Intelligent Autonomous Agents

Agents and Rational Behavior:

Adversarial Agents

Ralf Möller, Rainer Marrone
Hamburg University of Technology

Adversarial Agents

- Competitive environments
  - Agents' goals are in conflict,
  - Giving rise to adversarial search problems …
  - … often known as games.
- Mathematical game theory
  - Branch of economics, views any multiagent environment as a game, …
  - … provided that the impact of each agent on the others is significant,
  - regardless of whether the agents are cooperative or competitive.

Games

- Agents must anticipate what other agents do
- Criteria:
  - **Abstraction**: To describe a game we must capture every *relevant* aspect of the game.
  - **Accessible environments:** Such games are characterized by perfect information
  - **Search:** game-playing then consists of a search through possible game positions
  - **Unpredictable opponent:** introduces **uncertainty** thus game-playing must deal with **contingency problems**

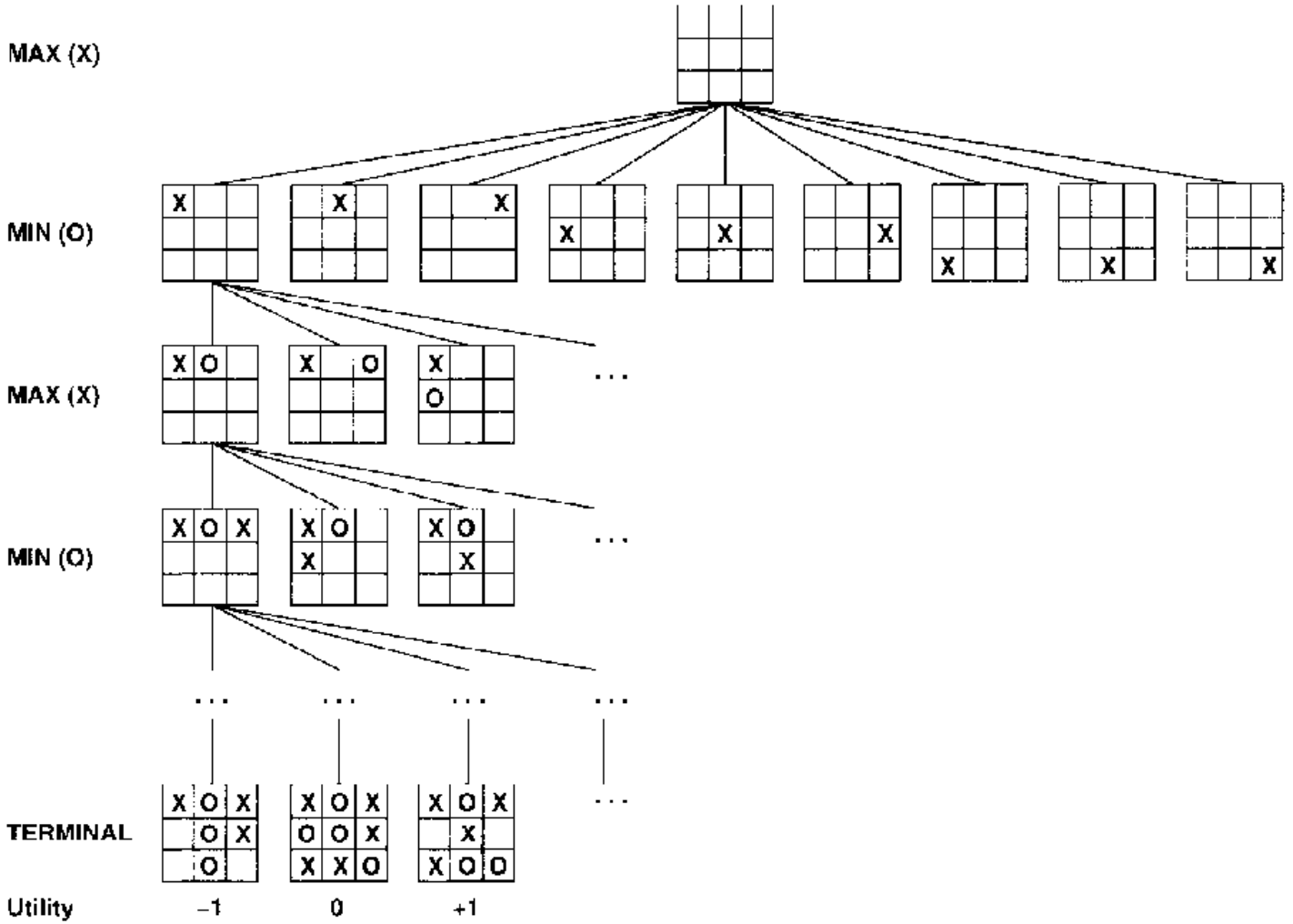
Two-player games

- A game formulated as a search problem:
  - Initial state: ?
  - Operators: ?
  - Terminal state: ?
  - Utility function: ?

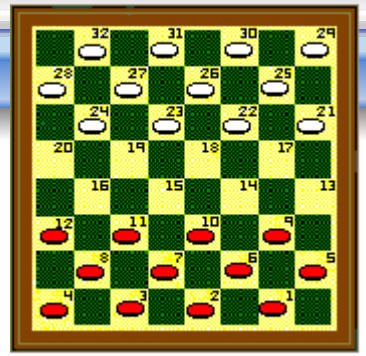
Two-player games

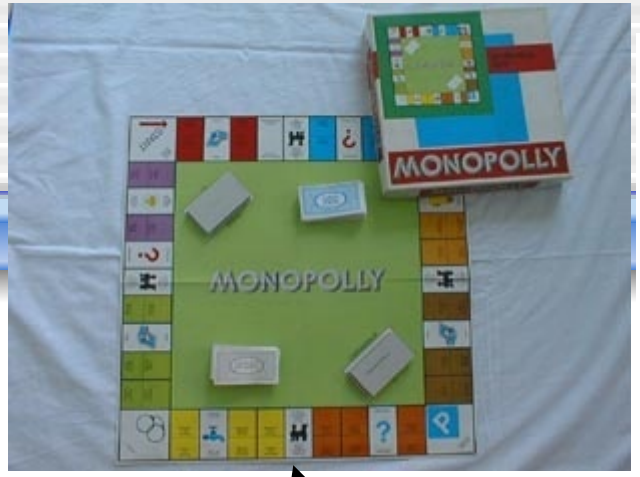
- A game formulated as a search problem:
  - Initial state: board position and turn
  - Operators: definition of legal moves
  - Terminal state: conditions for when game is over
  - Utility function:
    a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win (AKA **payoff function**)

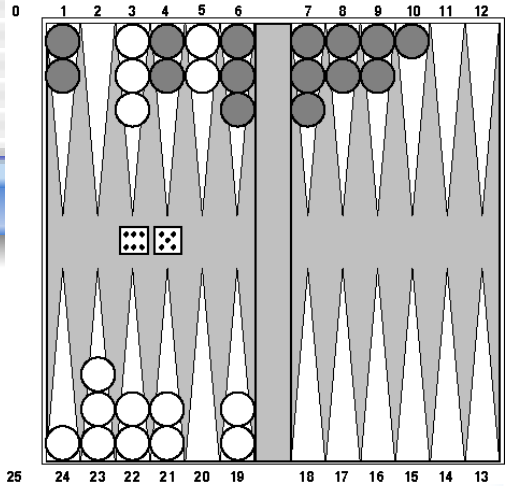
Example: Tic-Tac-Toe



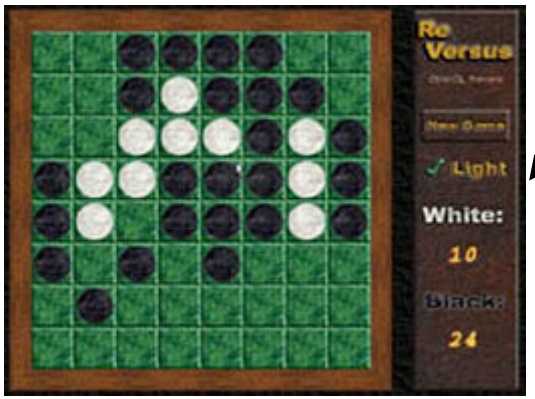
Types of Games

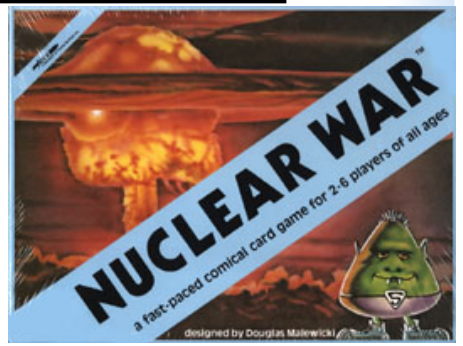






	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war



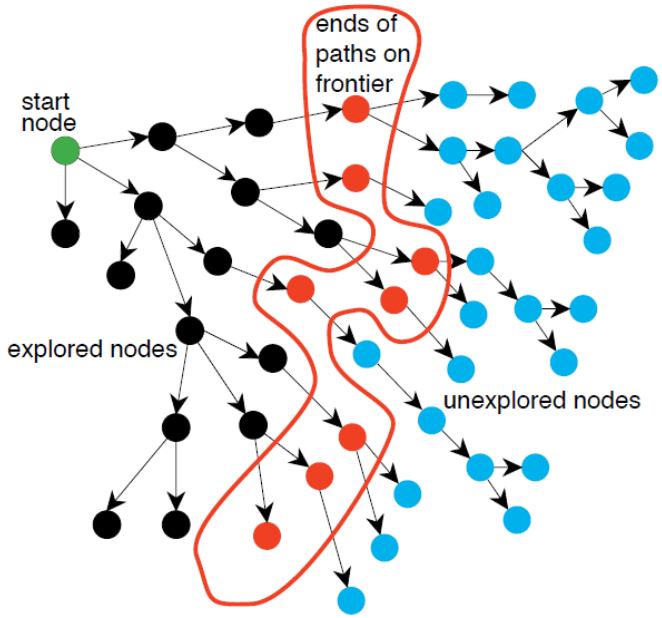


Searching

- For small games, search the game tree for an optimal solution
  - Depth first search
  - Breadth first search
  - Informed/Uniformed search

Searching

- Standard search methods:
  - Depth first
  - Breadth first
- Alternative approach selective expansion of search space



Heuristic search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- $h(n)$ uses only readily obtainable information (that is easy to compute) about a node.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from n to a goal that has path length less than $h(n)$.

