Tampere University

Adversarial search

CHAPTER 5 IN THE TEXTBOOK

Tampere University

Types of Games

- Many different kinds of games!

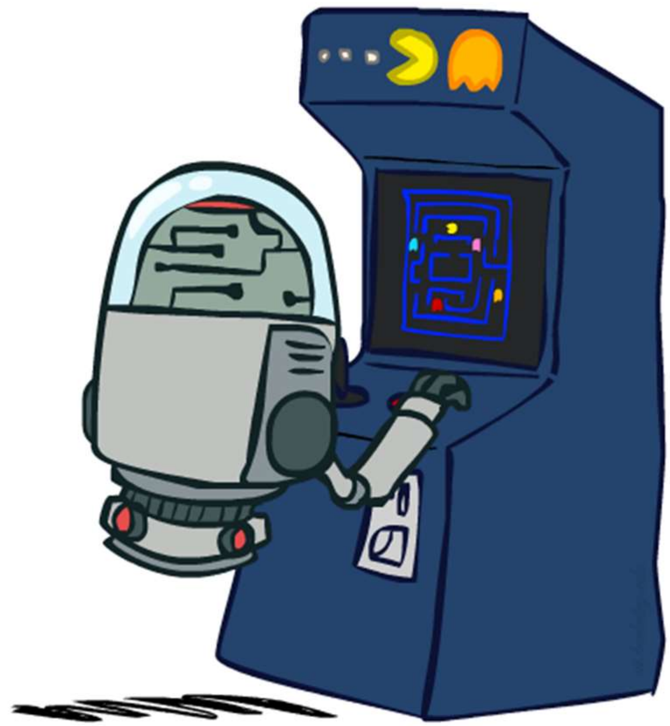
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

- Want algorithms for calculating a strategy (policy) which recommends a move from each state

Tampere University

Deterministic Games

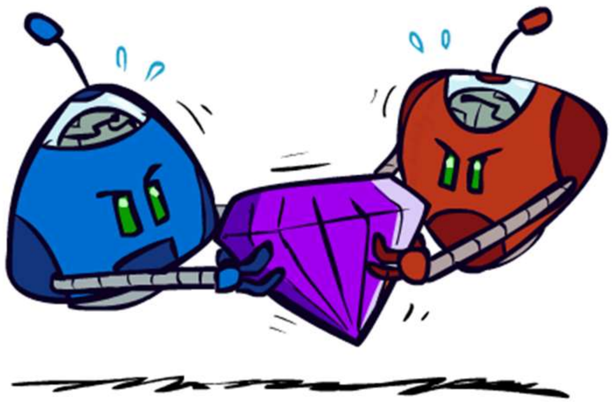
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: $P = \{1 \dots N\}$ (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: $S \times A \rightarrow S$
  - Terminal Test: $S \rightarrow \{T, F\}$
  - Terminal Utilities: $S \times P \rightarrow \mathbb{R}$

