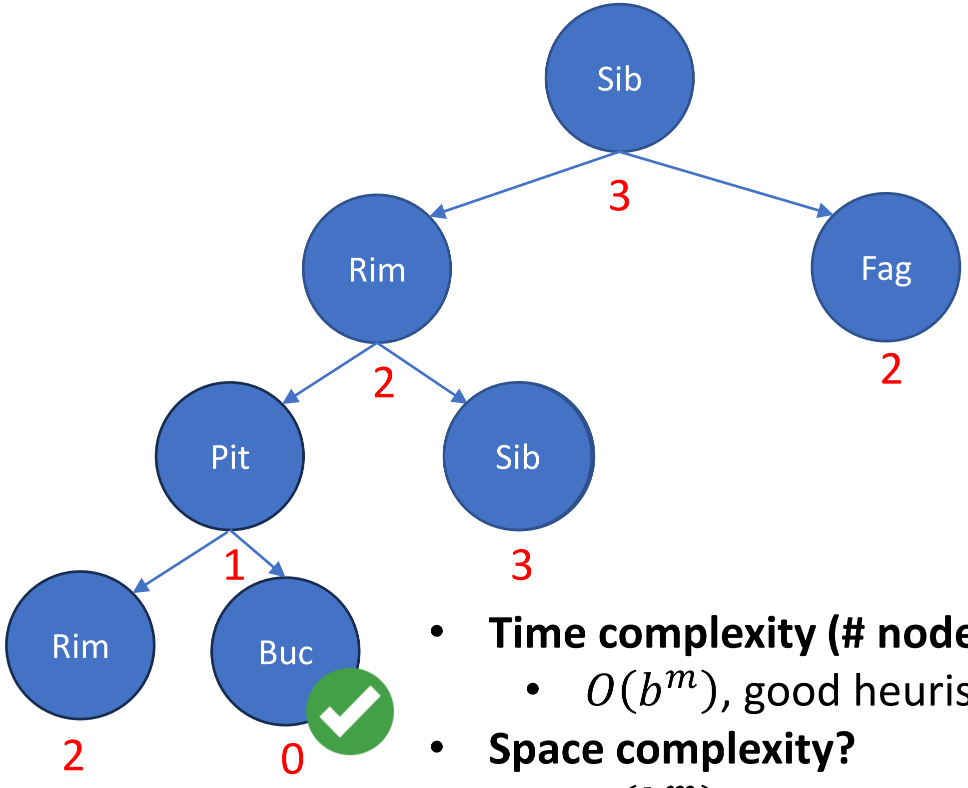
```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

Evaluation function f(n):
estimate the "goodness" of a state

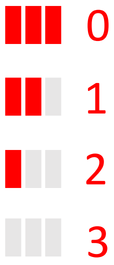
f(n) = h(n)

Heuristic: estimated cost from n to goal





Signal to ~distance:



- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?**

Greedy Best-first Search

```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
   state = frontier.pop()
   if state is goal: return solution
   for action in actions(state):

      next state = transition(state, action)
      frontier.add(next state)
return failure
```

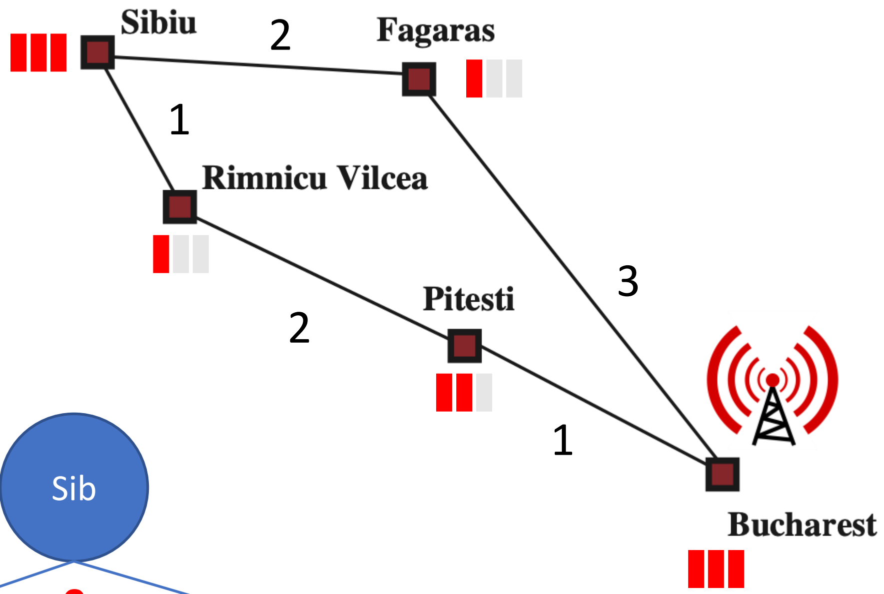
Evaluation function f(n):
estimate the "goodness" of a state

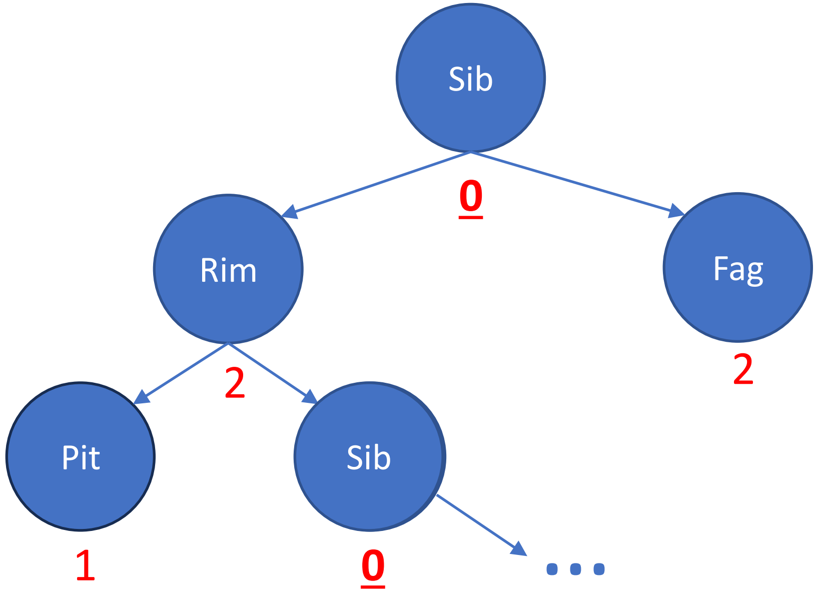
f(n) = h(n)

Heuristic: estimated cost from n to goal

Doesn't consider the cost so far!

Anomaly!





Signal to ~distance:
0
1
2
3

- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?** No
- **Optimal?** No

A* Search

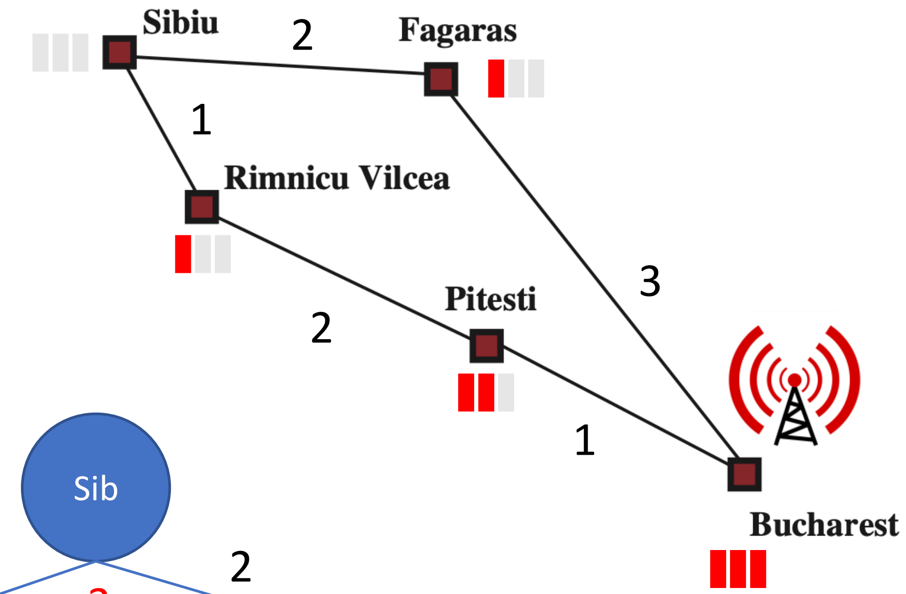
```
create frontier : priority queue f(n)

insert initial state
while frontier is not empty:
    state = frontier.pop()
    if state is goal: return solution
    for action in actions(state):

