ST. FRANCIS XAVIER UNIVERSITY

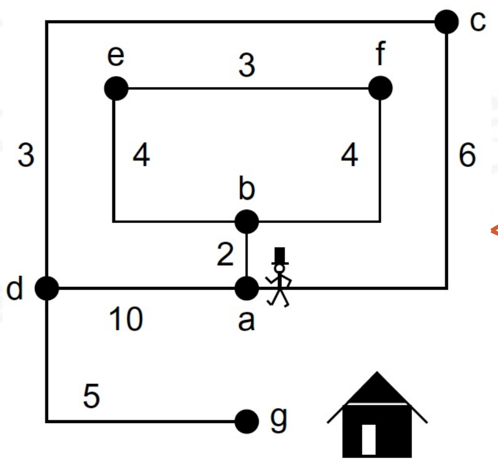
CSCI-564 CONSTRAINT PROCESSING AND HEURISTIC SEARCH

LECTURE 16 – ADVERSARY SEARCH

Dr. Jean-Alexis Delamer

Recap

We can represent most of problems as a shortest path problem.

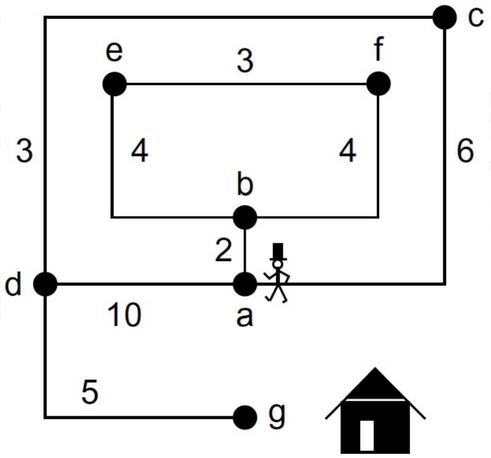


Optimal solution
- Blind search
- Informed search

Suboptimal solution
- State space pruning
- Real-Time Search

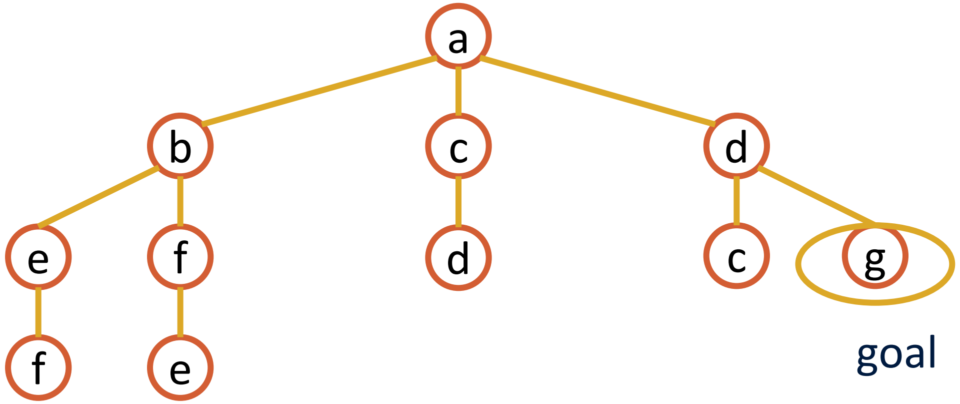
Recap

We can represent most of problems as a shortest path problem.



Blind search

DFS, BFS



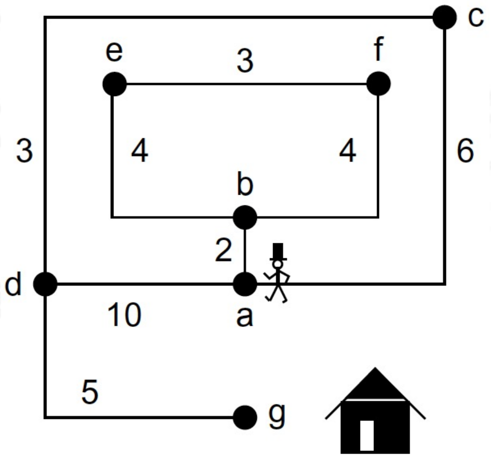
Dijkstra

k=0		j						
		a	b	c	d	e	f	g
i	a	0	2	6	10	∞	∞	∞
	b	2	0	∞	∞	4	4	∞
	c	6	∞	0	3	∞	∞	∞
	d	10	∞	3	0	∞	∞	5
	e	∞	4	∞	∞	0	3	∞
	f	∞	4	∞	∞	3	0	∞
	g	∞	∞	∞	5	∞	∞	0

Heavy computation.

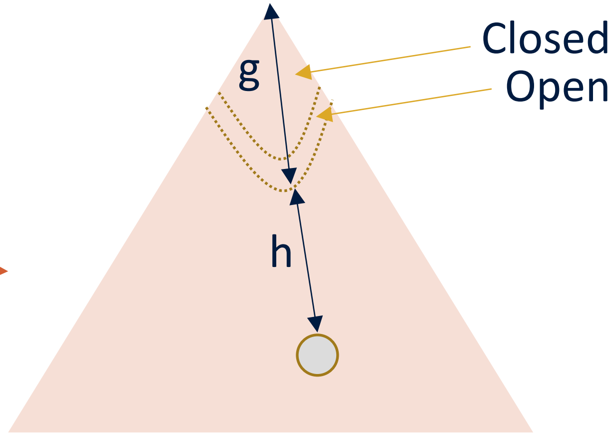
Recap

We can represent most of problems as a shortest path problem.



Help the search effort, but some problems can be too large.

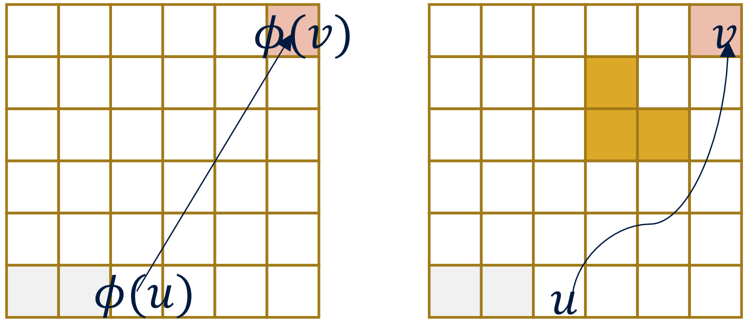
Informed search: Using a heuristic to guide the search



A*

The h-value can be calculated on an abstract problem.

Simplified version



Concrete problem

Abstraction of the problem

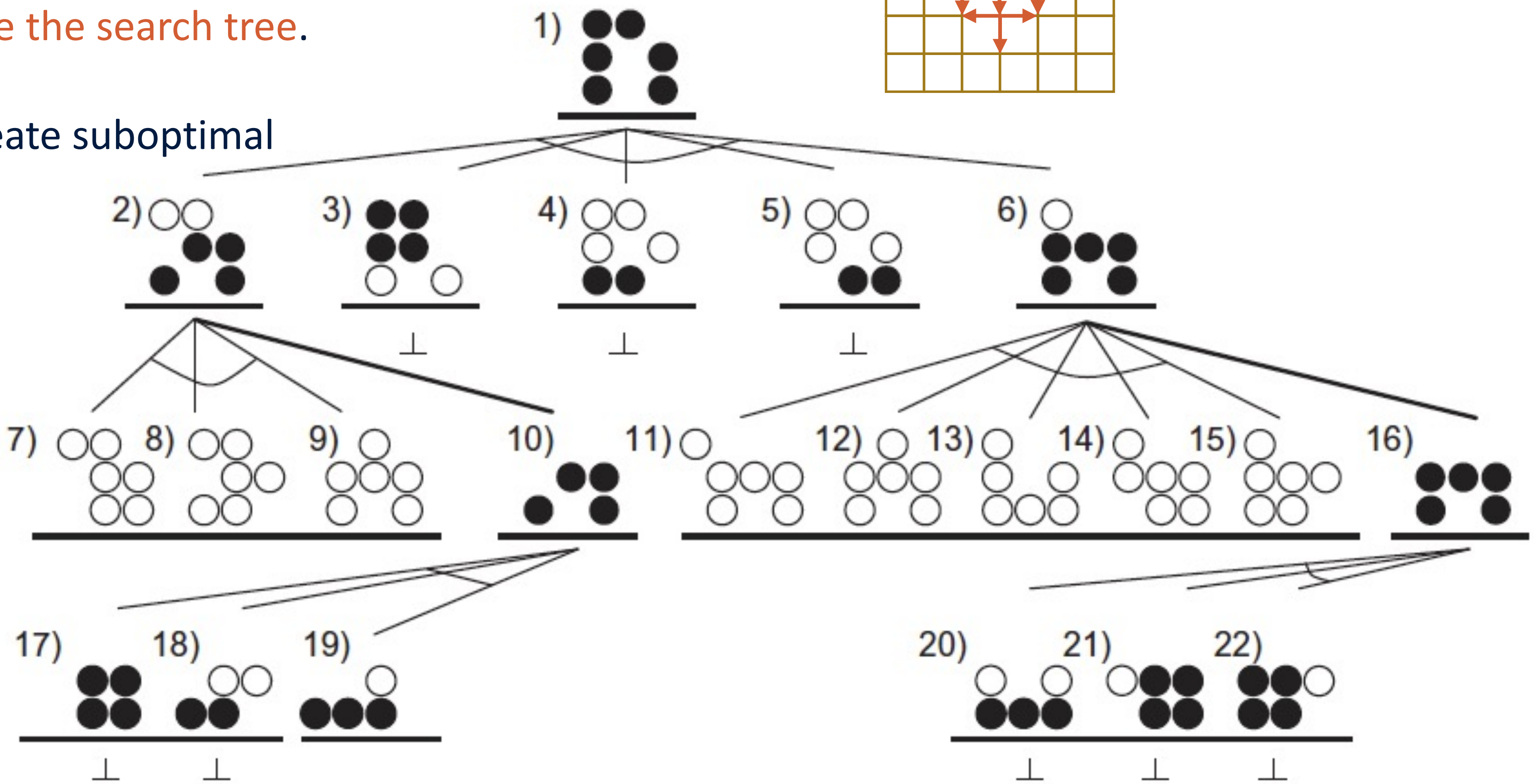
StFX

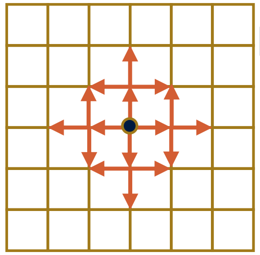
Recap

When the problem is too large, we can try to prune the search tree.

Pruning can create suboptimal solution.

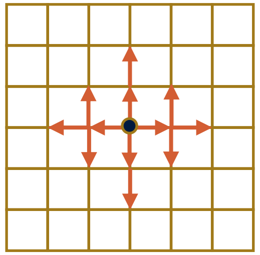
Detect and prune dead-ends.





Reject the following duplicates

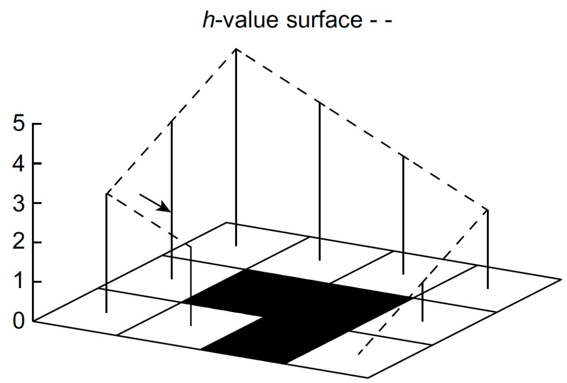
$D = \{DR, DL, UR, UL\}$

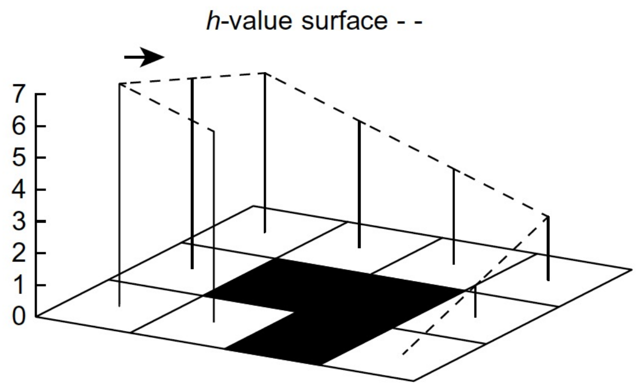


Recap

Or we can use Real-Time search algorithm.

- Suboptimal solution
- Decide the time allowed to the search between each action.