Example Heuristic Functions

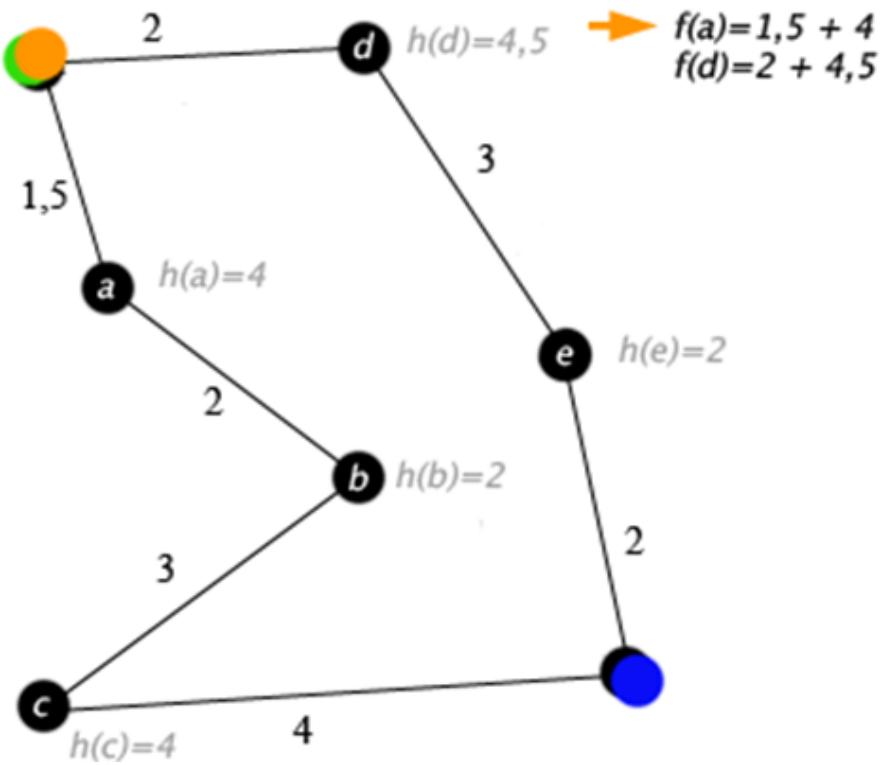
- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the straight-line distance from n to the closest goal as the value of $h(n)$.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.

Example A* Search

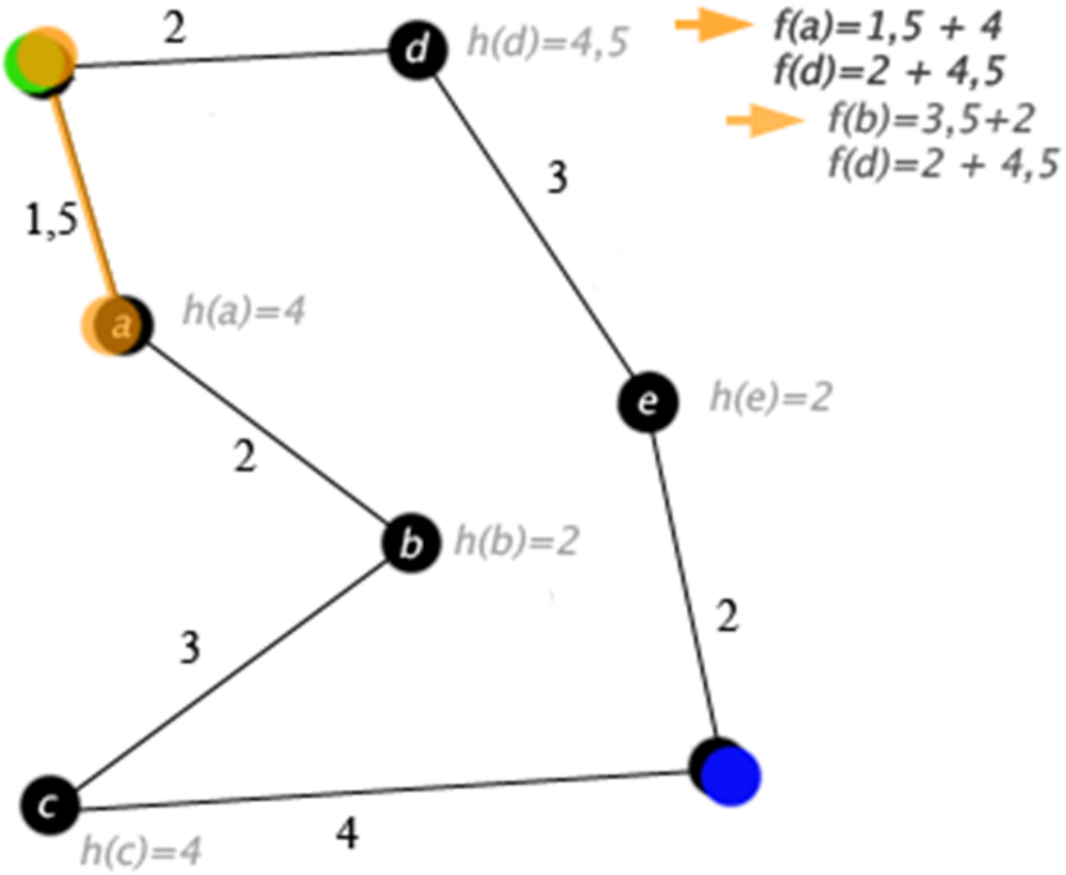
- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$. $f(p)$ estimates the total path cost of going from a start node to a goal via p.

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{cost(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

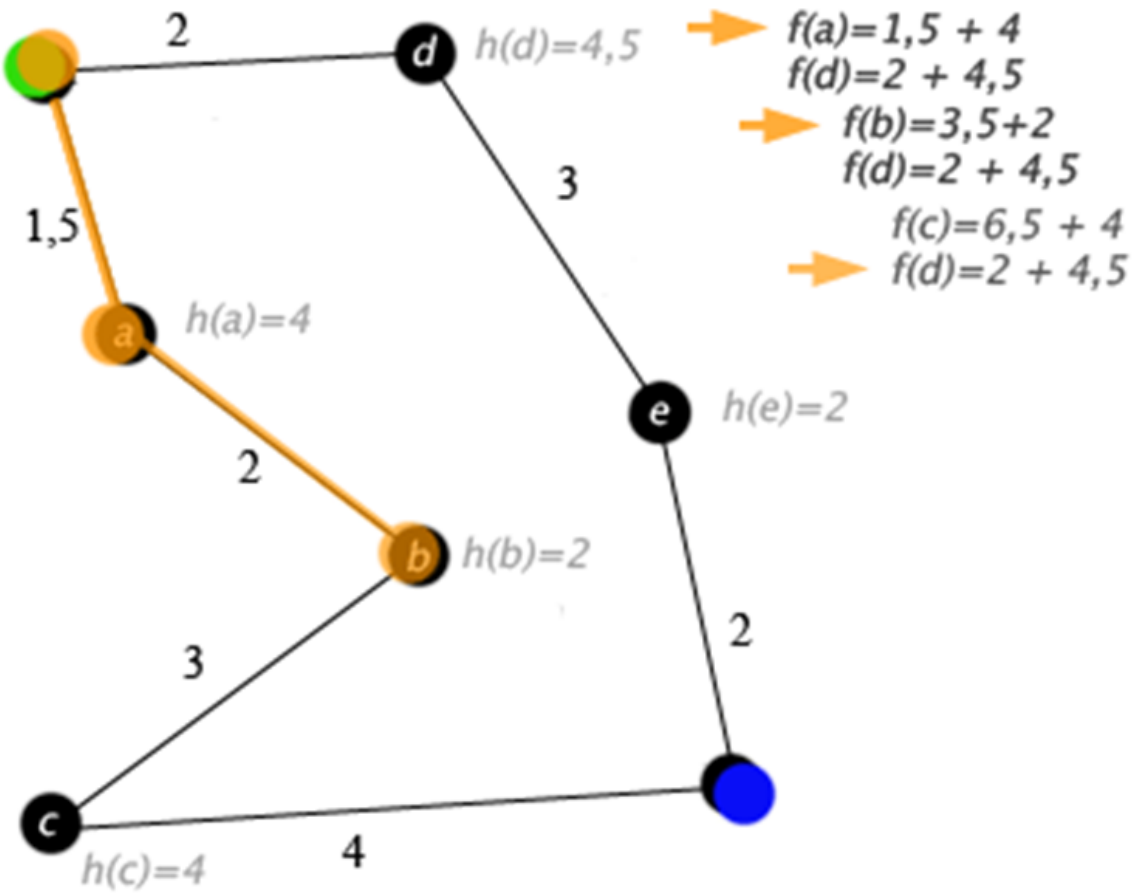
Path finding example – A*



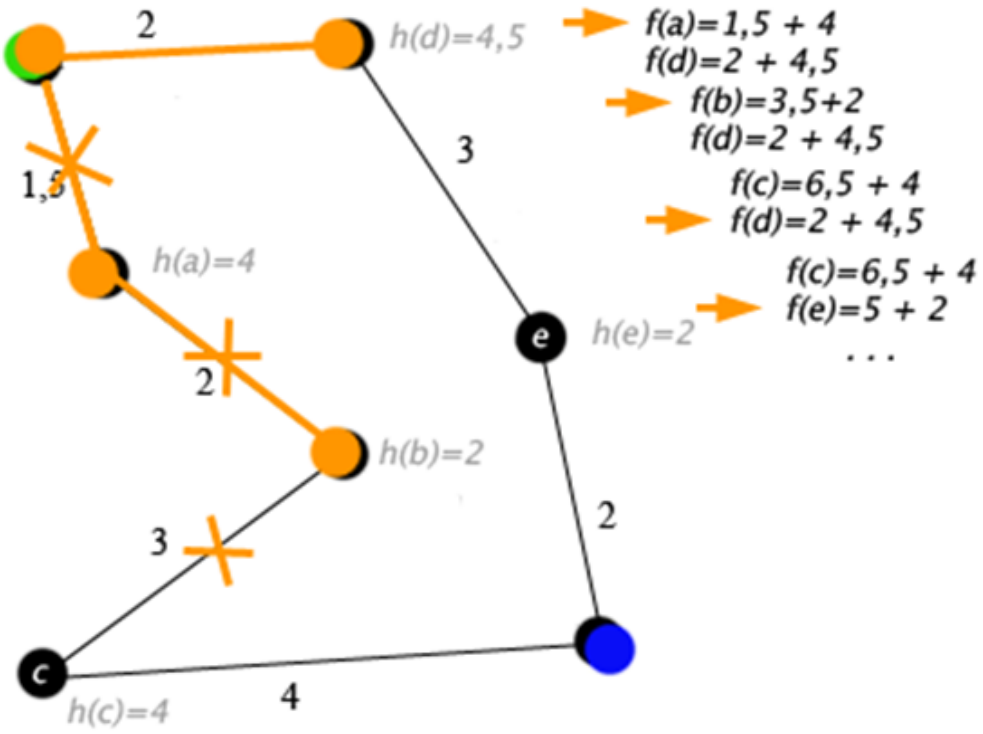
A* example



A* example


2
d
h(d)=4,5
f(a)=1,5 + 4
f(d)=2 + 4,5
f(b)=3,5+2
f(d)=2 + 4,5
f(c)=6,5 + 4
f(d)=2 + 4,5
3
1,5
a
h(a)=4
e
h(e)=2
2
b
h(b)=2
2
3
c
h(c)=4
4