        next state = transition(state, action)
        frontier.add(next state)
return failure
```

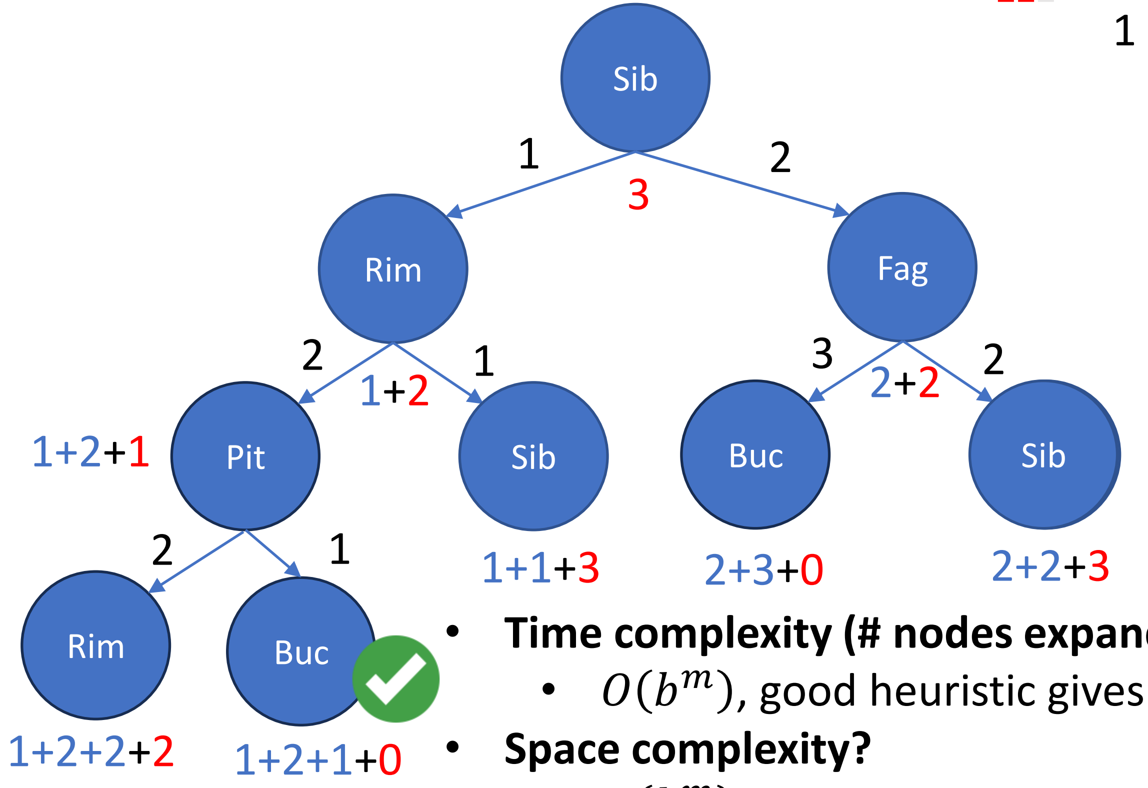
Evaluation function f(n):
estimate the "goodness" of a state

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

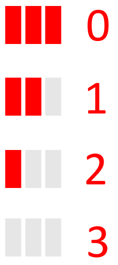
Cost so far to reach n

Heuristic: estimated cost from n to goal





Signal to ~distance:



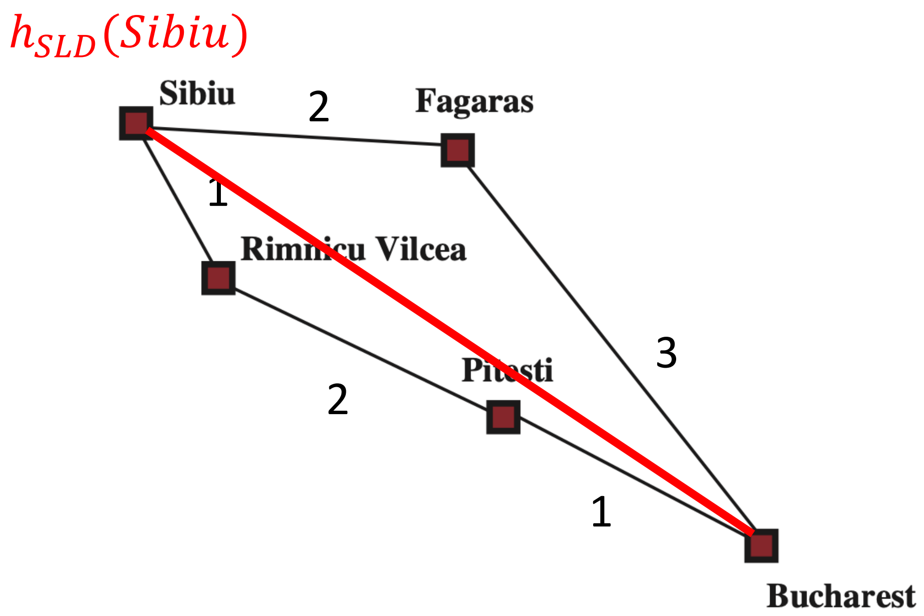
- **Time complexity (# nodes expanded)?**
  - $O(b^m)$, good heuristic gives improvement
- **Space complexity?**
  - $O(b^m)$, keep all nodes in memory
- **Complete?** Yes
- **Optimal?** Yes*

Admissible Heuristics

A heuristic h(n) is **admissible** if for every node n, h(n) ≤ h*(n), where h*(n) is the **true cost** to reach the goal state from n.

An admissible heuristic **never over-estimates** the cost to reach the goal, i.e., it is a **conservative estimate.**

Theorem: if h(n) is admissible, A* using **tree search** is optimal

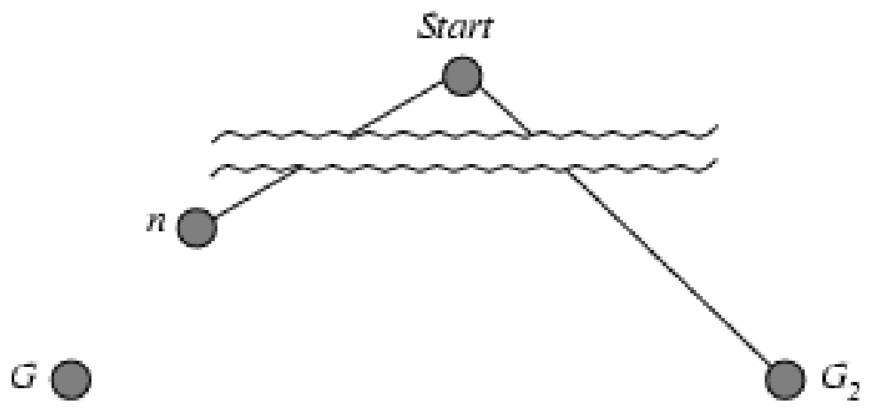


Example: $h_{SLD}(n)$ never overestimates the actual road distance

Admissible Heuristics: Optimality

Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

$f(G) = g(G) + h(G) = g(G) + 0$
$f(G_2) = g(G_2) + h(G_2) = g(G_2) + 0$
$g(G_2) > g(G)$ since G_2 is suboptimal
$f(G_2) > f(G)$



$h(n) \leq h^*(n)$ since h is admissible
$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f(G)$

$f(n) \leq f(G) < f(G_2)$, G_2 will never be expanded

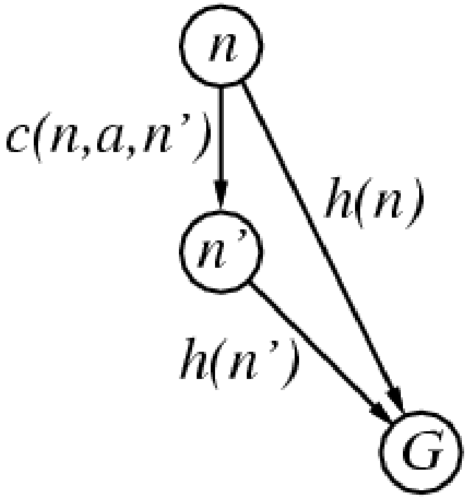
Consistent Heuristics

A heuristic $h(n)$ is **consistent** if for every node n, every successor n' of n generated by any action a, $h(n) \le c(n,a,n') + h(n')$

If h is consistent, we have

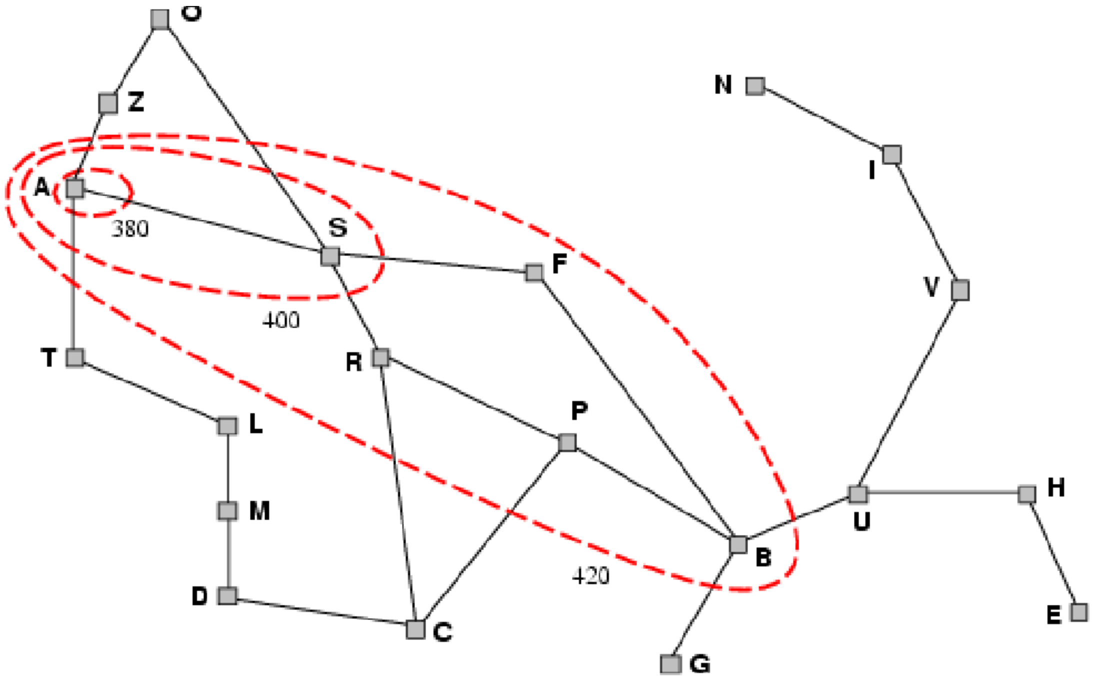
$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n,a,n') + h(n') \\
&\ge g(n) + h(n) = f(n)
\end{aligned}
$$

i.e., f(n) is **non-decreasing** along any path



Theorem: If h(n) is consistent, A* using **graph search** is optimal

Consistent Heuristics: Optimality



A * expands nodes in order of increasing f-cost

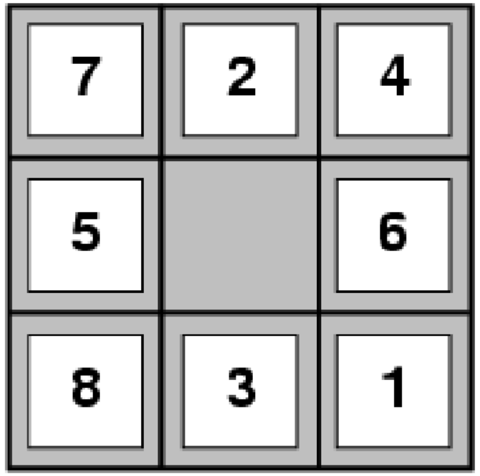
Gradually adds "f-contours" of nodes

Contour i has all nodes with $f = f^{(i)}$, where $f^{(i)} < f^{(i+1)}$

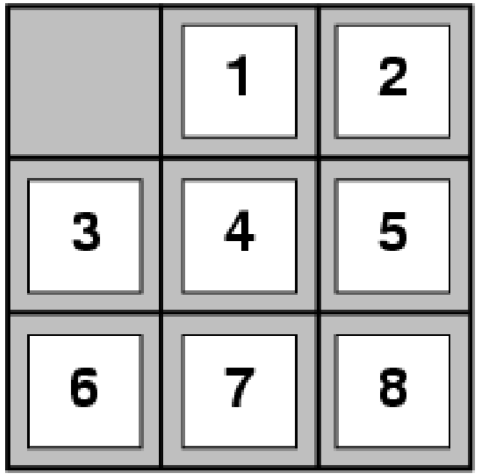
Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1.

h_2 is better for search.



Start State



Goal State

Heuristics

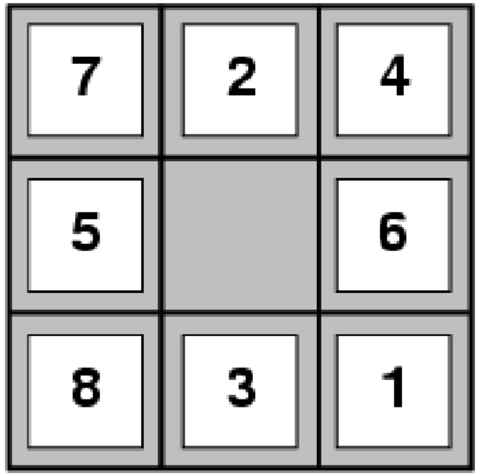
- h_1 number of misplaces tiles
- h_2 total Manhattan distance

h_2 dominates h_1

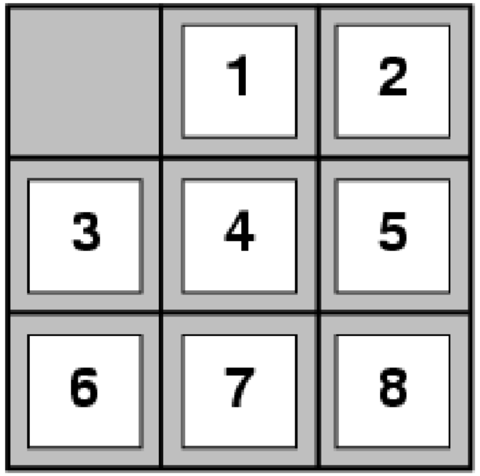
If each tile is at most one distance away from the goal, then h_2 = h_1, otherwise h_2 > h_1

“Inventing” Admissible Heuristics

A problem with **fewer restrictions** on the actions is called a [relaxed problem](). The cost of an optimal solution to a relaxed problem is an **admissible heuristic** for the original problem.