Adversary Search

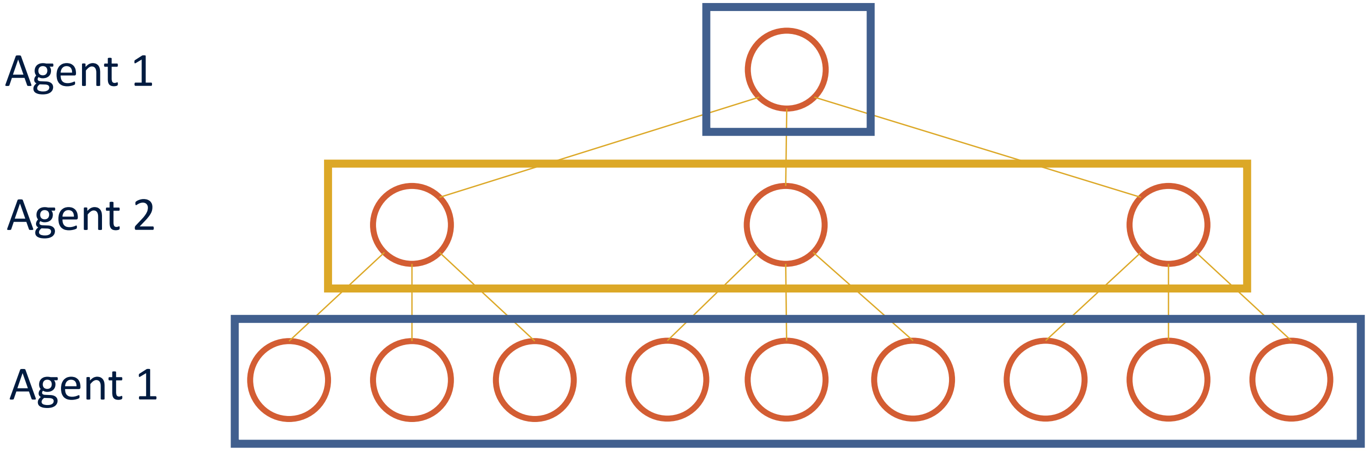
- We didn't discuss problem with adversaries.

- How can we model problem in which you compete with another agent?
  - One way is to represent the problem as a game.
  - Usually referred as Game Theory.

Adversary Search

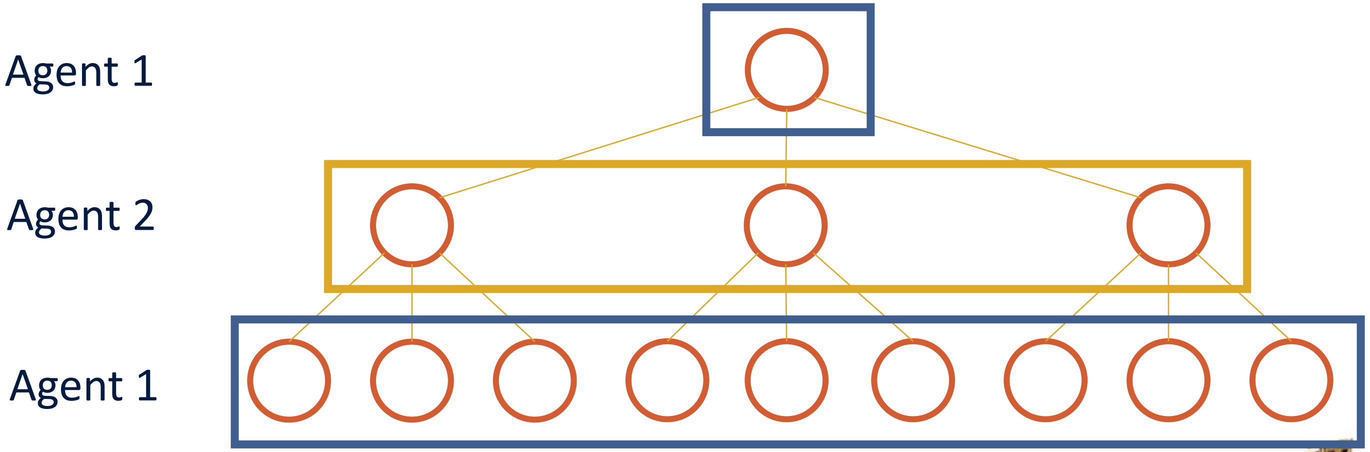
- What difference does it make to search of the optimal solution?
  - Adversaries introduce uncertainty.

- You don't decide of all the actions in the search tree.



Adversary Search

- In game theory, we are not talking of optimal solution.
- We want to find the optimal strategy.
- What is the difference?
  - Optimal strategies result in perfect play.
  - The players take actions alternately and independently.



Adversary Search

- We will focus on algorithms such as negmax and minimax.
- With one pruning strategy: $\alpha\beta$.

- In game theory:
  - The search trees are rather depth-bounded than cost-bounded.
  - The value are computed with a static evaluation function.
  - Why?

Nondeterministic environments

- Not only problems with adversary agents can be represented this way.
- In nondeterministic or probabilistic environments:
  - We include problem where the “adversary” is the unpredictable behavior of nature.
  - The outcome of an executed action in a state is not unique.
    - The ack of knowledge for modeling the real world precisely.
    - Sensors and actuators that are imprecise.
    - Etc.

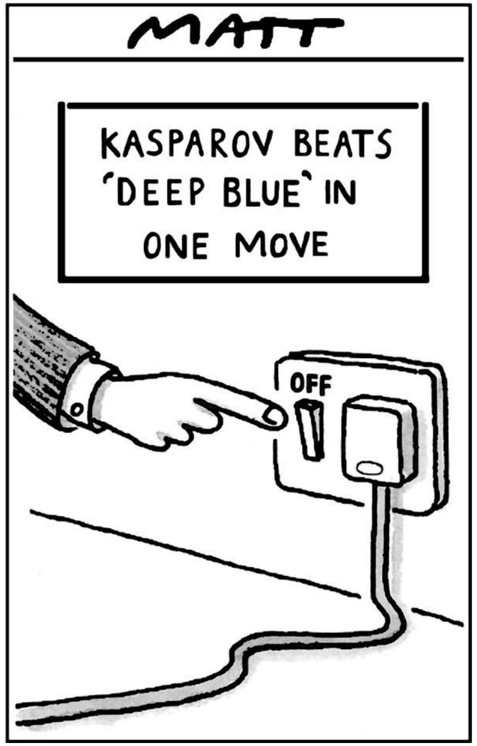
Nondeterministic environments

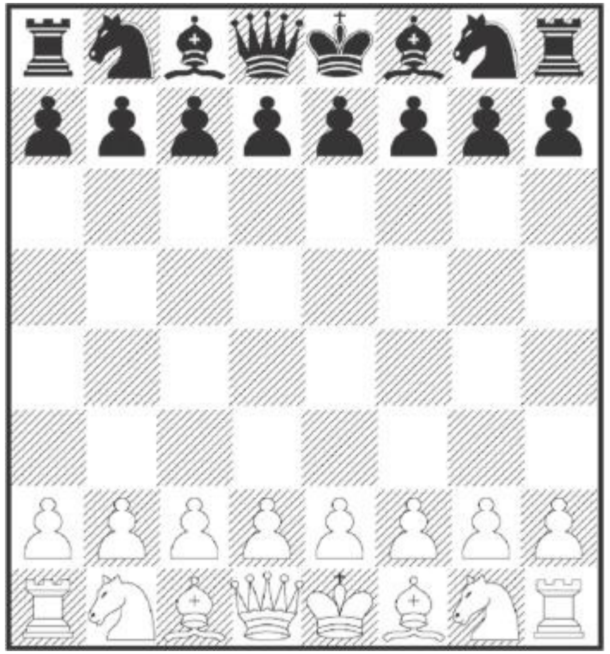
- Solutions to nondeterministic problems are not sequences of actions.
  - Why?
- Solutions are presented as mappings from state to actions.
  - We call it policies.
- It requires state space traversal to return a solution.
  - Policy are often represented as value function.
    - Assigns a value to each state.

Two-Player Games

- Chess is the typical example for two-player games.

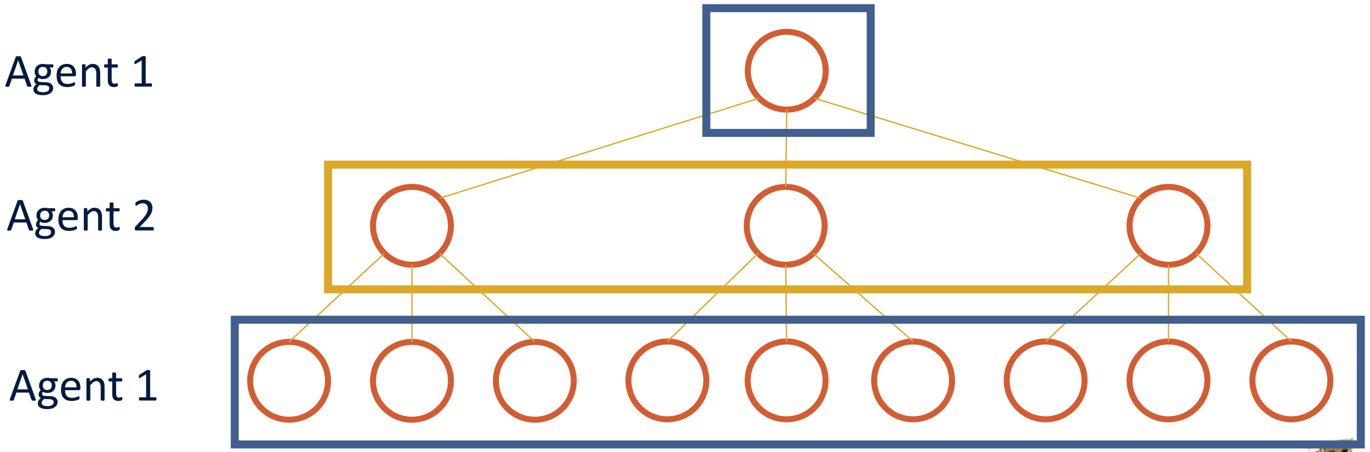
- One of the main successes in AI.





Two-Player Games

- To select an optimal move in a two-player game, we construct a game tree.
  - A node represent a board configuration.
  - The root is the current configuration.
  - The children are reachable configurations.

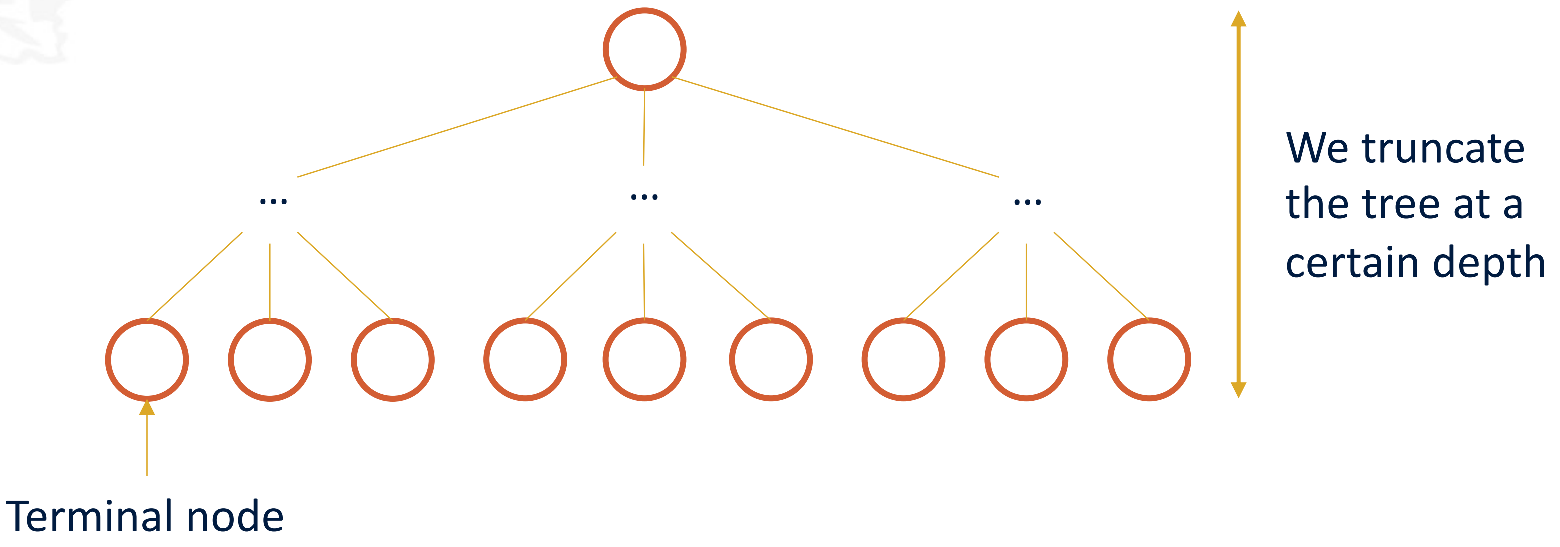


Two-Player Games

- Game theory is an entire field of research.
- We will focus on zero-sum games.
  - The win of one player is the loss of the other.
- And games with perfect information.