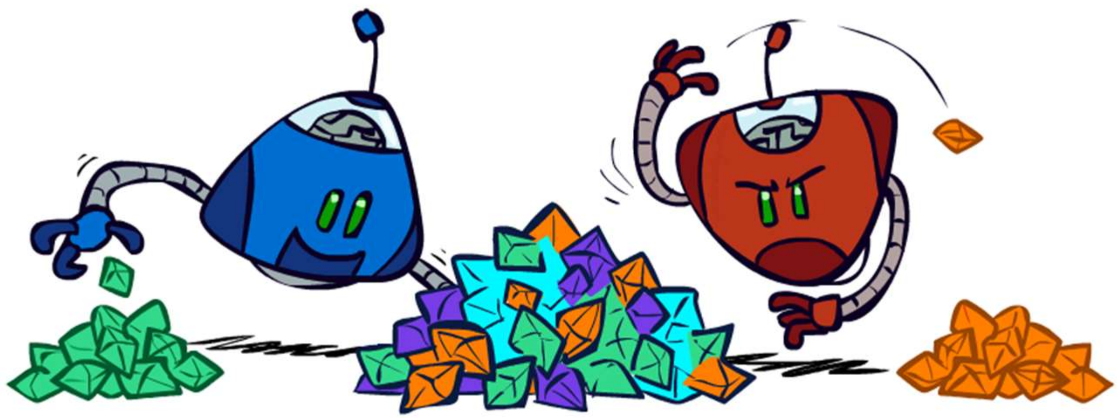


- Solution for a player is a policy: $S \rightarrow A$

Tampere University

Zero-Sum Games



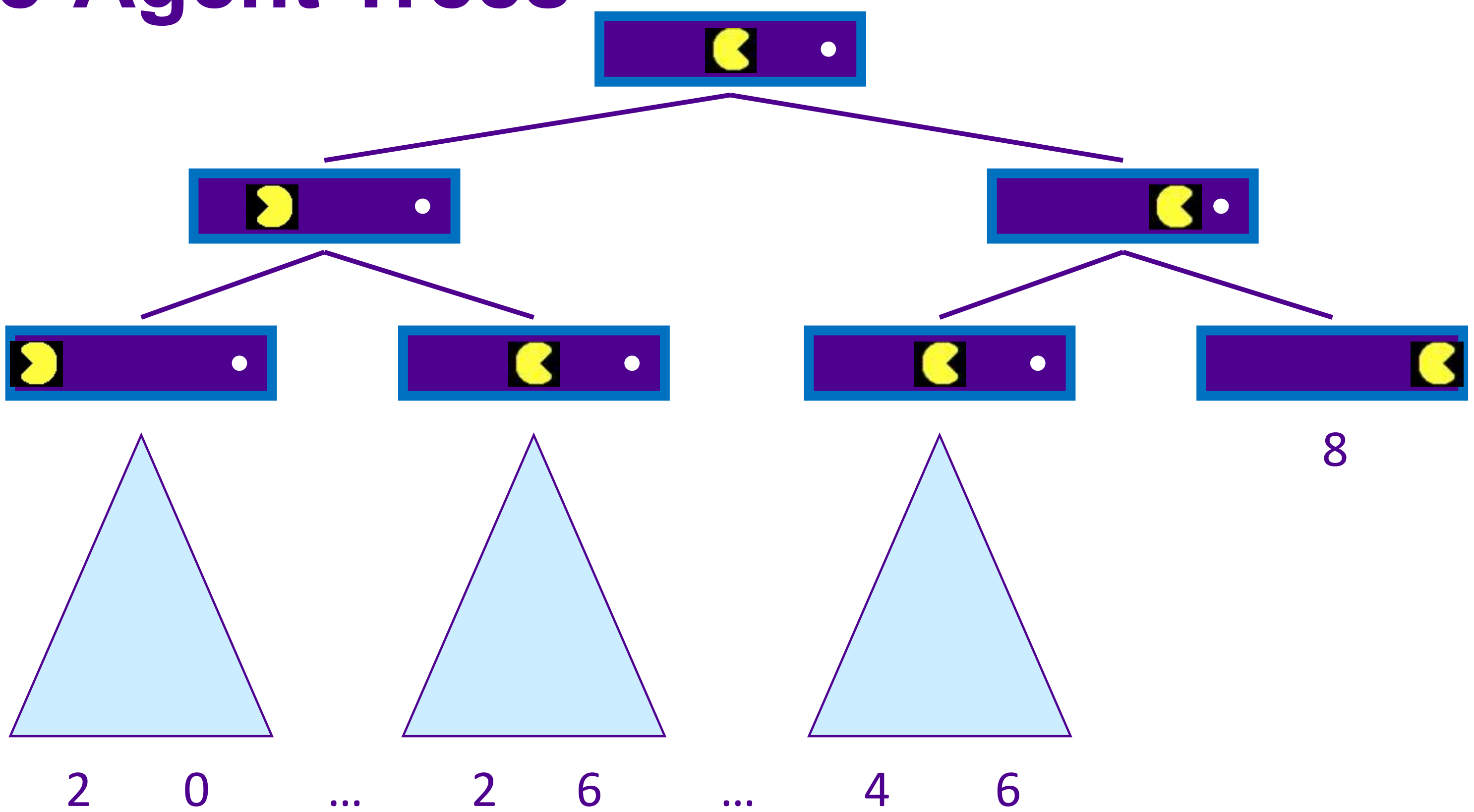


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

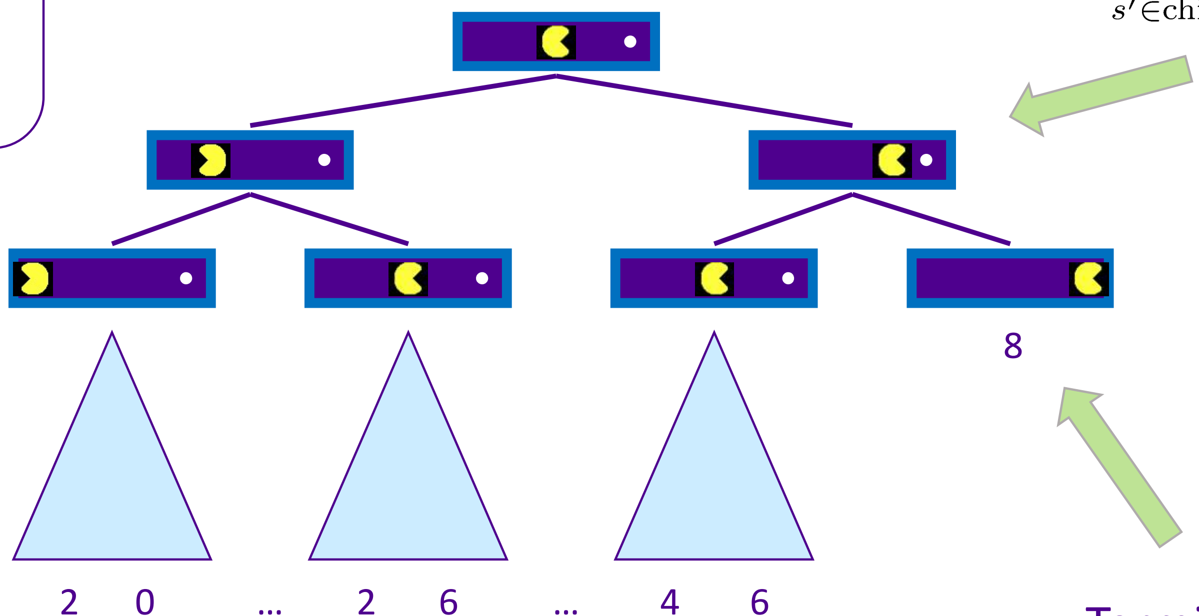
Tampere University

Single-Agent Trees



Value of a State

Value of a state:
The best achievable outcome (utility) from that state



