CS2109S: Introduction to AI and Machine Learning

Lecture 3:
Local and Adversarial Search

28 January 2025

DO NOT CLOSE YOUR POLLEVERYWHERE APP

There will be activities ahead

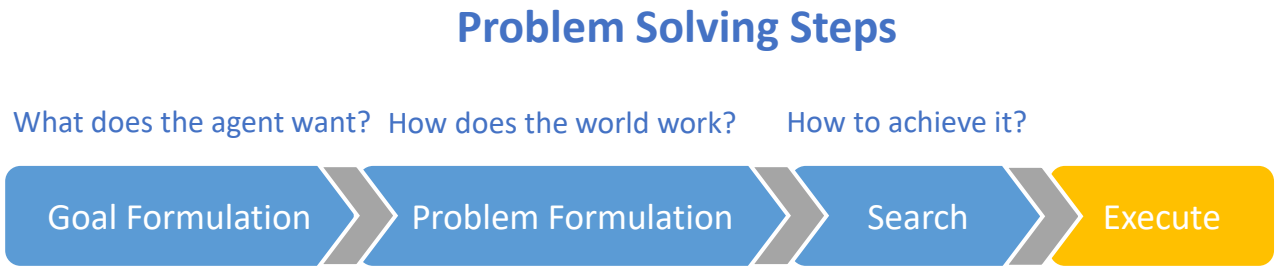
Chinese New Year (CNY) Tutorial

- **Optional**, **500 free *"Hong Bao"* EXP** for everyone
- Offline, online, and recorded tutorials are available. Please check the announcement in Coursemology.

Recall: Designing an Agent

To solve a problem using search, the agent needs to have:

- A goal, or a set of goals
- **A model of the environment**
- **A search algorithm**



Outline

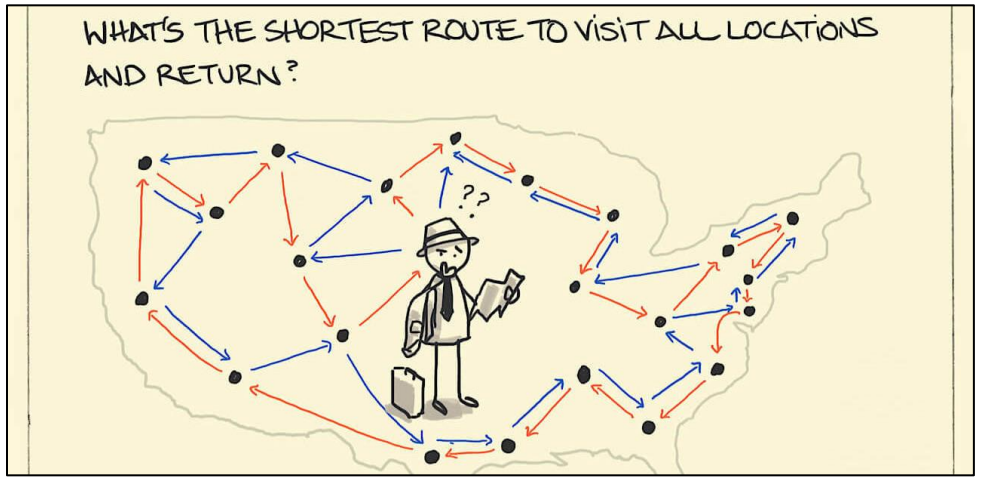
- Local Search
  - Problem Formulation
  - Hill-climbing
- Adversarial Search
  - Problem Formulation
  - Minimax
  - Alpha-beta Pruning
  - “Real-world” Games

Outline

- **Local Search**
  - Problem Formulation
  - Hill-climbing
- Adversarial Search
  - Problem Formulation
  - Minimax
  - Alpha-beta Pruning
  - “Real-world” Games

Difficult Search & Optimization Problems

- **(NP-hard) combinatorial optimization problems**
  - Example: Travelling Salesperson Problem (TSP)
- **Constraint Satisfaction Problems (CSPs)**
  - Example: Map coloring, Scheduling problems
- **Continuous Optimization Problems**
  - Example: training neural networks, non-linear programming problems



Credit: sketchplanations

Credit: Classic Computer Science Problems in Java, David Kopec

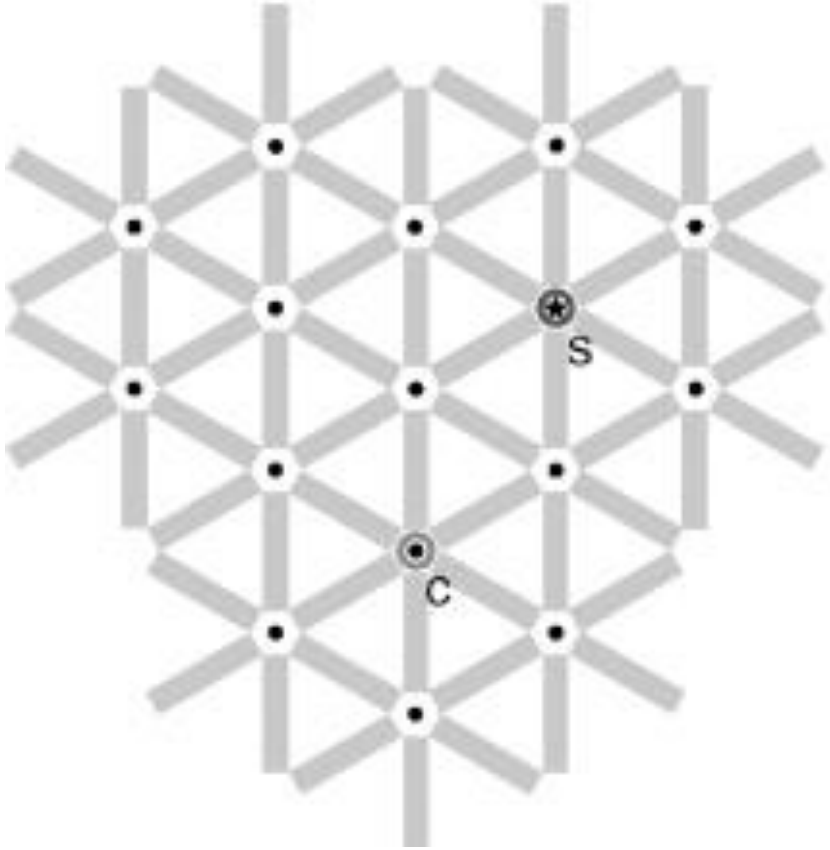
Previously: Systematic Search

- Traverse the state space in a **systematic** manner
- Typically, **complete** and **optimal** under certain assumptions
  - E.g., BFS, UCS, IDS, A*
- Solving difficult search and optimization problems with this paradigm is intractable: not solvable within a reasonable amount of time

Local Search

- Traverse the state space by considering only **locally-reachable** states:
  - Start at a (random) position in a state space
  - Iteratively move from a state to another neighboring state via perturbation (most common) or construction (more on these later).
  - The solution is the final state
- Typically, **incomplete** and **suboptimal**
- **Any-time** property: longer runtime, better solution
  - Can obtain a “good enough” solution for difficult search and optimization problems

Local Search



Holger H. Hoos, Thomas Stützle, in Stochastic Local Search, 2005

State Space and Search Space

- **State Space**:
  All possible configurations or states of a system.
- **Search Space**:
  The subset of the **state space** that will be explored to find a solution.

Local Search – Type

- **Perturbative search** (most common, discussed in this lecture)
  - Search space: complete candidate solutions
  - Search step: modification of one/more solutions
  - Example (path finding): swap a path with another path
- **Constructive search**
  - Search space: partial candidate solutions
  - Search step: extension with one/more solution components
  - Example (path finding): from one city, go to nearest neighbor city

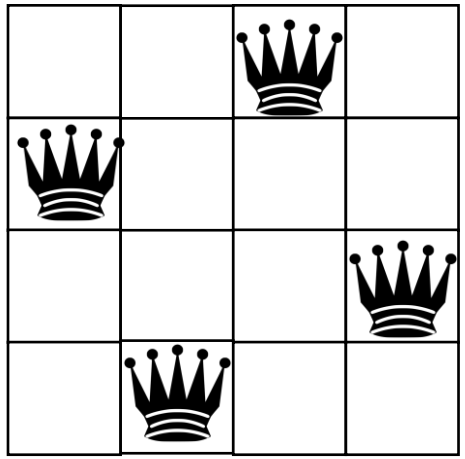
Problem Formulation in Systematic Search

- **States:** Represent the actual configurations or conditions of the problem. Each state corresponds to a specific situation or configuration of the problem domain.
- **Initial State:** The starting configuration
- **Goal Test / State(s)**
- **Actions**
- **Transition model**
- **Action cost function**

Perturbative Problem Formulation in Local Search

- **States:** Represent different configurations or candidate solutions within the search space. They may not directly map to an actual problem state but are used to represent potential solutions.
- **Initial State:** The starting configuration (candidate solution).
- **Goal Test (optional):** Check if a current state is the desired solution.
- **Successor Function:** A function that generates neighboring states (candidate solutions) by applying small modifications or steps from the current state. This defines the local search space.

Example: N-Queens



States	Configurations of queens on the board
Initial state	Random placement of queens, one queen on each column
Goal test	No two queens threaten each other, i.e., no two queens share the same row, column, or diagonal
Successor function	Move a queen to a different row within the same column

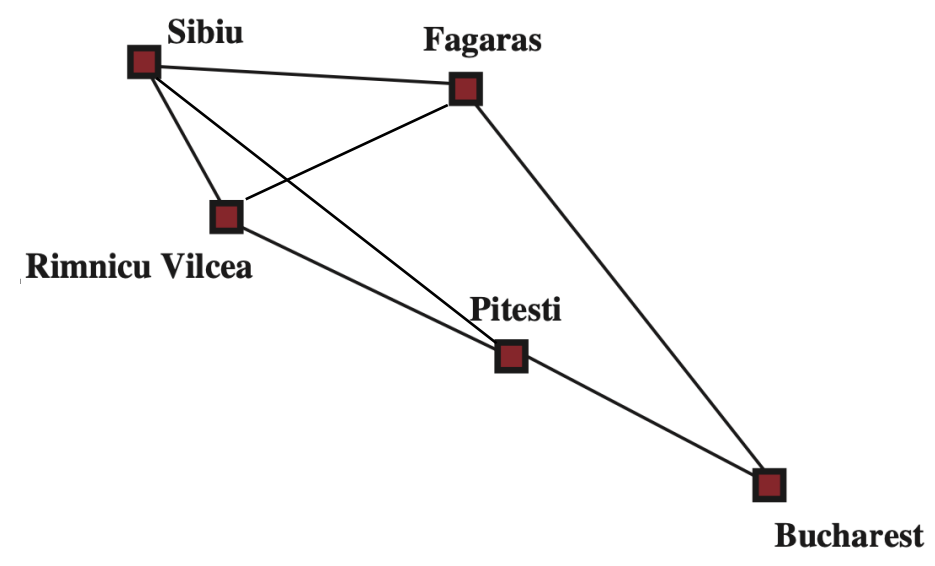
Example: Minimizing a Function

Find an **integer** x that minimize $f(x) = (x-2)^2$

States	All possible integers
Initial state	Random integer
Goal test	None
Successor function	Make small changes to x, either by increasing or decreasing it by 1

PollEverywhere

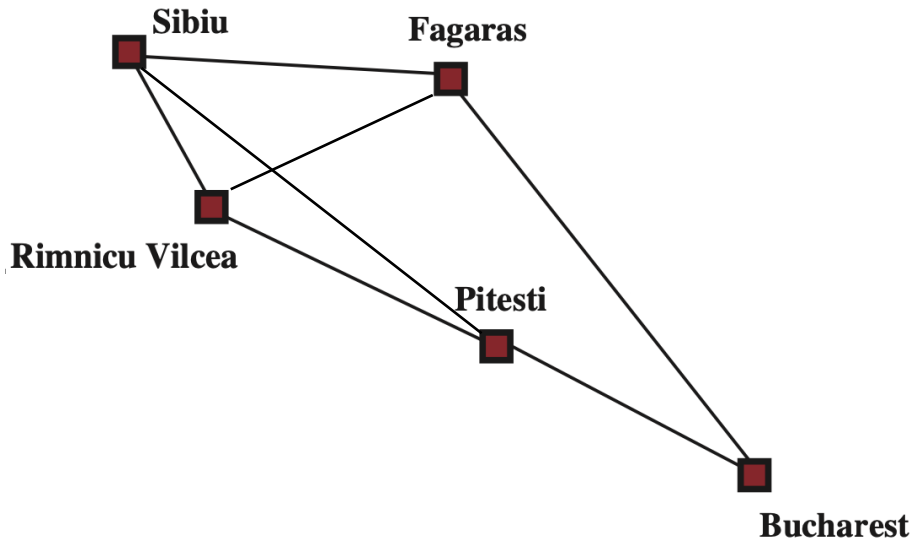
PollEV.com



In local search, the solution is the final state rather than the path taken to reach it. Can local search be used to solve shortest path problems?

A. Yes
B. No
C. It depends

Example: Path Finding



States	Paths from Sibiu to Bucharest
Initial state	Random path from Sibiu to Bucharest
Goal test	None
Successor function	Replace a subpath between any arbitrary two cities in the current path with other paths

Example:

Sibiu – Fagaras – Rimnicu Vilcea – Pitesti – Bucharest

↓

Sibiu – Fagaras – Bucharest

Sibiu – Fagaras – Rimnicu Vilcea – Sibiu – Pitesti – Bucharest

Evaluation Function

An **evaluation function** (also known as a **fitness function** or **objective function**) is a mathematical function **eval**(state) used to assess the quality or desirability of a state within a given search space.

Depending on the problem and the algorithm, we may want to either minimize or maximize this function.

Evaluation Function - Examples

- In the **n-queens problem**, an evaluation function can be the number of "safe" queens, i.e., queens not attacking each other in the same row, column, or diagonal. The goal is to maximize this number.

- In the problem of **minimizing a function**, the evaluation function is the (objective) function itself (below). The goal is to minimize it.