Game Tree Search

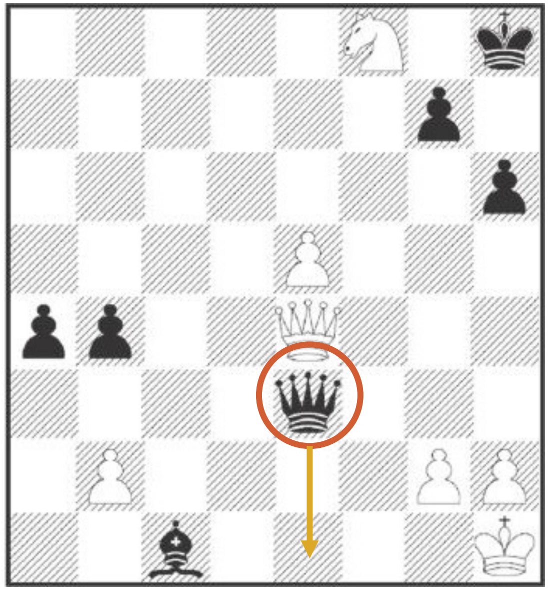
- The tree is generally too large in two-player games.
- Terminal nodes are evaluated by a heuristic procedure called static evaluation function.
- Making decision before knowing the optimal action is similar to real-time search.



Game Tree Search

- Static evaluator:
  - Doesn't need to be correct.
  - But it needs to yield the correct values for terminal positions (Goals).
  - And that higher values correlate with better positions.

- There is no notion of admissibility.



Game Tree Search

- The performance of the algorithm depends on the static evaluation function.

- What is a good static evaluation function?
  - Balance between giving an indicative value without excessive computational cost.
  - Usually developed by expert in laborious, meticulous trial-and-error experimentation.
- For complex problem, a perfect evaluation function doesn't exist.

Game Tree Search

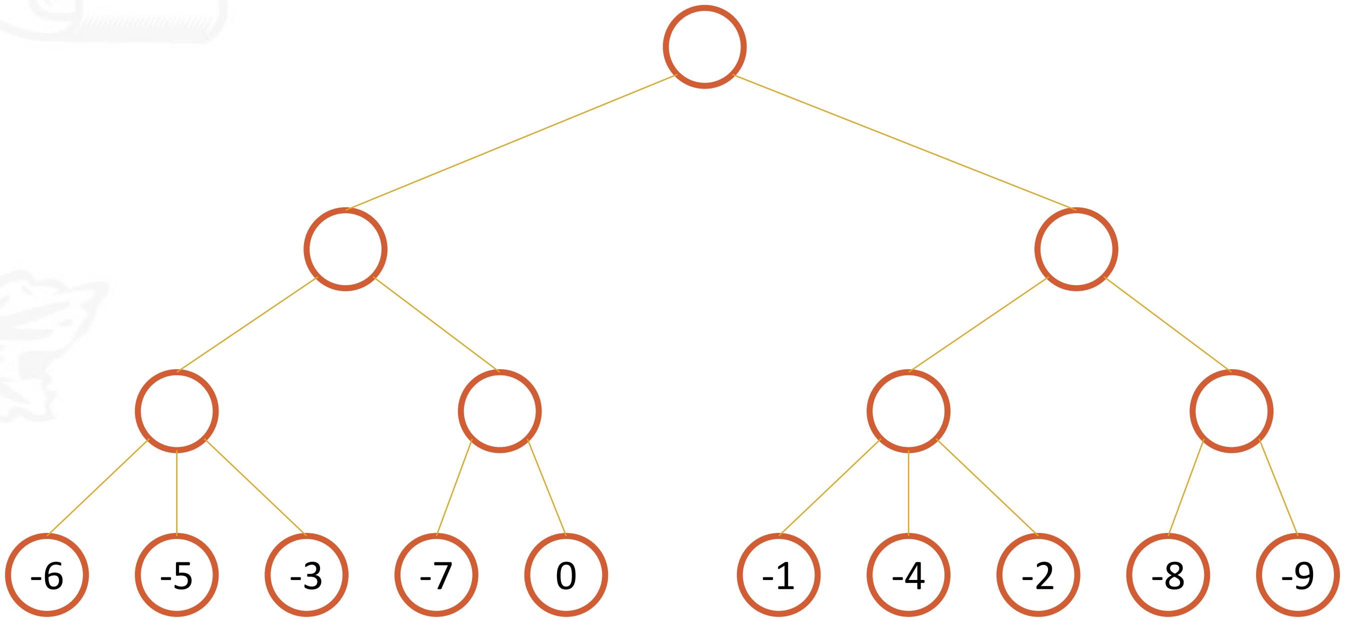
- A first algorithm: negmax.
- The idea is simple:
  - We assume that the value of a position for the first agent is the negative of its value for the second agent.
  - Meaning that we choose a move that maximizes his worst-case return.

Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

if $(\mathit{leaf}(u))$ **return** $\mathit{Eval}(u)$	;; No successor, static evaluation
$\mathit{res} \leftarrow -\infty$	;; Initialize value *res* for current frame
for each $v \in \mathit{Succ}(u)$	;; Traverse successor list
$\mathit{res} \leftarrow \max\{\mathit{res}, -\mathit{Negmax}(v)\}$	;; Update value *res*
return *res*	;; Return final evaluation

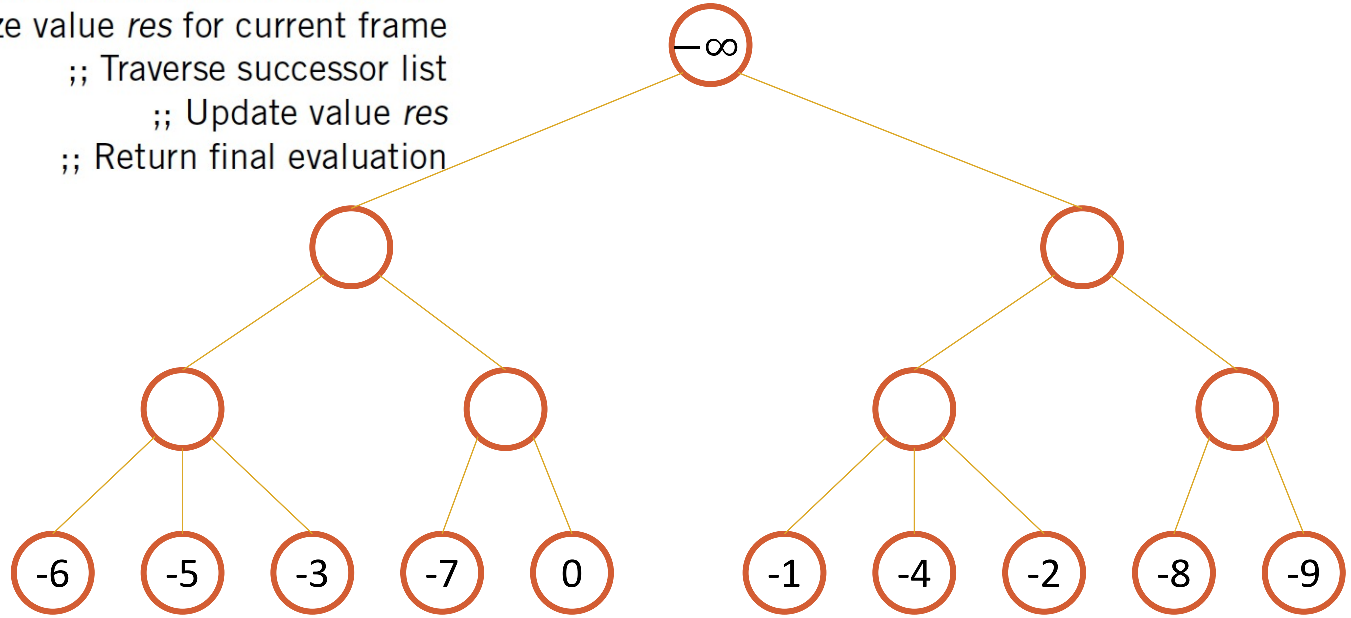
Game Tree Search



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

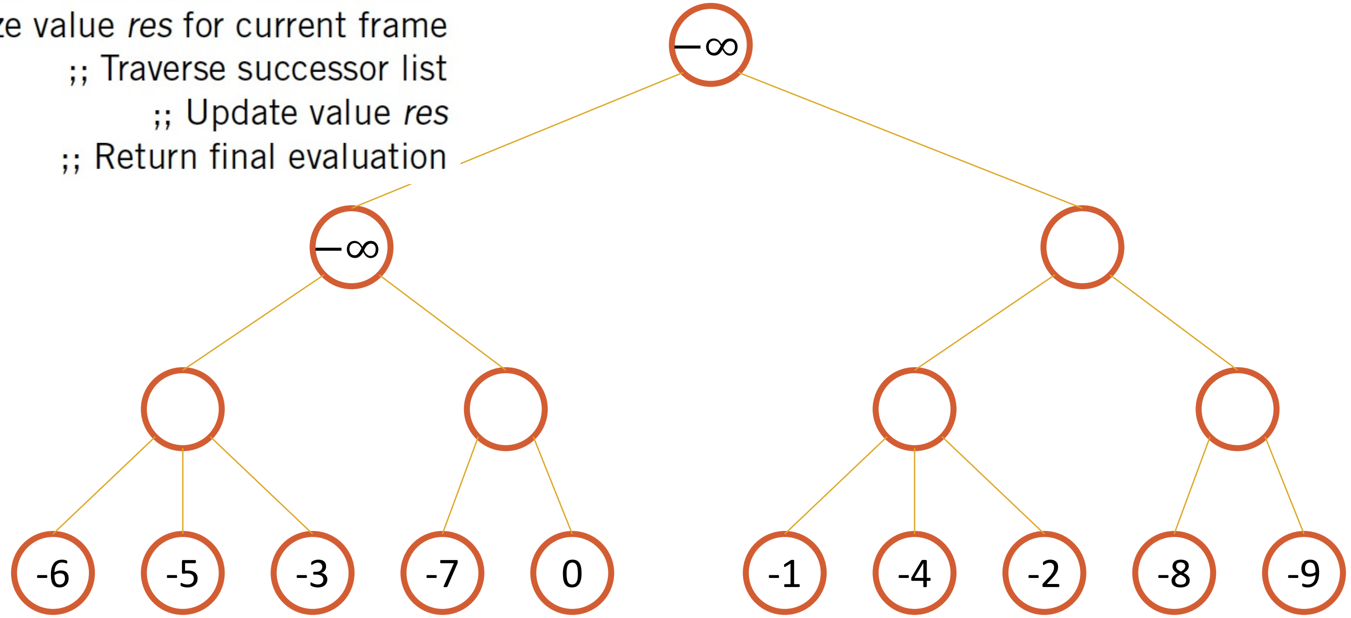
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
    $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