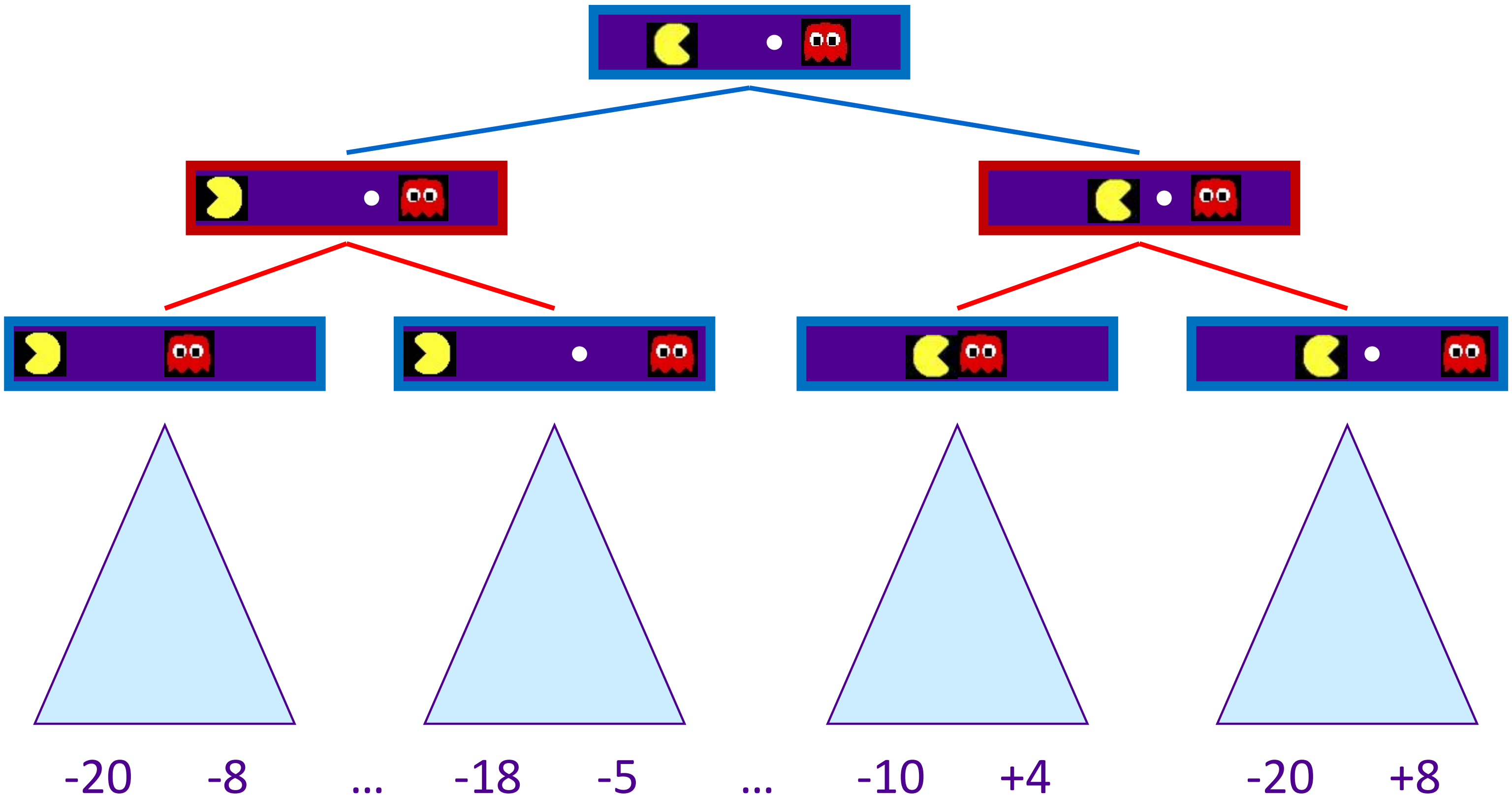
Non-Terminal States:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

$$V(s) = \text{known}$$

Adversarial Game Trees



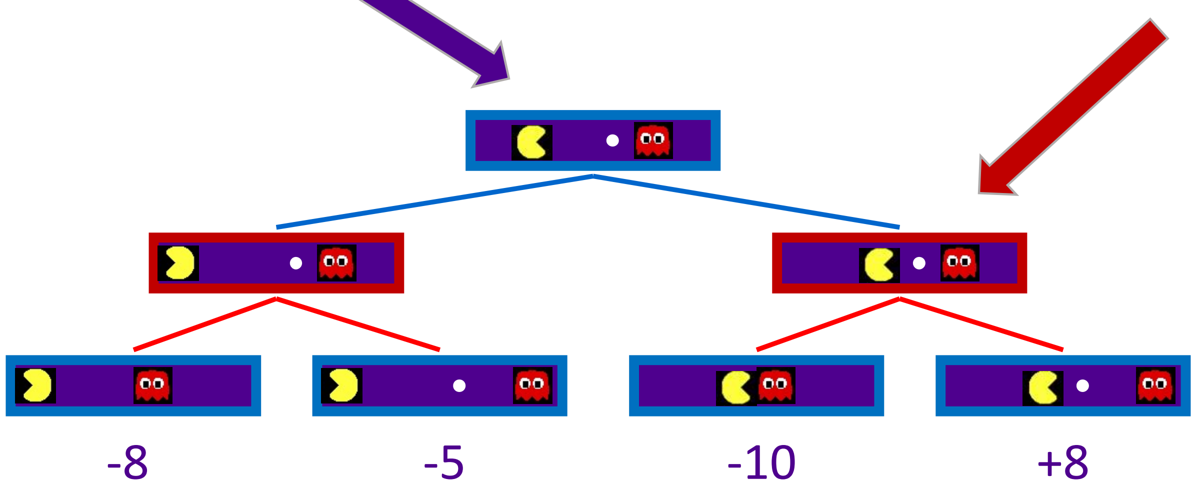
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

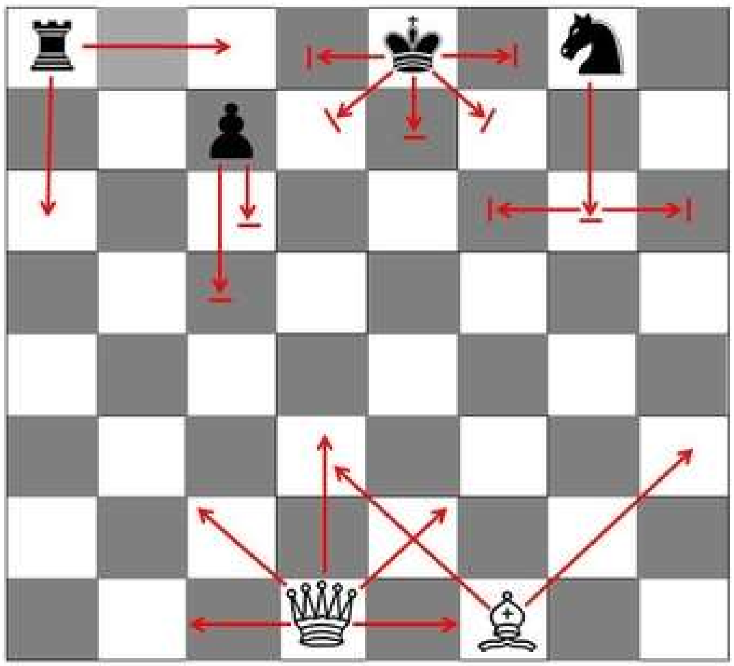
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