Start State



Goal State

Original:
A tile can only move to adjacent blank

Relaxations:

- Each tile can move anywhere
  - h_1 number of misplaces tiles
- Each tile can move to any adjacent square
  - h_2 total Manhattan distance

Variants of A*

- Iterative Deepening A* (IDA*)
  - Use iterative deepening search
  - Cutoff using f-cost [$f(n) = g(n) + h(n)$] instead of depth
- Simplified Memory-bounded A* (SMA*)
  - Drop the nodes with worst f-cost if memory is full

Summary: Informed Search

- Informed search: <u>guide</u> search with domain information
- Best-first search
  - Greedy best-first search
    - $f(n) = h(n)$, **heuristic** estimate of cost from n to goal
  - A* search
    - $f(n) = g(n) + h(n)$, cost so far + heuristic
- Heuristics
  - Admissible: $h(n) \leq h^*(n)$
  - Consistent: $h(n) \leq c(n,a,n') + h(n')$
  - Dominant: if $h1(n) \leq h2(n)$, h2 dominant
- Creating admissible heuristic: **true cost** of the **relaxed problem**
- A* variants: IDA*, SMA*
  - Idea: prune to save memory

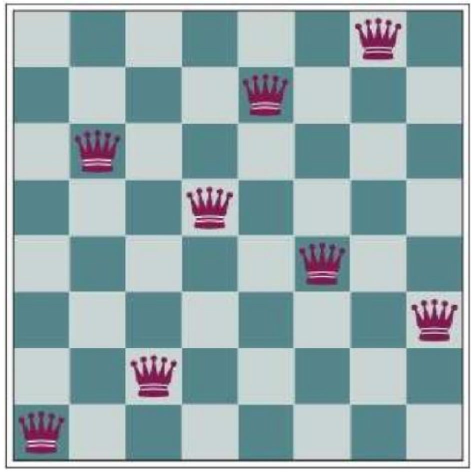
Outline

- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
  - Variants of A*
- **Local search**
  - Hill climbing
  - Simulated annealing
  - Beam search
  - Genetic algorithms
- Adversarial search
  - Games
  - Minimax
  - Alpha-beta pruning
  - Handling large/infinite game trees
  - Optimizing the search

Local Search

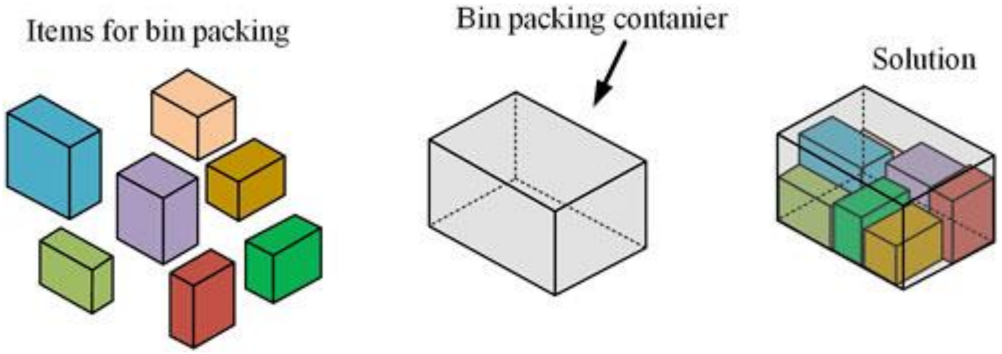
Previously: path to a goal is a solution

There are problems where <u>path is not important</u> (state is the solution).



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

Credit: Encyclopaedia Britannica

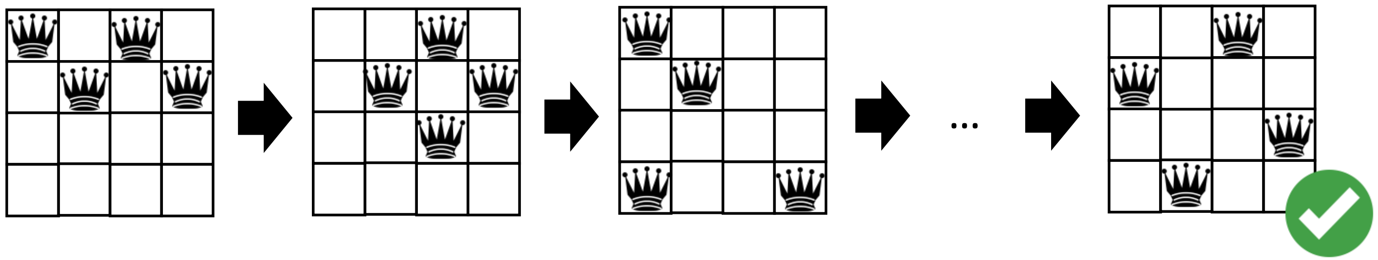


Credit: Frontiersin

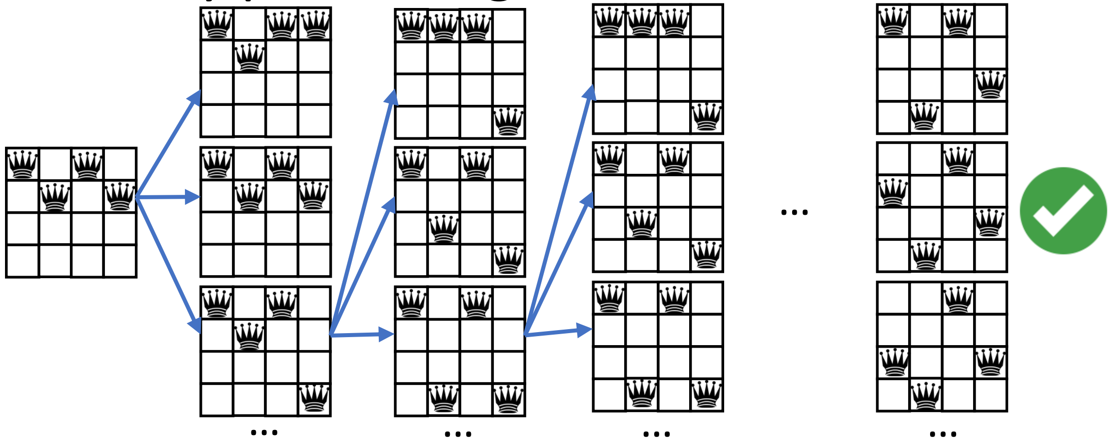
How do we solve these problems? **Start with a state** (solution) then **try to improve** it

Trivial Algorithms

- Random sampling
  - Generate a state randomly until a solution is found



- Random walk
  - Randomly pick a neighbour from the current state

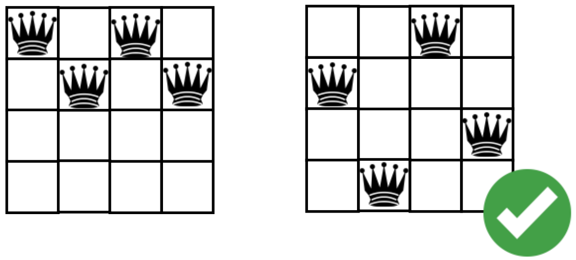


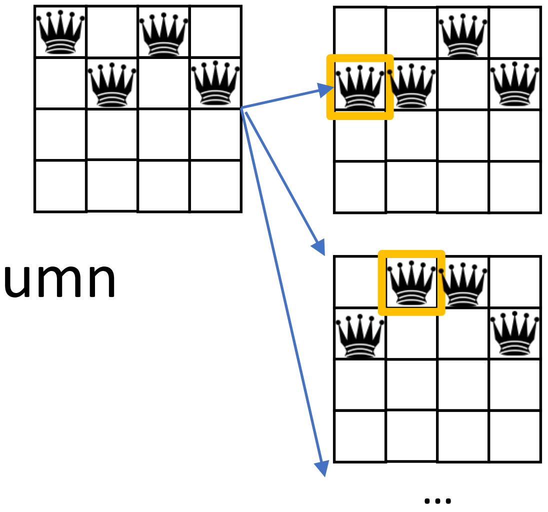
Can we do better?

Formulation

Formally, a local search problem can be formulated as follows.

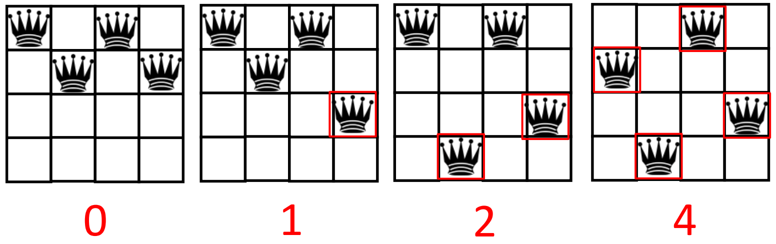
- **States (state space)**
  - State representation: grid with N queens
- **Initial state**
  - N queens positioned randomly on the grid
- **Goal test**
  - No queens are attacking each other
- **Successor function**
  - Move a single queen to another square on the same column





Evaluation Function

Output the value (“goodness”) of a state.

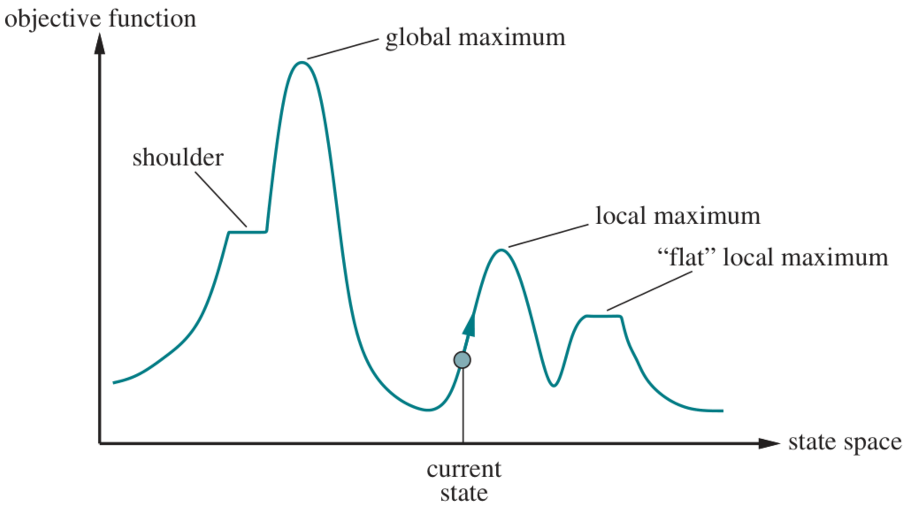


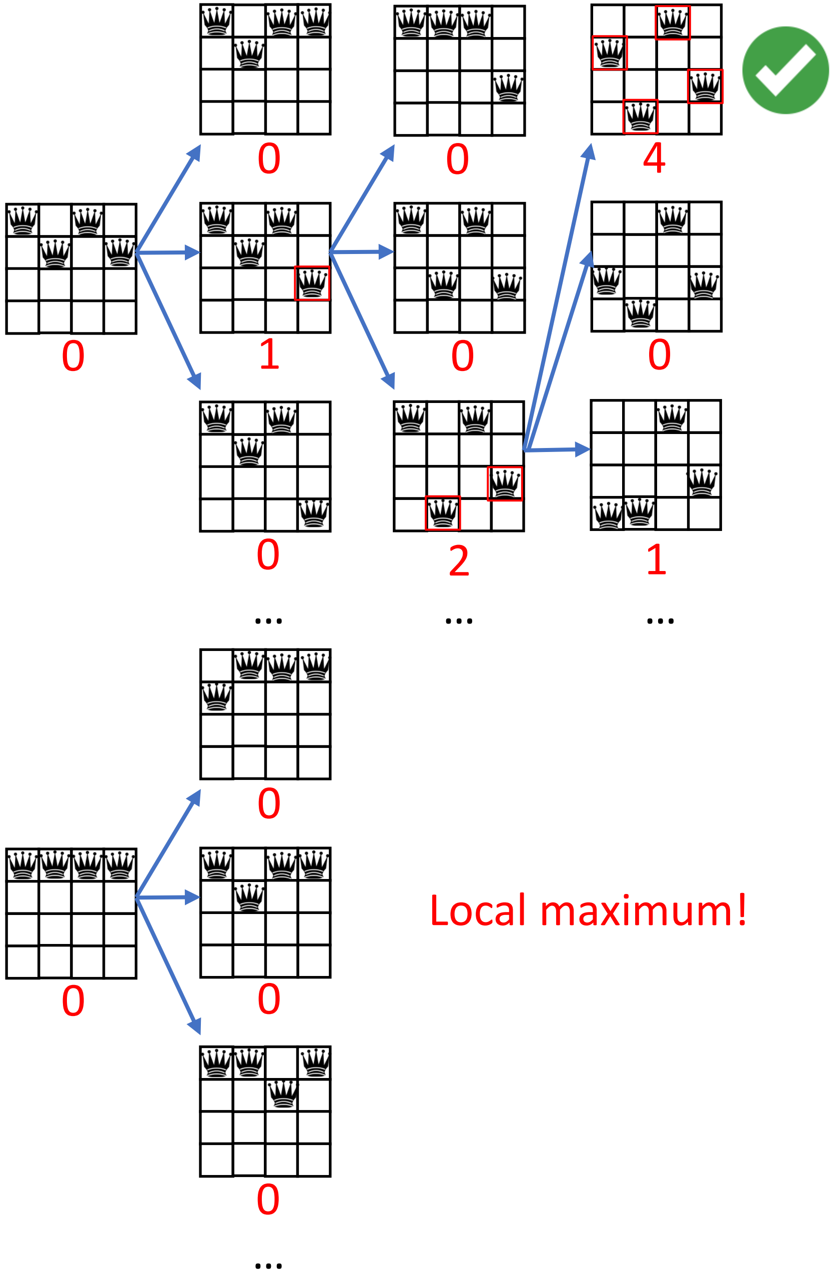
Can be either:

- Heuristic function
  - # of well-positioned queens
- Objective function
  - $J(x) = -x^2$

Hill climbing algorithm