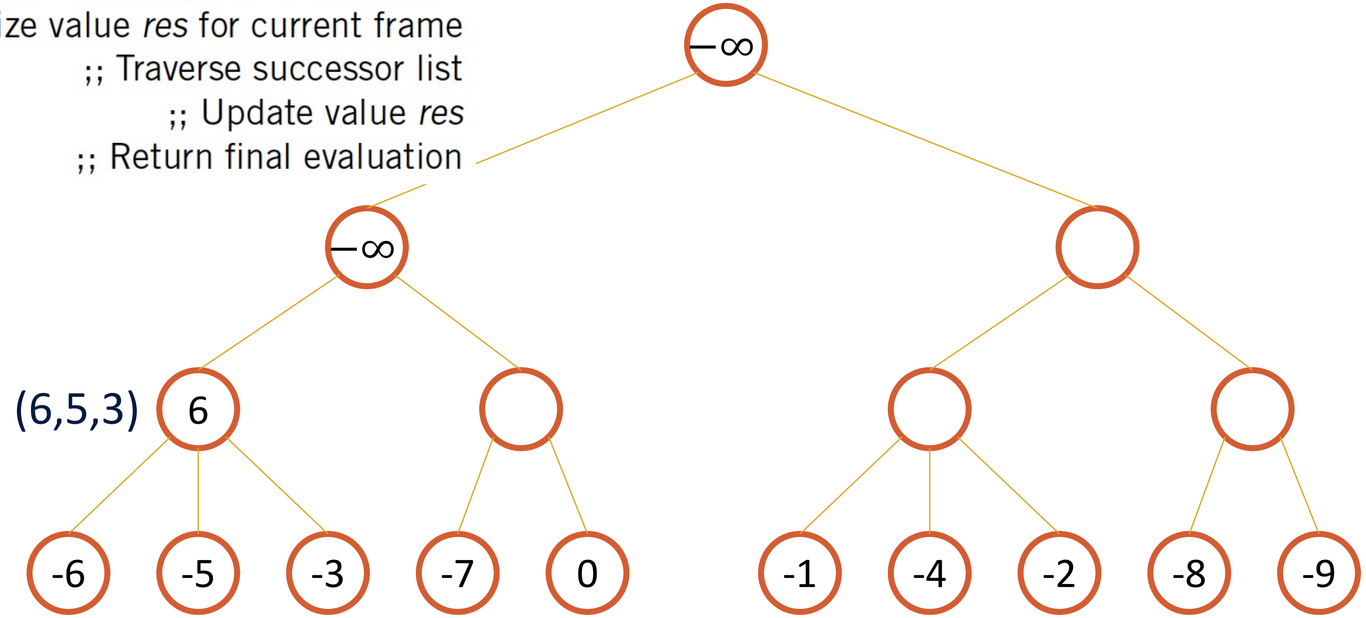
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
  $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

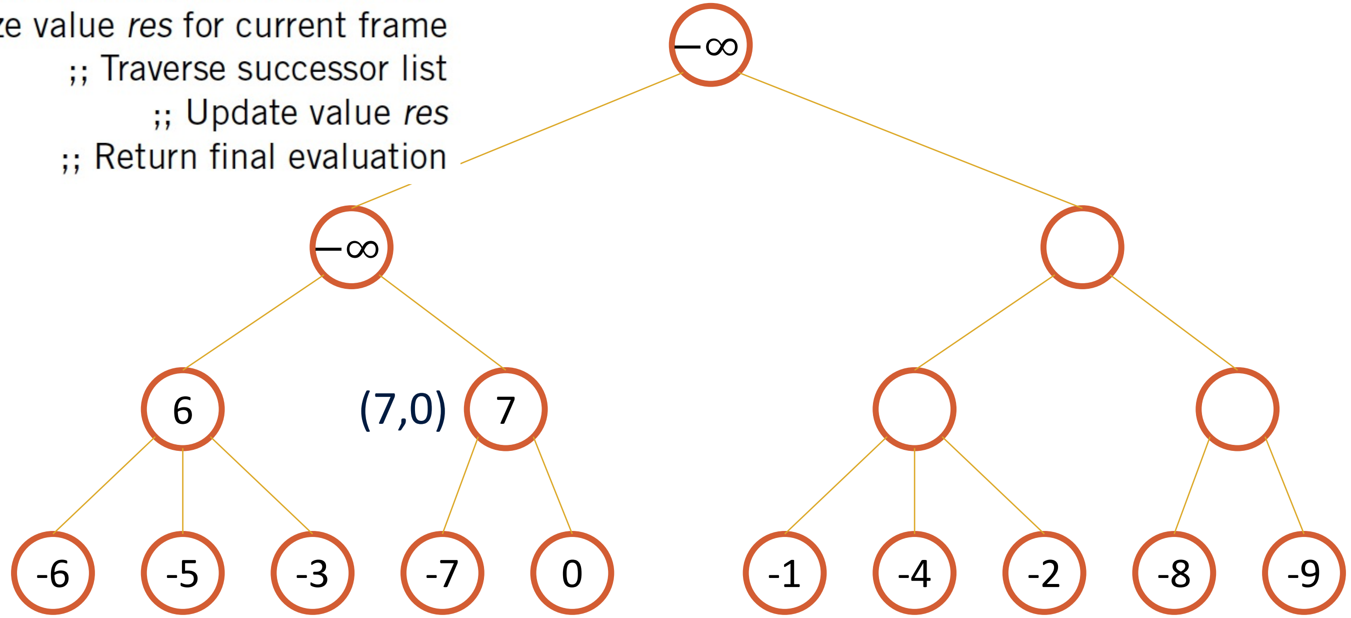
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
    $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

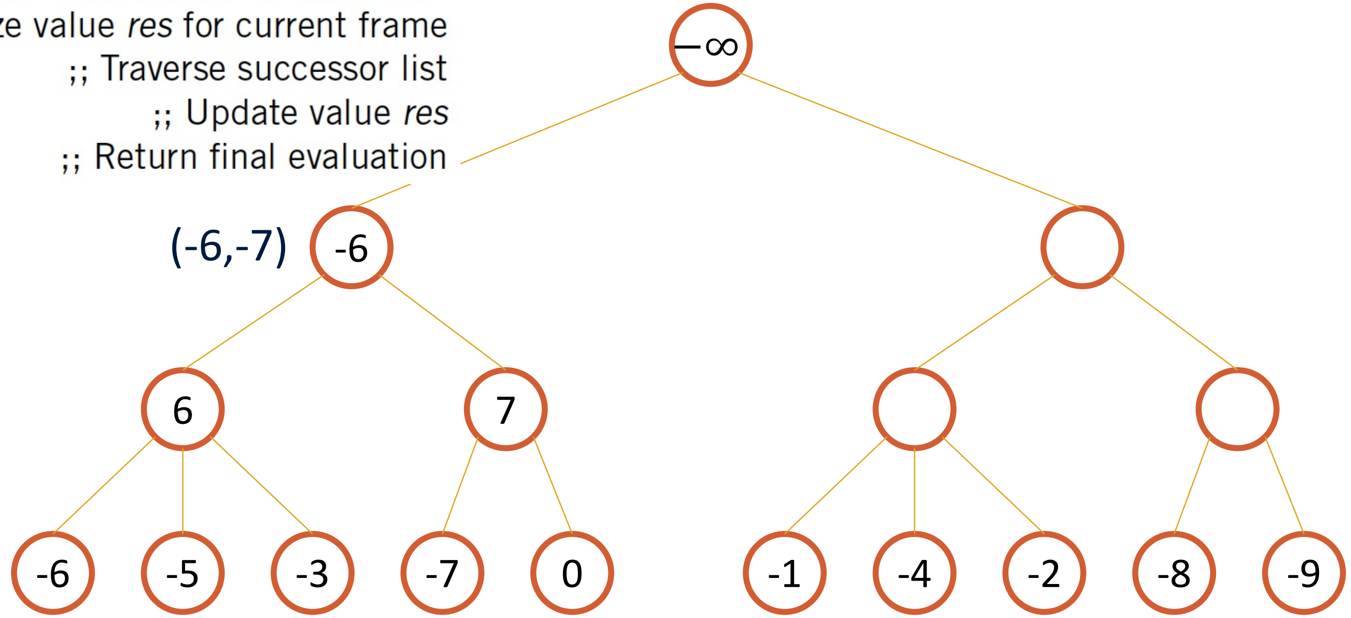
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
    $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

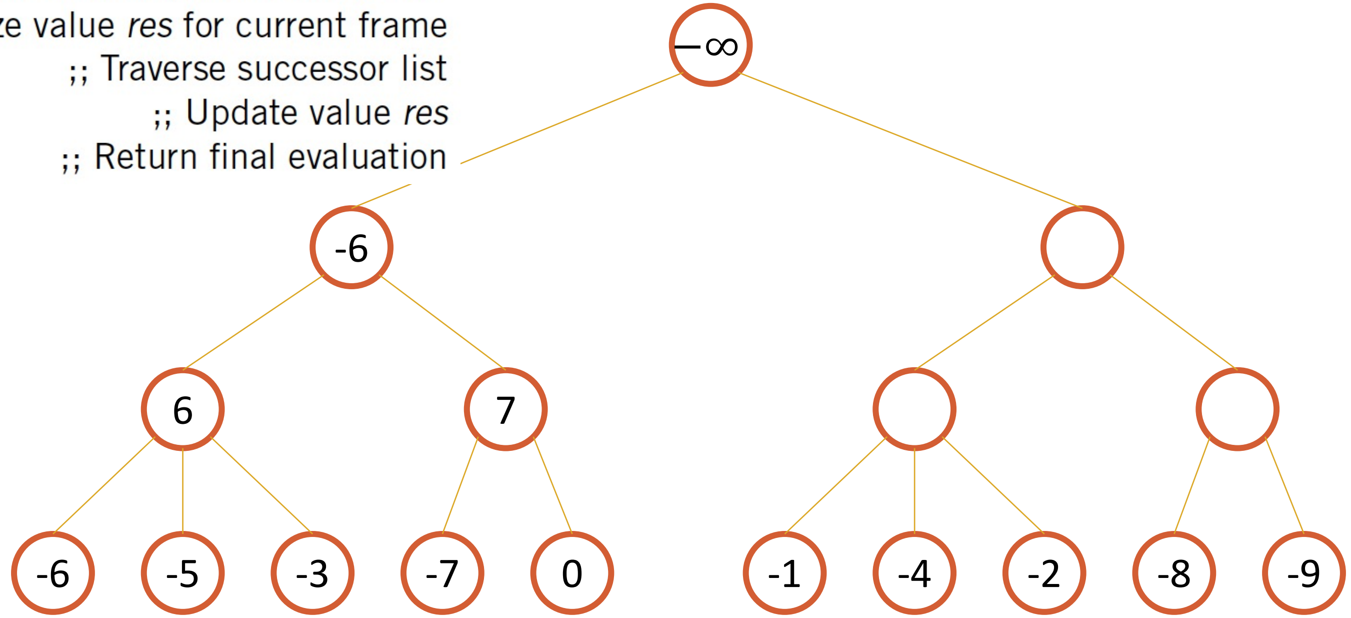
if ($leaf(u)$) **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
  $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

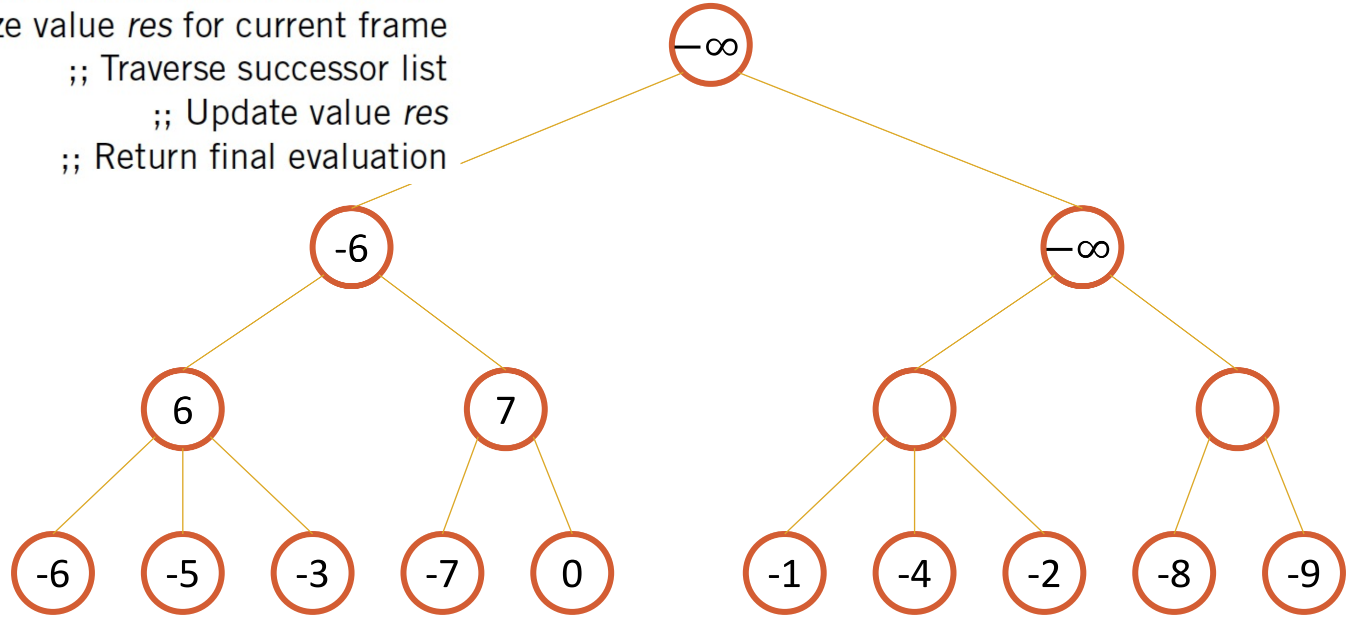
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
    $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