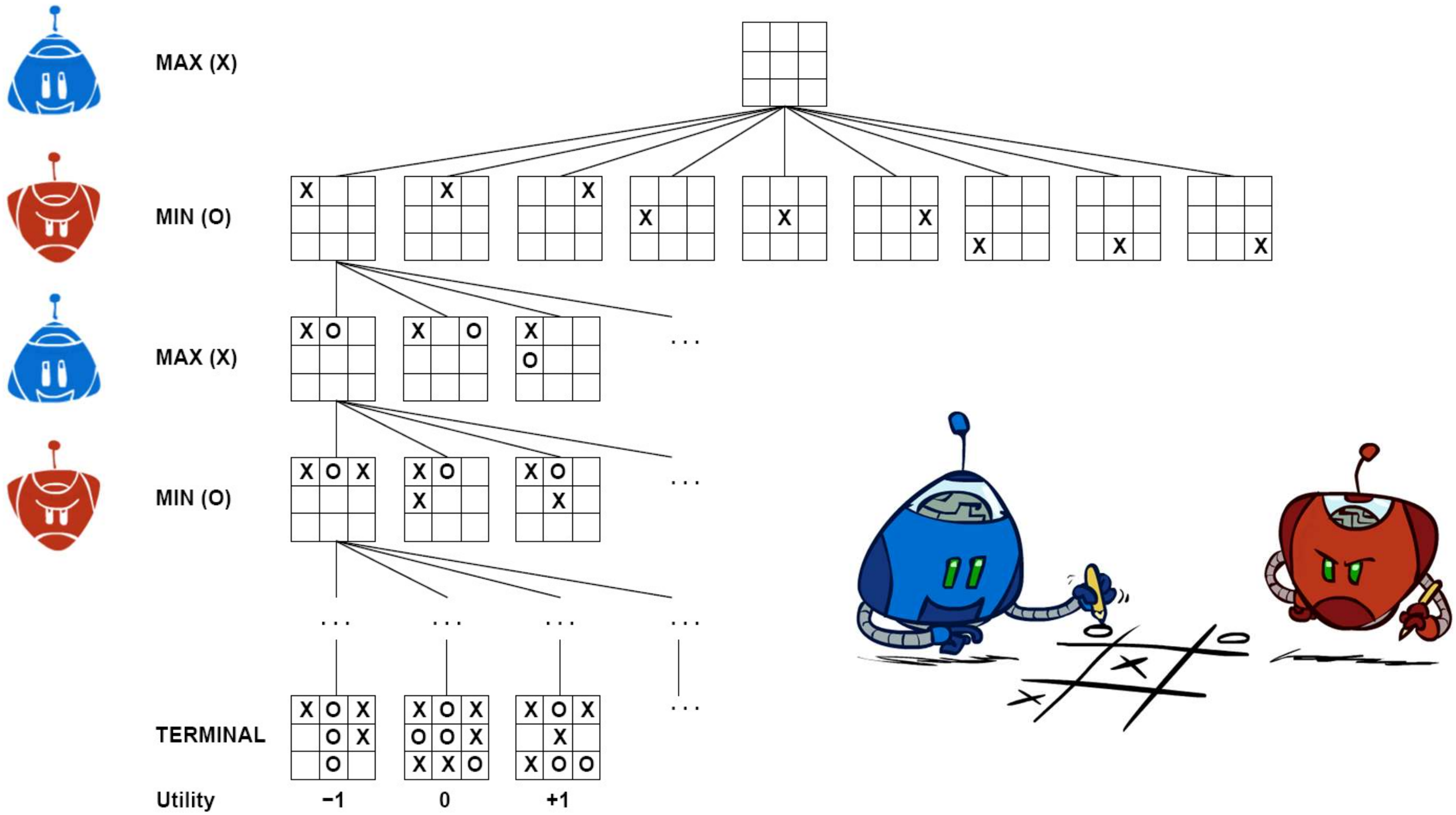
$$V(s) = \text{known}$$

Tampere University



- In principle Chess is easy to solve:
  - A finite number of moves for both players to choose from
  - Build up a tree where the players take turns to choose piece movements on the board configuration that has evolved
  - No matter what search strategy we use, in practice all board configurations cannot be evaluated

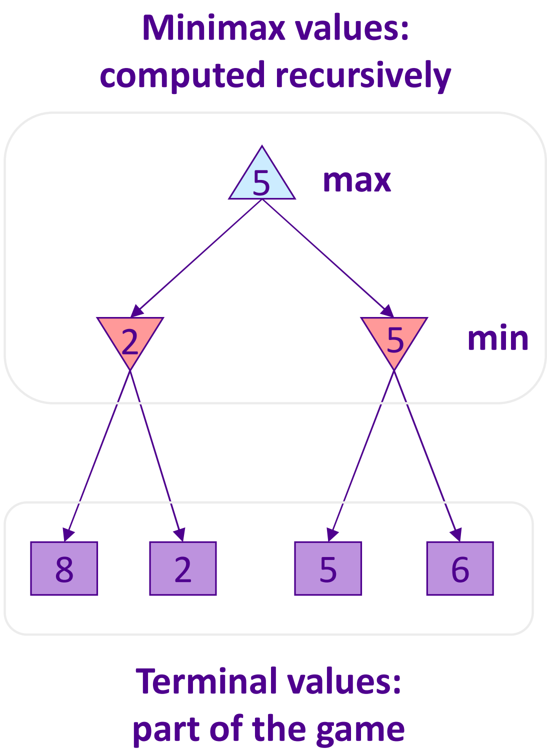
Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