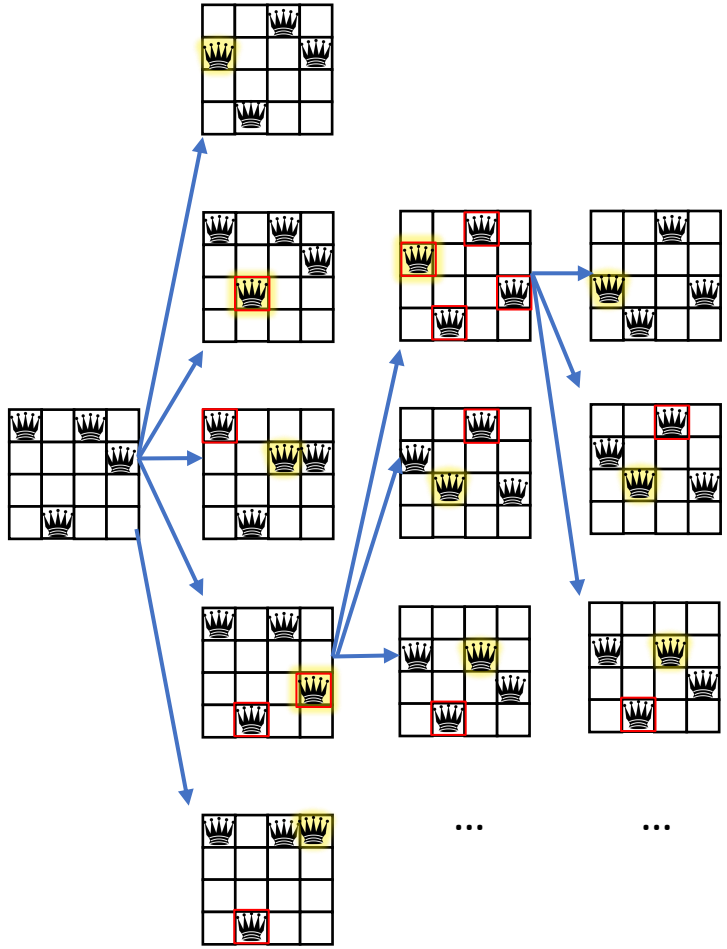
$$f(x) = (x-2)^2$$

Note: A minimization objective $f(x)$ can be transformed into a maximization objective by defining a new objective: $\hat{f}(x) = -f(x)$, and vice versa.

Hill climbing algorithm

(Steepest Ascent Search, Greedy Local Search)

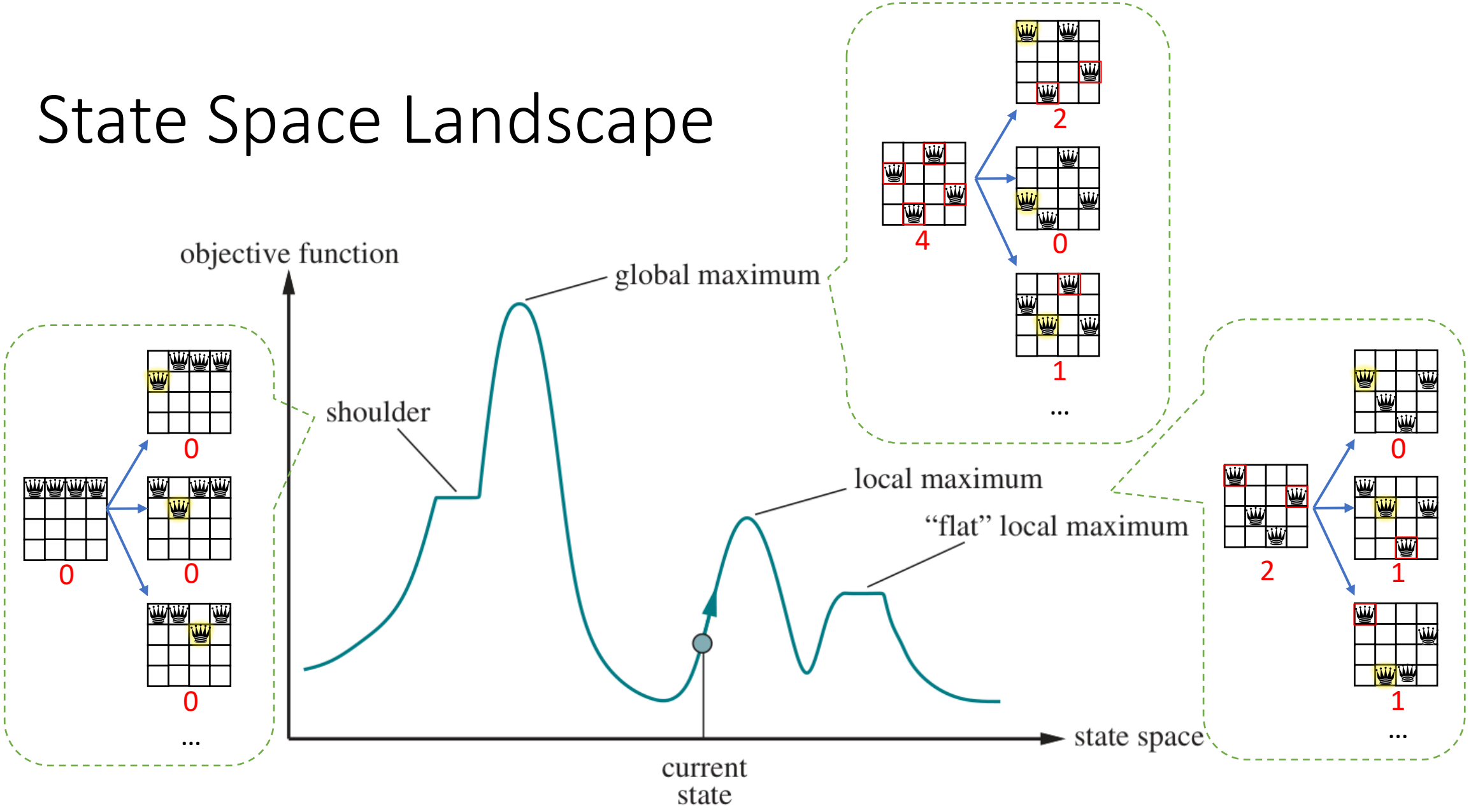
```
current_state = initial_state
while True:
    best_successor = a highest-valued successor state of current_state
    if eval(best_successor) <= eval(current_state):
        return current_state
    current_state = best_successor
```



State Space Landscape

- **Global Maximum**: The overall **highest point** or solution across the entire search space that represents the optimal solution.
- **Local Maximum**: A point in the search space that is **higher than its immediate neighbors**, but there may be a higher value elsewhere in the search space. It represents a local optimum, not necessarily the best solution globally.
- **Shoulder**: A region in the search space where the objective function has a **flat** area, making it difficult for the algorithm to move past it towards a better solution.

State Space Landscape



Summary: Local Search

- There are problems with a very large state space
  - Good enough solution is okay
- Local Search
  - Hill-climbing (greedy): **pick the best** among neighbours, repeat
- State space landscape
  - Local maximum
  - Global maximum
  - Shoulder

Further Reading (Optional)

- Simulated annealing (AIAMA 4.1.2)
- Escape techniques
  - Random restarts (AIAMA 4.1.1)
  - Tabu search (AIAMA Ch. 4, pg. 154)
- Local beam search (AIAMA 4.1.3)
- Genetic algorithm (AIAMA 4.1.4)

Break

A BRIEF HISTORY OF CHESS

TEDEd

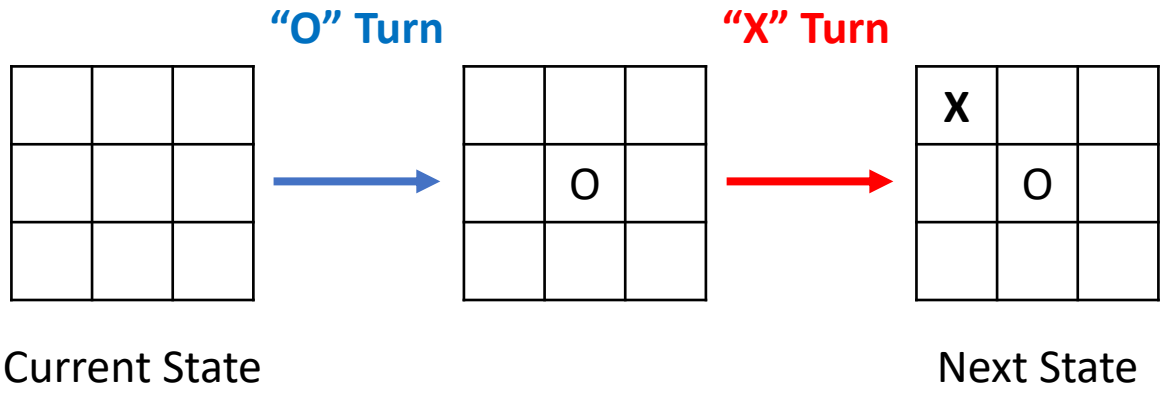
Outline

- Local Search
  - Problem Formulation
  - Hill-climbing
- **Adversarial Search**
  - Problem Formulation
  - Minimax
  - Alpha-beta Pruning
  - “Real-world” Games

Multi-agent Problems

Multi-agent problems involve **multiple agents** interacting within an environment, where each agent has its own goal(s) and decision-making process. The agents may cooperate, compete, or both, depending on the problem.

The next state of the environment depends not only on our action but also on the action of other agents, which are assumed to be strategic, i.e., they try to reach their goal(s).



Multi-agent Problems – Types

- **Cooperative:** Agents work together towards a common goal
  - Example: robotic swarms
- **Competitive:** Agents have opposing goals
  - Example: chess, go
- **Mixed-Motive:** Some cooperation and some competition
  - Example: auctions, the Prisoner's Dilemma
- …

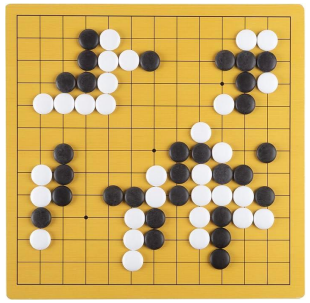
Competitive Multi-agent Problems

Conflicting Goals: Agents have opposing goals, and one agent's success might directly impact the failure of the other.

Example:

- **Adversarial Games**: Players (agents) compete against each other.
  - **Two-player zero-sum games**: one player's gain is the other player's loss. Example: chess, go, tic-tac-toe, 2-player poker.
  - …
- …

Credit: chess.com



Credit: Amazon.sg

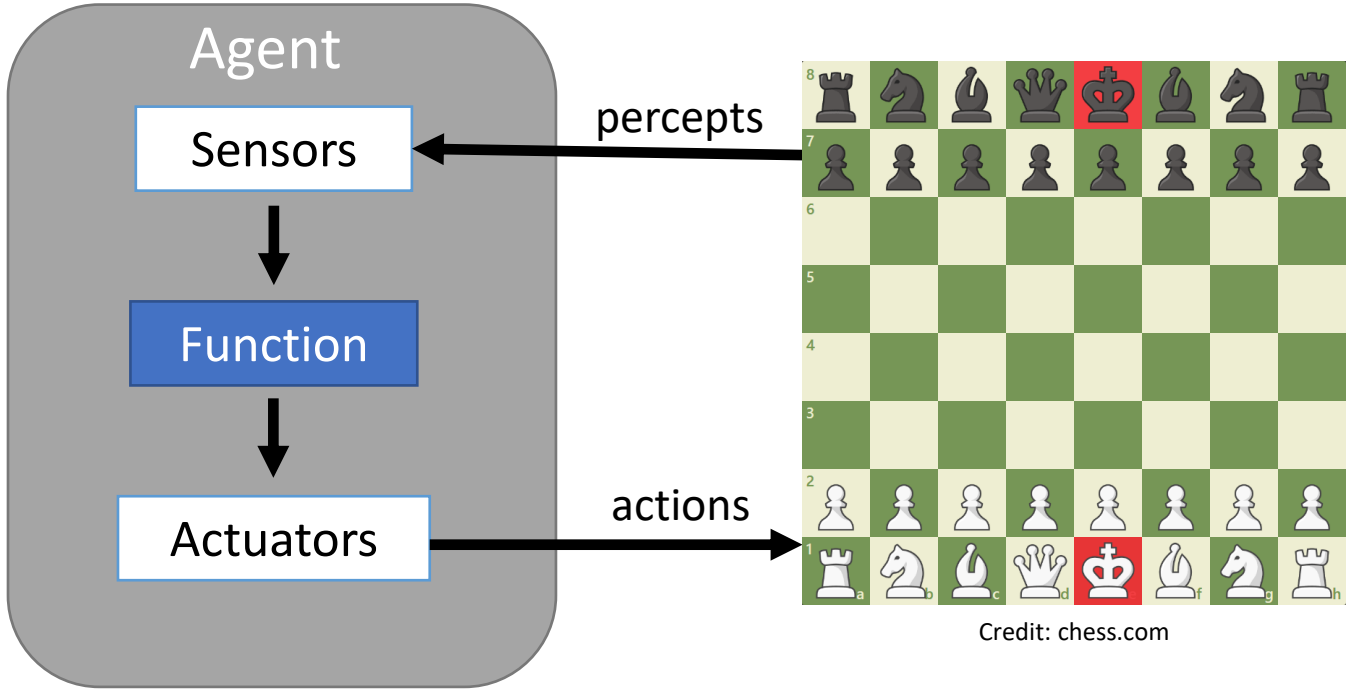
Credit: Wikipedia

Designing an Agent

for **competitive multi-agent problems** (i.e., "classical" **adversarial games**)

Utility-based agent

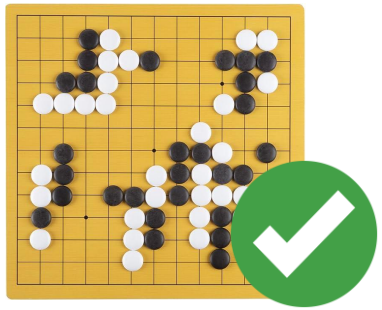
Adversarial search



Credit: chess.com

Fully observable
Deterministic
Discrete

No infinite runs
Two-player zero-sum
Turn-taking



Credit: Amazon.sg

Credit: Wikipedia

Credit: Toys R Us

Games – Important Terms

- **Player** (Agent): An entity making decisions or taking actions in a game.
- **Turn**: A phase where a player makes a move, after which control passes to the next player.
- **Move** (Action): A decision or action by a player that changes the game state.
- **End State** (Terminal State): A state where the game ends, with no further moves possible.
- **Winning Condition**: Criteria that define when a player wins the game.
- **Game Tree** (Game Graph or Search Tree): A diagram of possible game states, where nodes represent states and edges represent moves.