```
current = initial state
loop:
    neighbor = a highest value successor of current
    if value(neighbor) <= value(current):
        return current
    current = neighbor
```





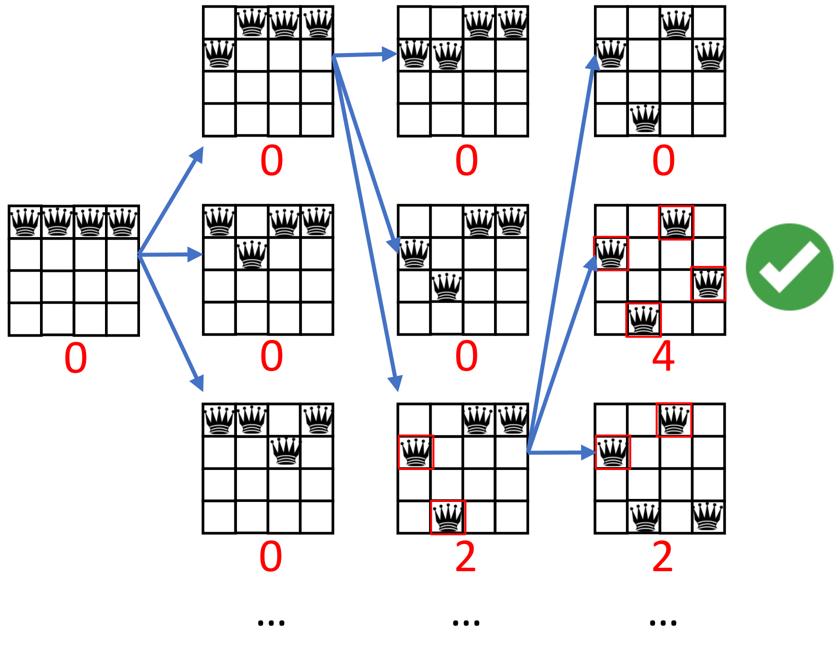
Simulated Annealing