A* example


2
h(d)=4,5
f(a)=1,5 + 4
f(d)=2 + 4,5
f(b)=3,5+2
f(d)=2 + 4,5
3
1,5
f(c)=6,5 + 4
f(d)=2 + 4,5
h(a)=4
f(c)=6,5 + 4
f(e)=5 + 2
. . .
e
h(e)=2
2
h(b)=2
2
3
c
4
h(c)=4

A* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(p)$.
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

A*

- A* heuristics reduces the number of branches to be evaluated in typical cases
- A* always finds the optimal solution.

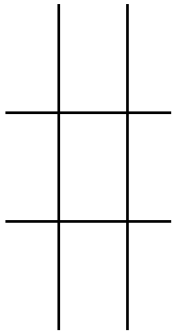
- NOW we are interested in searching in the context of two-player, zero-sum games.

The minimax algorithm

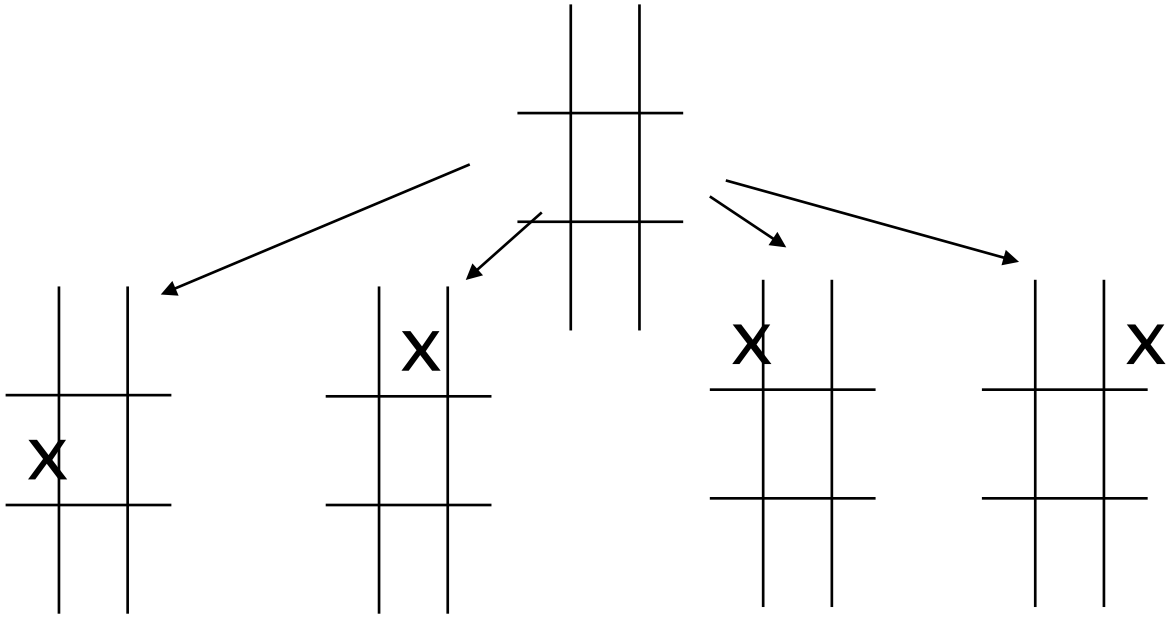
- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff **against best play**
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