Problem Formulation in Systematic Search

- **States**
- **Initial State**
- **Goal Test / State(s)**

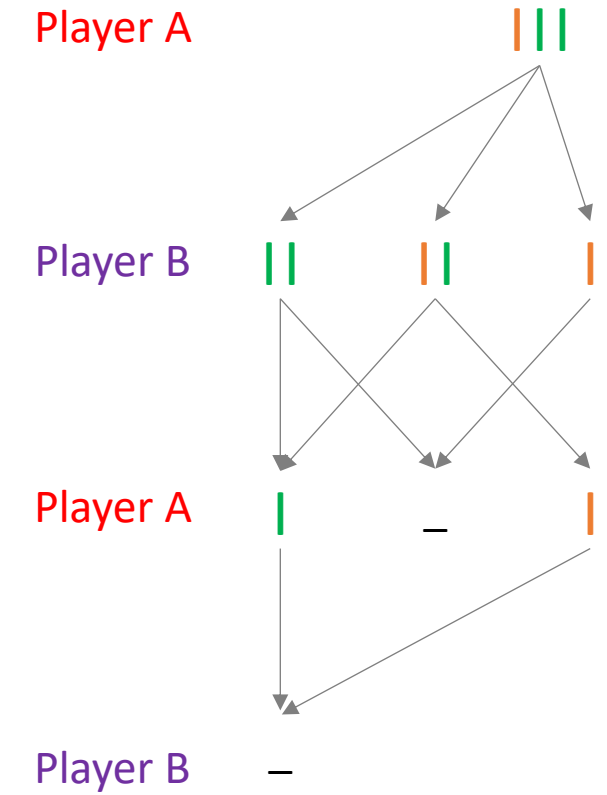
- **Actions**
- **Transition**
- **Action cost function**

Problem Formulation in Adversarial Search

- **States**
- **Initial State**
- **Terminal States** : states where the game ends, such as win/lose/draw states.
  - This set of states could be defined as a function.
- **Actions**
- **Transition**
- **Utility Function**: output the value of a state from the perspective of our agent.
  - In many games, such as chess or tic-tac-toe, the outcome for player A is a win, loss, or draw. For simplicity, here, we assign utility values of +1, -1, and 0, respectively, to represent these outcomes. Our agent wants to maximize the utility.
  - The utility of non-terminal states is typically not known in advance and must be computed.

Example – Sticks

- Two piles of sticks (green and orange) are given
- Player can take any number of sticks from a single pile
- The player who cannot make any move loses



States	Configurations of piles
Initial state	A number of green sticks and orange sticks
Terminal states	No stick
Actions	Take 1,2,... sticks from one of the piles
Transition	Remove the sticks from the pile
Utility function	+1, 0, -1 for win, draw, lose, respectively

Minimax

- Algorithm for two-player zero-sum game
- Take the view of player A: try to maximize the outcome of the game
- Player B tries to minimize outcome of Player A, i.e., maximize their own outcome

Minimax

Expand function **expand**(state)

- Outputs a set of (action, next_state) pairs.

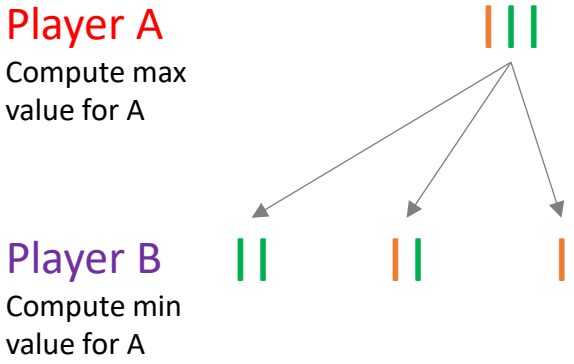
Terminal function **is_terminal**(state)

- Outputs true/false if state is a finished-game state.

Utility function: **utility**(state)

- Outputs a score for the state from the point of view of player A.

Minimax



Let's look at player A and a current state of the game

```
def max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞ // smallest value possible
    for next_state in expand(state):
        v = max(v, minimum value for player A in the next_state)
    return v
```

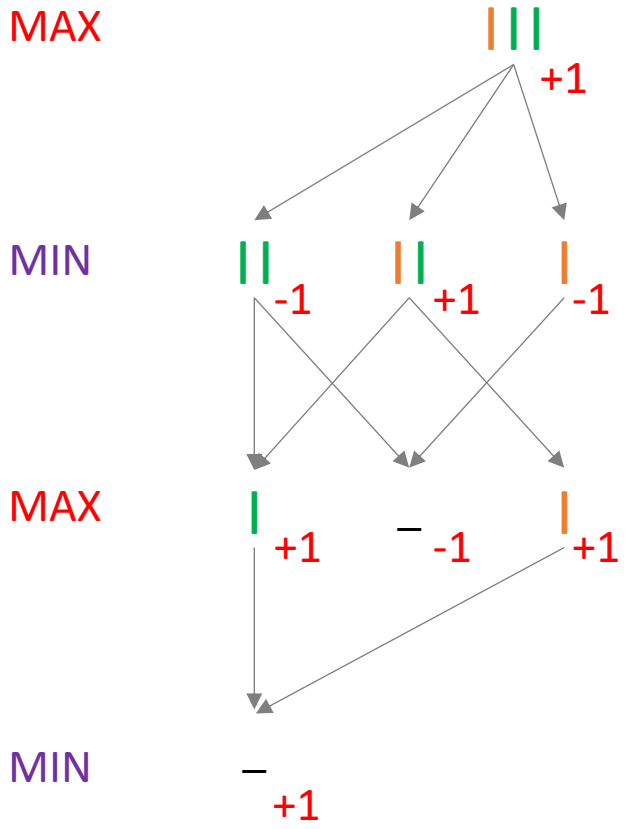
Assumes opponent play **optimally**:
trying to **minimize** player A value

Minimax

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

def max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞ // smallest value possible
    for next_state in expand(state):
        v = max(v, min_value(next_state))
    return v

def min_value(state):
    if is_terminal(state): return utility(state)
    v = ∞ // largest value possible
    for next_state in expand(state):
        v = min(v, max_value(next_state))
    return v
```

Player A = MAX

Player B = MIN



Minimax – Analysis

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

def max_value(state):
    if is_terminal(state): return utility(state)
    v = -∞ // smallest value possible
    for next_state in expand(state):
        v = max(v, min_value(next_state))
    return v

def min_value(state):
    if is_terminal(state): return utility(state)
    v = ∞ // largest value possible
    for next_state in expand(state):
        v = min(v, max_value(next_state))
    return v
```

Time Complexity?

Exponential w.r.t. maximum depth: $O(b^m)$

Branching factor b, max-depth m

Space Complexity? Polynomial

Complete? Yes, if tree is finite

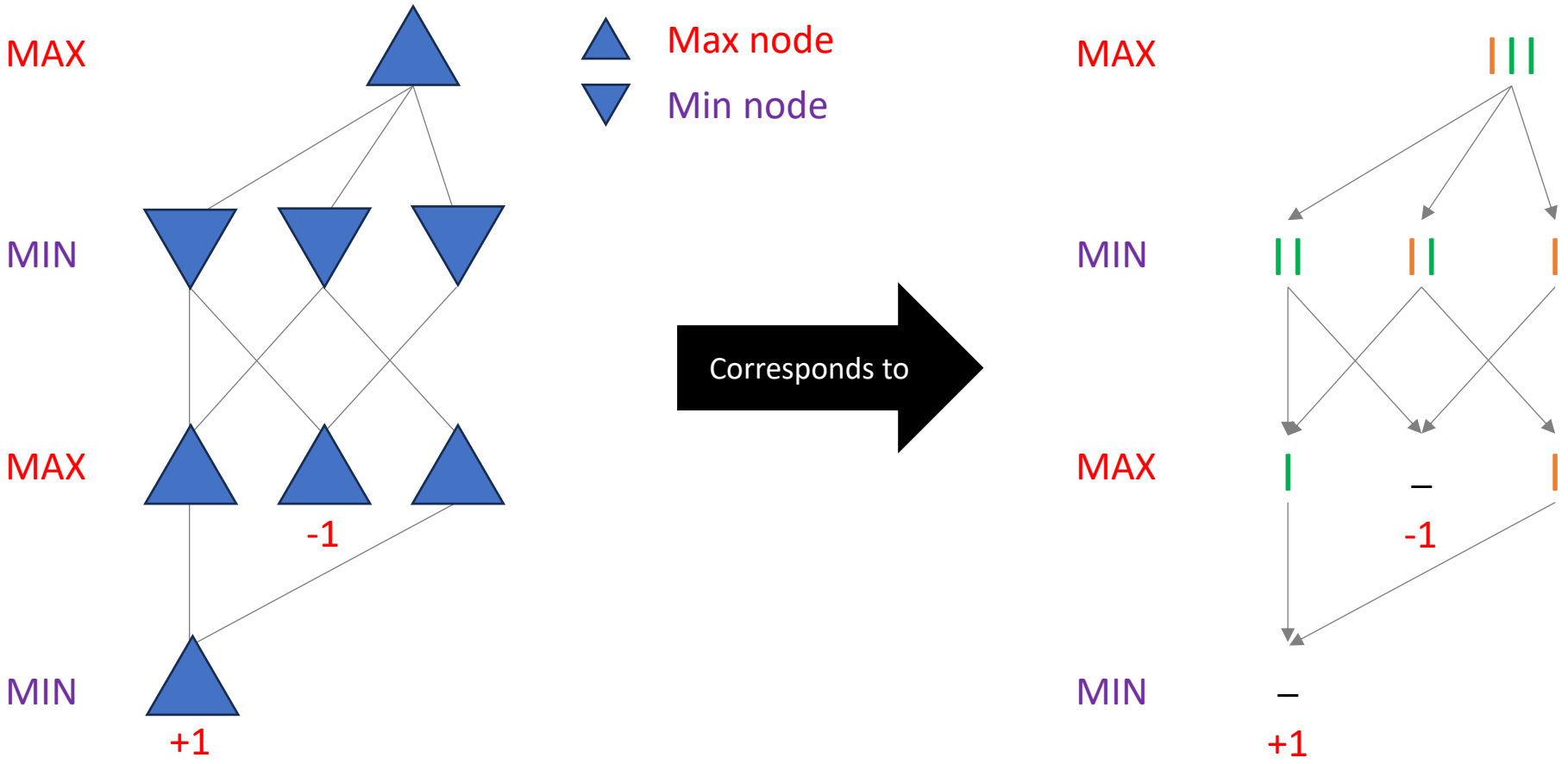
Optimal? Yes, against optimal opponent

Minimax – Optimality

- Algorithm returns an action that **assumes Player B plays optimally** until the end of the game.
- If the **opponent plays sub-optimally**, it can be shown that Player A's action **never has a lower utility** than the utility against an optimal opponent. However, there may be other strategies that perform even better (e.g., end the game faster) against a suboptimal opponent.

Game Tree

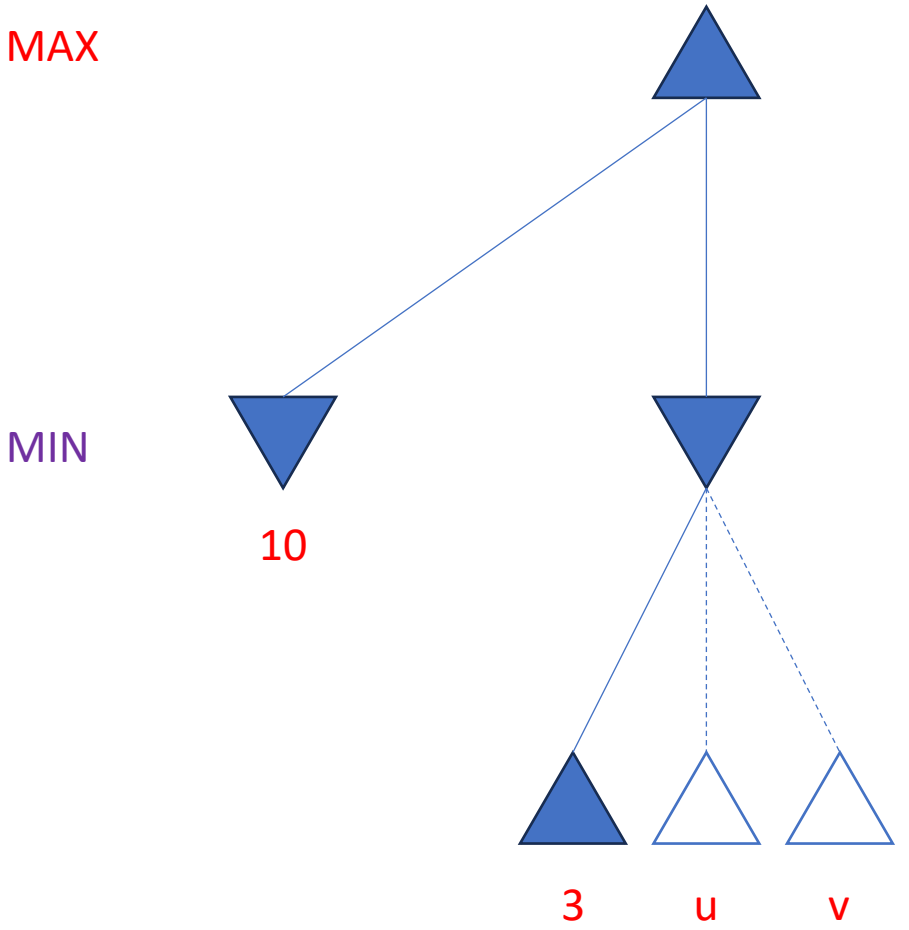
Sometimes, we want to talk about a game tree without defining the game. In this case, we will use an **abstract** game tree.