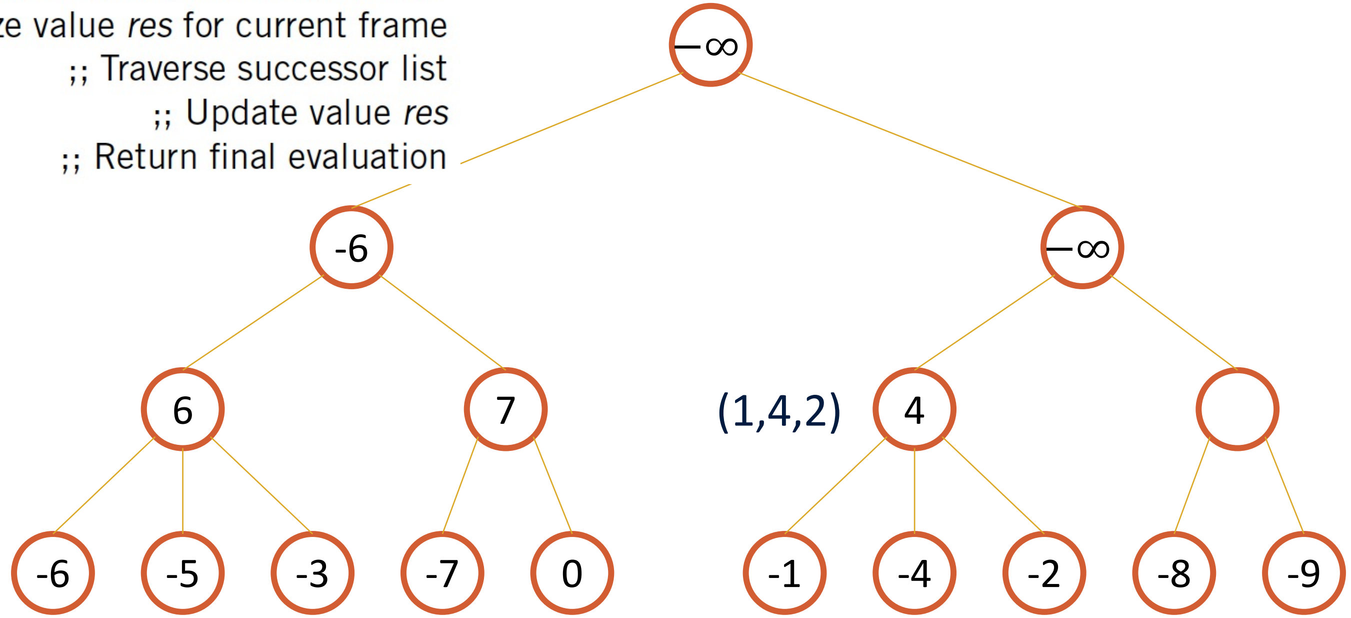
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
    $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

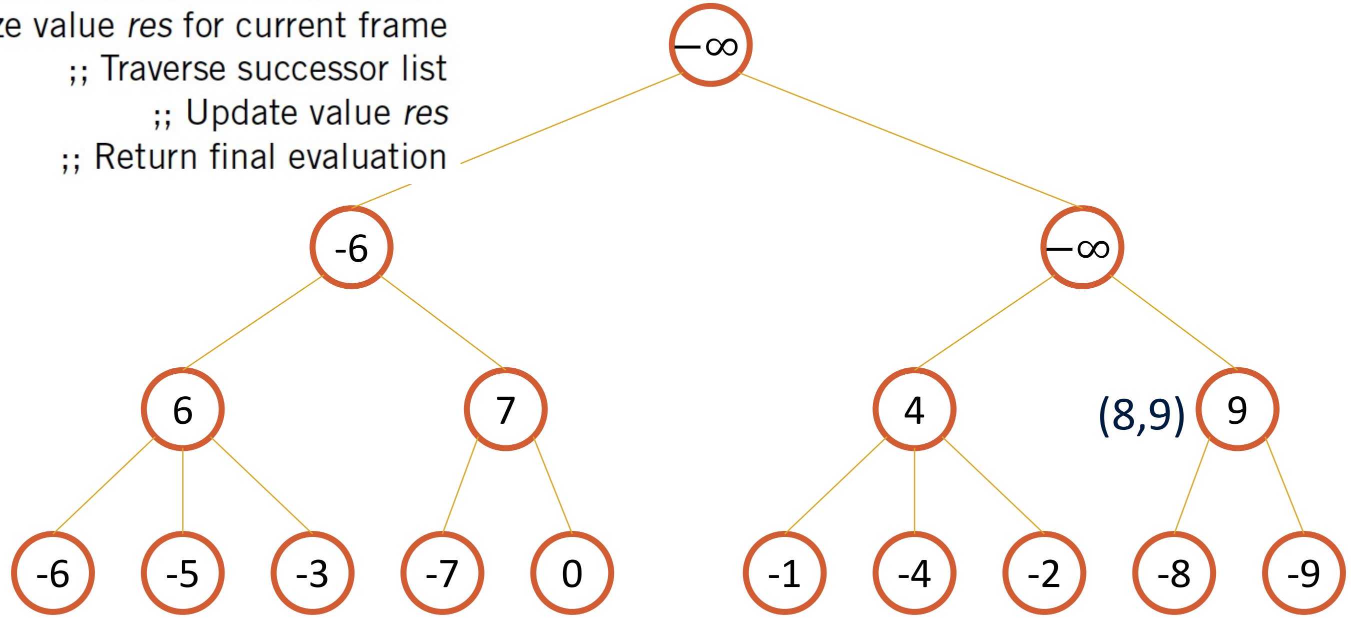
if $(leaf(u))$ **return** $Eval(u)$	;; No successor, static evaluation
$res \leftarrow -\infty$	;; Initialize value *res* for current frame
for each $v \in Succ(u)$	;; Traverse successor list
$\quad res \leftarrow \max\{res, -Negmax(v)\}$	;; Update value *res*
return *res*	;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

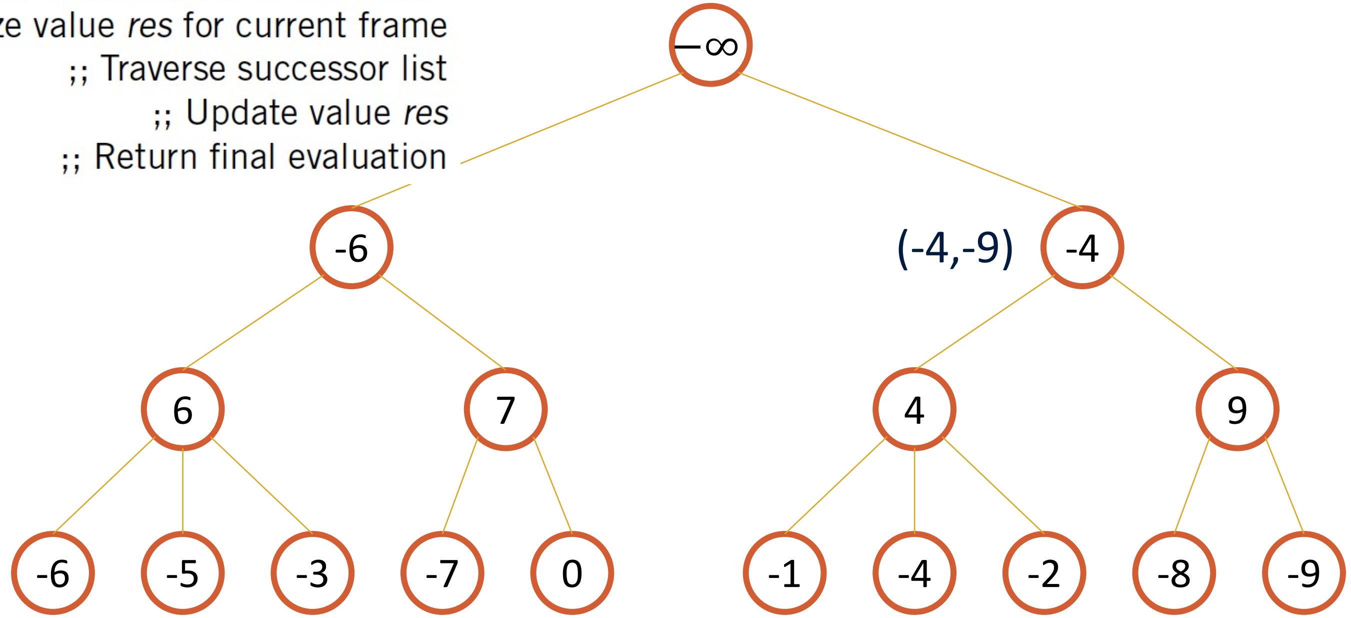
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
$res \leftarrow -\infty$;; Initialize value *res* for current frame
for each $v \in Succ(u)$;; Traverse successor list
  $res \leftarrow \max\{res, -Negmax(v)\}$;; Update value *res*
return *res* ;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

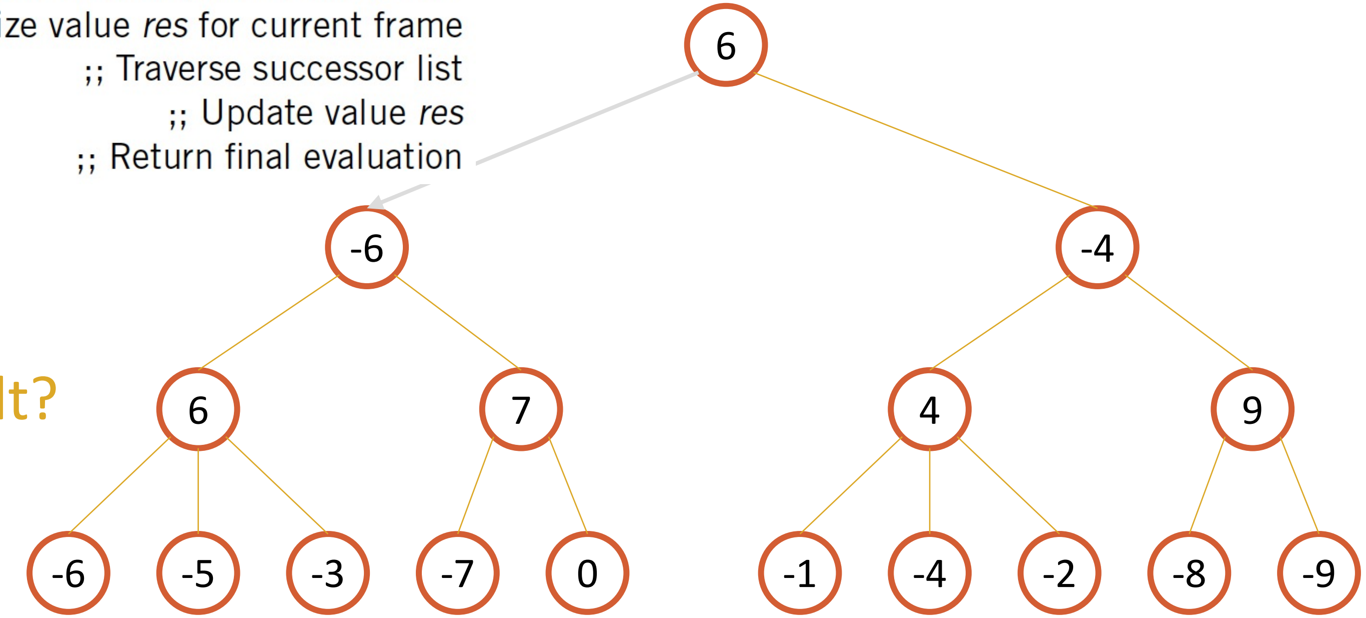
if $(leaf(u))$ **return** $Eval(u)$	;; No successor, static evaluation
$res \leftarrow -\infty$	;; Initialize value *res* for current frame
for each $v \in Succ(u)$	;; Traverse successor list
$\quad res \leftarrow \max\{res, -Negmax(v)\}$	;; Update value *res*
return *res*	;; Return final evaluation



Game Tree Search

Procedure Negmax
Input: Position u
Output: Value at root

if $(leaf(u))$ **return** $Eval(u)$	;; No successor, static evaluation
$res \leftarrow -\infty$	;; Initialize value *res* for current frame
for each $v \in Succ(u)$	;; Traverse successor list
$\quad res \leftarrow \max\{res, -Negmax(v)\}$	;; Update value *res*
return *res*	;; Return final evaluation



What do you think about this result?