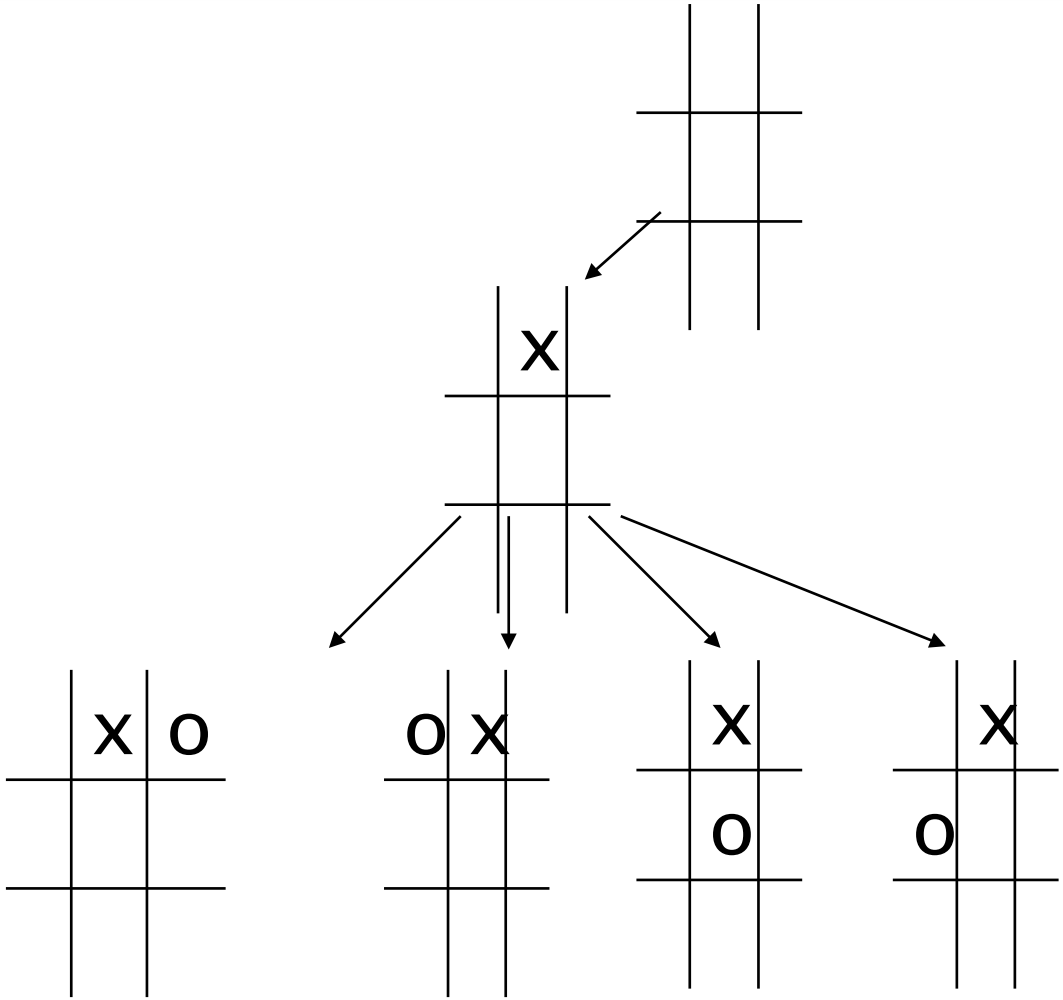
Generate Game Tree



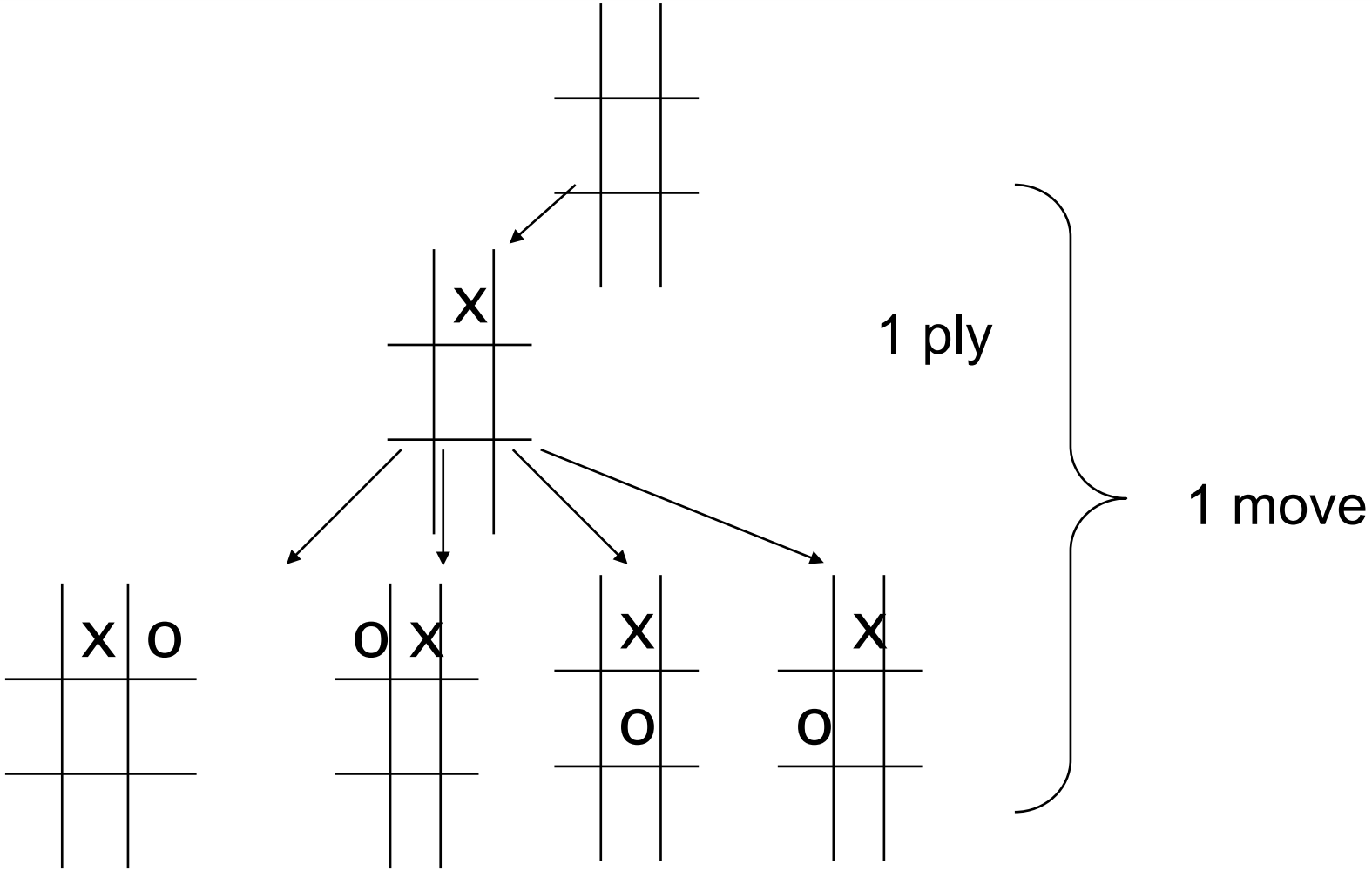
Generate Game Tree



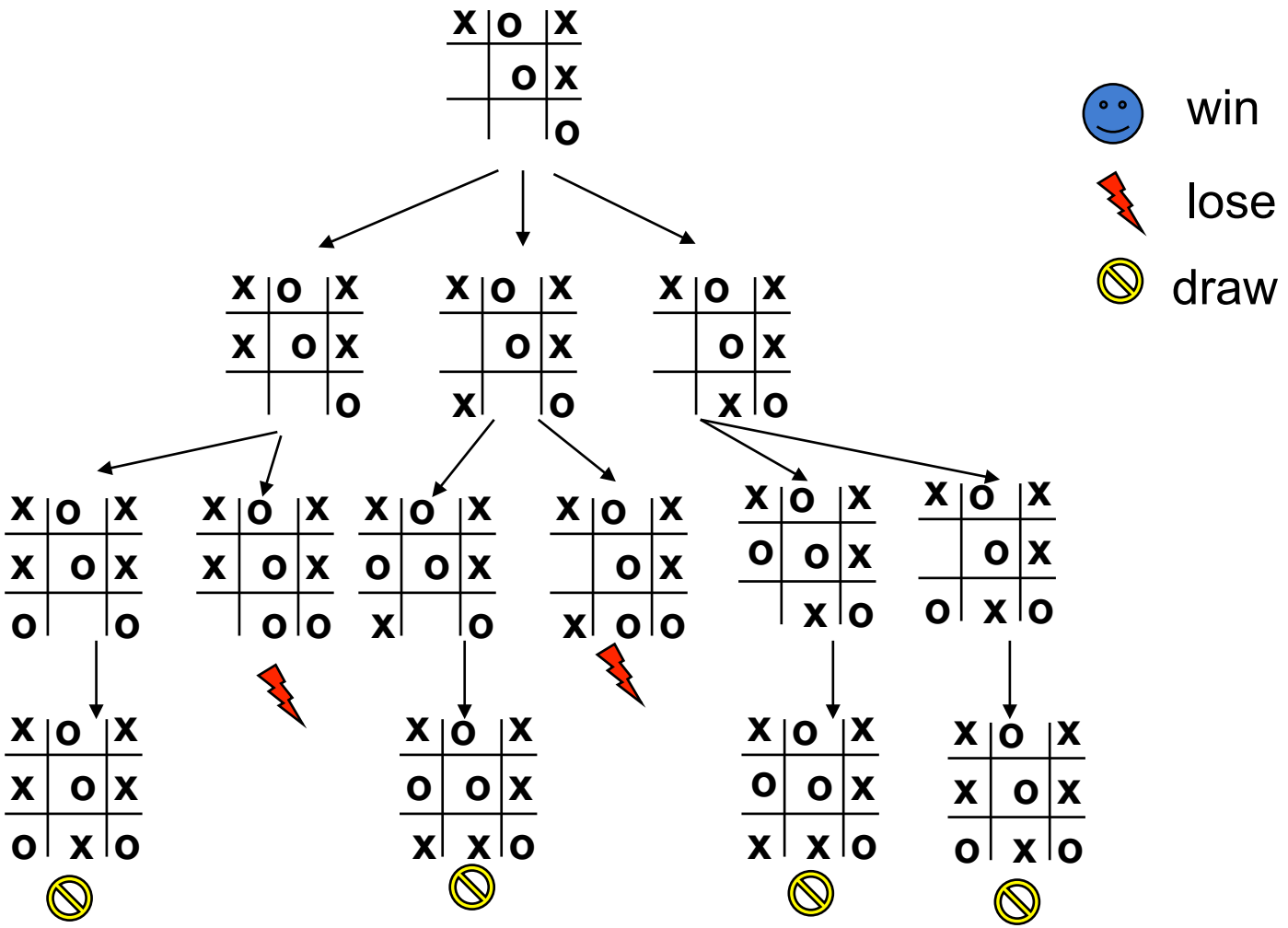
Generate Game Tree



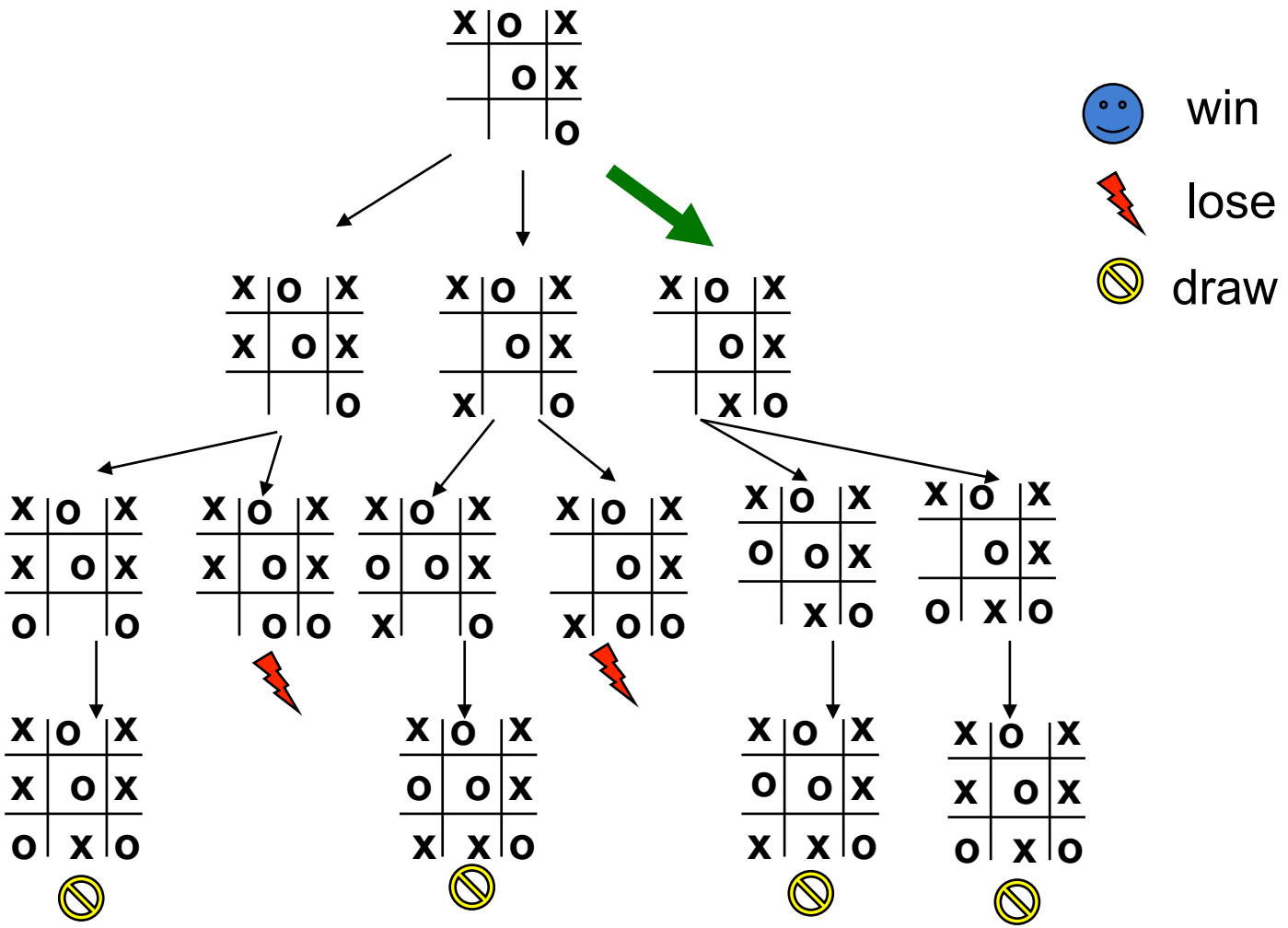
Generate Game Tree



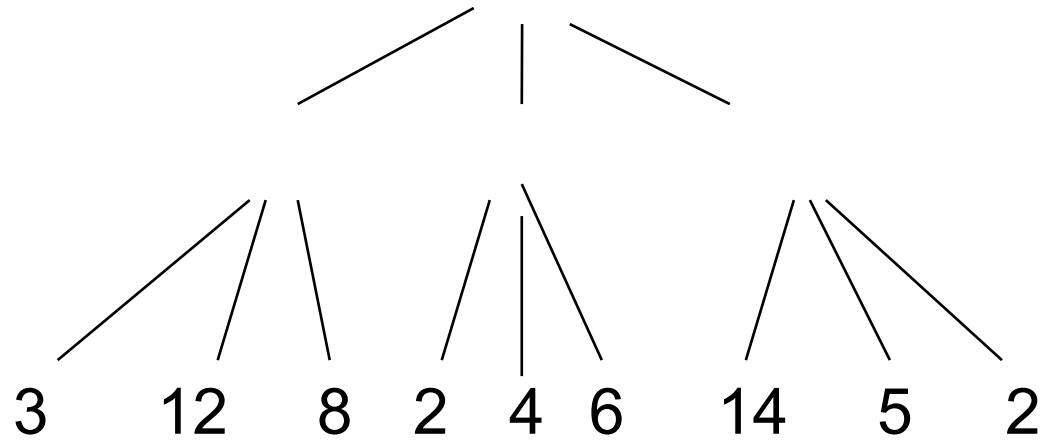
A subtree



What is a good move?