```
current = initial state
for t = 1 … ∞:
   T = schedule(t)
   if T = 0:
      return current
   next = a randomly selected successor of current
   if value(next) > value(current) or Prob(next, current, T):
      current = next
```

Allow "bad moves" from time to time

$$\textbf{Prob}(next, current, T) = e^{\frac{value(next) - value(current)}{T}}$$

Select a "bad move"



Escape!

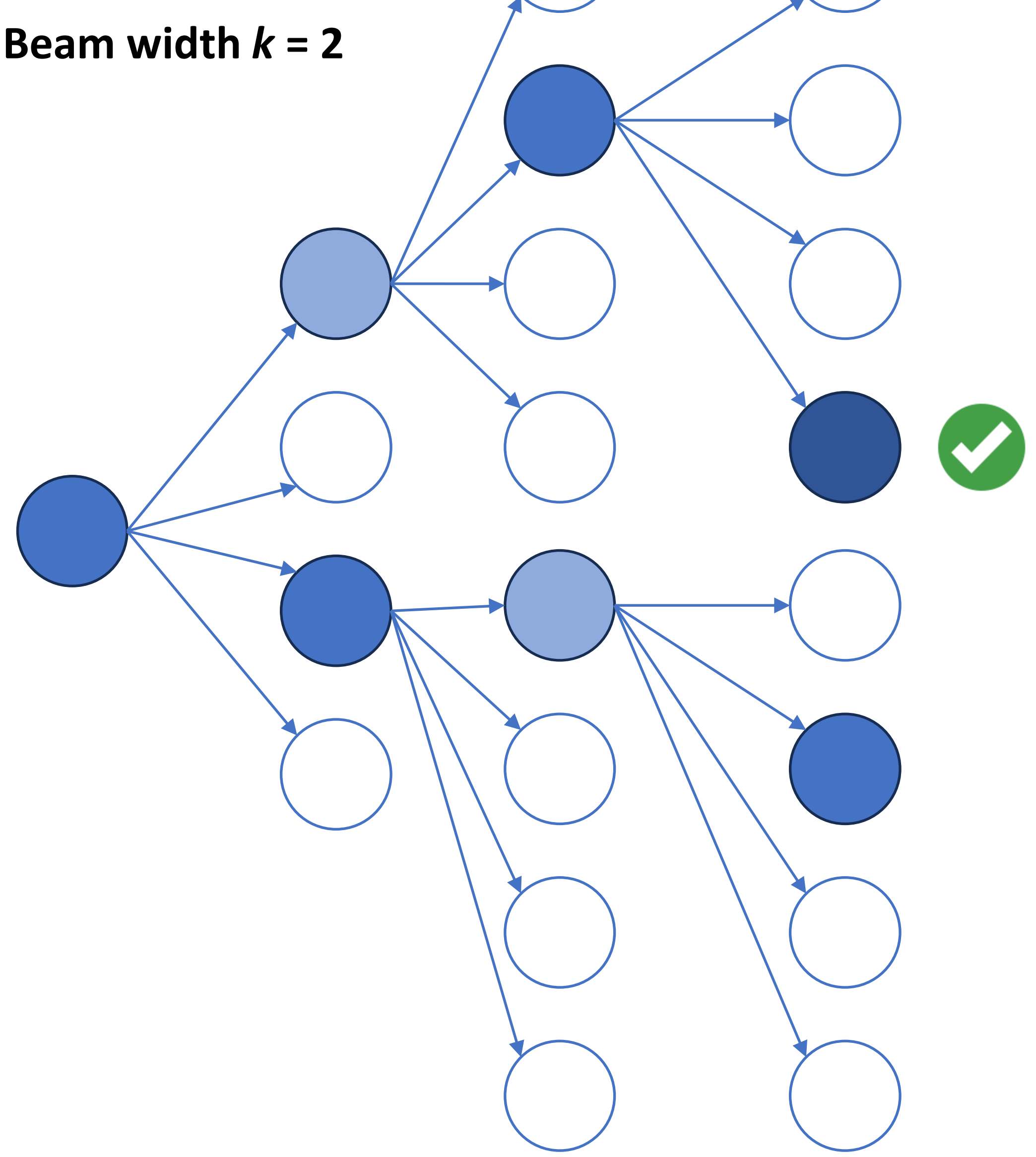
Theorem: if T decreases slowly enough, simulated annealing will find a global optimum with high probability

Beam Search

Perform *k* hill-climbing in parallel

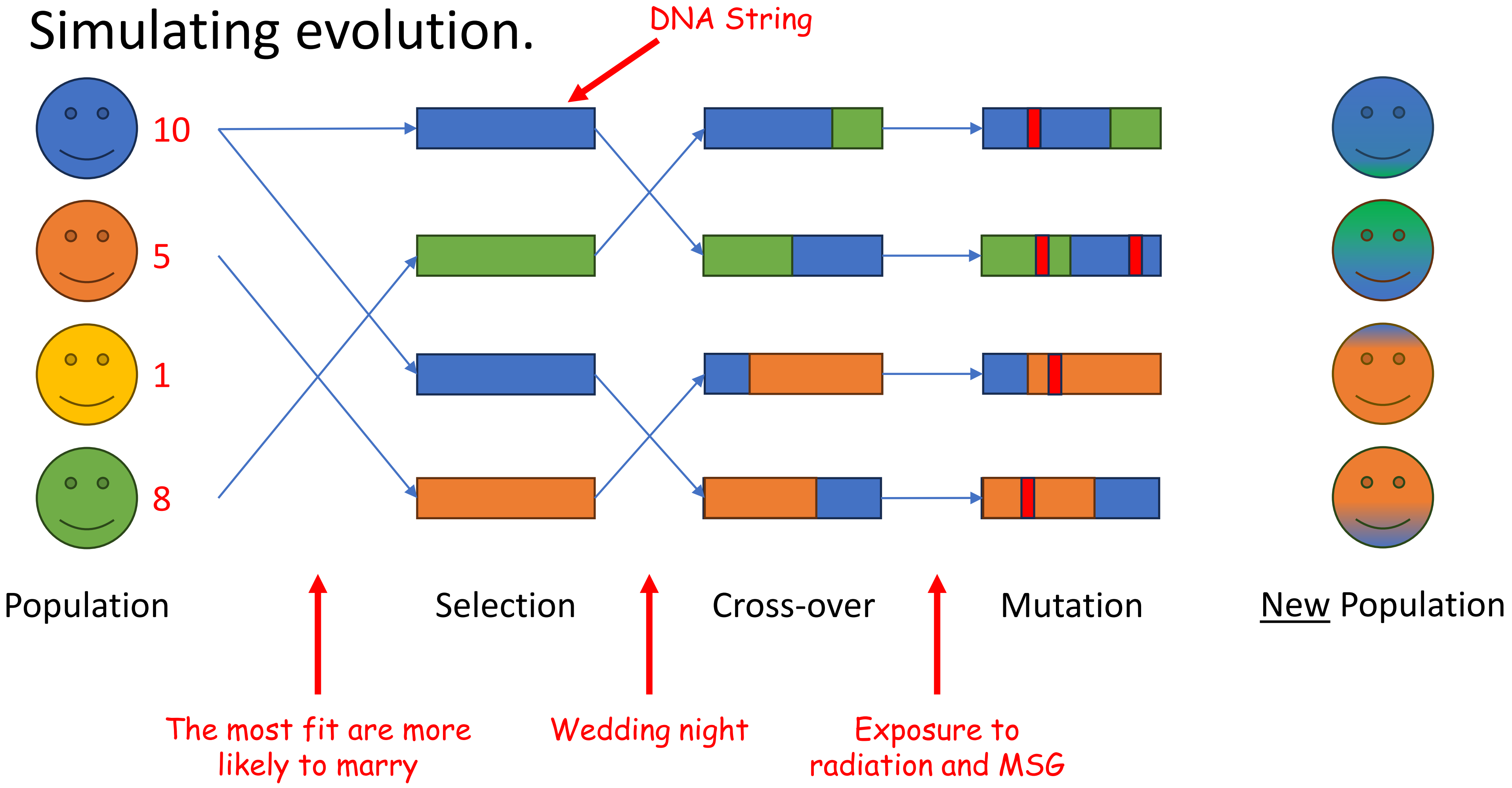
Variants:

- Local beam search
  - Choose *k* successors <u>deterministically</u>
- Stochastic beam search
  - Choose *k successors* <u>probabilistically</u>

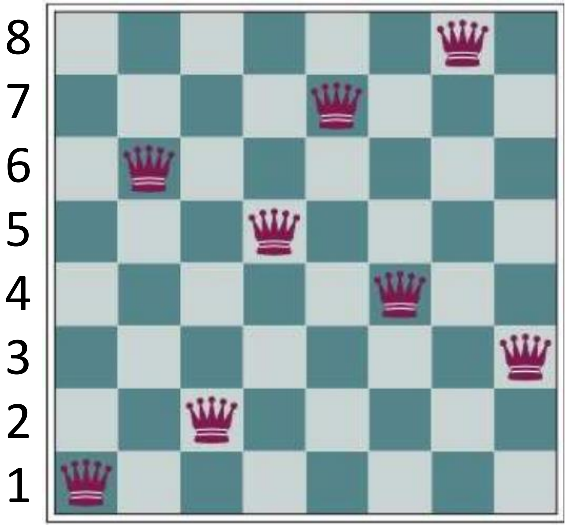


Genetic Algorithm

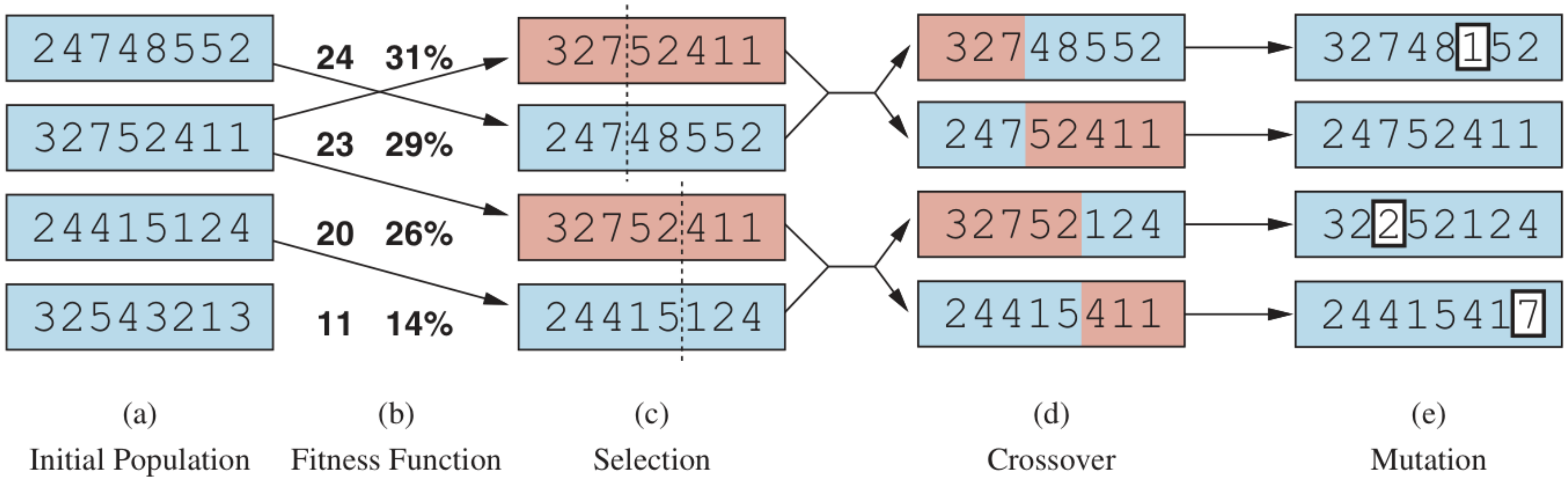
Simulating evolution.

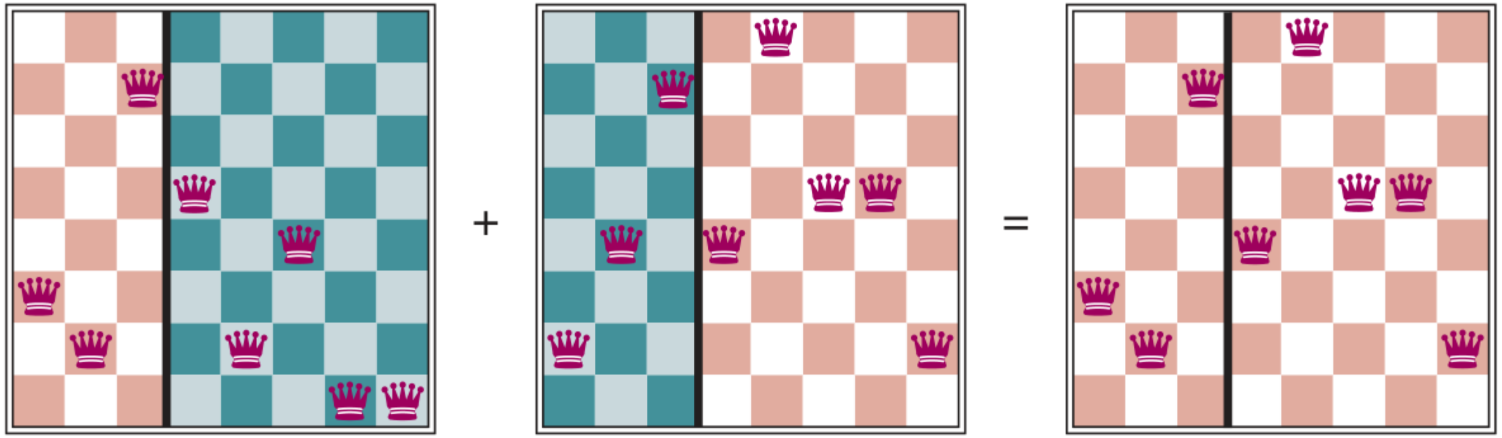


Genetic Algorithm: 8-Queens



DNA String: 16257483





Summary: Local Search

- Local search: **path is not important**, just want the state
  - Goal state is unknown
- Trivial algorithms
  - Random sampling: **random sample a state** until solution is found
  - Random walk: **go to random neighbours** until solution is found
- Algorithms:
  - Hill-climbing: **pick the best** among neighbours, repeat
  - Simulated Annealing: hill-climbing but **allow bad moves** from time to time
  - Beam search: do **k hill-climbing in parallel**
  - Genetic algorithm: **marry** the best, **mutate**, repeat

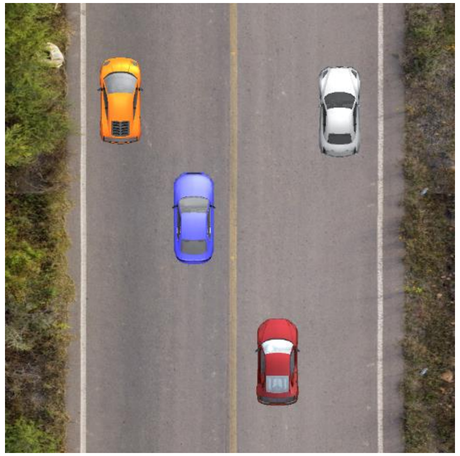
Outline

- Informed search algorithms
  - Greedy best-first search
  - A* search
  - Heuristics
  - Variants of A*
- Local search
  - Hill climbing
  - Simulated annealing
  - Beam search
  - Genetic algorithms
- **Adversarial search**
  - Games
  - Minimax
  - Alpha-beta pruning
  - Handling large/infinite game trees
  - Optimizing the search

Games

Search problems:

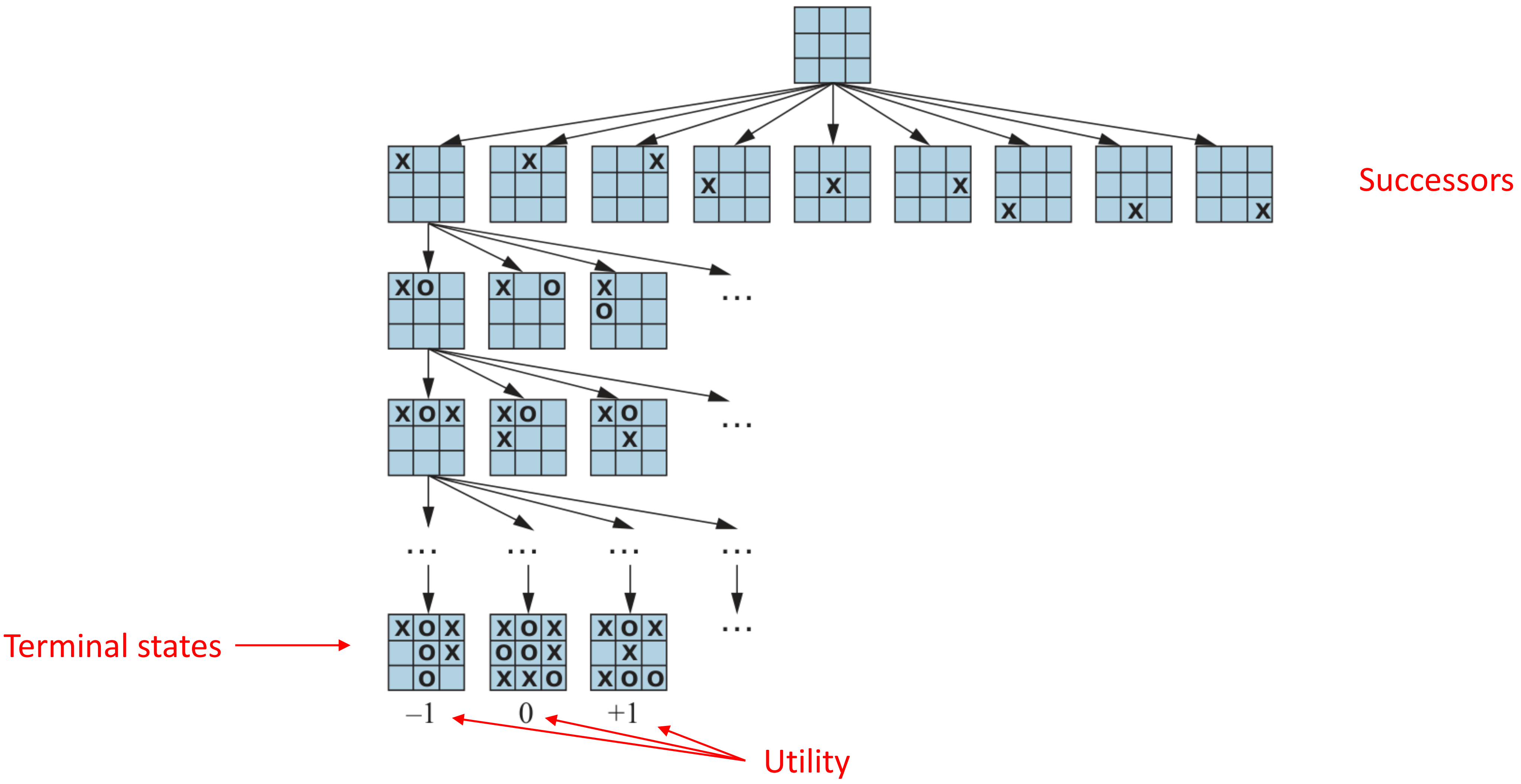
The environment reacts either **deterministically** or **stochastically**



Games:

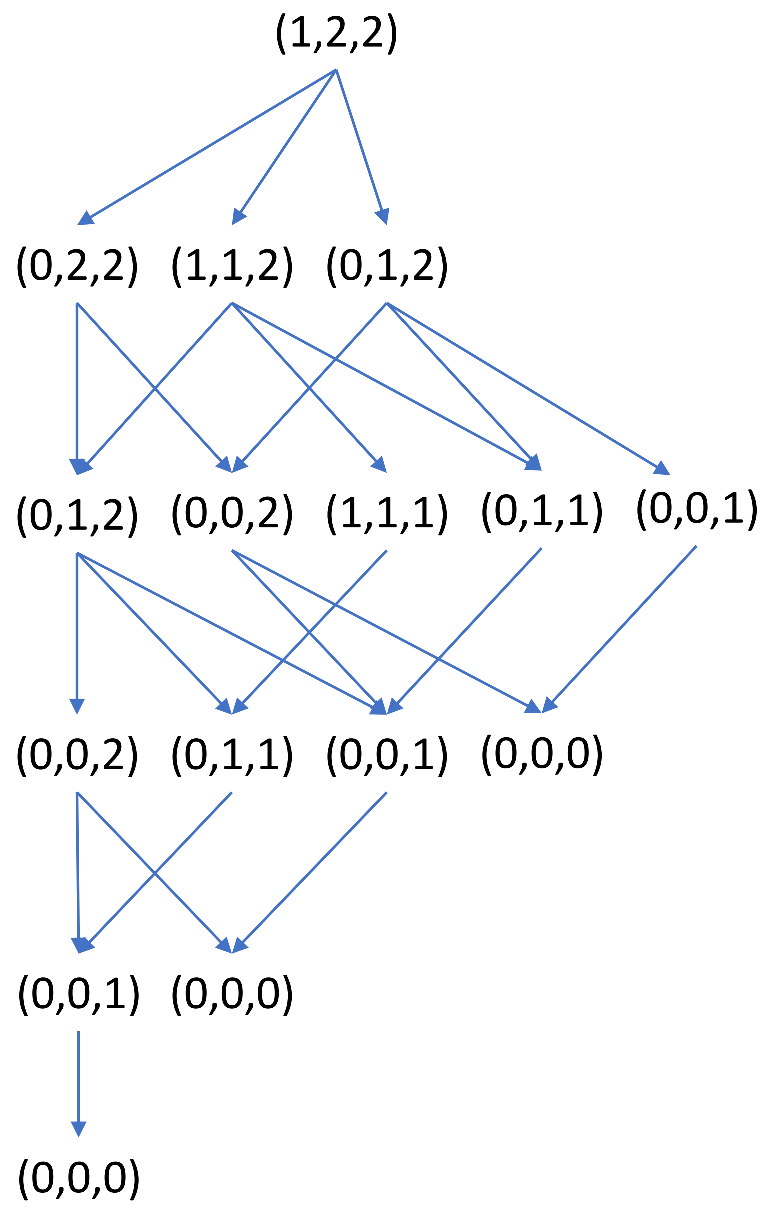
- The opponent reacts “unpredictably” (**strategic** environment)
- Usually have time limits

Example: Tic-tac-toe



Example: Game of NIM

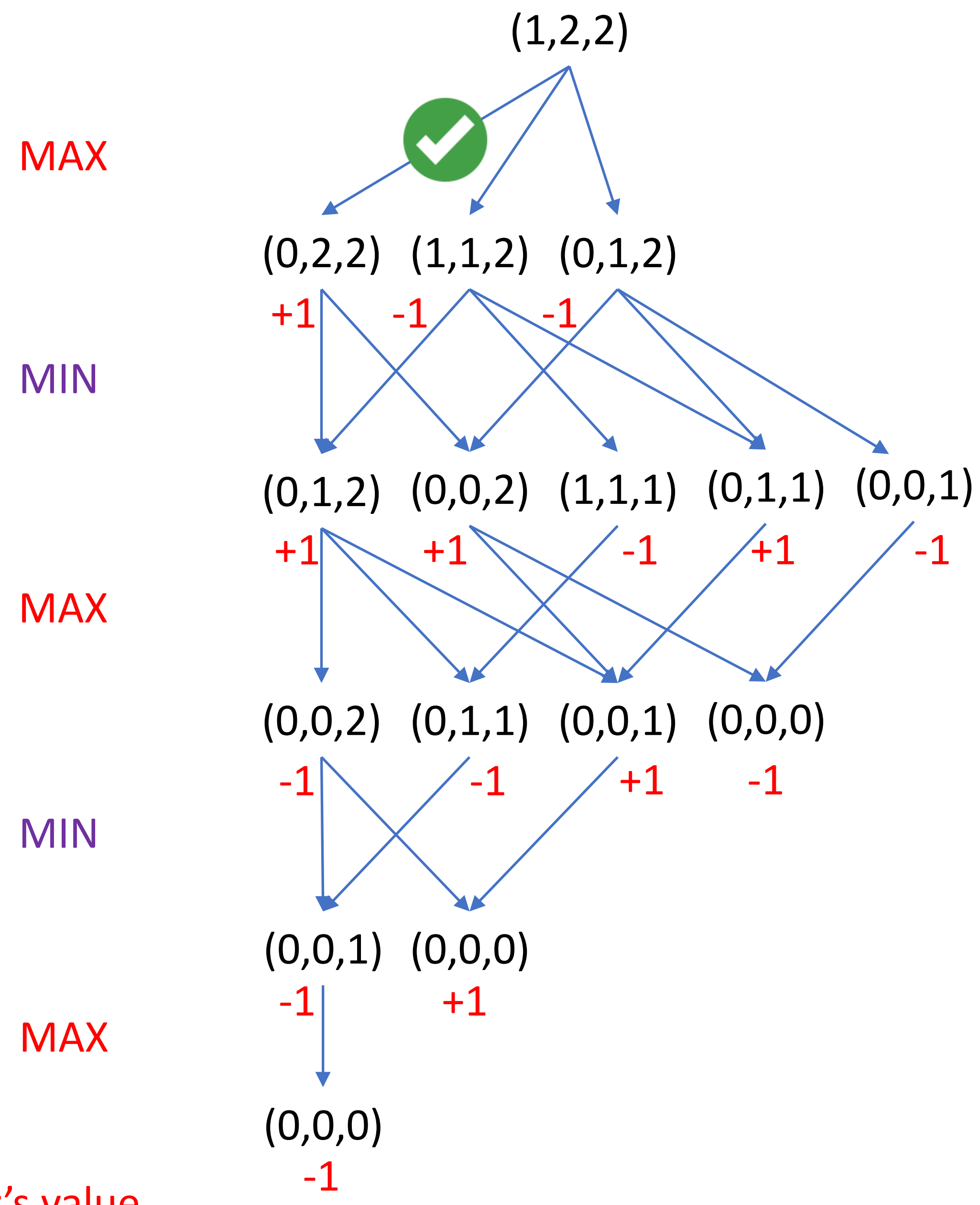
- Several piles of sticks are given
  - Representation: **monotone** sequence of integers, e.g., (1,3,5)
- A player may **remove any number of sticks from one pile** in one turn
  - Example: (1,3,5) -> (1,1,3)
- The player who takes the last stick <u>loses</u>



Minimax