Game Tree Search

- A second algorithm: minimax

- The idea is similar:
  - The tree consists of two different types of nodes.
    - The MIN nodes for the player that tries to minimize the payoff.
    - The MAX nodes for the players that tries to maximize the payoff.
  - The agent try to maximize its payoff during its turn.
  - And the adversary try to minimize the payoff of the agent during his turn.

Game Tree Search

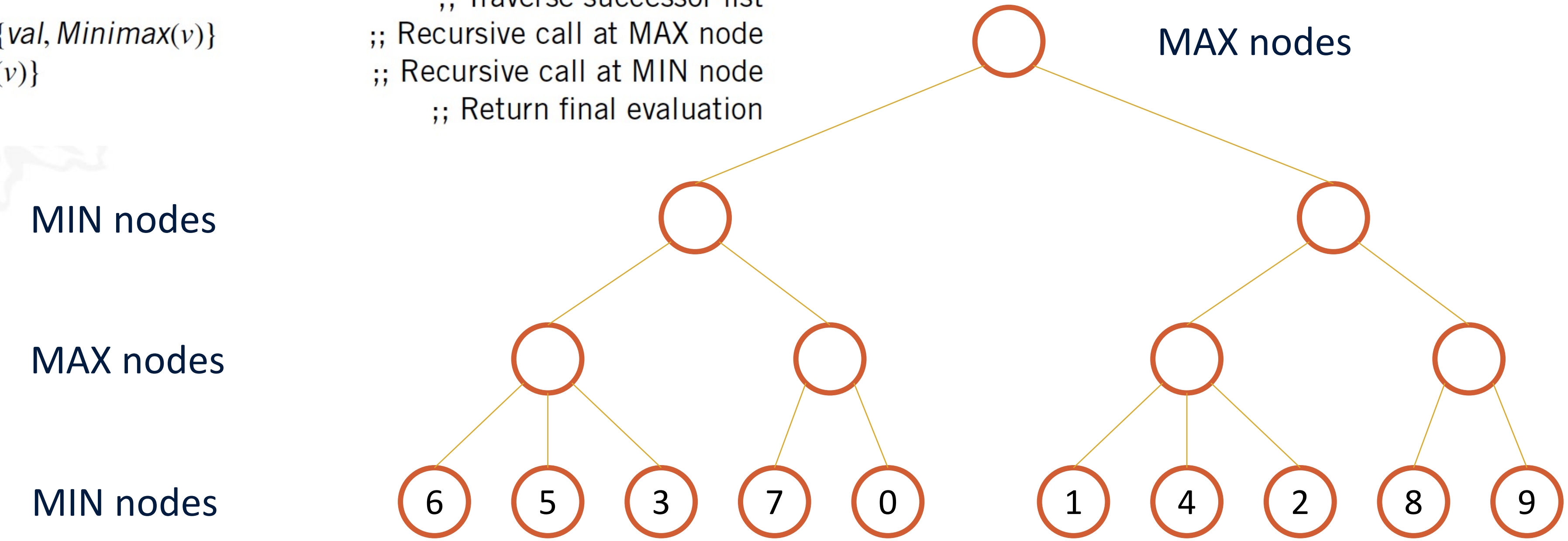
Procedure Minimax
Input: Position u
Output: Value at root

```
if (leaf(u)) return Eval(u)                           ;; No successor, static evaluation
if (max-node(u)) val ← −∞                             ;; Initialize return value for MAX node
else val ← +∞                                         ;; Initialize return value for MIN node
for each v ∈ Succ(u)                                  ;; Traverse successor list
   if (max-node(u)) val ← max{val, Minimax(v)}        ;; Recursive call at MAX node
   else val ← min{val, Minimax(v)}                    ;; Recursive call at MIN node
return res                                            ;; Return final evaluation
```

Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

if $(\mathit{leaf}(u))$ **return** $\mathit{Eval}(u)$	;; No successor, static evaluation
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow -\infty$	;; Initialize return value for MAX node
else $\mathit{val} \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in \mathit{Succ}(u)$	;; Traverse successor list
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow \max\{\mathit{val}, \mathit{Minimax}(v)\}$	;; Recursive call at MAX node
else $\mathit{val} \leftarrow \min\{\mathit{val}, \mathit{Minimax}(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation

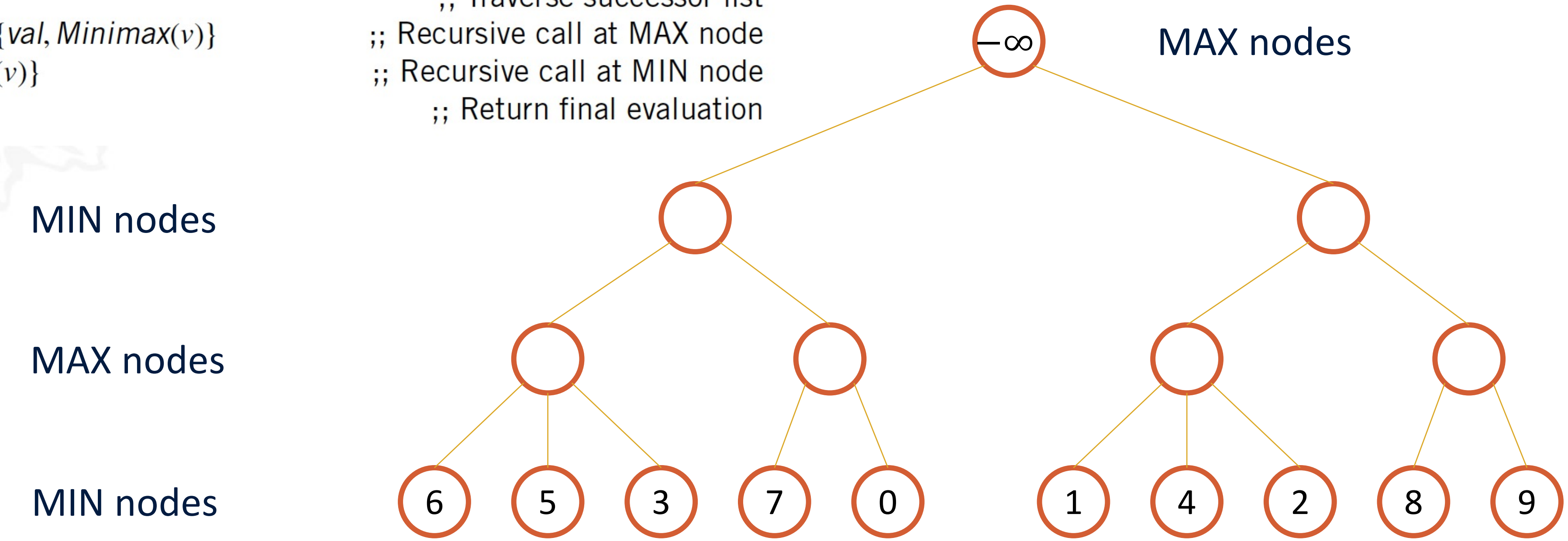


StFX

Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

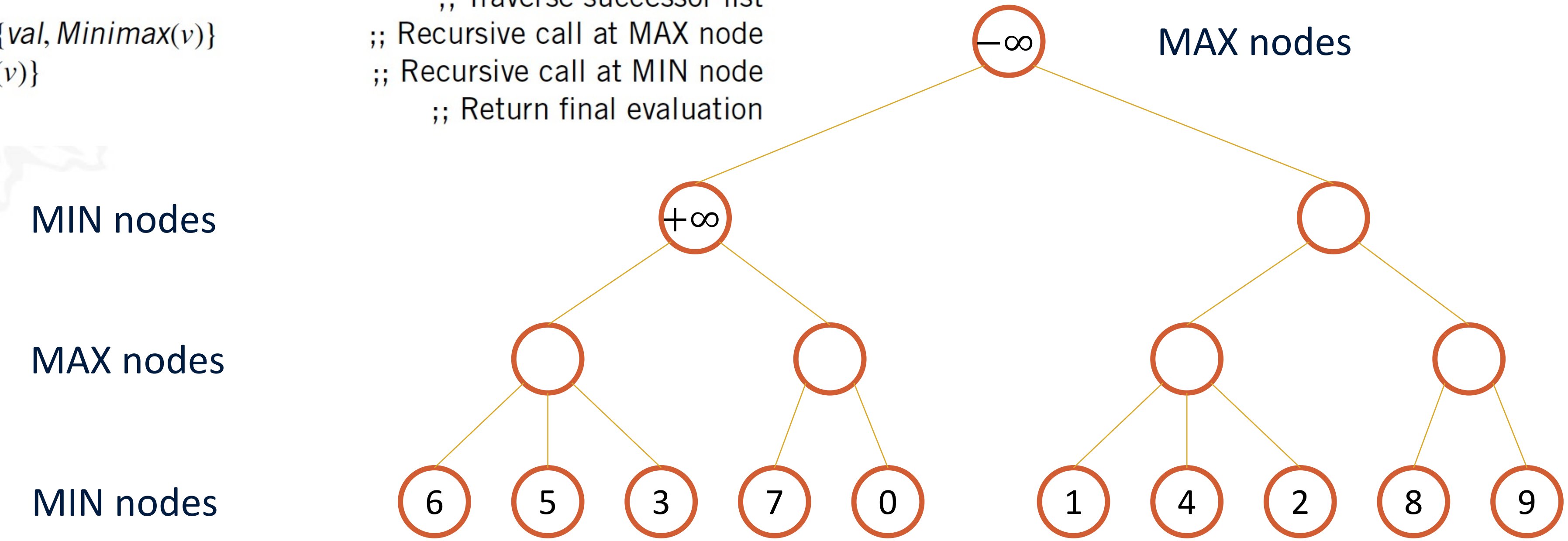
if $(\mathit{leaf}(u))$ **return** $\mathit{Eval}(u)$;; No successor, static evaluation
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow -\infty$;; Initialize return value for MAX node
else $\mathit{val} \leftarrow +\infty$;; Initialize return value for MIN node
for each $v \in \mathit{Succ}(u)$;; Traverse successor list
  if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow \max\{\mathit{val}, \mathit{Minimax}(v)\}$;; Recursive call at MAX node
  else $\mathit{val} \leftarrow \min\{\mathit{val}, \mathit{Minimax}(v)\}$;; Recursive call at MIN node
return res ;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

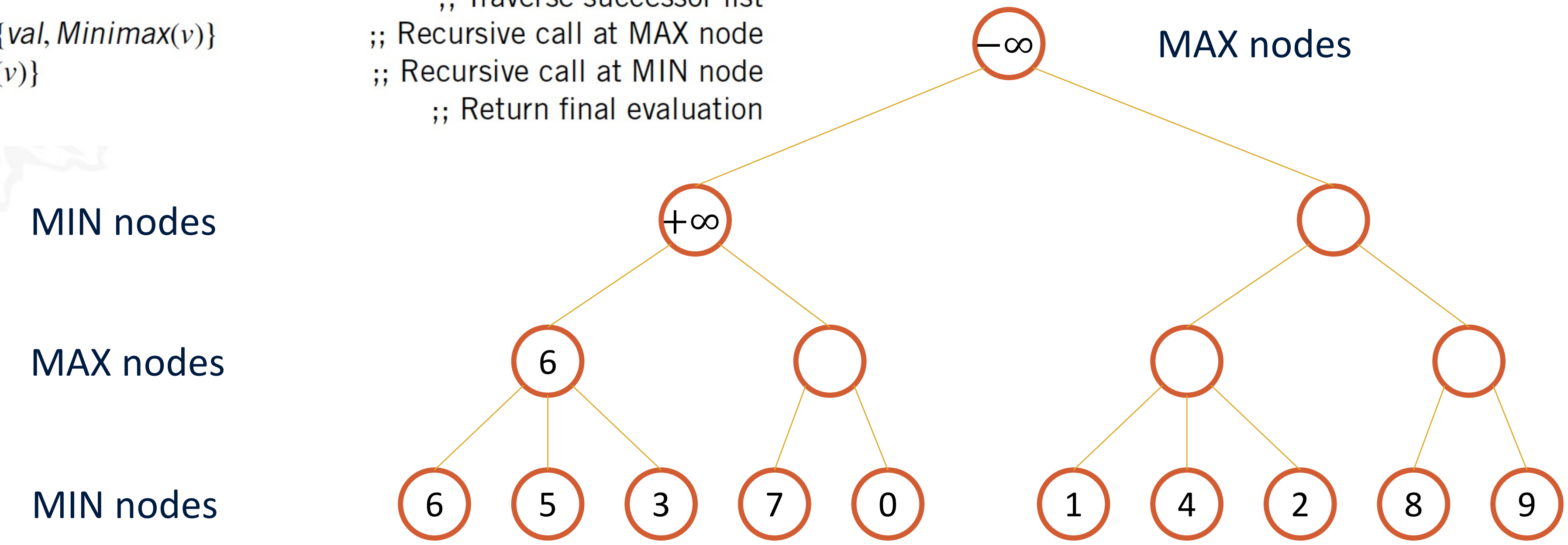
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
if $(max\text{-}node(u))$ $val \leftarrow -\infty$;; Initialize return value for MAX node
else $val \leftarrow +\infty$;; Initialize return value for MIN node
for each $v \in Succ(u)$;; Traverse successor list
  if $(max\text{-}node(u))$ $val \leftarrow \max\{val, Minimax(v)\}$;; Recursive call at MAX node
  else $val \leftarrow \min\{val, Minimax(v)\}$;; Recursive call at MIN node
return res ;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

if ($leaf(u)$) **return** $Eval(u)$	;; No successor, static evaluation
if ($max\text{-}node(u)$) $val \leftarrow -\infty$	;; Initialize return value for MAX node
else $val \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in Succ(u)$	;; Traverse successor list
if ($max\text{-}node(u)$) $val \leftarrow \max\{val, Minimax(v)\}$	;; Recursive call at MAX node
else $val \leftarrow \min\{val, Minimax(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation

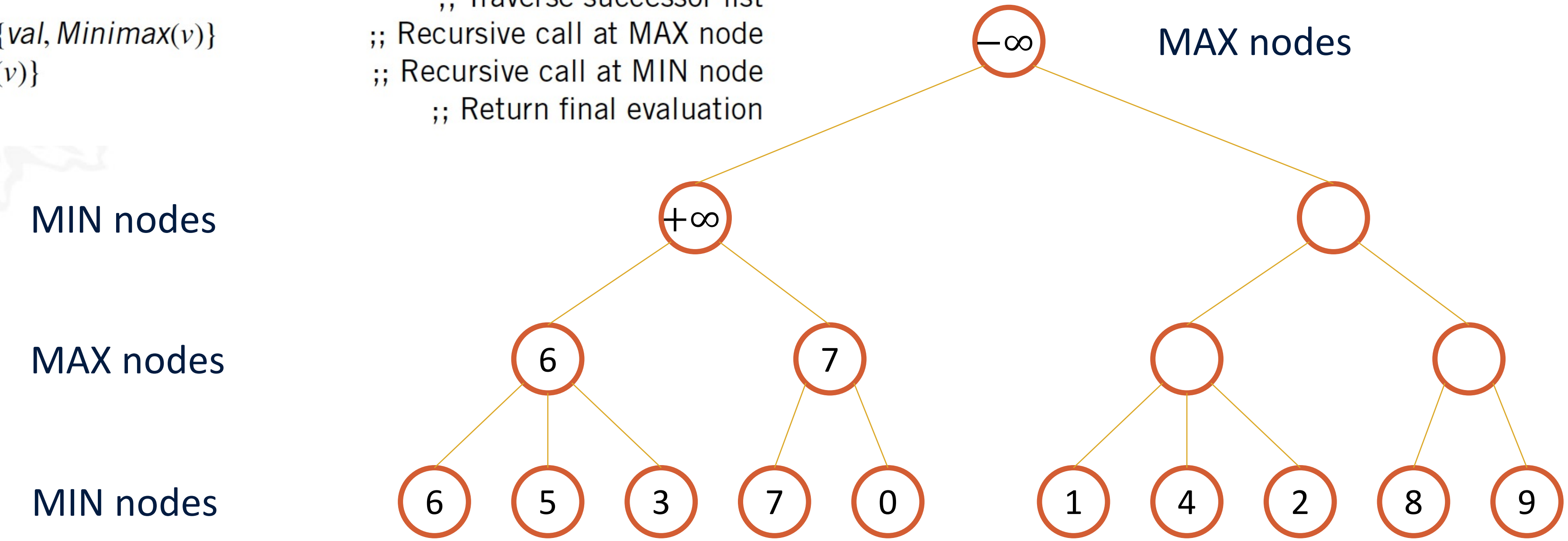


StFX

Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

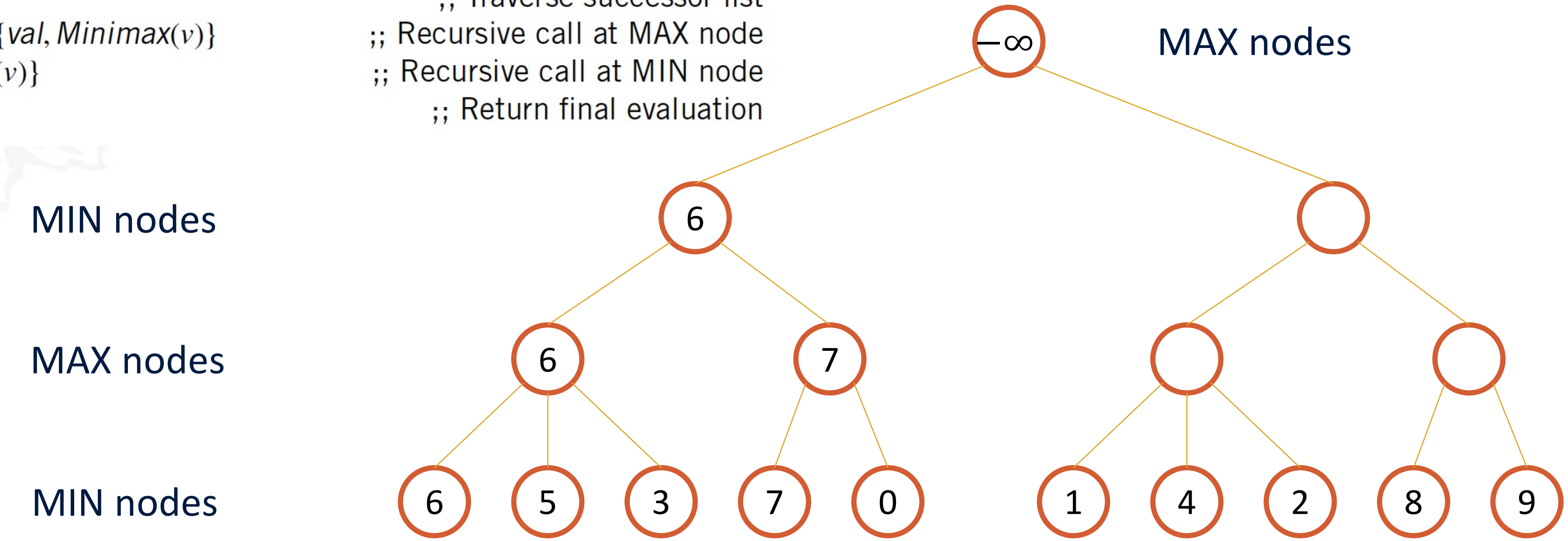
if ($leaf(u)$) **return** $Eval(u)$	;; No successor, static evaluation
if ($max\text{-}node(u)$) $val \leftarrow -\infty$	;; Initialize return value for MAX node
else $val \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in Succ(u)$	;; Traverse successor list
if ($max\text{-}node(u)$) $val \leftarrow \max\{val, Minimax(v)\}$	;; Recursive call at MAX node
else $val \leftarrow \min\{val, Minimax(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

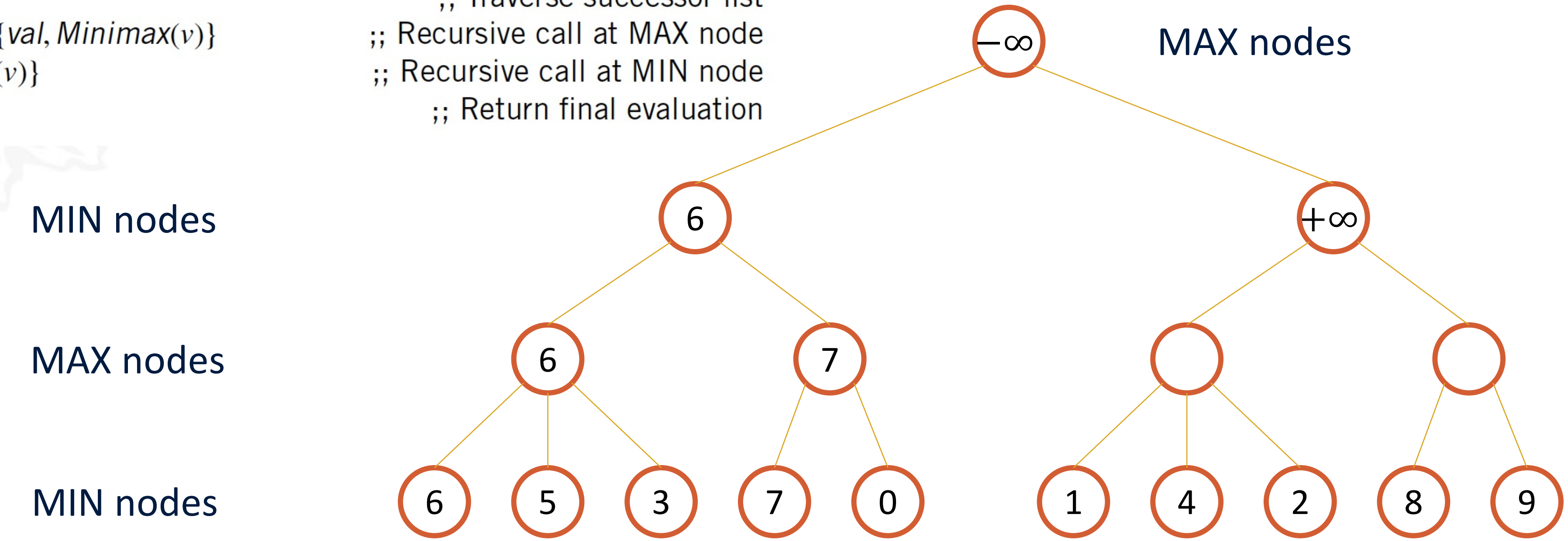
if ($leaf(u)$) **return** $Eval(u)$	;; No successor, static evaluation
if ($max\text{-}node(u)$) $val \leftarrow -\infty$	;; Initialize return value for MAX node
else $val \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in Succ(u)$	;; Traverse successor list
if ($max\text{-}node(u)$) $val \leftarrow \max\{val, Minimax(v)\}$	;; Recursive call at MAX node
else $val \leftarrow \min\{val, Minimax(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

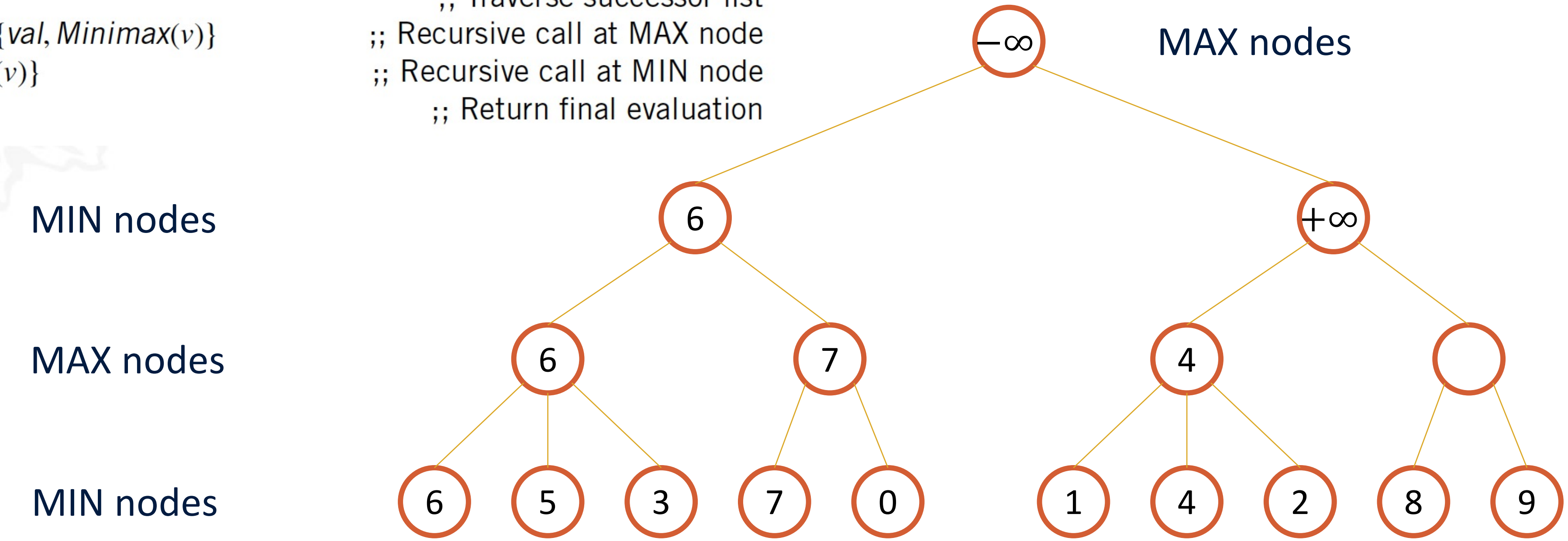
```
if (leaf(u)) return Eval(u)                      ;; No successor, static evaluation
if (max-node(u)) val ← −∞                        ;; Initialize return value for MAX node
else val ← +∞                                    ;; Initialize return value for MIN node
for each v ∈ Succ(u)                             ;; Traverse successor list
    if (max-node(u)) val ← max{val, Minimax(v)}  ;; Recursive call at MAX node
    else val ← min{val, Minimax(v)}              ;; Recursive call at MIN node
return res                                       ;; Return final evaluation
```