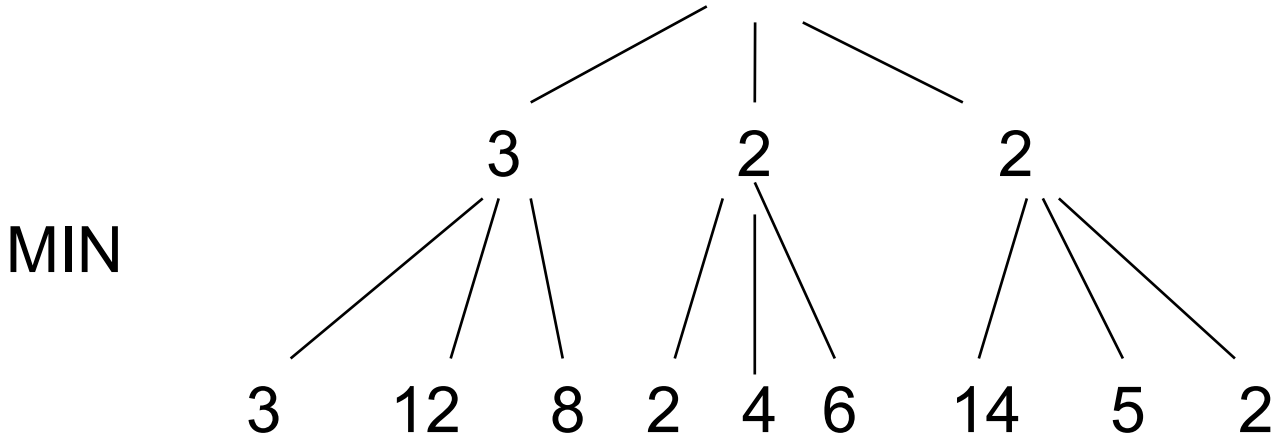


Minimax



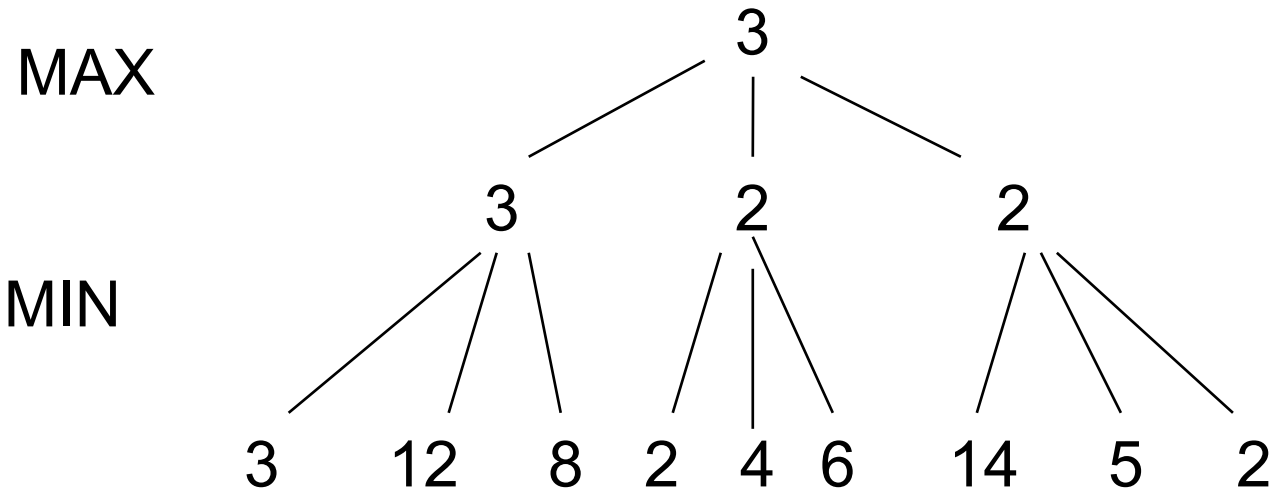
- Minimize opponent's chance
- Maximize your chance

Minimax



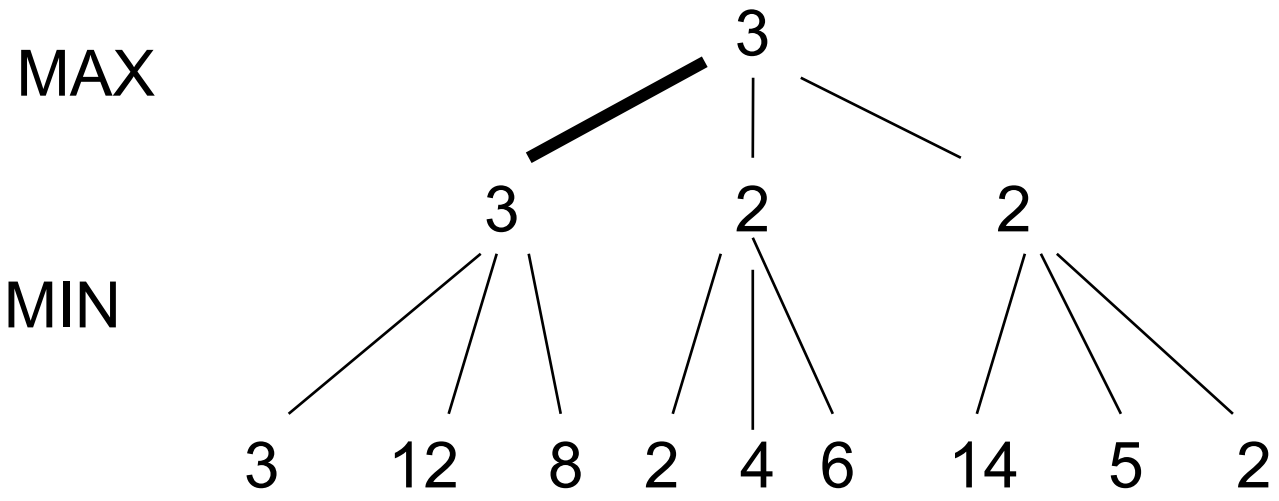
- Minimize opponent's chance
- Maximize your chance

Minimax



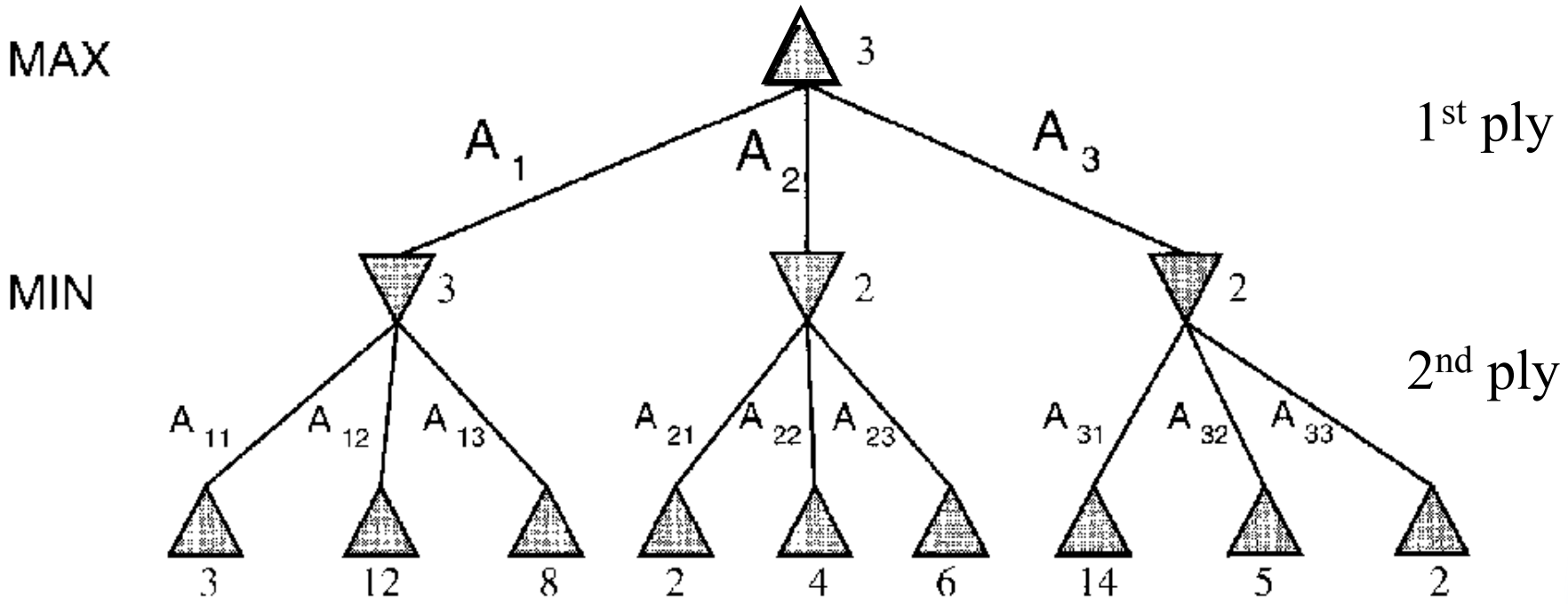
- Minimize opponent's chance
- Maximize your chance

Minimax



- Minimize opponent's chance
- Maximize your chance

minimax = maximum of the minimum



Minimax: Recursive implementation

```
function MINIMAX-DECISION(game) returns an operator
    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]

---

function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Complete: ?
Optimal: ?

Time complexity: ?
Space complexity: ?

Minimax: Recursive implementation

```
function MINIMAX-DECISION(game) returns an operator
    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Complete: Yes, for finite state-space
Optimal: Yes

Time complexity: $O(b^m)$
Space complexity: O(bm) (= DFS Does not keep all nodes in memory.)