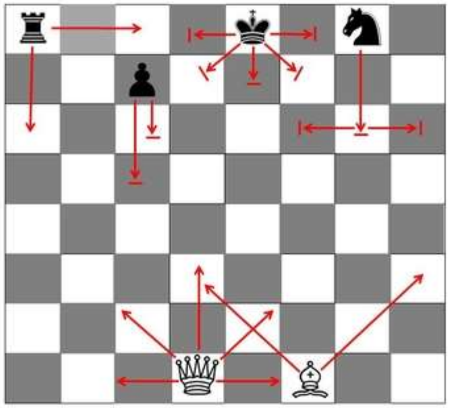
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Tampere University

Optimal Decisions in Games

- The two players: min and max
- The initial board position is like the rules of the game dictate and max is the first to move
- Successor function $S(n)$ determines legal moves and resulting states
- A terminal test determines when the game is over
- max (min) aims at maximizing (minimizing) the value of the utility function
- The initial state and the successor function determine a *game tree*, where the players take turns to choose an edge to travel



- In our quest for the optimal game strategy, we will assume that also the adversary is infallible
- Player $\min$ chooses the moves that are best for it
- To determine the optimal strategy, we compute for each node n its *minimax* value $MM(n)$:

$$MM(n) = \begin{cases} \textsc{PayOff}(n), & \text{if } n \text{ is a terminal state} \\ \max_{s \in S(n)} MM(s), & \text{if } n \text{ is a max node} \\ \min_{s \in S(n)} MM(s), & \text{if } n \text{ is a min node} \end{cases}$$

Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

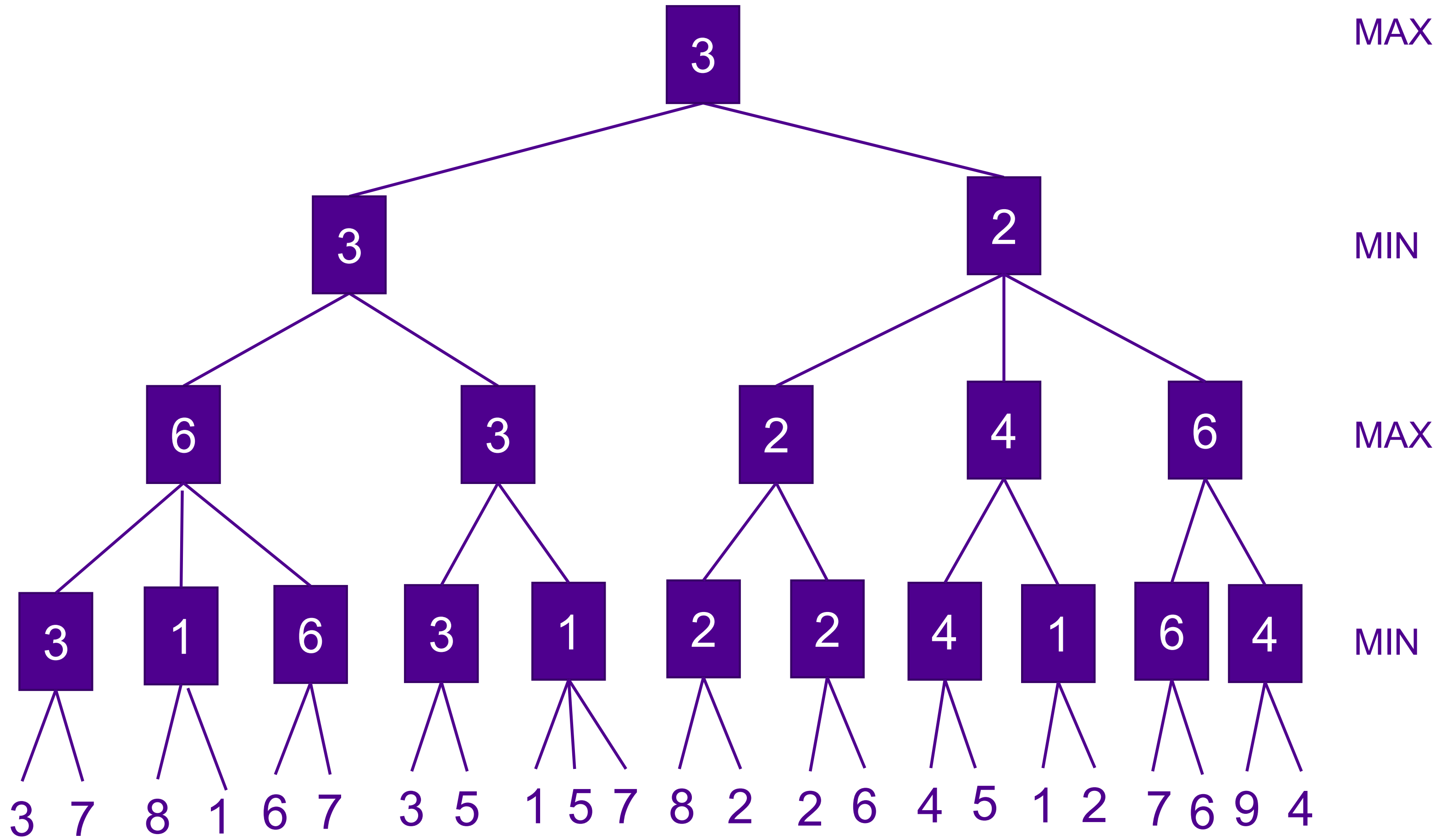
Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Tampere University



- The play between two optimally playing players is completely determined by the minimax values
- For $\max$ the minimax values gives the worst-case outcome — the opponent $\min$ is optimal
- If the opponent does not choose the best moves, then $\max$ will do at least as well as against $\min$
- Against suboptimal opponents there may be other strategies that do better than minimax
- The minimax algorithm performs a complete depth-first exploration of the game tree
- Therefore, the time complexity is $O(b^m)$, where b is the number of legal moves at each point and m is the maximum depth
- For real games, exponential time cost is totally impractical