```
def minimax(state):
   v = max_value(state)
   return action in successors(state) with value v

def max_value(state):
   if is_terminal(state): return utility(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state))
   return v

def min_value(state):
   if is_terminal(state): return utility(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
   return v
```

Assumes opponent play **optimally**: trying to **minimize** player's value

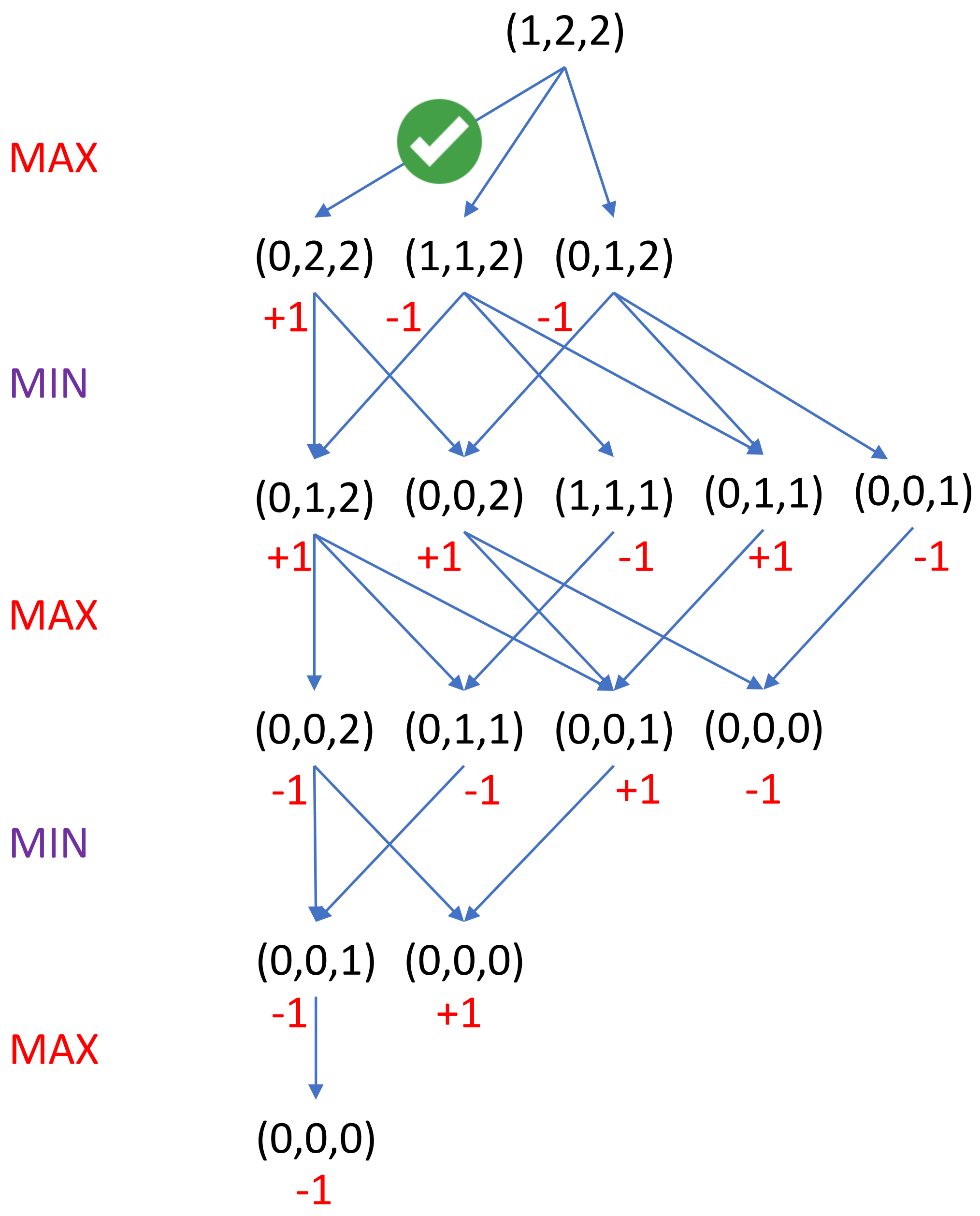


Minimax: Analysis

- **Complete?**
  - Yes, if tree is finite
- **Time Complexity?**
  - $O(b^m)$
- **Space Complexity?**
  - $O(bm)$, with depth first exploration
- **Optimal?**
  - Yes, against optimal opponent

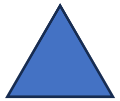
Chess: b ≈ 35, m ≈ 100 for “reasonable” games

What to do?



Alpha-beta Pruning

```
def alpha_beta_search(state):
   v = max_value(state, -∞, ∞)
   return action in successors(state) with value v

def max_value(state, α, β):
   if is_terminal(state): return utility(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state, α, β))
      if v >= β: return v
      α = max(α, v)
   return v

def min_value(state , α, β):
   if is_terminal(state): return utility(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
      if v <= α: return v
      β = min(β, v)
   return v
```



MAX

MIN

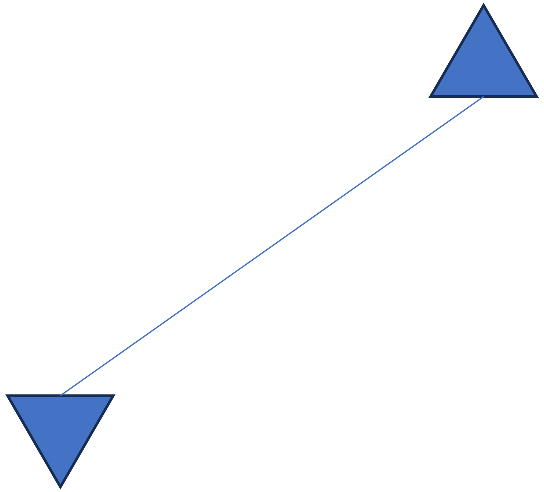
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```

MAX

$\alpha = -\infty$

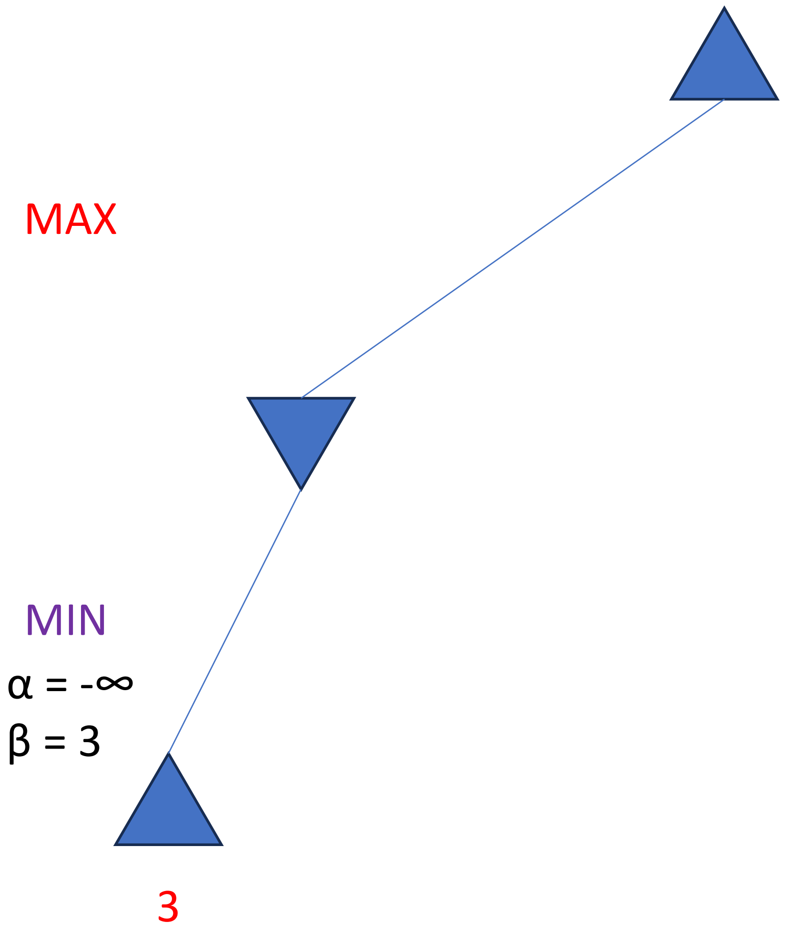
$\beta = \infty$



MIN

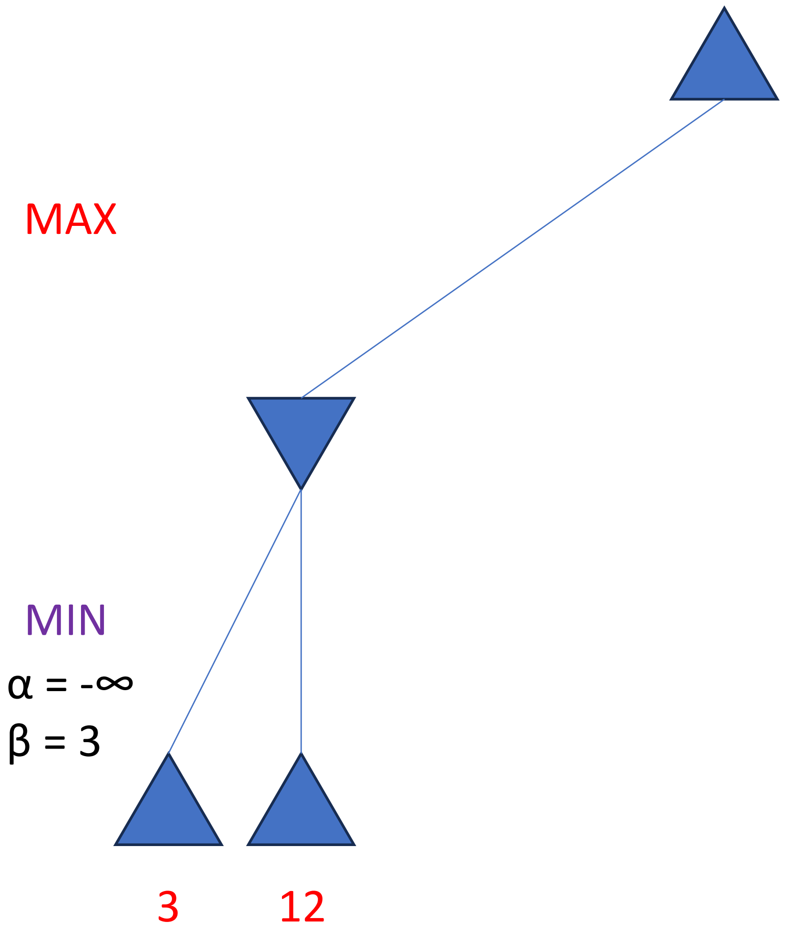
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



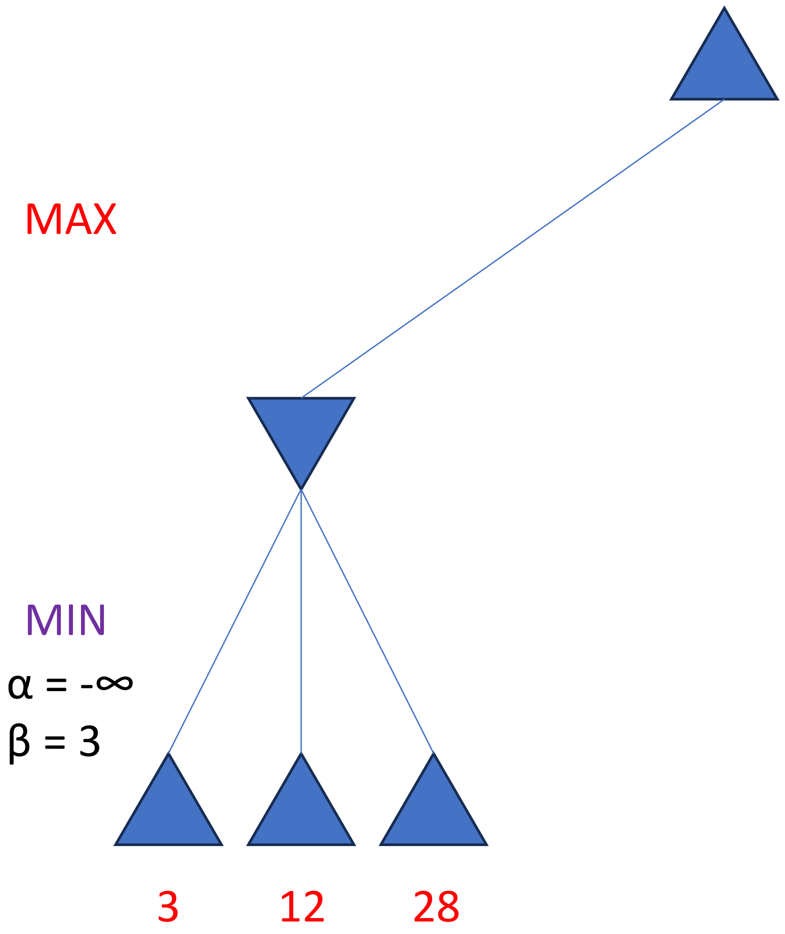
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```



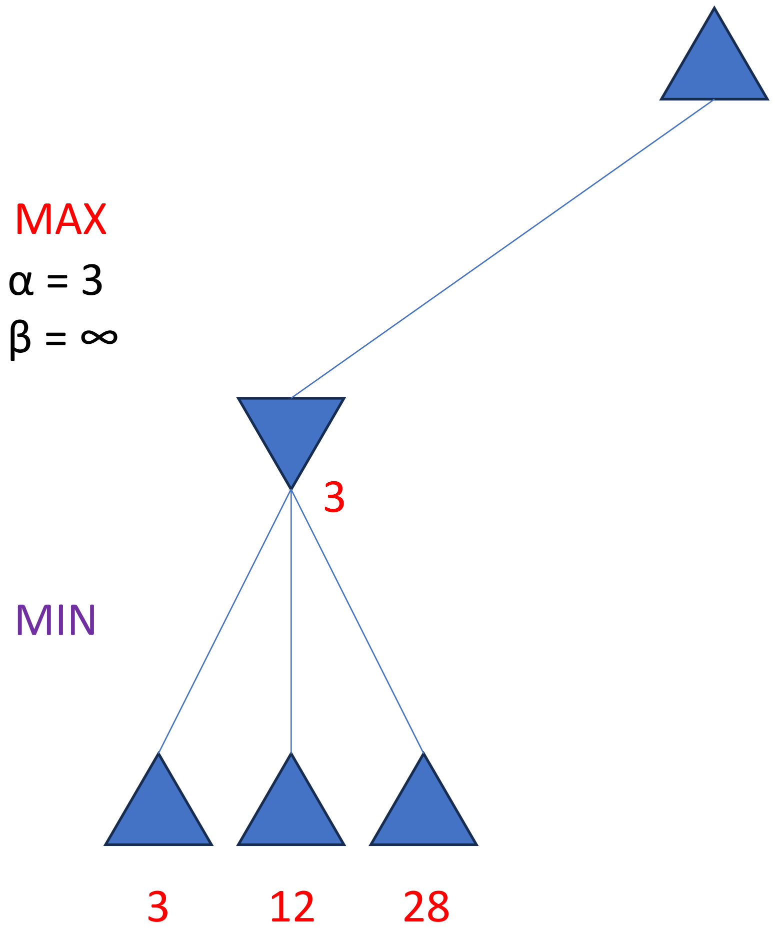
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```



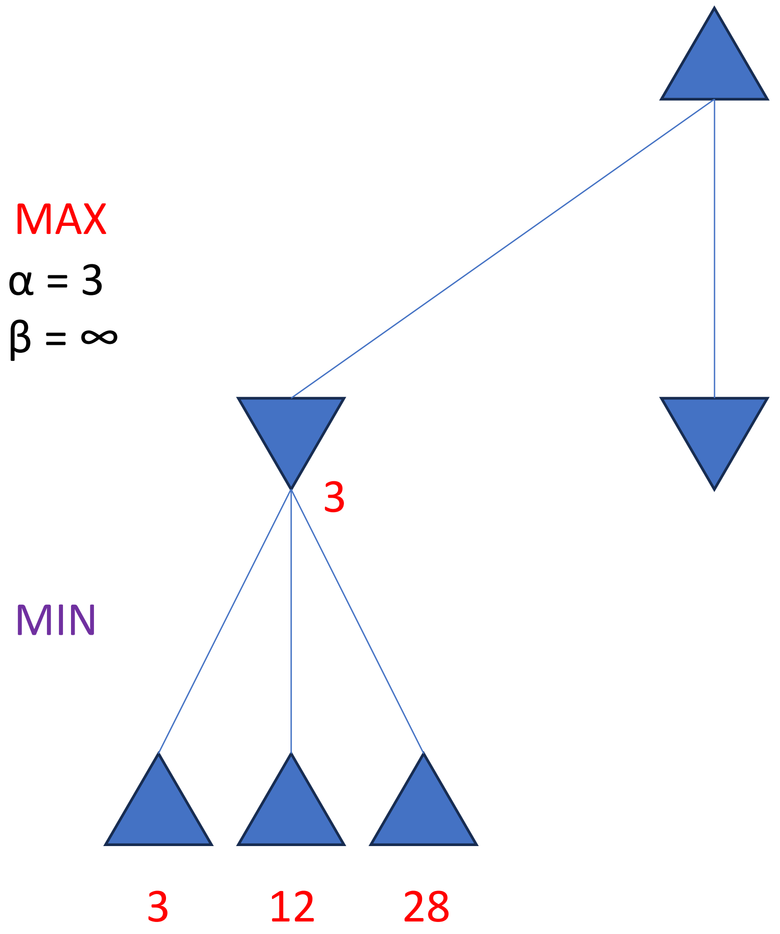
Alpha-beta Pruning