Game vs. search problem

"Unpredictable" opponent $\Rightarrow$ solution is a contingency plan

Time limits $\Rightarrow$ unlikely to find goal, must approximate

Plan of attack:

- algorithm for perfect play (Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
- pruning to reduce costs (McCarthy, 1956)

Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** $b = 35, m=100 \Rightarrow nodes = 35^{100}$
    if each node takes about 1 ns to explore
    then each move will take about ***10^{50} millennia***
    to calculate.
- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate

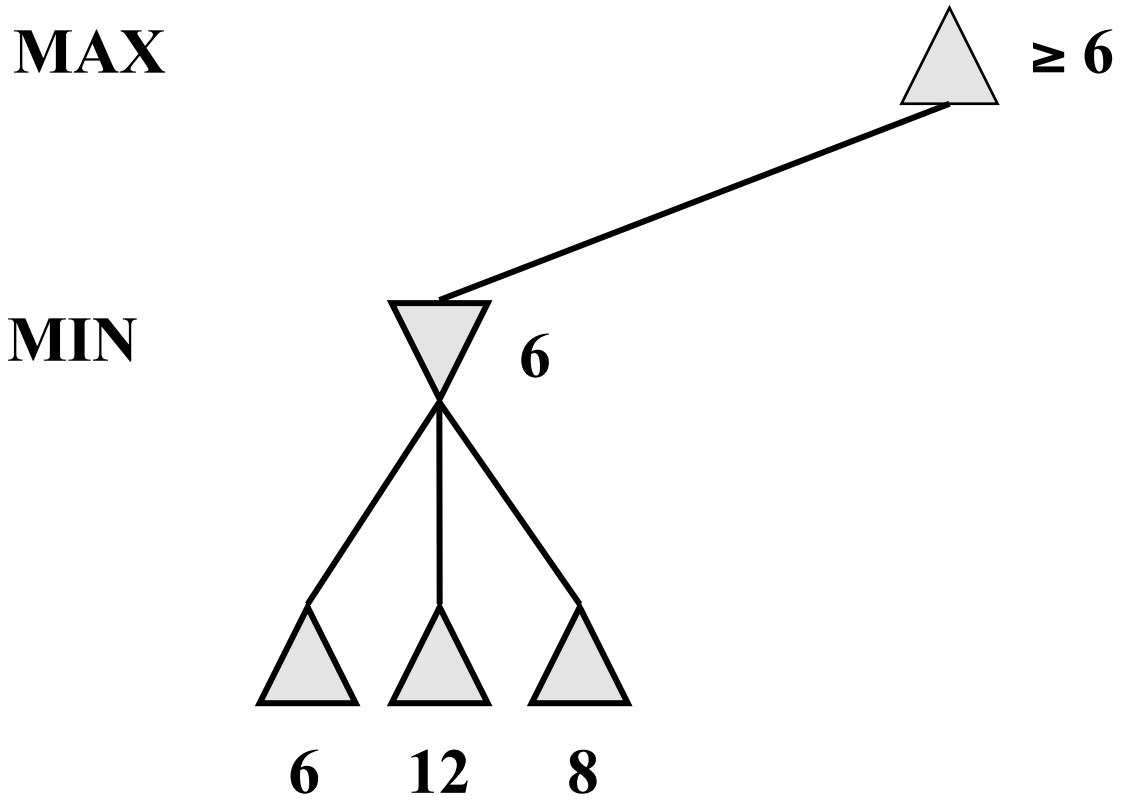
1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result.
2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.

1. α–β pruning

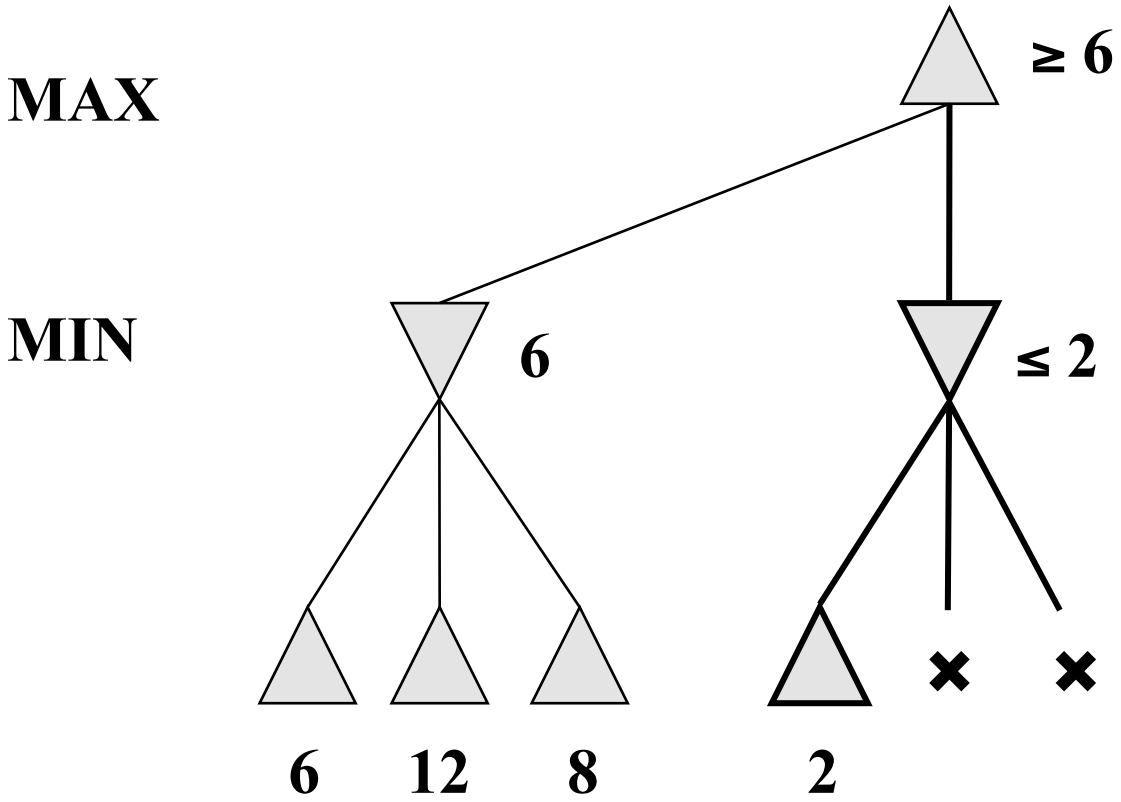
- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node (e.g. A*).
- **α–β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the *max* or *min* node, by just considering the values of nodes seen so far.

- Does it work? Yes, it roughly cuts the branching factor from b to $\sqrt{b}$ resulting in double as far look-ahead than pure minimax.