Pruning the game tree

Minimax explores full game tree

Evaluating a node is sometimes not useful (i.e., it won't change the decision)



Here, the utility is more general i.e., not only {-1,0,1}

Logical view:

MAX(10, MIN(3,u,v)) = 10

for all u,v.

Alpha-beta Pruning

Based on explored moves so far keep track of highest and lowest value the MAX player can obtain.

> α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
>
> β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

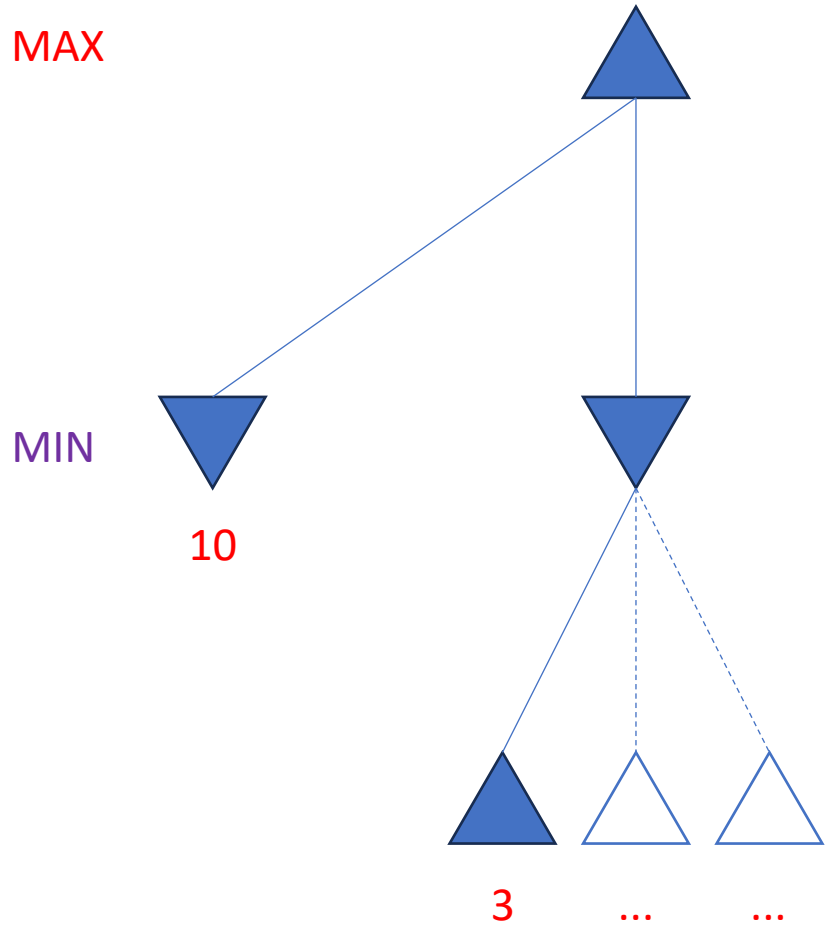
At the start: $\alpha = -\infty$, $\beta = \infty$.

Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

Let's look at the minimum value

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

```
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 next_state in expand(state):
        v = max(v, min_value(next_state, α, β))
        α = max(α, v)
        if v >= β: return v
    return v

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

Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

```
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 next_state in expand(state):
        v = max(v, min_value(next_state, α, β))
        α = max(α, v)
        if v >= β: return v
    return v

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

MAX

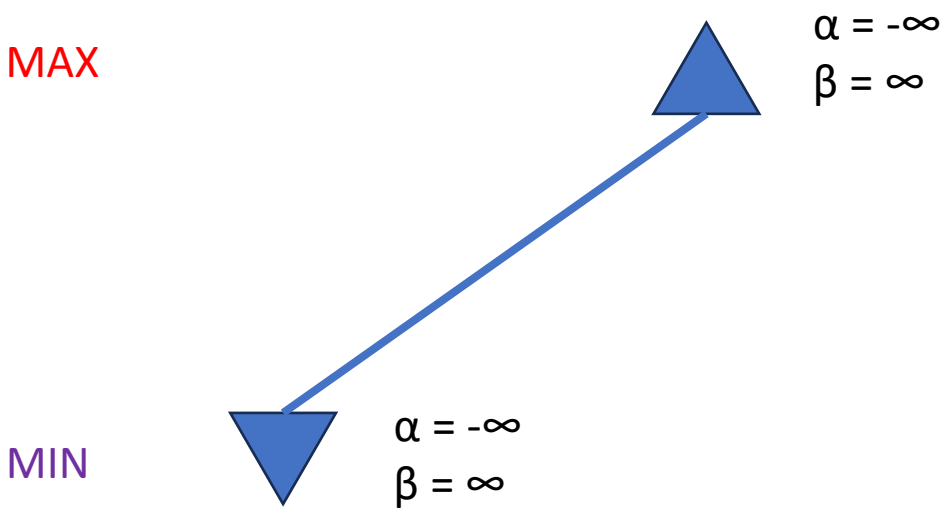
MIN

α = -∞
β = ∞

Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

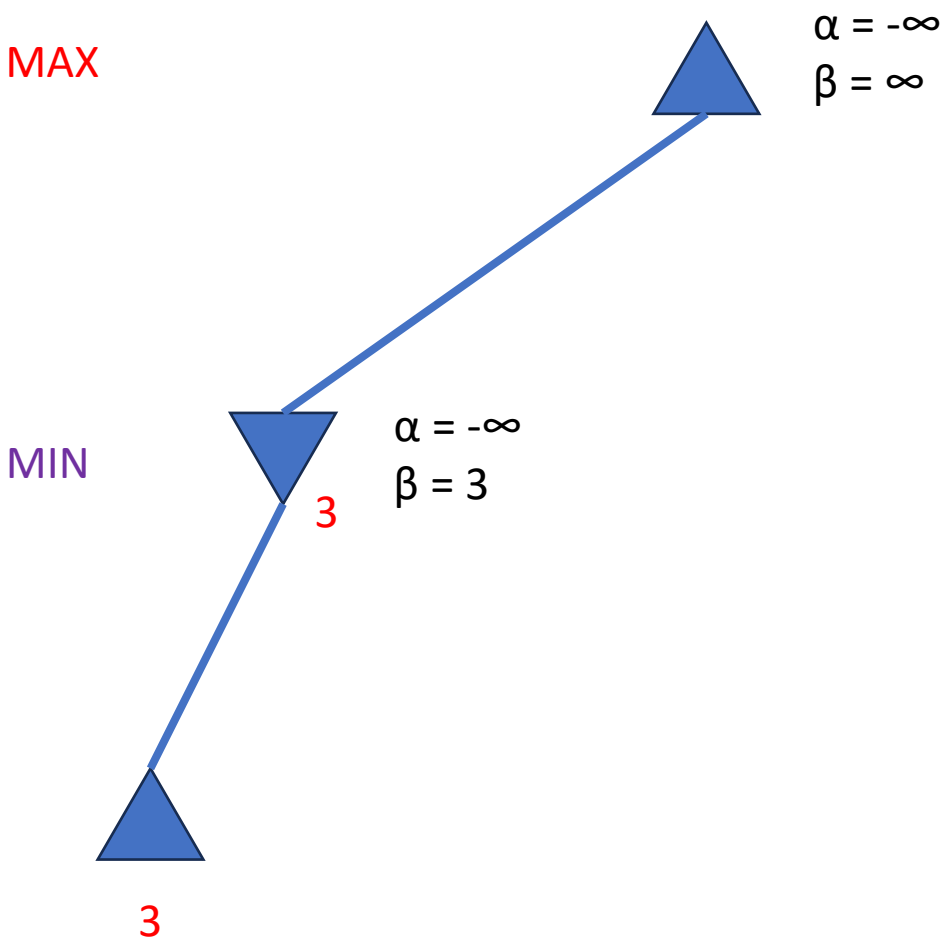
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

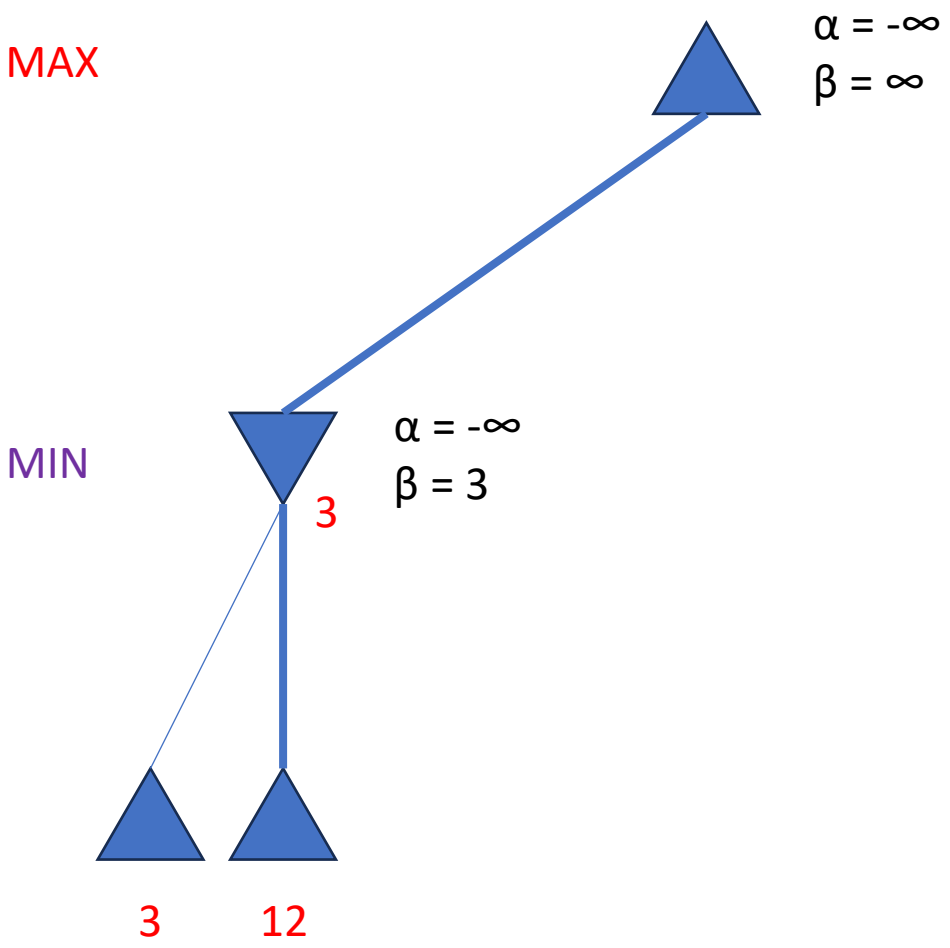
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

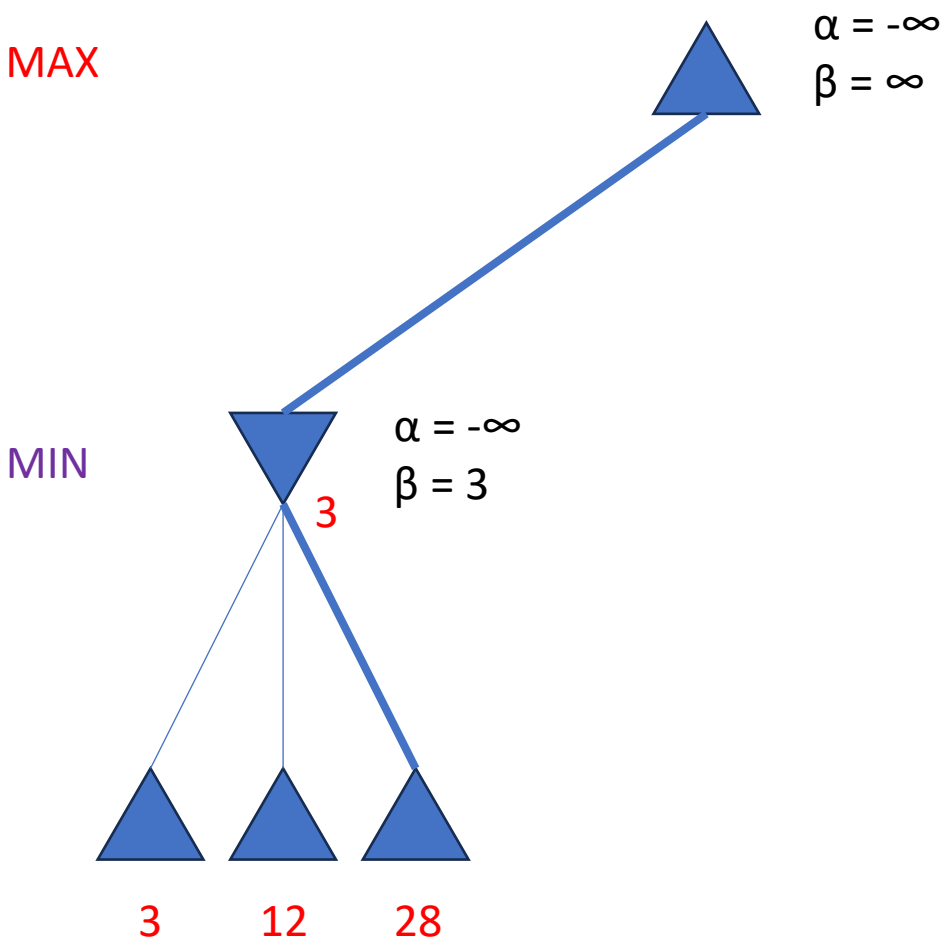
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