Tampere University

Video of Demo Min vs. Rnd (Min)

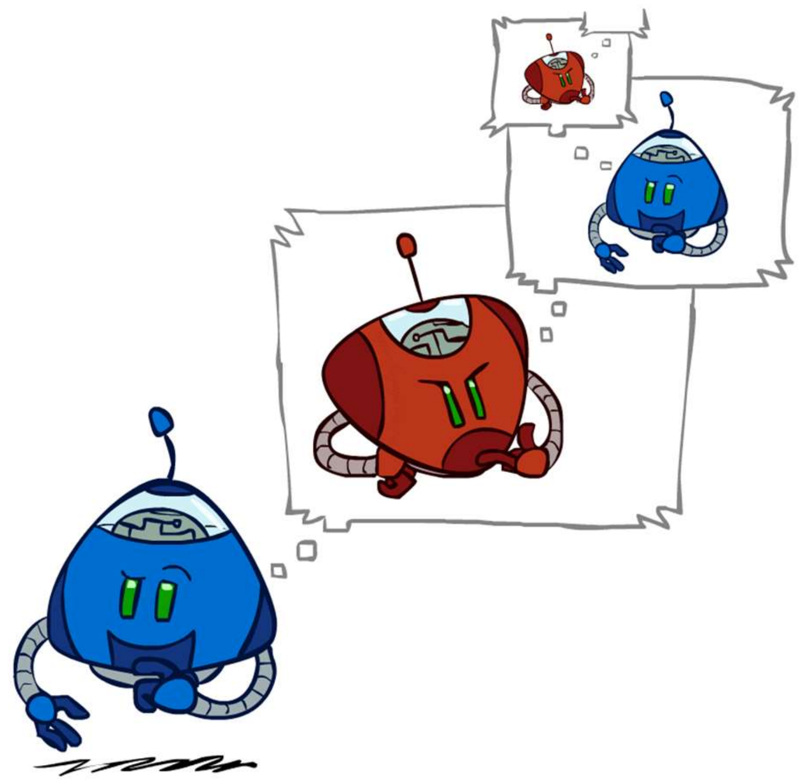
Video of Demo Min vs. Rnd (Rnd)

Tampere University

Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
- Optimal against a perfect player. Otherwise?



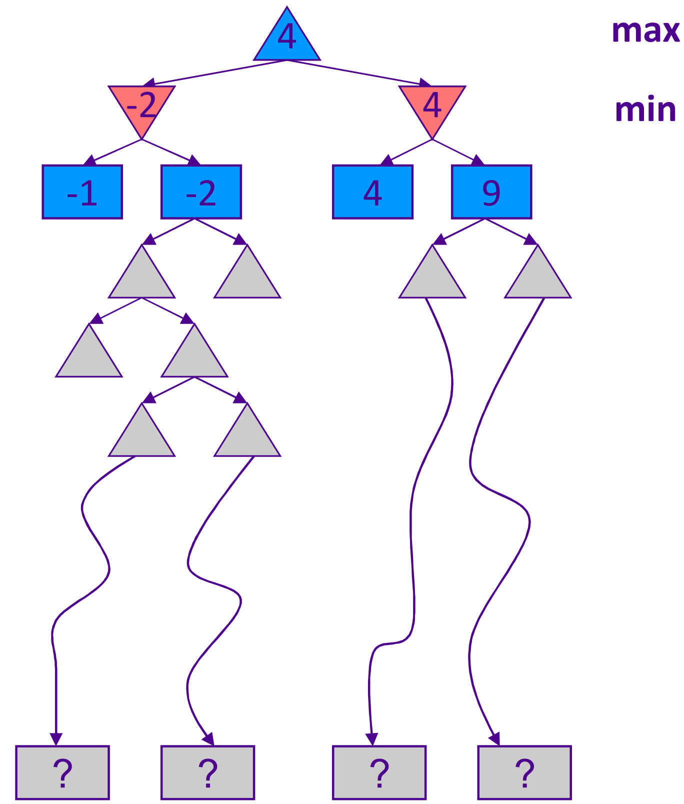
- Examining the game tree we could find out which successive moves lead White to win and which of them lead to a win for Black
- The branching factor for chess is $b = 35$, and the length of longest play is infinite
- No matter what search strategy we use, in practice all board configurations cannot be evaluated

- Instead use a **payoff** (or utility) **function** to estimate how the board configuration evolves as moves are chosen
- The simplest payoff could be determined at the end of the game; did we win (1), draw (½), or lose (0)
- More applicable is to estimate any board position by, e.g., summing up the (difference of) material values of remaining pieces

  pawn 1, knight 3, bishop 3, rook 5, queen 9

Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha - \beta$ reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Tampere University