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

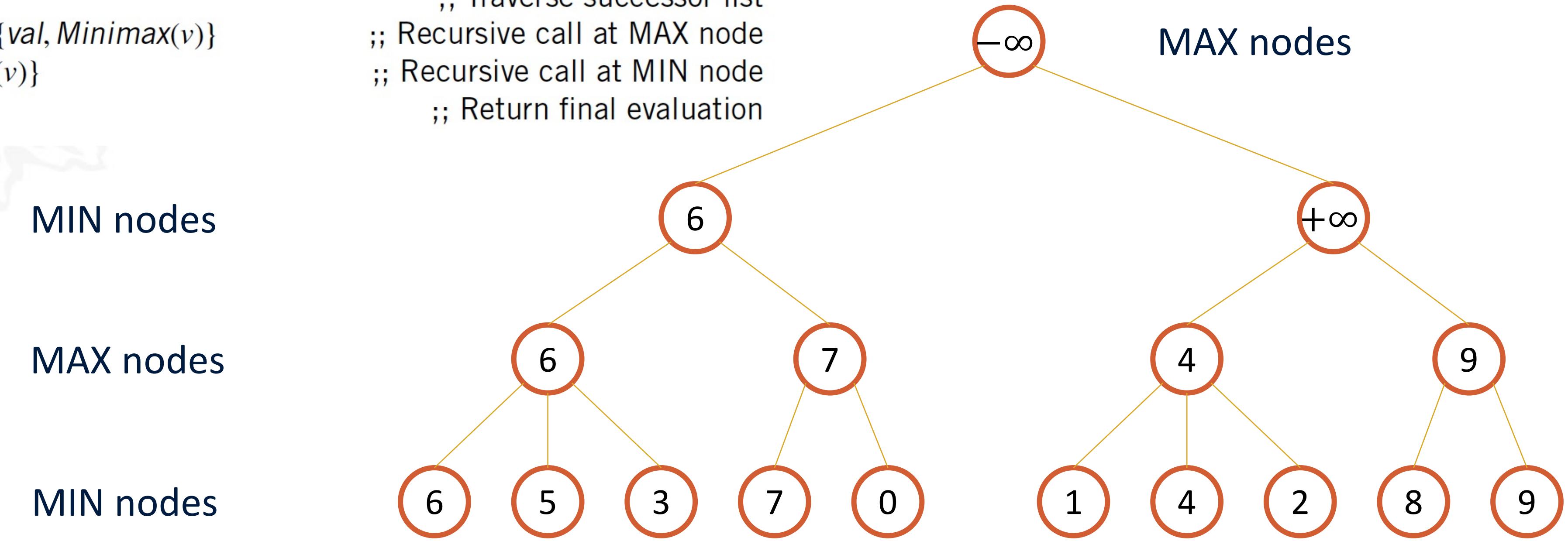
if $(leaf(u))$ **return** $Eval(u)$	;; No successor, static evaluation
if $(max\text{-}node(u))$ $val \leftarrow -\infty$	;; Initialize return value for MAX node
else $val \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in Succ(u)$	;; Traverse successor list
if $(max\text{-}node(u))$ $val \leftarrow \max\{val, Minimax(v)\}$	;; Recursive call at MAX node
else $val \leftarrow \min\{val, Minimax(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

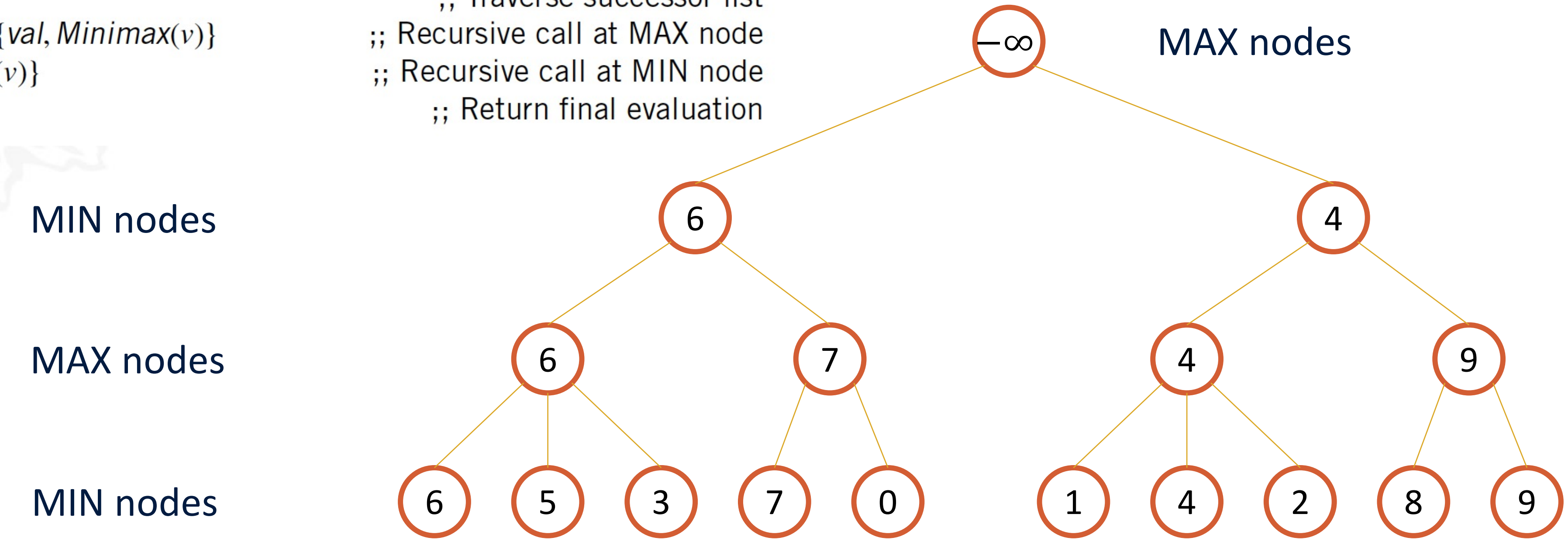
if $(\mathit{leaf}(u))$ **return** $\mathit{Eval}(u)$;; No successor, static evaluation
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow -\infty$;; Initialize return value for MAX node
else $\mathit{val} \leftarrow +\infty$;; Initialize return value for MIN node
for each $v \in \mathit{Succ}(u)$;; Traverse successor list
  if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow \max\{\mathit{val}, \mathit{Minimax}(v)\}$;; Recursive call at MAX node
  else $\mathit{val} \leftarrow \min\{\mathit{val}, \mathit{Minimax}(v)\}$;; Recursive call at MIN node
return res ;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

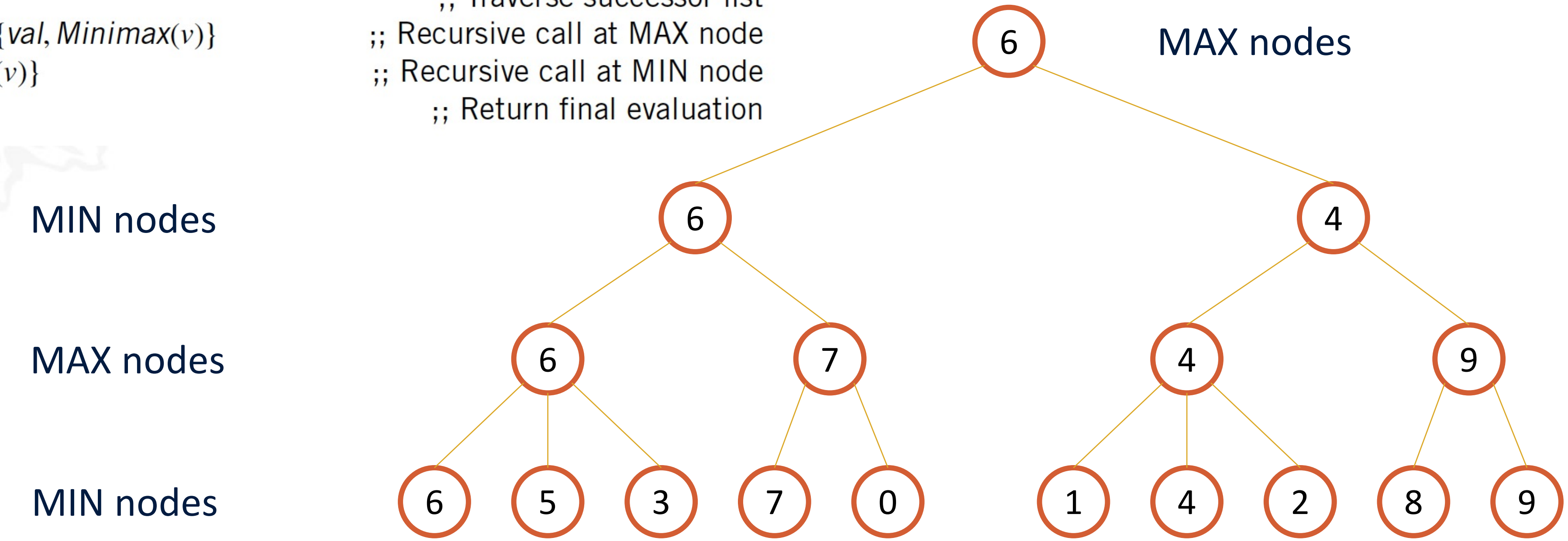
if $(leaf(u))$ **return** $Eval(u)$;; No successor, static evaluation
if $(max\text{-}node(u))$ $val \leftarrow -\infty$;; Initialize return value for MAX node
else $val \leftarrow +\infty$;; Initialize return value for MIN node
for each $v \in Succ(u)$;; Traverse successor list
  if $(max\text{-}node(u))$ $val \leftarrow \max\{val, Minimax(v)\}$;; Recursive call at MAX node
  else $val \leftarrow \min\{val, Minimax(v)\}$;; Recursive call at MIN node
return res ;; Return final evaluation



Game Tree Search

Procedure Minimax
Input: Position u
Output: Value at root

if $(\mathit{leaf}(u))$ **return** $\mathit{Eval}(u)$	;; No successor, static evaluation
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow -\infty$	;; Initialize return value for MAX node
else $\mathit{val} \leftarrow +\infty$	;; Initialize return value for MIN node
for each $v \in \mathit{Succ}(u)$	;; Traverse successor list
if $(\mathit{max\text{-}node}(u))$ $\mathit{val} \leftarrow \max\{\mathit{val}, \mathit{Minimax}(v)\}$	;; Recursive call at MAX node
else $\mathit{val} \leftarrow \min\{\mathit{val}, \mathit{Minimax}(v)\}$	;; Recursive call at MIN node
return res	;; Return final evaluation



What do you think about this result?

Game Search Tree

- The game tree cannot be fully evaluated.
  - So, we need to define a maximum depth.
- In practice, the depth that can be explored depends on a time limit.
  - The computation time is not known beforehand.
- Usually, you apply an iterative-deepening approach.
  - You increase the depth by 2 until the time available is exhausted.