```
def alpha_beta_search(state):
   v = max_value(state, -∞, ∞)
   return action in successors(state) with value v

def max_value(state, α, β):
   if is_terminal(state): return utility(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state, α, β))
      if v >= β: return v
      α = max(α, v)
   return v

def min_value(state , α, β):
   if is_terminal(state): return utility(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
      if v <= α: return v
      β = min(β, v)
   return v
```



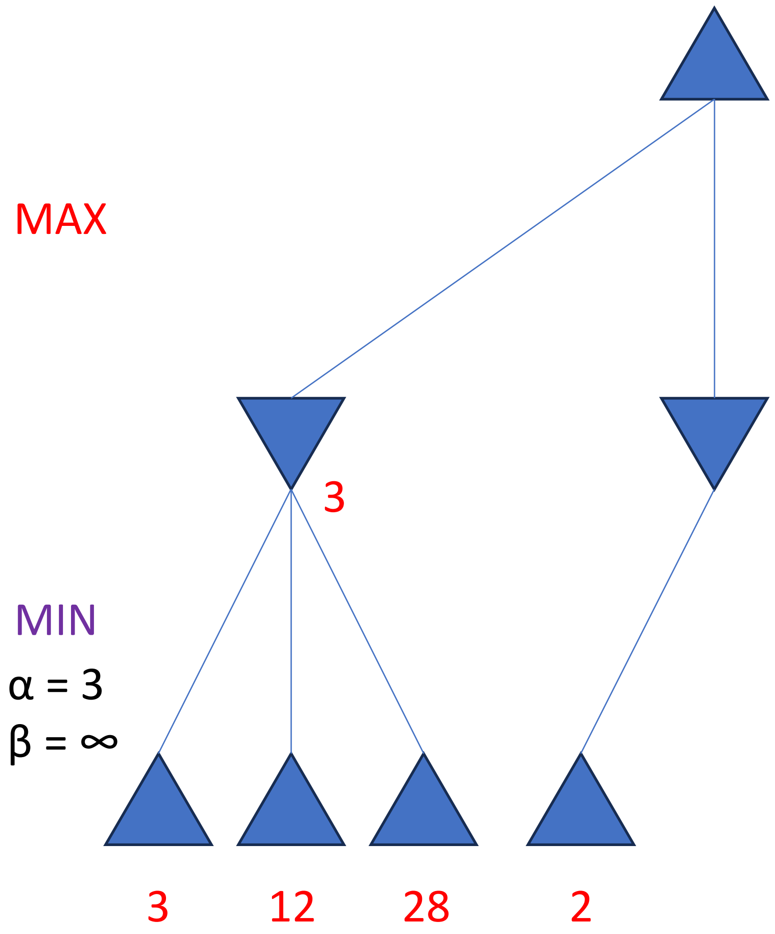
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```



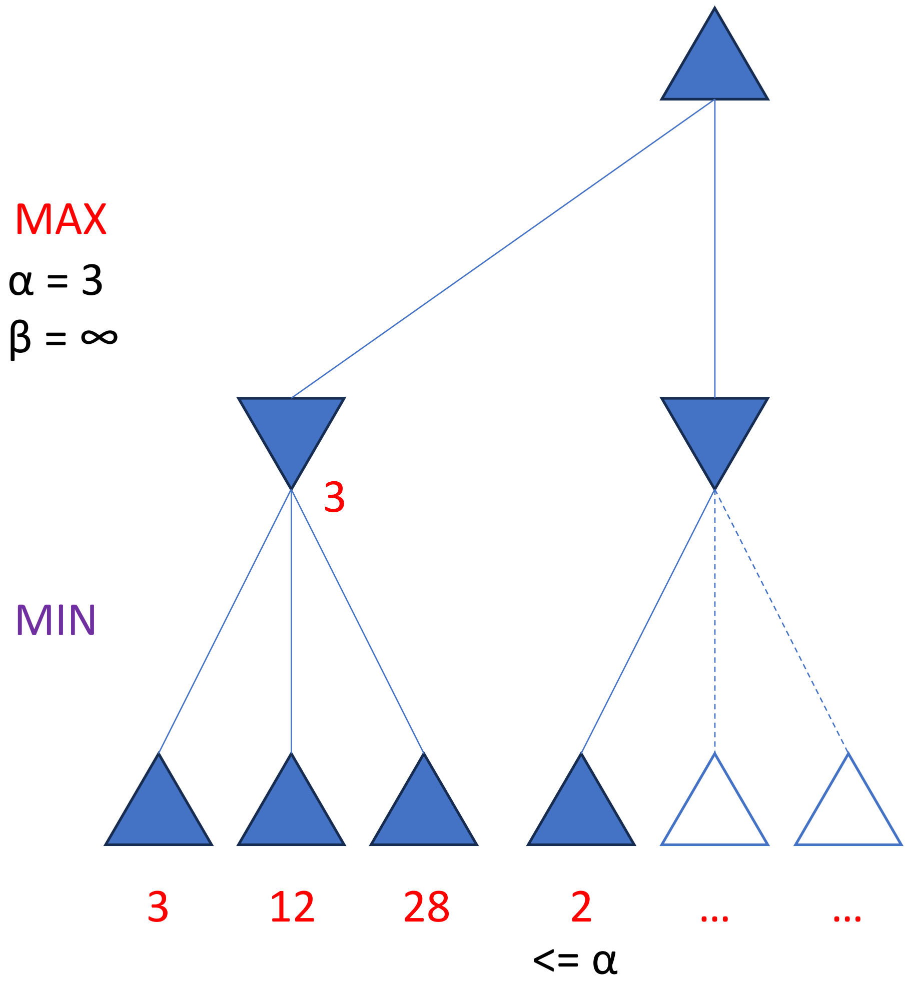
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



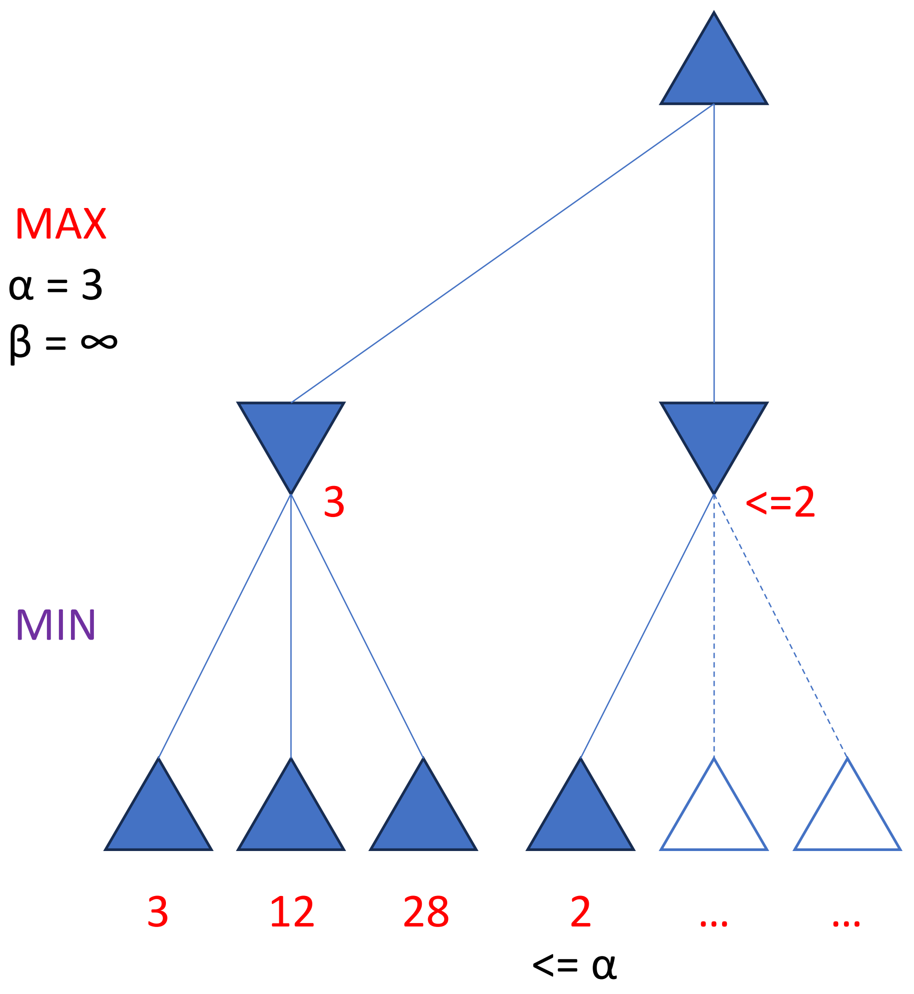
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```



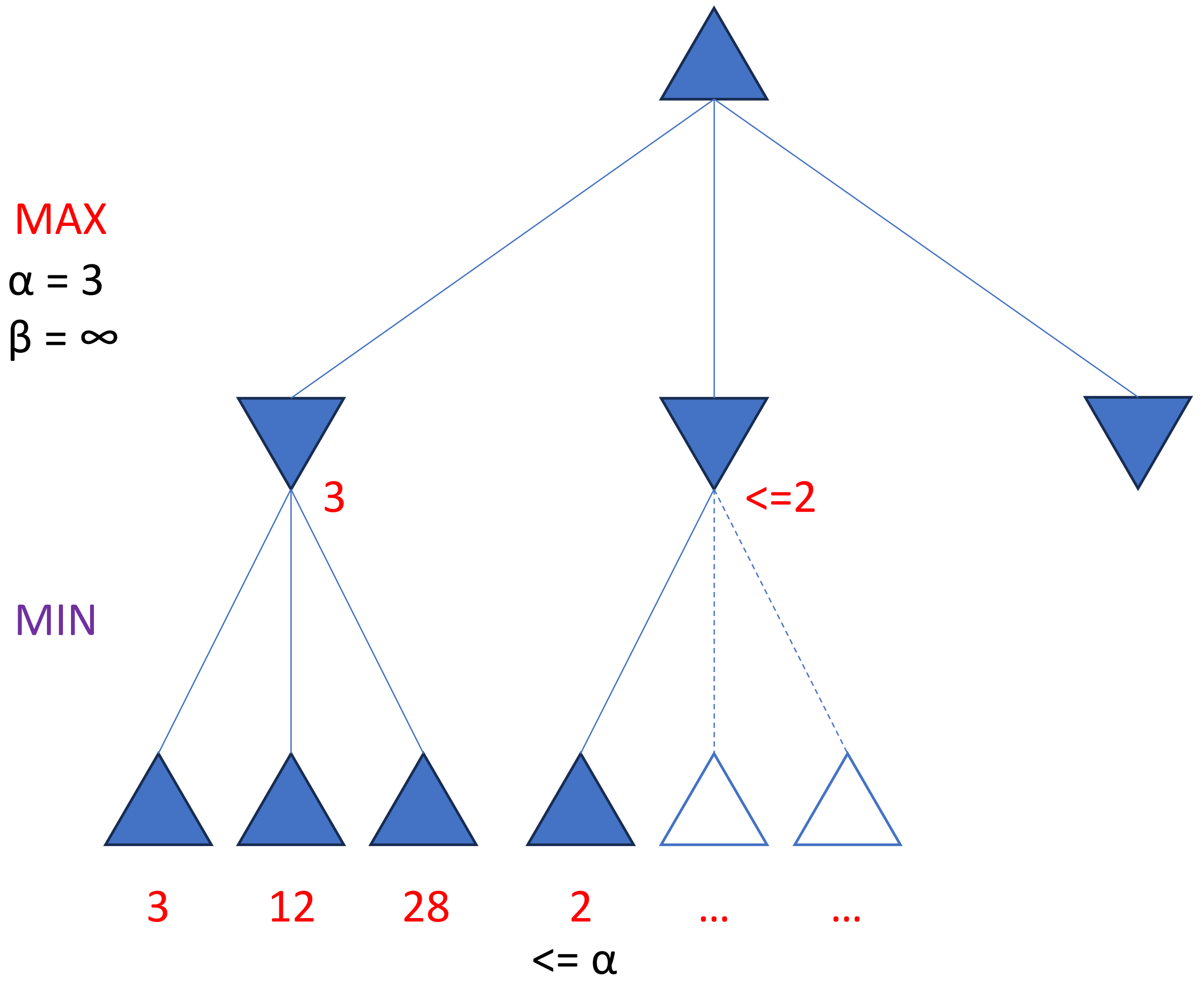
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



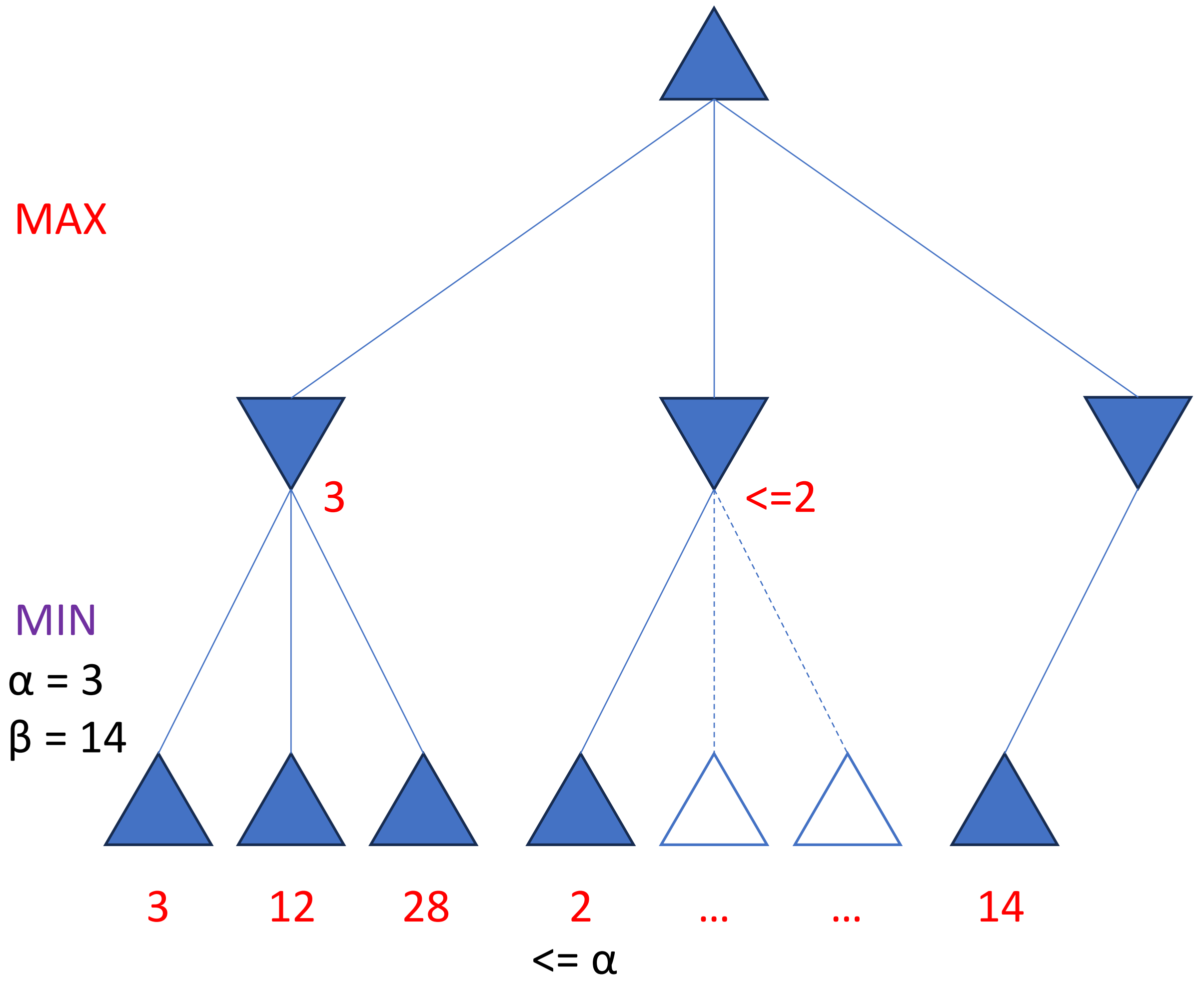
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



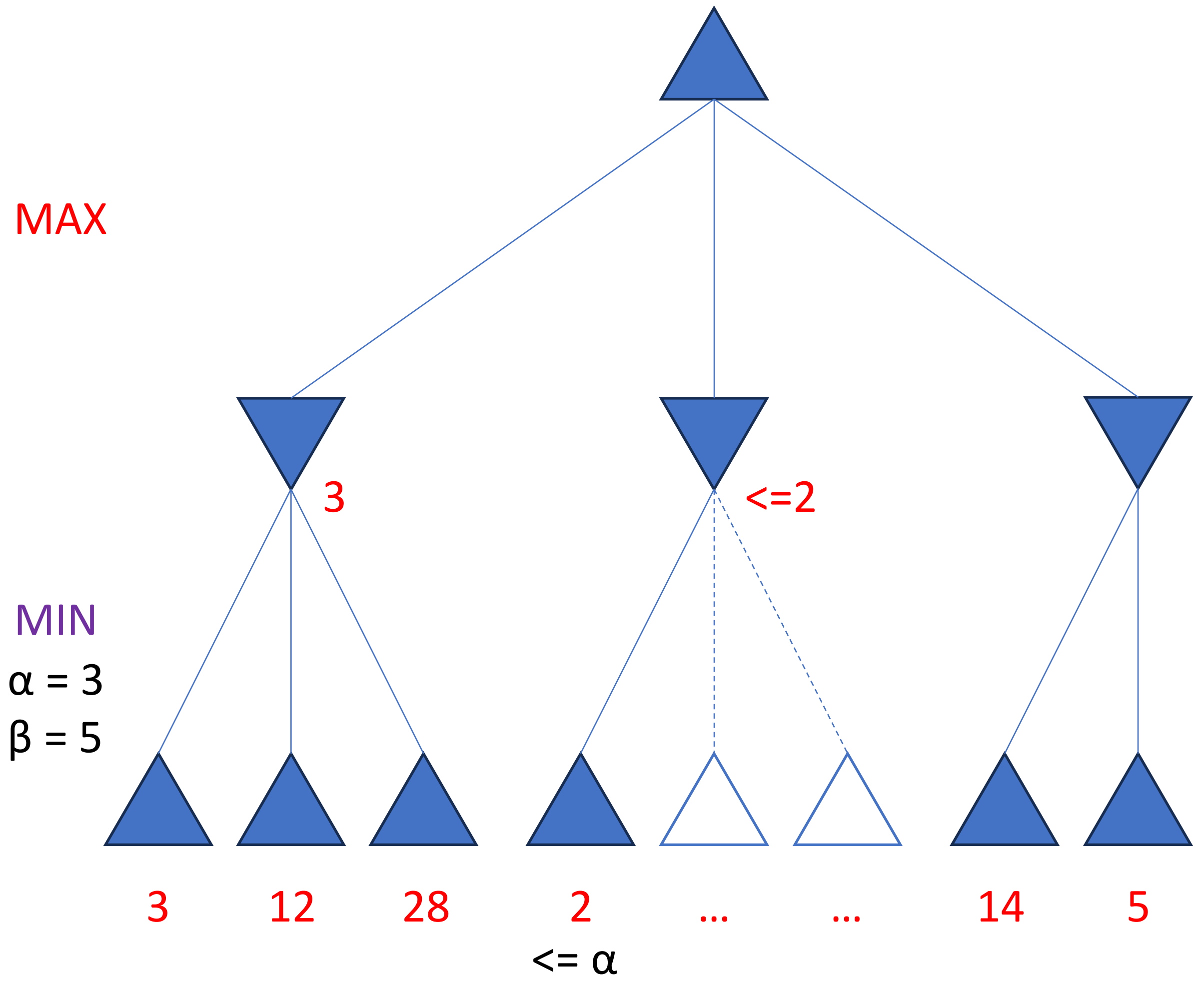
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



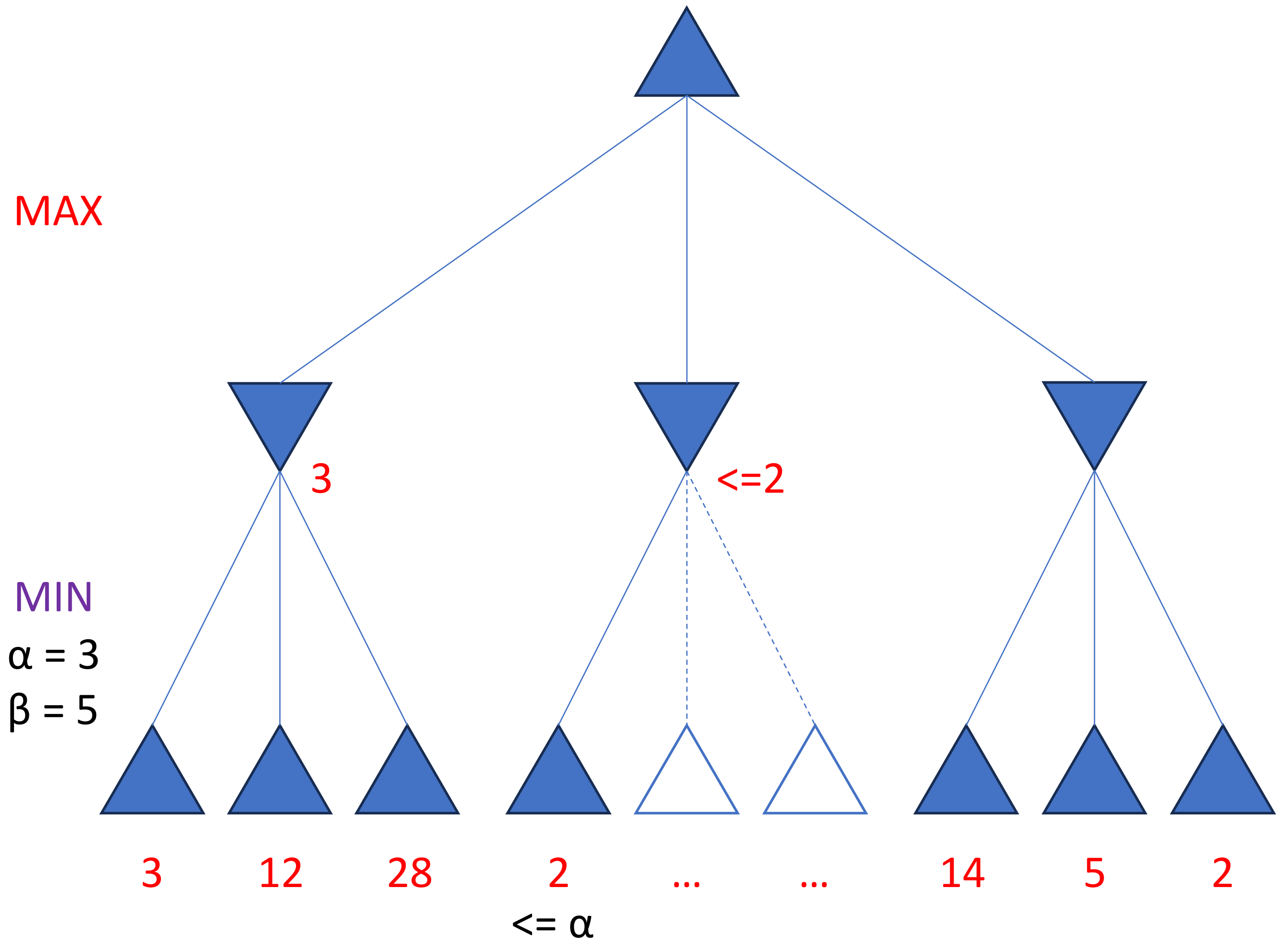
Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



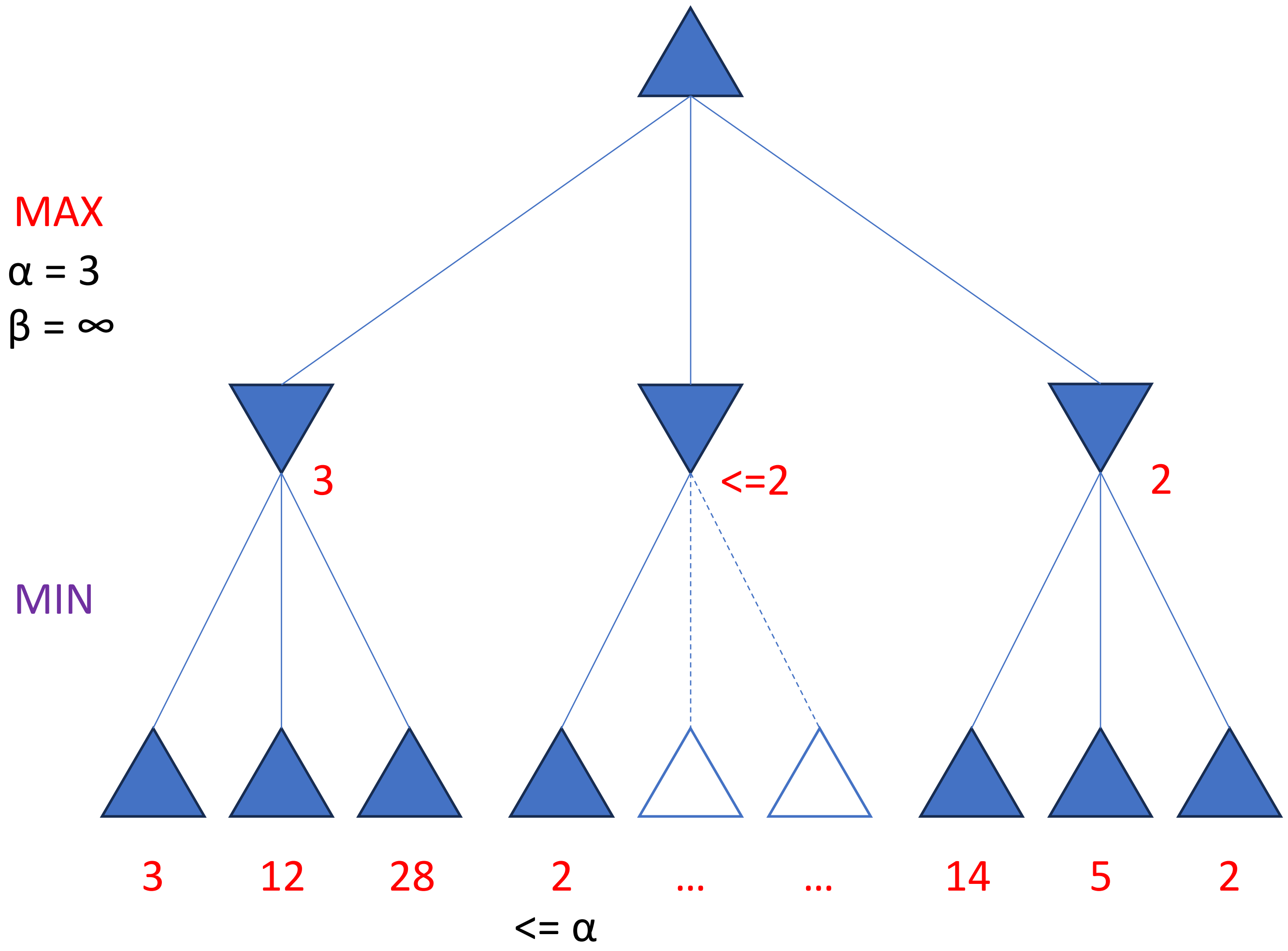
Alpha-beta Pruning