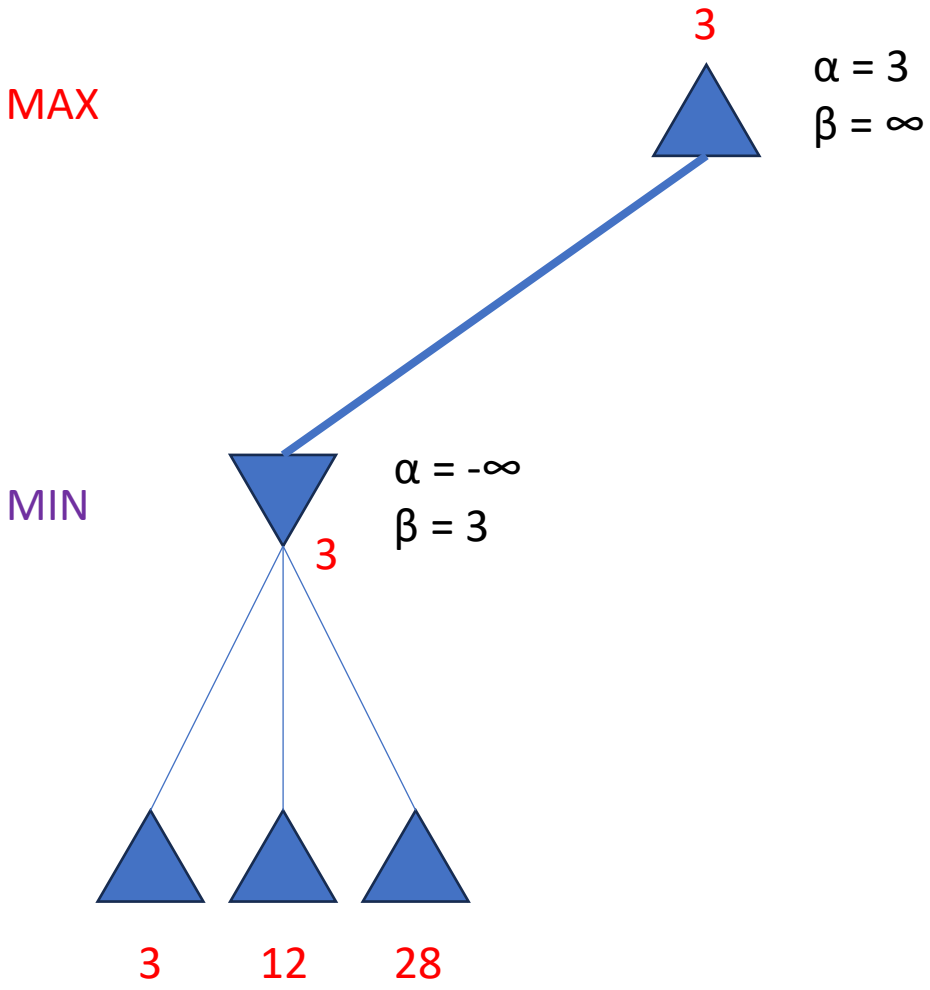
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

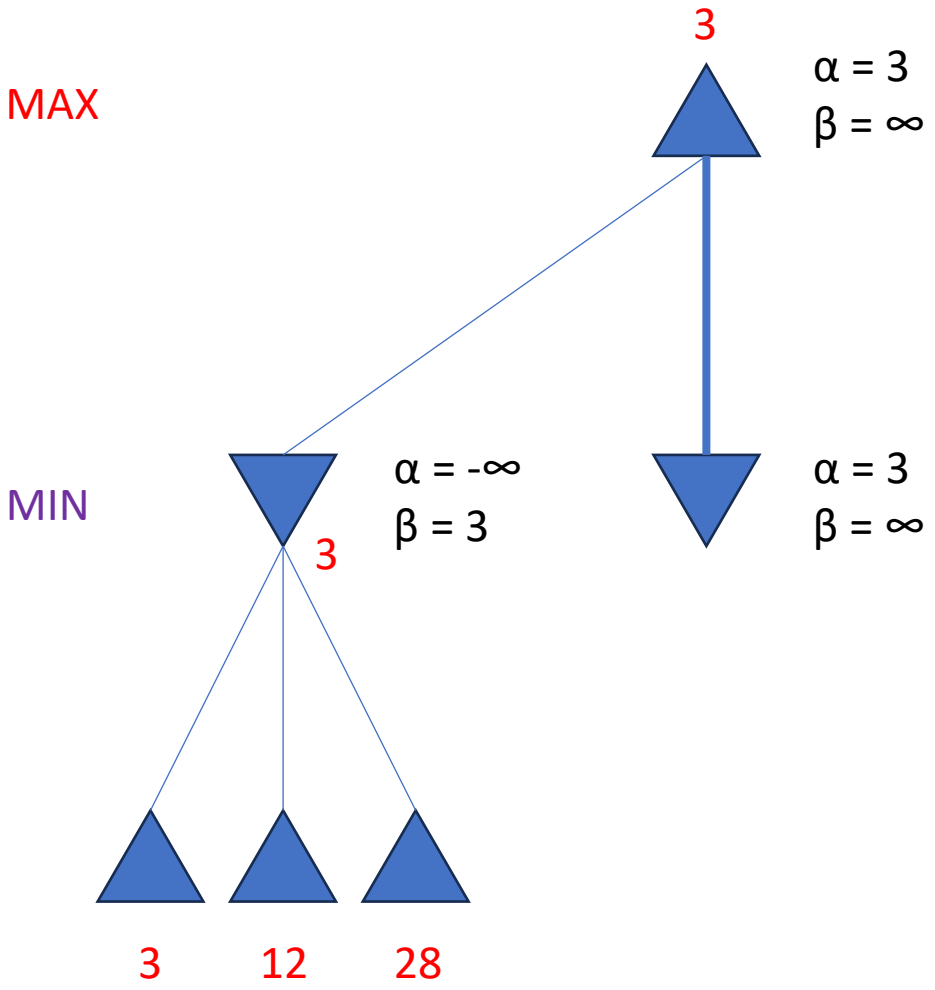
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

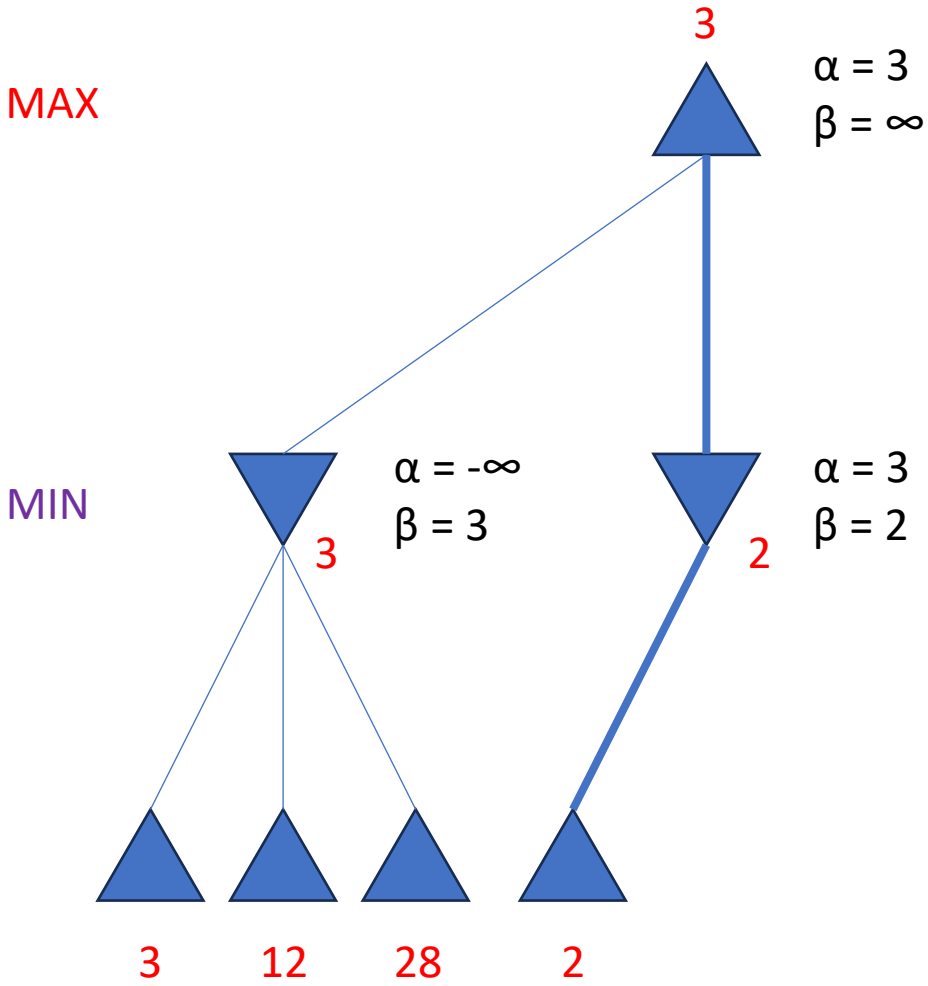
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

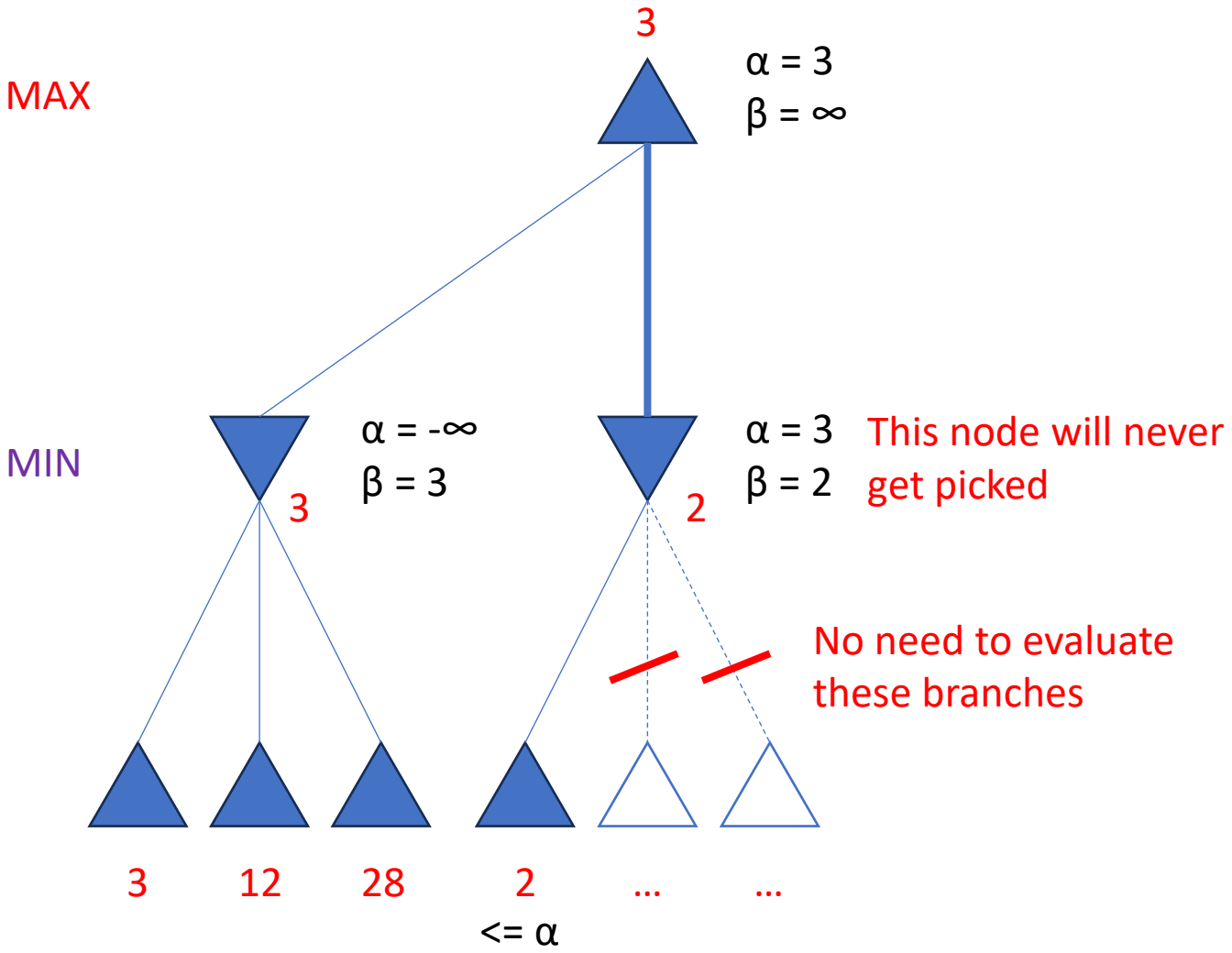
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