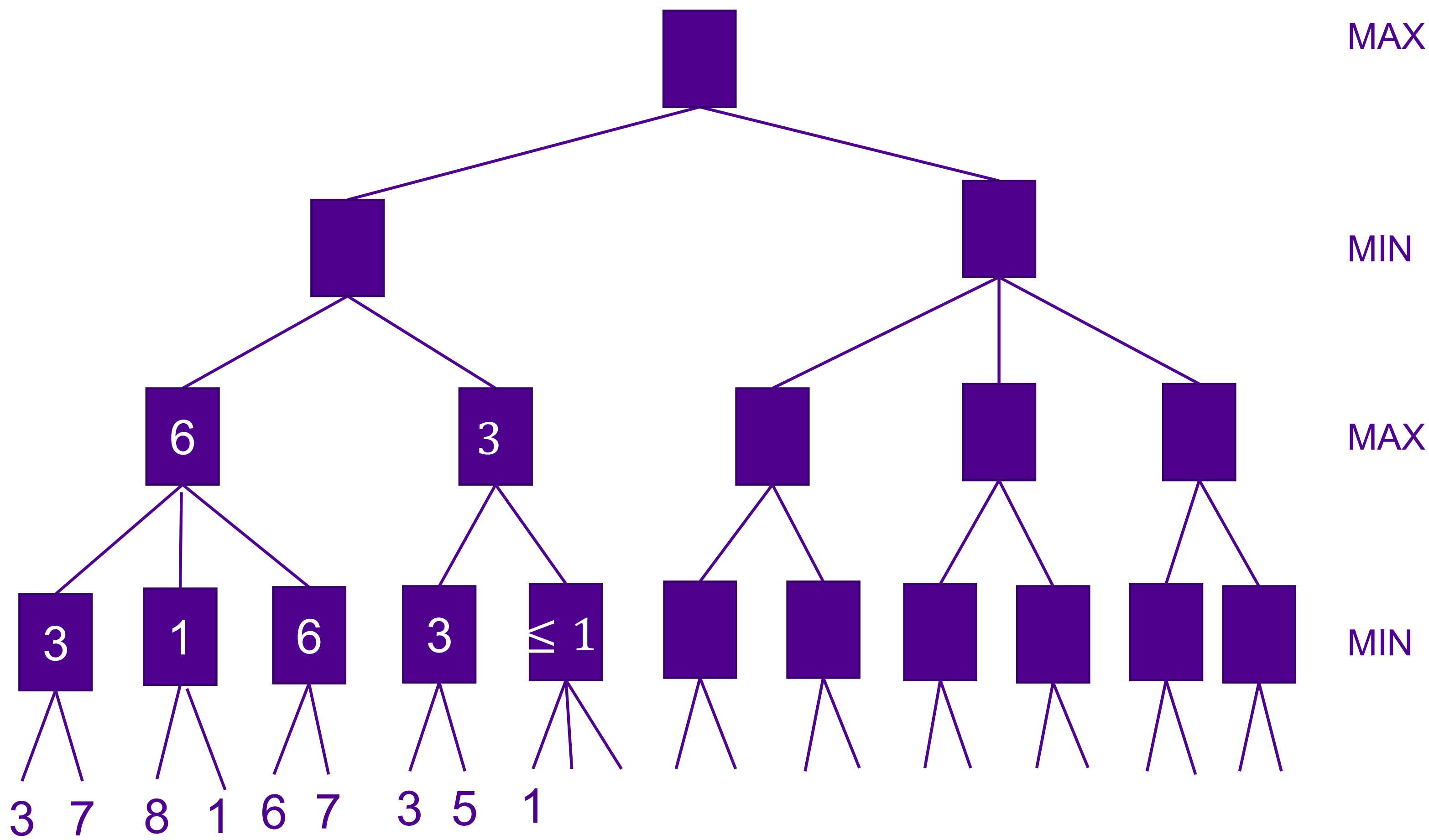
Video of Demo Limited Depth (2)

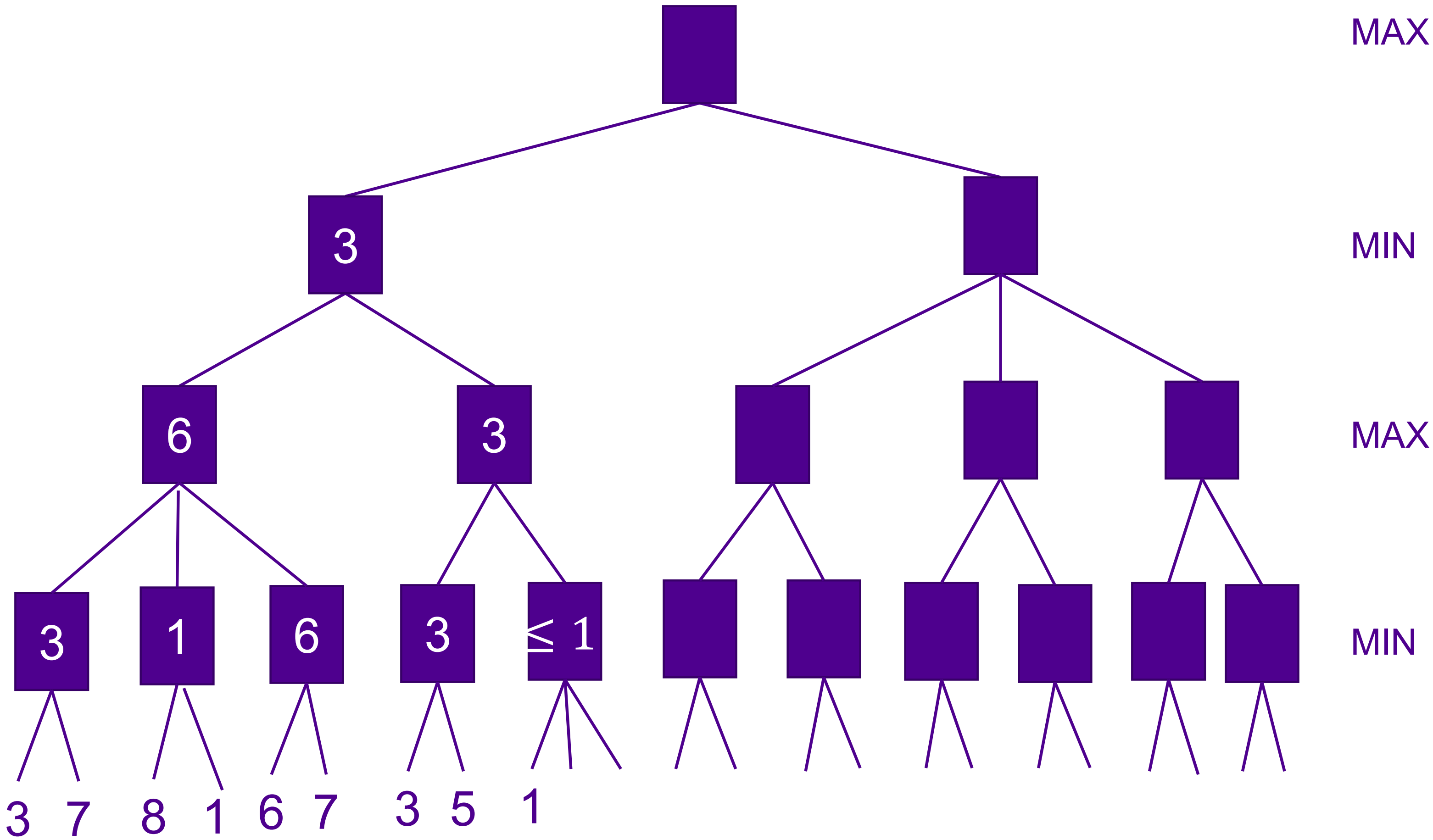
Video of Demo Limited Depth (10)