```
def alpha_beta_search(state):
   v = max_value(state, -∞, ∞)
   return action in successors(state) with value v

def max_value(state, α, β):
   if is_terminal(state): return utility(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state, α, β))
      if v >= β: return v
      α = max(α, v)
   return v

def min_value(state , α, β):
   if is_terminal(state): return utility(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
      if v <= α: return v
      β = min(β, v)
   return v
```



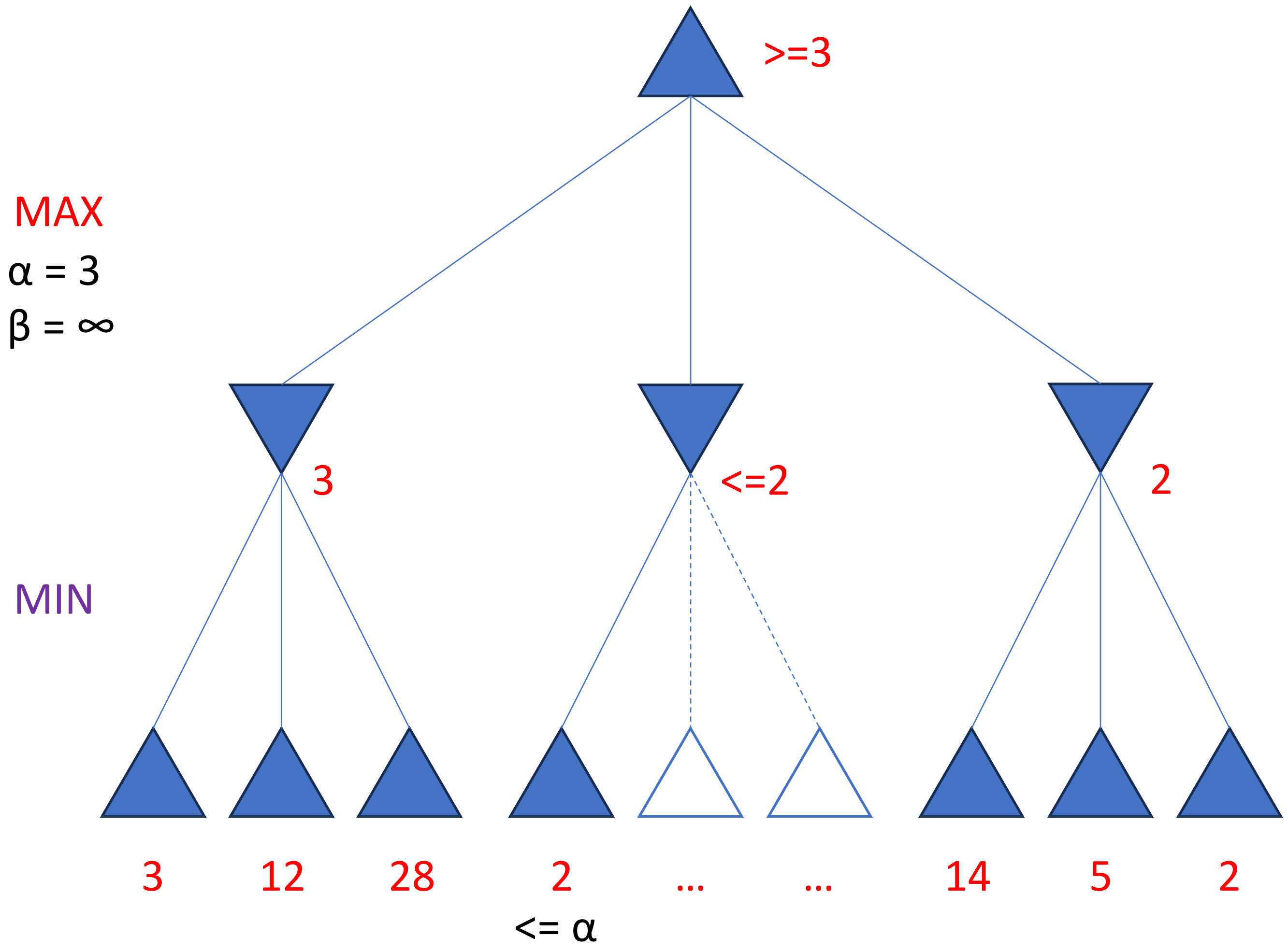
Alpha-beta Pruning

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state, α, β))
        if v >= β: return v
        α = max(α, v)
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
        if v <= α: return v
        β = min(β, v)
    return v
```



Alpha-beta Pruning

```
def alpha_beta_search(state):
  v = max_value(state, -∞, ∞)
  return action in successors(state) with value v

def max_value(state, α, β):
  if is_terminal(state): return utility(state)
  v = -∞
  for action, next_state in successors(state):
    v = max(v, min_value(next_state, α, β))
    if v >= β: return v
    α = max(α, v)
  return v

def min_value(state , α, β):
  if is_terminal(state): return utility(state)
  v = ∞
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    if v <= α: return v
    β = min(β, v)
  return v
```



Alpha-beta Pruning: Analysis

- Pruning doesn't affect final result
- Good move ordering improves effectiveness of pruning
  - "Perfect ordering": $O\left(b^{\frac{m}{2}}\right)$
- An example of **meta-reasoning**: reasoning about which computations to do first

Handling Huge/Infinite Game Trees

```
def minimax(state):
   v = max_value(state)
   return action in successors(state) with value v

def max_value(state):
   if is_terminal(state): return utility(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state))
   return v

def min_value(state):
   if is_terminal(state): return utility(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
   return v
```

```
def minimax_with_cutoff(state):
   v = max_value(state)
   return action in successors(state) with value v

def max_value(state):
   if is_cutoff(state): return eval(state)
   v = -∞
   for action, next_state in successors(state):
      v = max(v, min_value(next_state))
   return v

def min_value(state):
   if is_cutoff(state): return eval(state)
   v = ∞
   for action, next_state in successors(state):
      v = min(v, max_value(next_state))
   return v
```

is_cutoff

Return true if

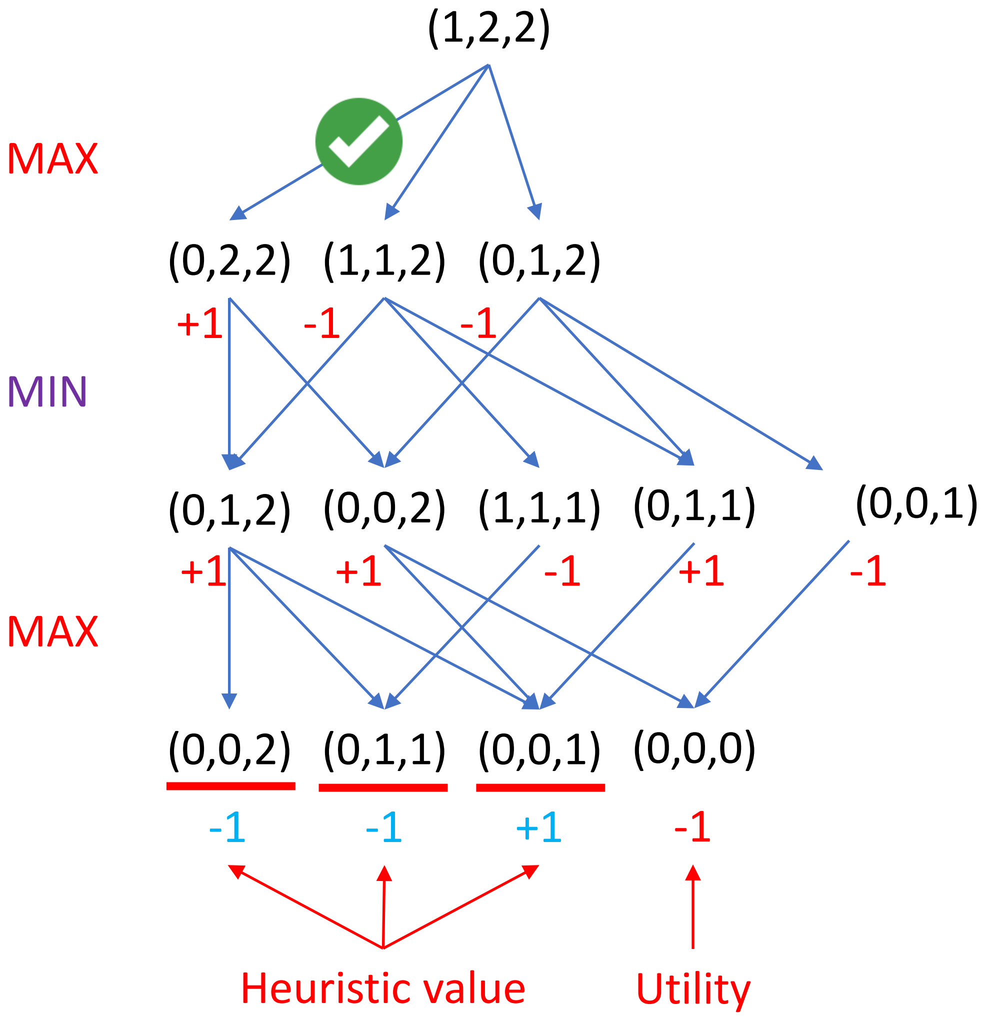
- State is terminal
- Cutoff is reached (e.g., time/depth limit is exceeded)

eval

Return

- Utility for terminal states
- Heuristic value for non-terminal states

Handling Huge/Infinite Game Trees




```
def minimax_with_cutoff(state):
    v = max_value(state)
    return action in successors(state) with value v

def max_value(state):
    if is_cutoff(state): return eval(state)
    v = -∞
    for action, next_state in successors(state):
        v = max(v, min_value(next_state))
    return v

def min_value(state):
    if is_cutoff(state): return eval(state)
    v = ∞
    for action, next_state in successors(state):
        v = min(v, max_value(next_state))
    return v
```

is_cutoff

Return true if

- State is terminal
- Cutoff is reached (e.g., time/depth limit is exceeded)

eval

Return

- Utility for terminal states
- Heuristic value for non-terminal states

Evaluation Functions

How to design a good evaluation function?

- Need to understand the problem domain

For chess, typically linear weighted sum of features:

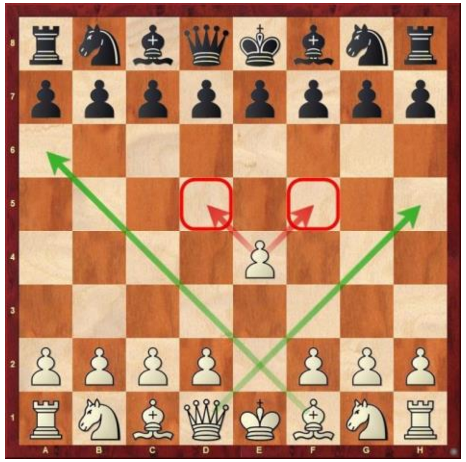
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s), w_i \in \mathbb{R}$$

Example features:

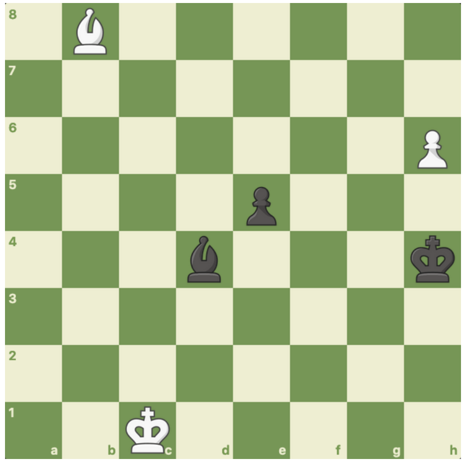
- Number of white queens – number of black queens
- Number of player's pieces – number of opponent's pieces

Optimizing the Search

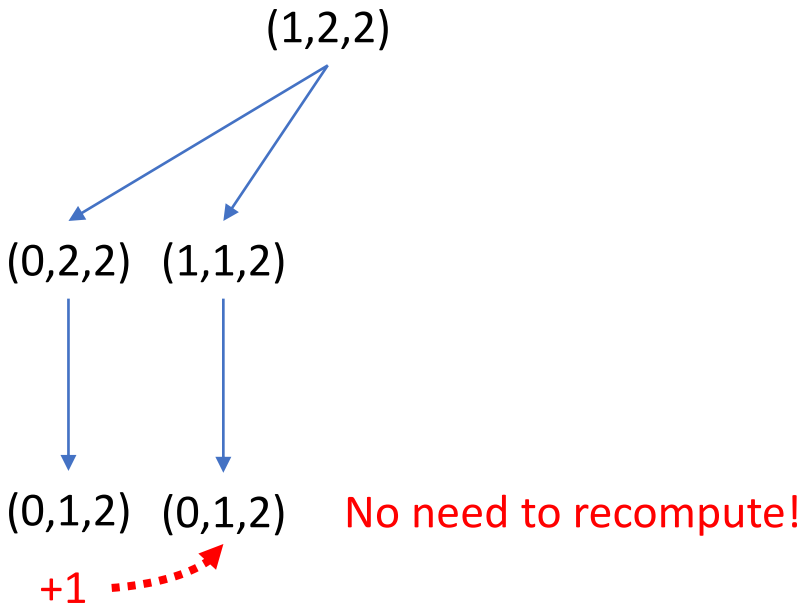
- **Transposition table**
- **Precomputation of best moves** in the opening and closing games



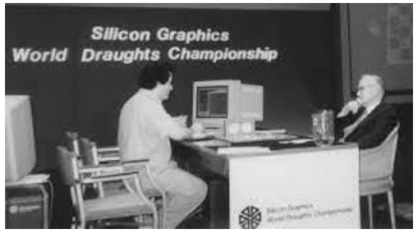
Credit: thechessworld.com



Credit: chess.com



AI in Games



Credit: AAAI

Credit: IEEE Spectrum

Credit: Guardian

- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.
- **Chess:** Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- **Othello:** human champions refuse to compete against computers, who are too good.
- **Go:** AlphaGo defeated 18-time world champion Lee Sedol in 2016. Instead of using probability algorithms hard-coded by human programmers, AlphaGo uses neural networks to estimate its probability of winning.

Summary: Adversarial Search

- Games vs Search Problems:
  - Search problems: **predictable** world, deterministic and stochastic
  - Games: **“unpredictable”** opponent, strategic
- Minimax:
  - Assume **opponent behaves optimally**
  - Pick move that **maximizes** player’s **utility**
- Alpha-beta pruning
  - **Prune branches** that are useless (**will not change the decision**)
- Handling large/infinite game trees:
  - **Cutoff** the search in the middle of the game and use **evaluation function**
- Optimizing the search:
  - Transposition table: **reuse** values
  - **Precomputation** of good moves

Summary

- Informed search algorithms
  - Greedy best-first search: f(n) = g(n)
  - A* search: f(n) = g(n) + h(n)
  - Heuristics: admissibility, consistency, dominance
  - Variants of A*: IDA*, SMA*
- Local search
  - Hill climbing: pick best neighbor, repeat
  - Simulated annealing: allow bad moves from time to time, escape local optima
  - Beam search: parallel hill-climbing
  - Genetic algorithms: inspired by evolution
- Adversarial search
  - Games: opponent "unpredictable"
  - Minimax: max then min value
  - Alpha-beta pruning: prune unuseful branch
  - Handling large/infinite game trees: cutoff, eval
  - Optimizing the search: transposition table, precomputation

Coming Up Next Week

- **Machine Learning**
- **Decision Trees**
  - Entropy and Information Gain
  - Different types of attributes
  - Pruning
- **Performance Measures**
  - Precision, Recall, F1
  - ROC

To Do

- **Lecture Training 3**
  - +100 Free EXP
  - +50 Early bird bonus
- **Problem Set 1**
  - Due Saturday, 2nd September (Tomorrow)