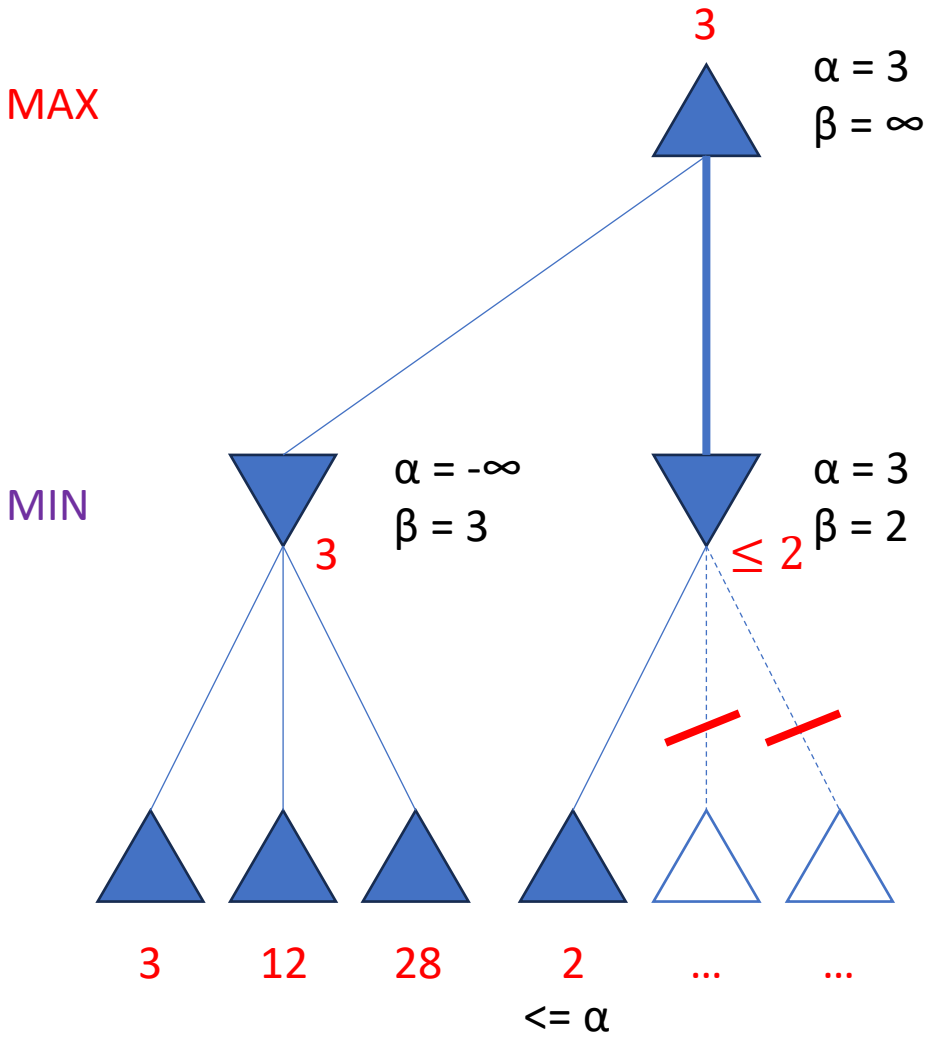
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

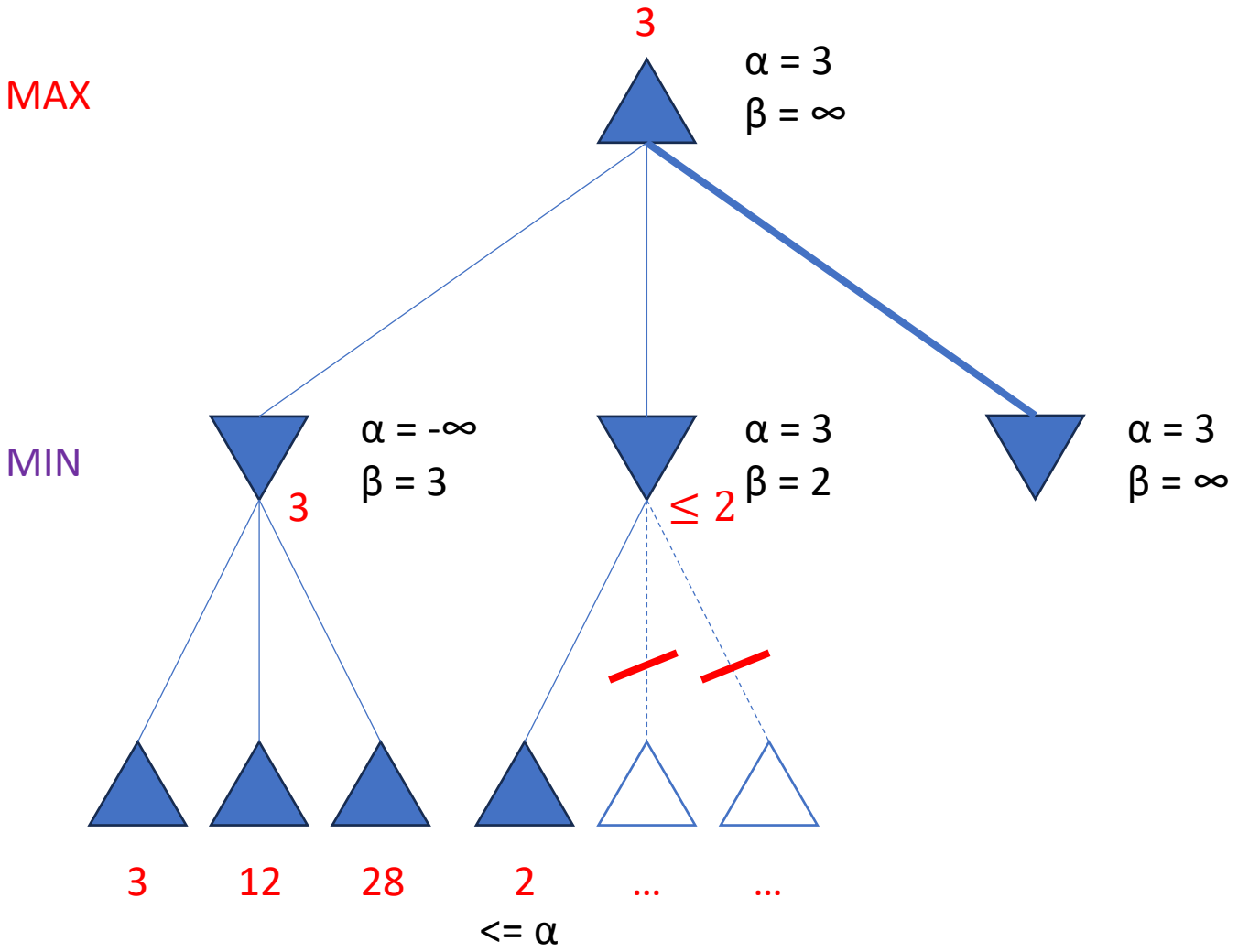
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

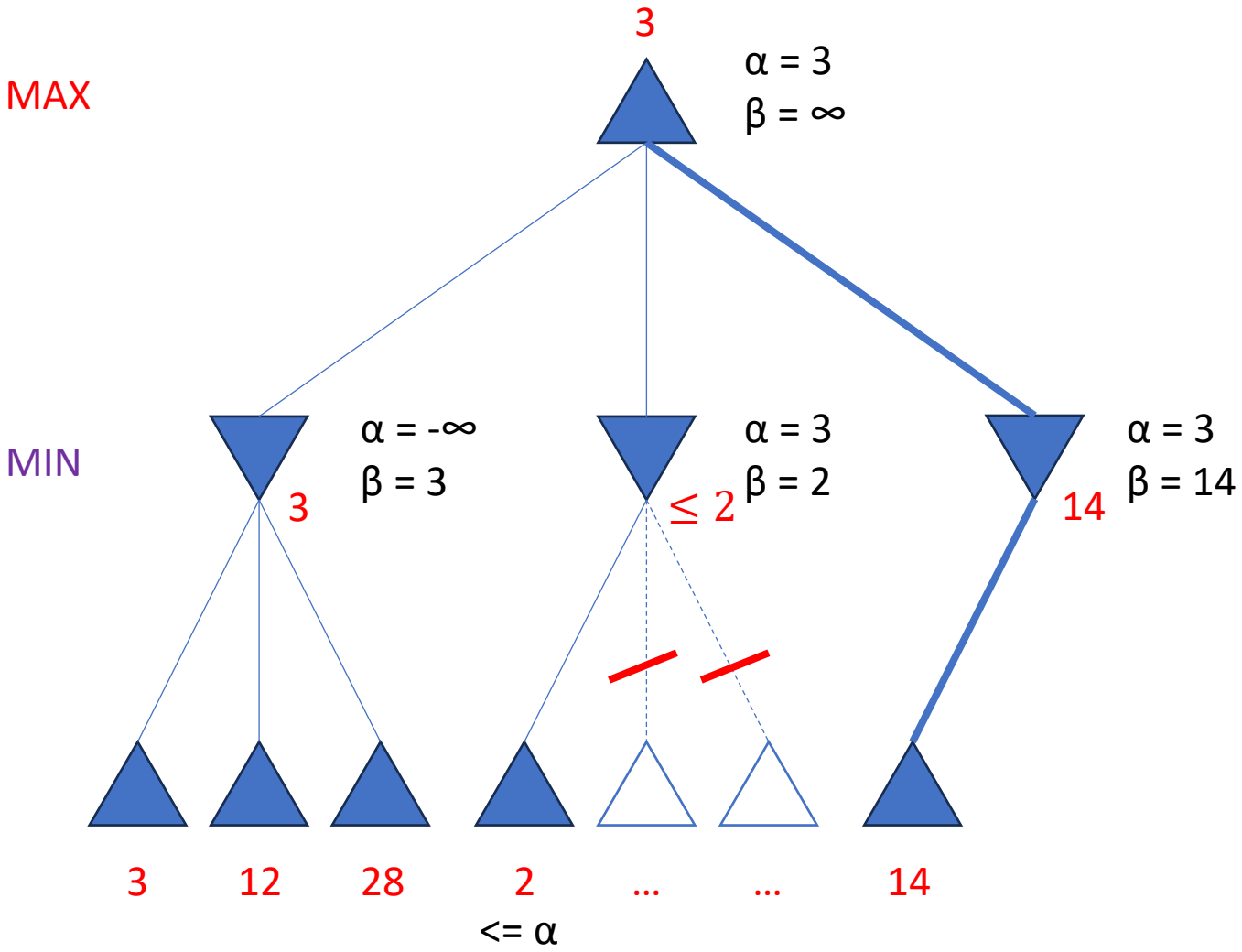
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

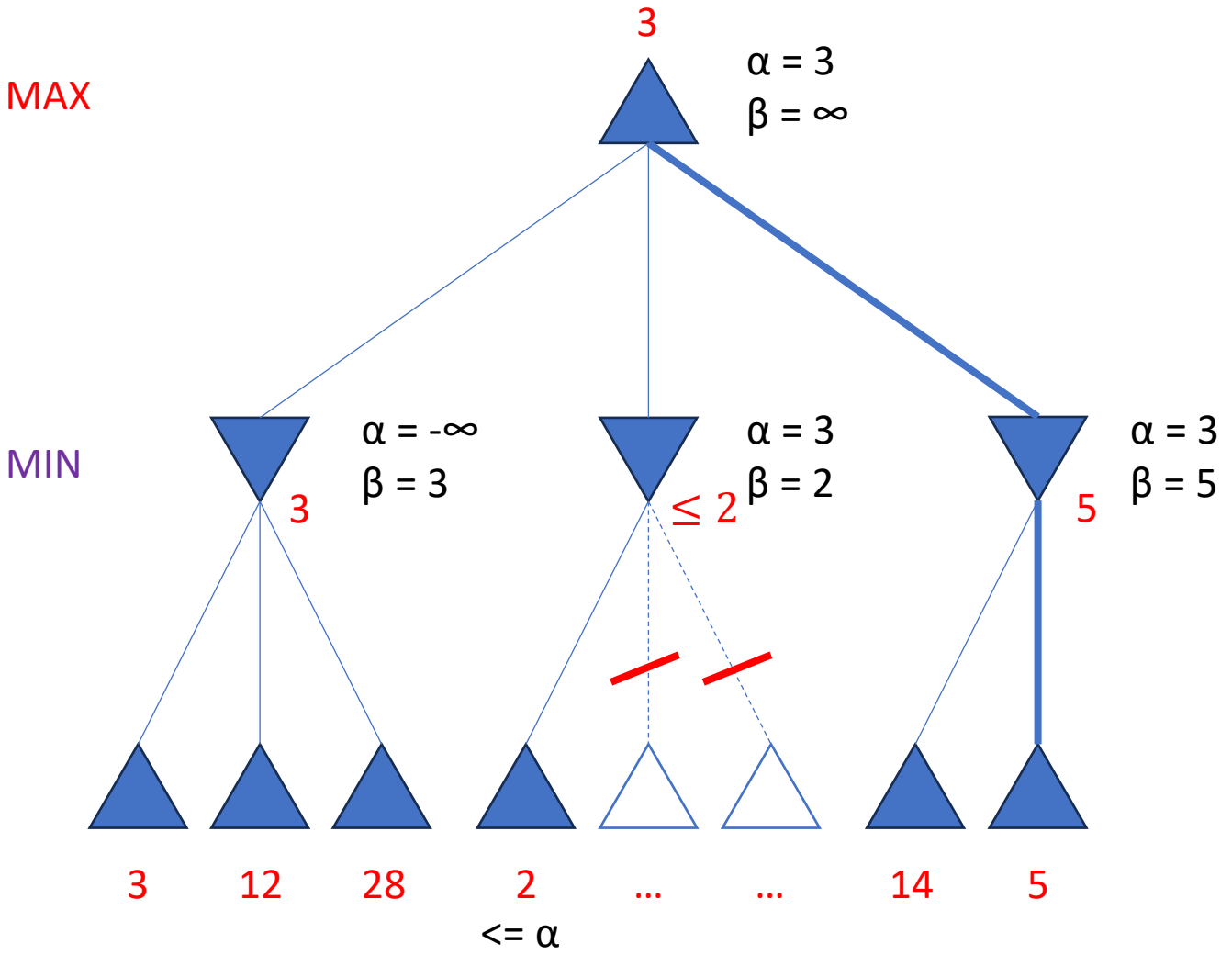
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