Tampere University

Alpha-beta pruning

- The exponential complexity of minimax search can be alleviated by pruning the nodes of the game tree that get evaluated
- It is possible to compute the correct minimax decision without looking at every node in the game tree
- Alpha-beta pruning gets its name from the parameters that describe bounds on the backed-up values that appear anywhere along the path
  - α = the value of the best (highest-value) choice we have found so far at any choice point along the path for $\max$
  - β = the value of the best (lowest-value) choice we have found so far at any choice point along the path for $\min$

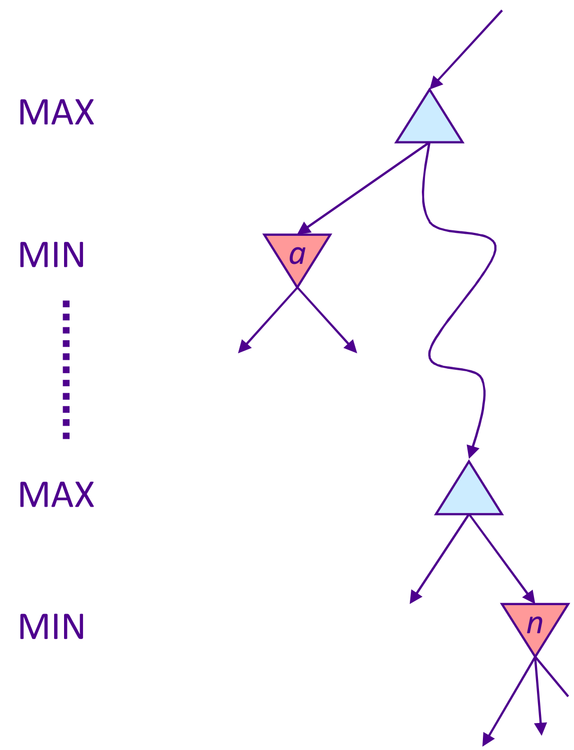

MAX
MIN
MAX
MIN
6
3
3
1
6
3
≤ 1
3 7 8 1 6 7 3 5 1


MAX
MIN
MAX
MIN
3
6
3
3
1
6
3
≤ 1
3 7 8 1 6 7 3 5 1

Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node n
  - We're looping over n's children
  - n's estimate of the childrens' min is dropping
  - Who cares about n's value? MAX
  - Let a be the best value that MAX can get at any choice point along the current path from the root
  - If n becomes worse than a, MAX will avoid it, so we can stop considering n's other children (it's already bad enough that it won't be played)

- MAX version is symmetric



Tampere University

Alpha-Beta Implementation

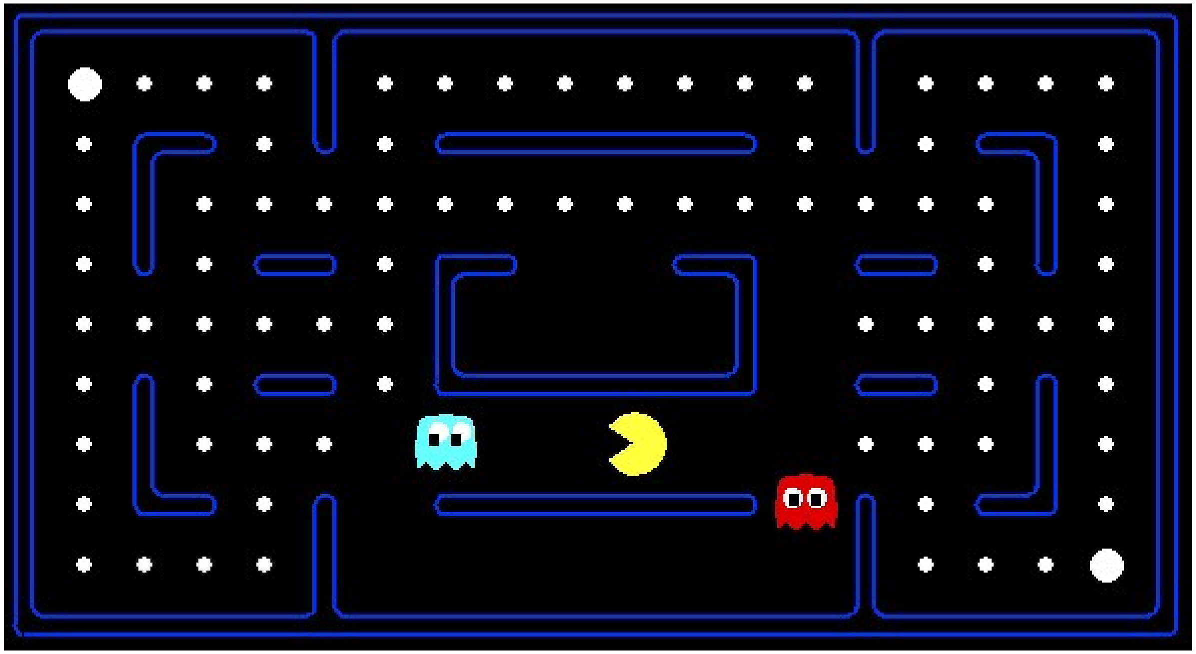
α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Tampere University

Behavior from Computation



Video of Demo Mystery Pacman