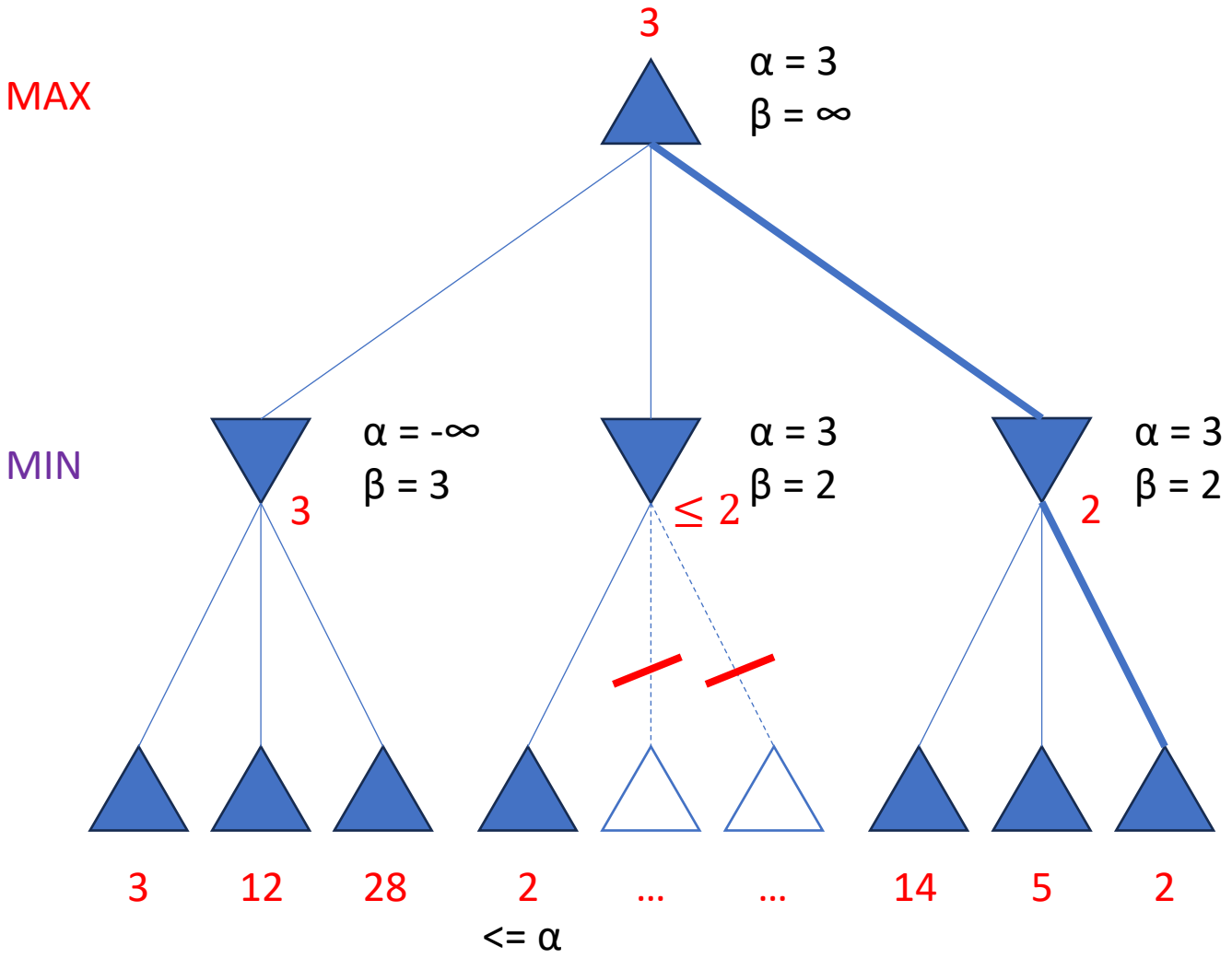
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

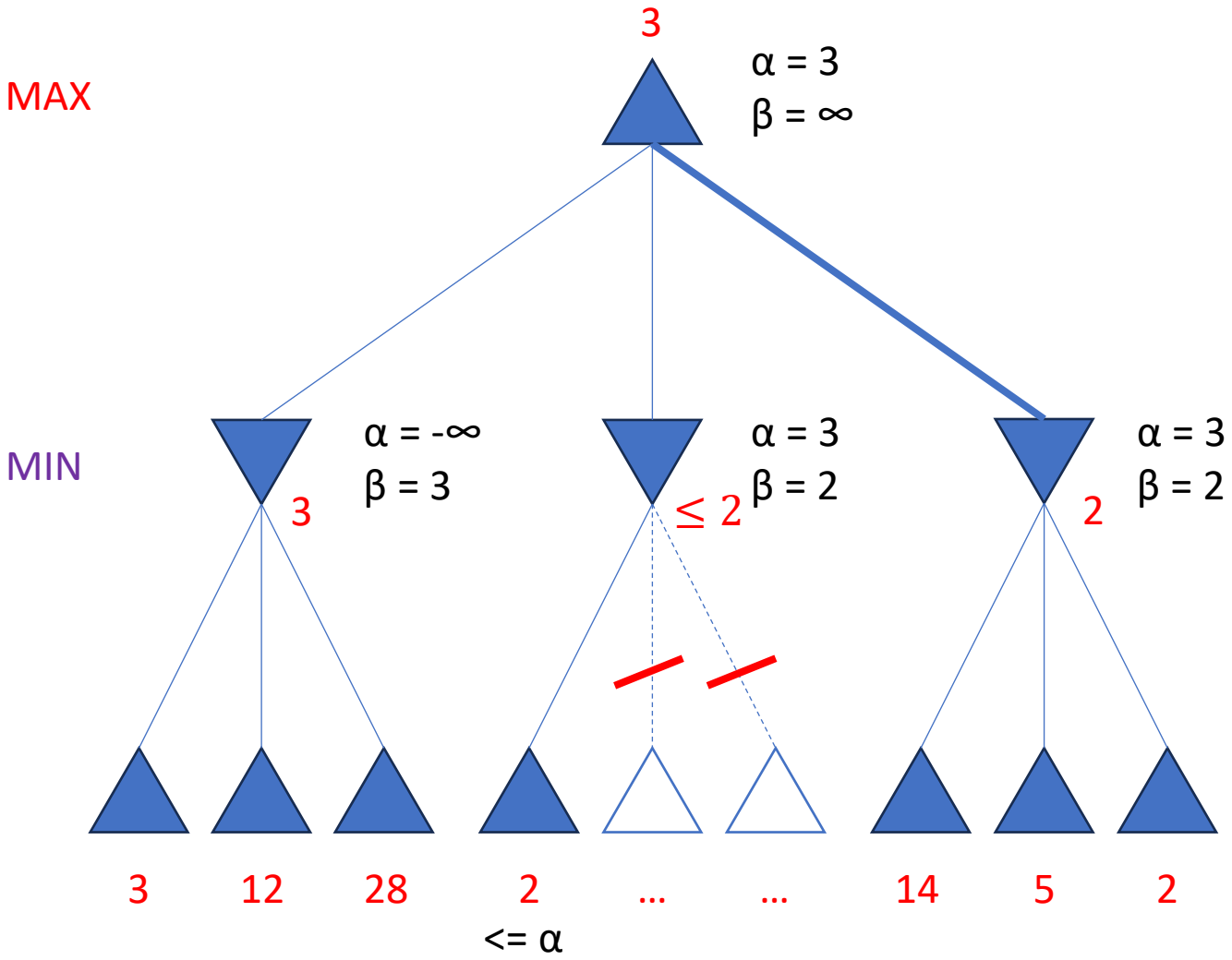
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

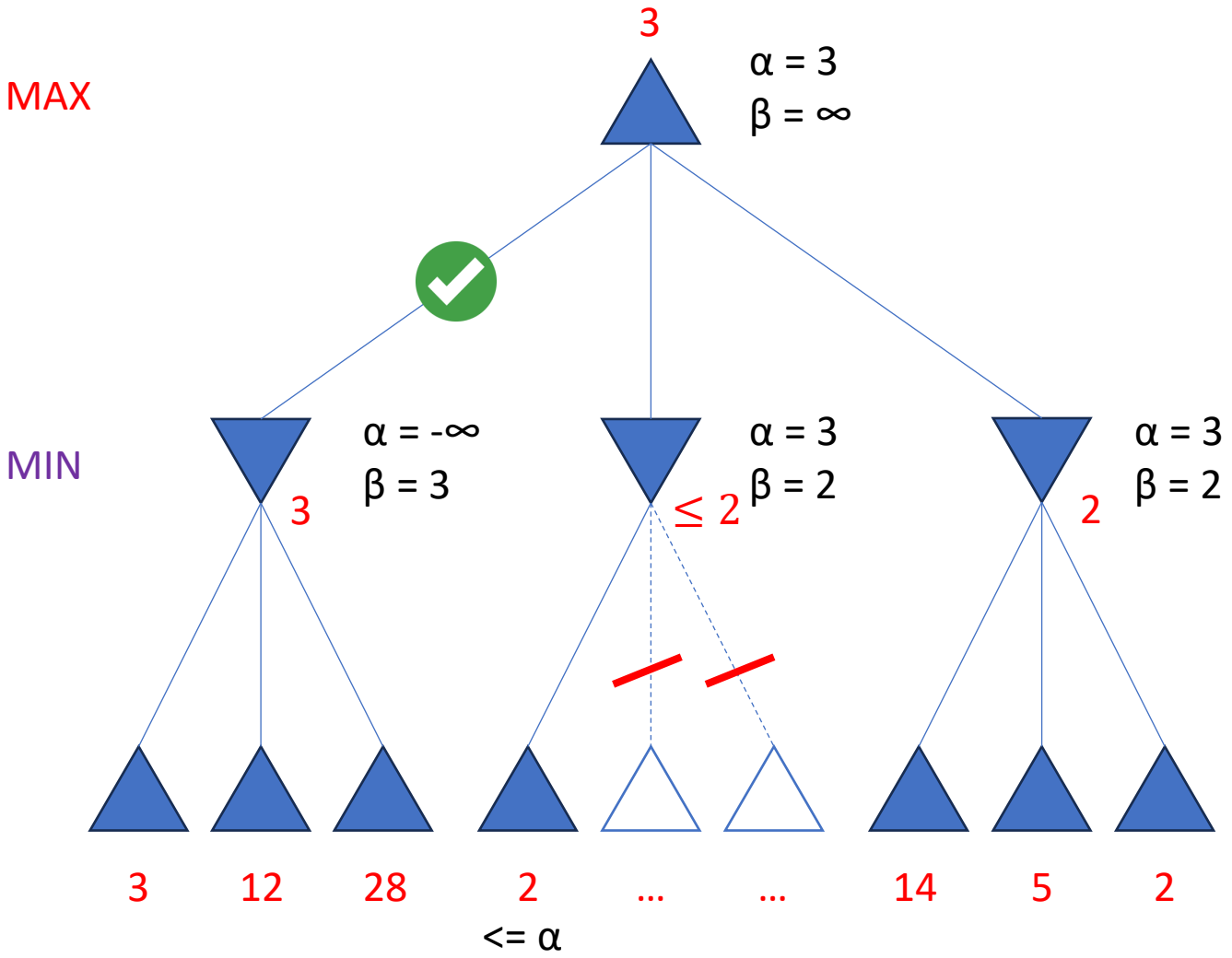
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning

α = highest value for MAX
β = lowest value for MAX

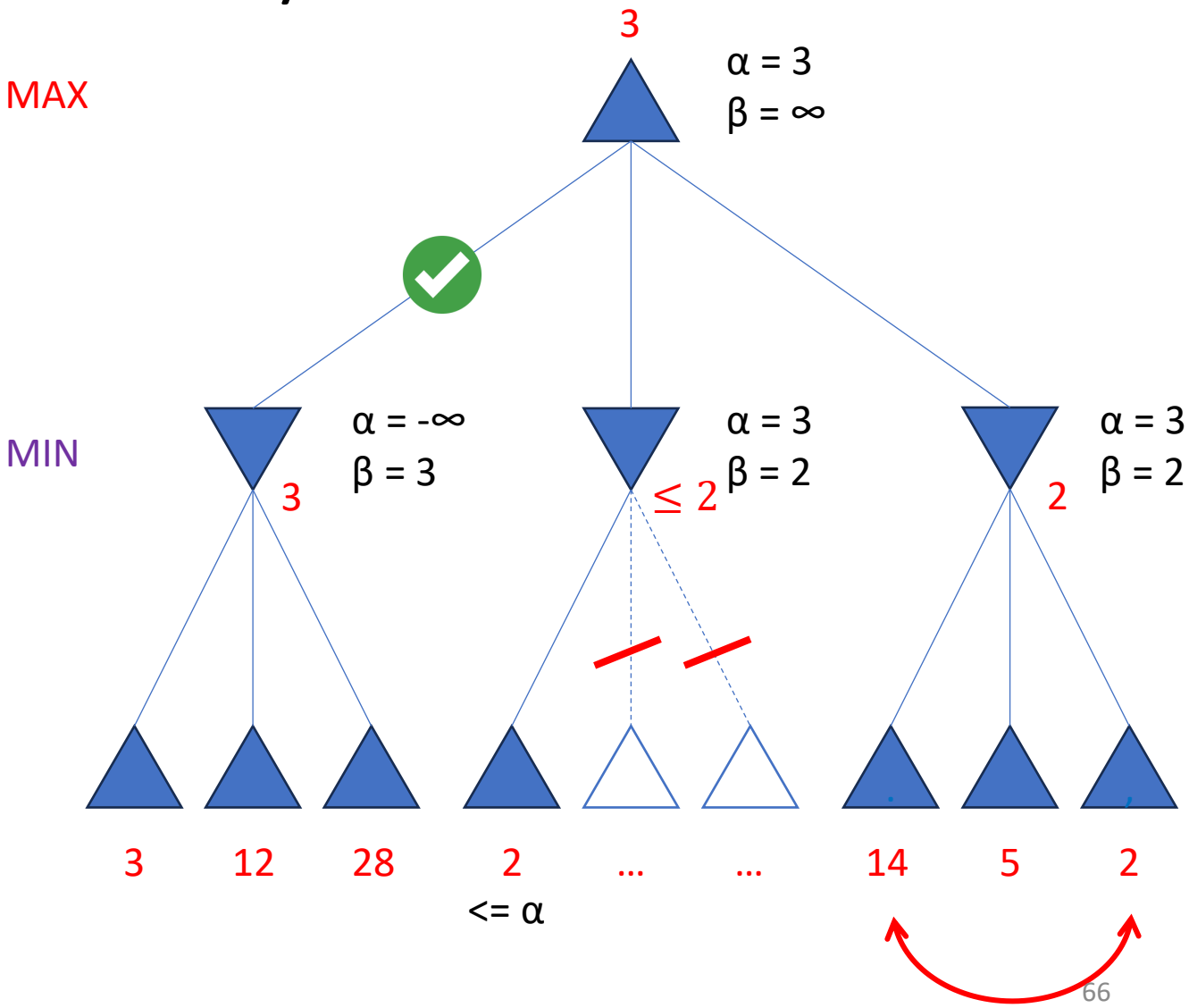
```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

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

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



Alpha-beta Pruning – Analysis

α = highest value for MAX
β = lowest value for MAX

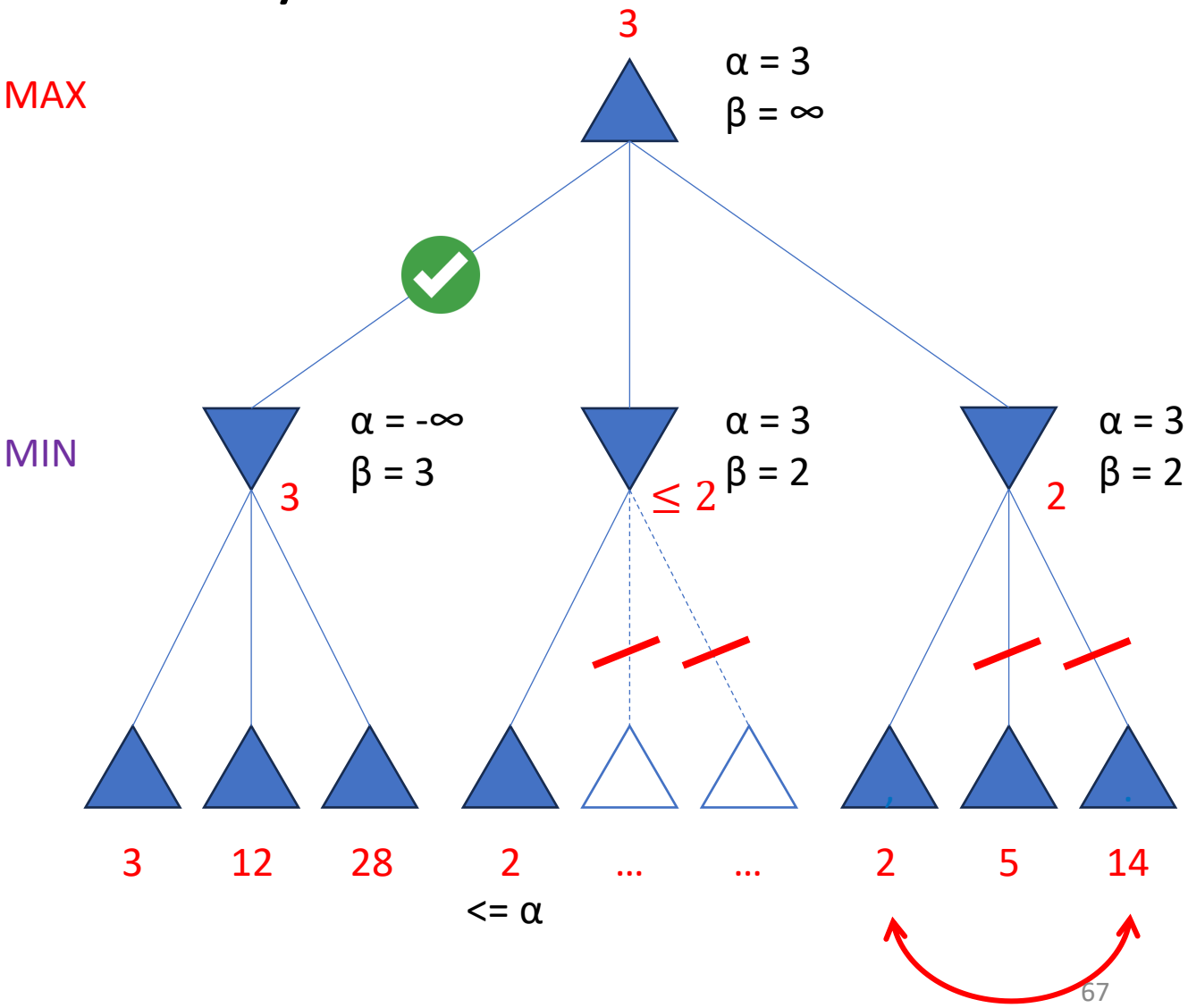
- Pruning doesn't affect the final result
- Good move ordering improves effectiveness of pruning



Alpha-beta Pruning – Analysis

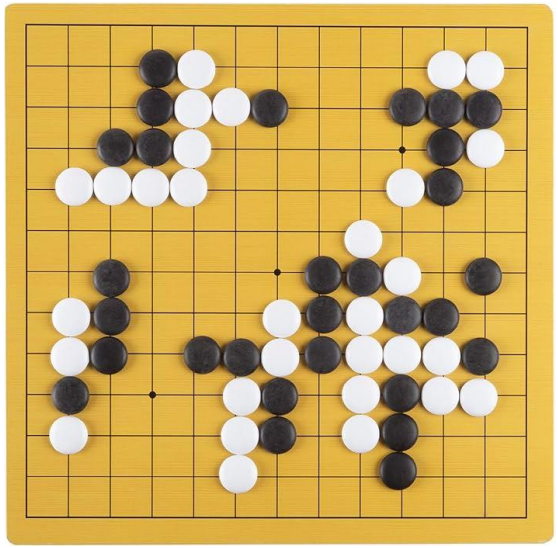
α = highest value for MAX
β = lowest value for MAX

- Pruning doesn't affect the 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



“Real-world” Games

Credit: chess.com



Credit: Amazon.sg

$b \approx 35, m \approx 80$ for “reasonable” games

Time complexity: $b^m = 35^{80} \approx 10^{123}$

$b \approx 250, m \approx 150$ for “reasonable” games

Time complexity: $b^m = 250^{150} \approx 10^{360}$

Number of particles in the universe: $\sim 10^{80}$

Imperfect Decisions by Using Cutoff

In real-world games, calculating the utility of a state by fully exploring the entire game tree with Minimax is often computationally infeasible.

Instead of evaluating the full tree, we can apply a **cutoff** strategy, halting the search in the middle and estimating the value of (mid-game) states using an evaluation function.

The cutoff is typically based on constraints such as the available computational budget (e.g., **time** or **search depth**).

Evaluation Function

- **Evaluation function** is a function that assigns a score to a game state, indicating how favorable it is for a player.
  - If terminal states: use utility(state)
  - Otherwise: use a heuristic
- **Heuristic function** is a function that *estimates* the utility of a state.
  - A heuristic should be somewhere between a loss and a win
    - i.e., $\forall_{state}\ utility(loss) \leq heuristic(state) \leq utility(win)$
  - Computation must not be too long
  - In adversarial search, admissibility and consistency properties do not apply

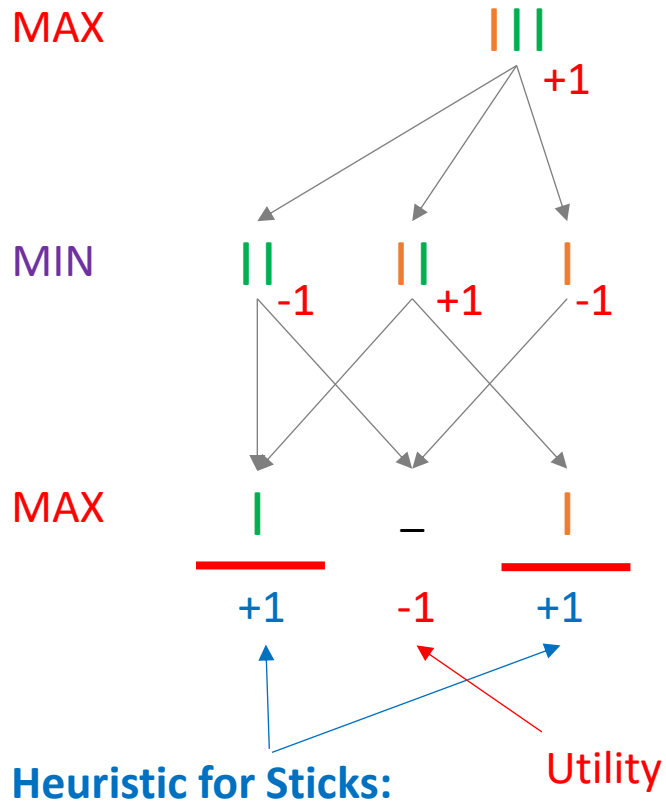
“Inventing” a Heuristic

- Relaxed game – in principle, but usually too costly in practice
  - **Relaxed Chess**: each piece can teleport and perform self-sacrificial attack
    - Heuristic: number of agent’s pieces – number of opponent’s pieces
- Expert knowledge
  - **Chess**: often uses features like material count, mobility, safety, etc.
- Learn from data
  - **Go**: Heuristic function could be learned from labeled (state, utility) dataset. The dataset could be generated using some expert agent or by solving the game (or some portions of the game)
  - AlphaGo uses *reinforcement learning* and *self-play* to learn a utility function

Minimax with Cutoff

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

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

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

Max depth = 2



Heuristic for Sticks:

+1 If the number of sticks ≥ 1

-1 otherwise

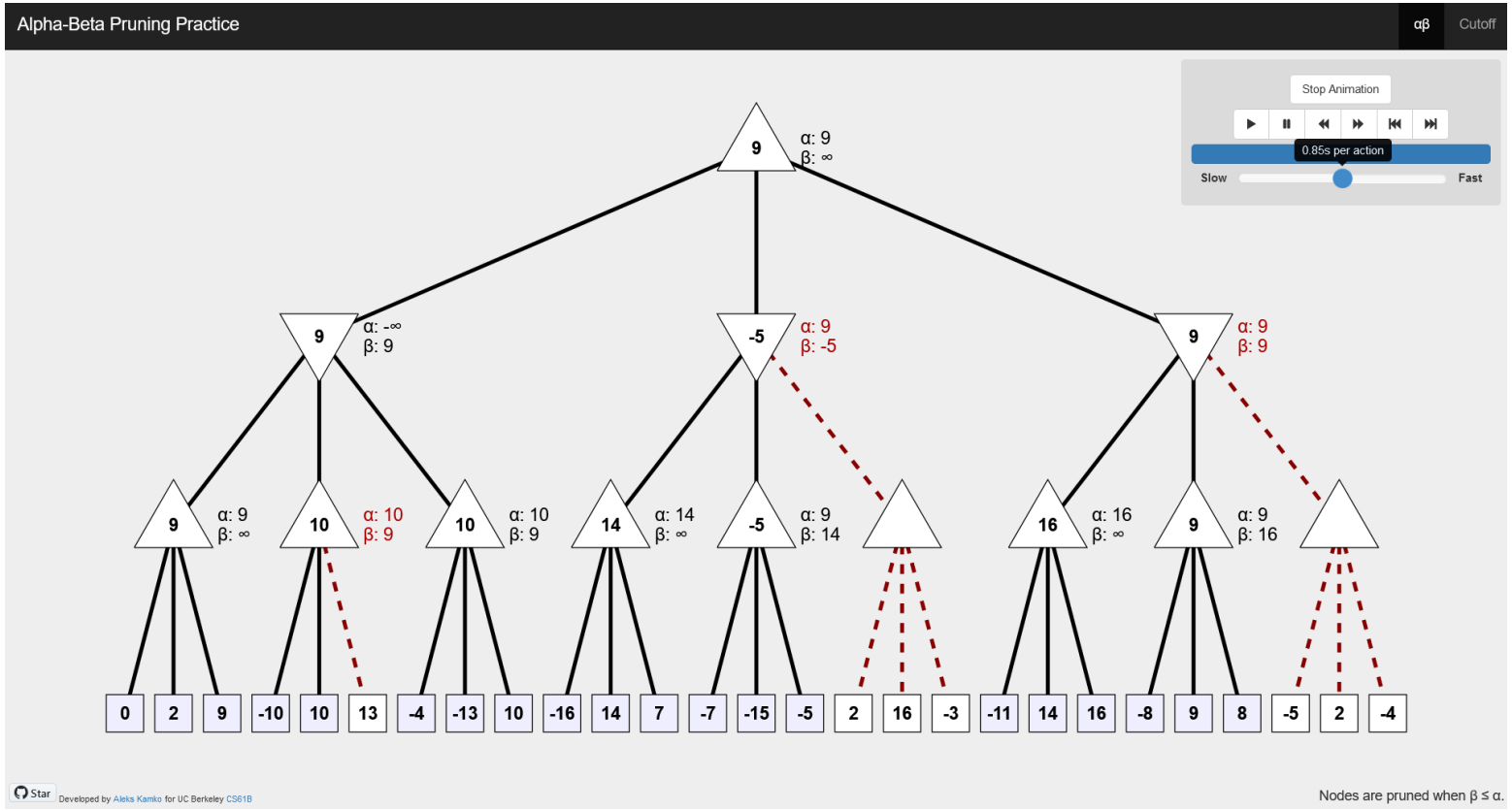
Summary: Adversarial Search

- Games vs Search Problems:
  - Search problems: **predictable** world, deterministic and stochastic
  - Games: **“unpredictable”** opponent, strategic
- Minimax:
  - Assume **opponent behaves optimally**
  - Pick a 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**

Further Reading (Optional)

- Move ordering in alpha-beta pruning (AIAMA 5.2.4)
- Optimizing the search with lookup (AIAMA 5.3.4)
- Monte-carlo tree search (AIAMA 5.4)
- AlphaGo (Mastering the game of Go without human knowledge, Silver et al, 2017)

Resources



https://pascscha.ch/info2/abTreePractice/

Note: can't be accessed from NUS network

Summary

- Local search
  - Hill climbing: pick the best the neighbour, repeat
- Adversarial search
  - Games: opponent strategic
  - Minimax: max then min value
  - Alpha-beta pruning: prune branches that don’t affect decision
  - Handling large game trees: cutoff, evaluation/heuristic function

Coming Up Next Week

- **Intro to Machine Learning**
- **Decision Trees**
  - Entropy and Information Gain
  - Different types of attributes
  - Pruning

To Do

- **Lecture Training 3**
  - +250 EXP
  - +100 Early bird bonus (extended to February 3rd)
- **Tutorial starts this week!**
  - 500 Free “Hong Bao” EXP for everyone
- **Problem Set 1**
  - Deadline: extended to Monday, 3 February, 23:59