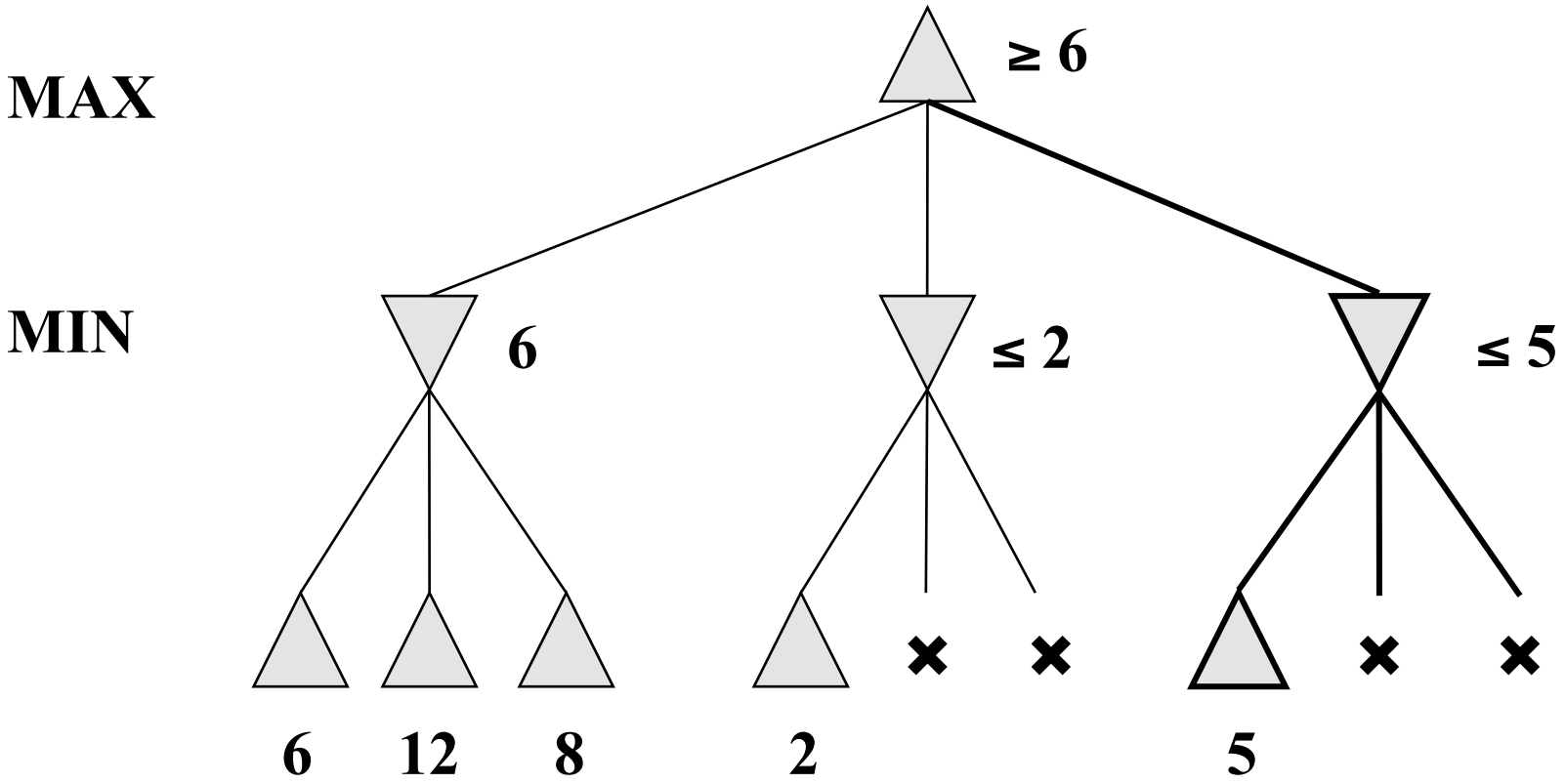
α–β pruning: example



α–β pruning: example

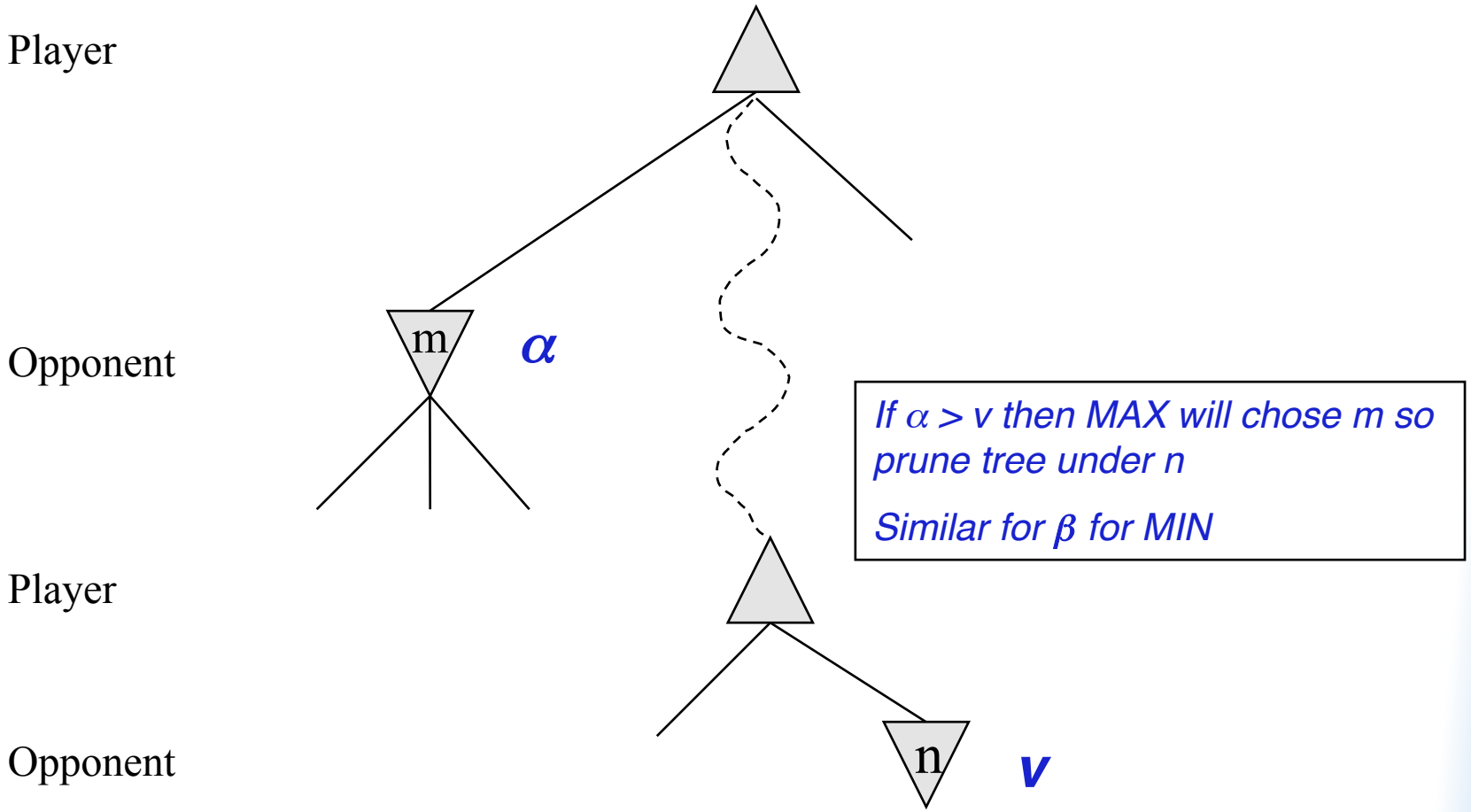


α–β pruning: example



MINIMAX(root) = max(min(6,12,8),min(2,a,b),min(5,b,d))
= max(6,z,y) where z=min(2,a,b)≤ 2 and y=min(5,b,d) ≤ 5
= 6

α–β pruning: general principle

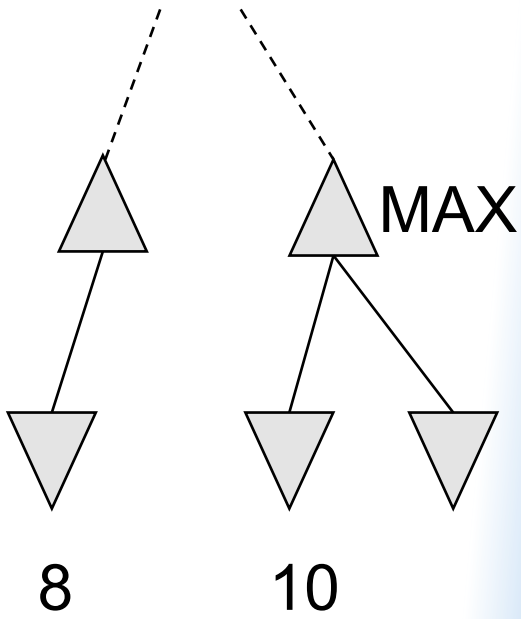


More on the α–β algorithm

- Because minimax is depth-first, let's consider nodes along a given path in the tree. Then, as we go along this path, we keep track of:
  - α : Best choice so far for MAX
  - β : Best choice so far for MIN

The α–β algorithm:

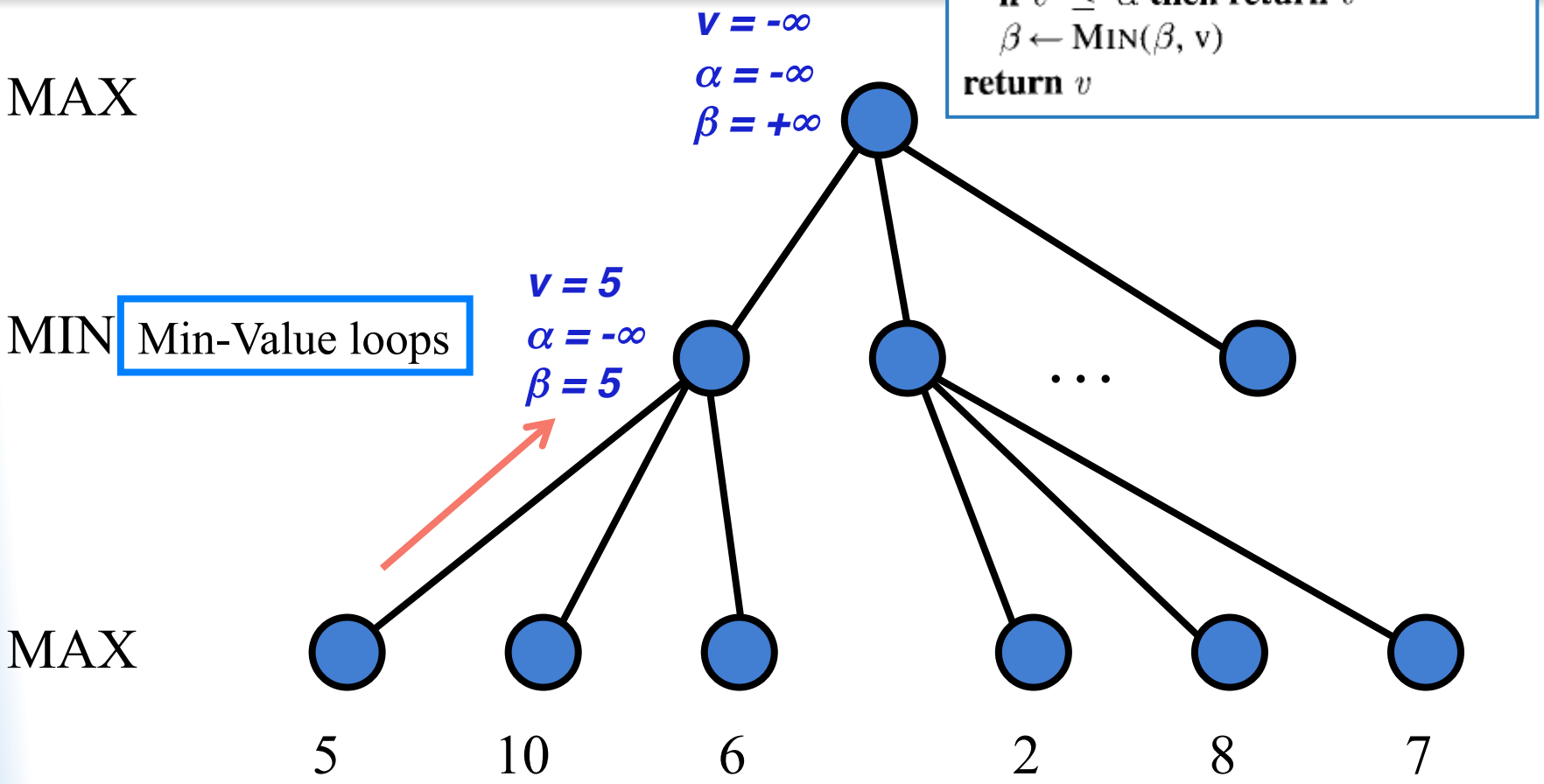
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```



More on the α–β algorithm

In Min-Value:

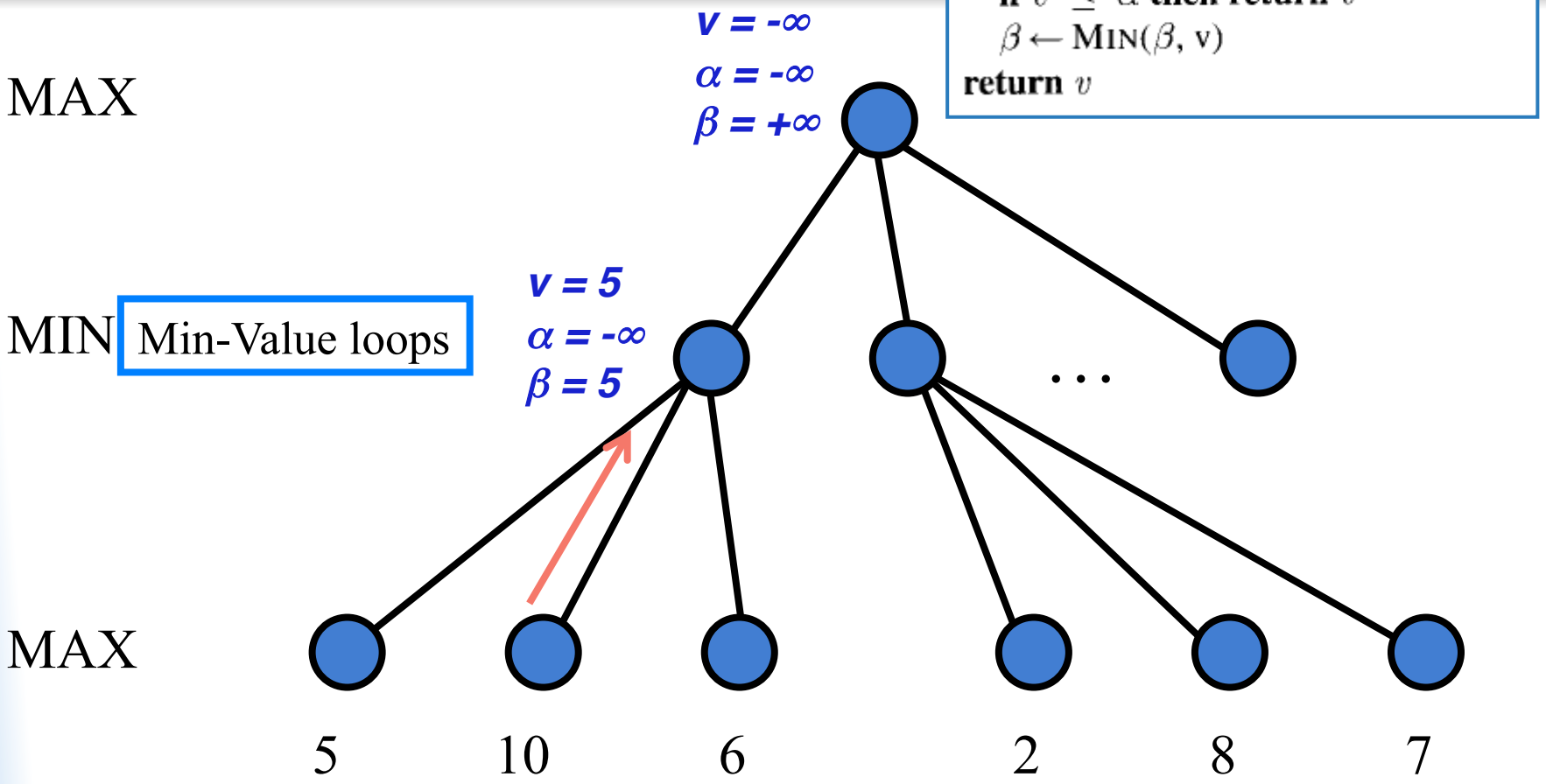
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Min-Value:

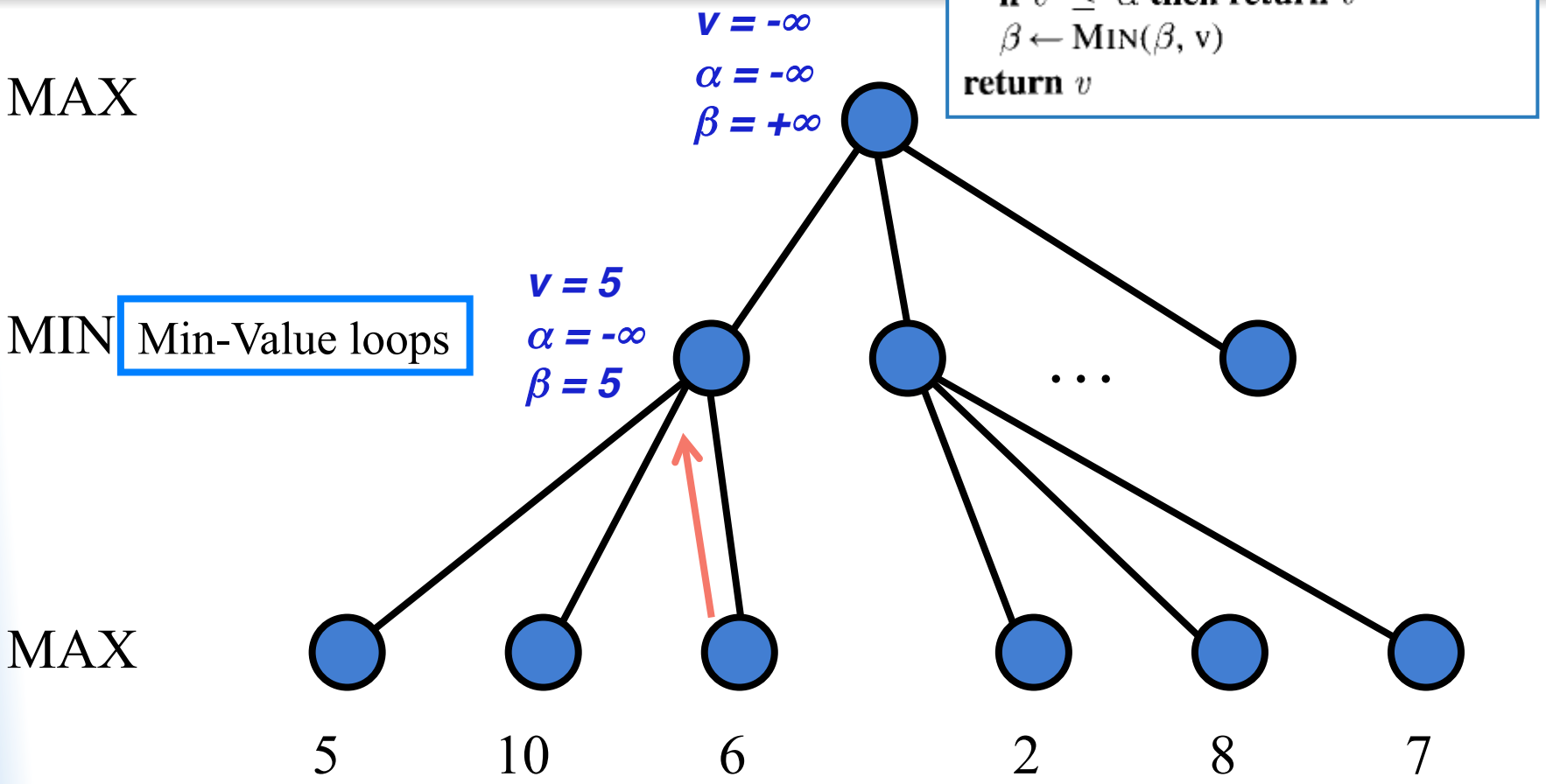
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Min-Value:

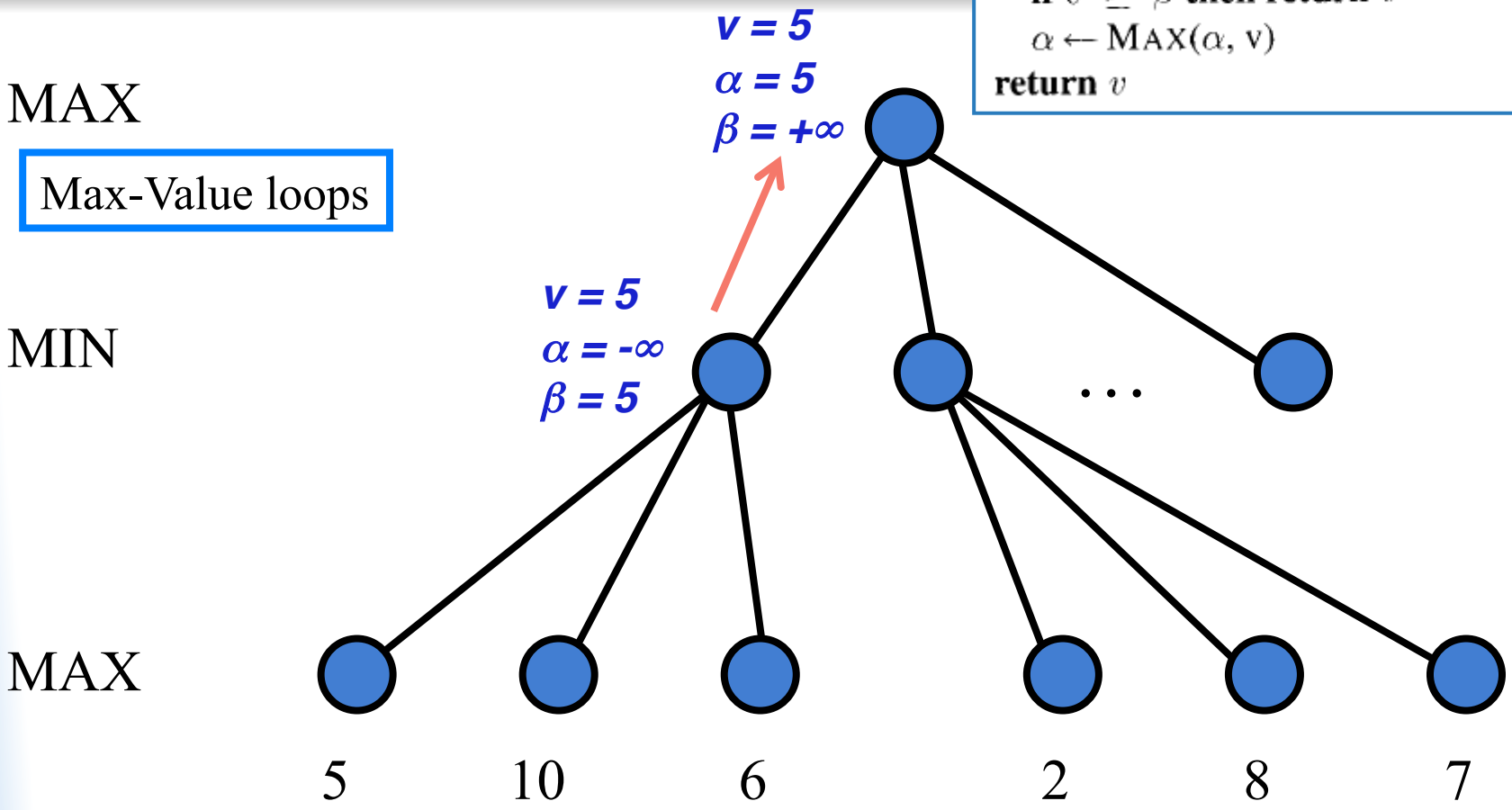
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Max-Value:

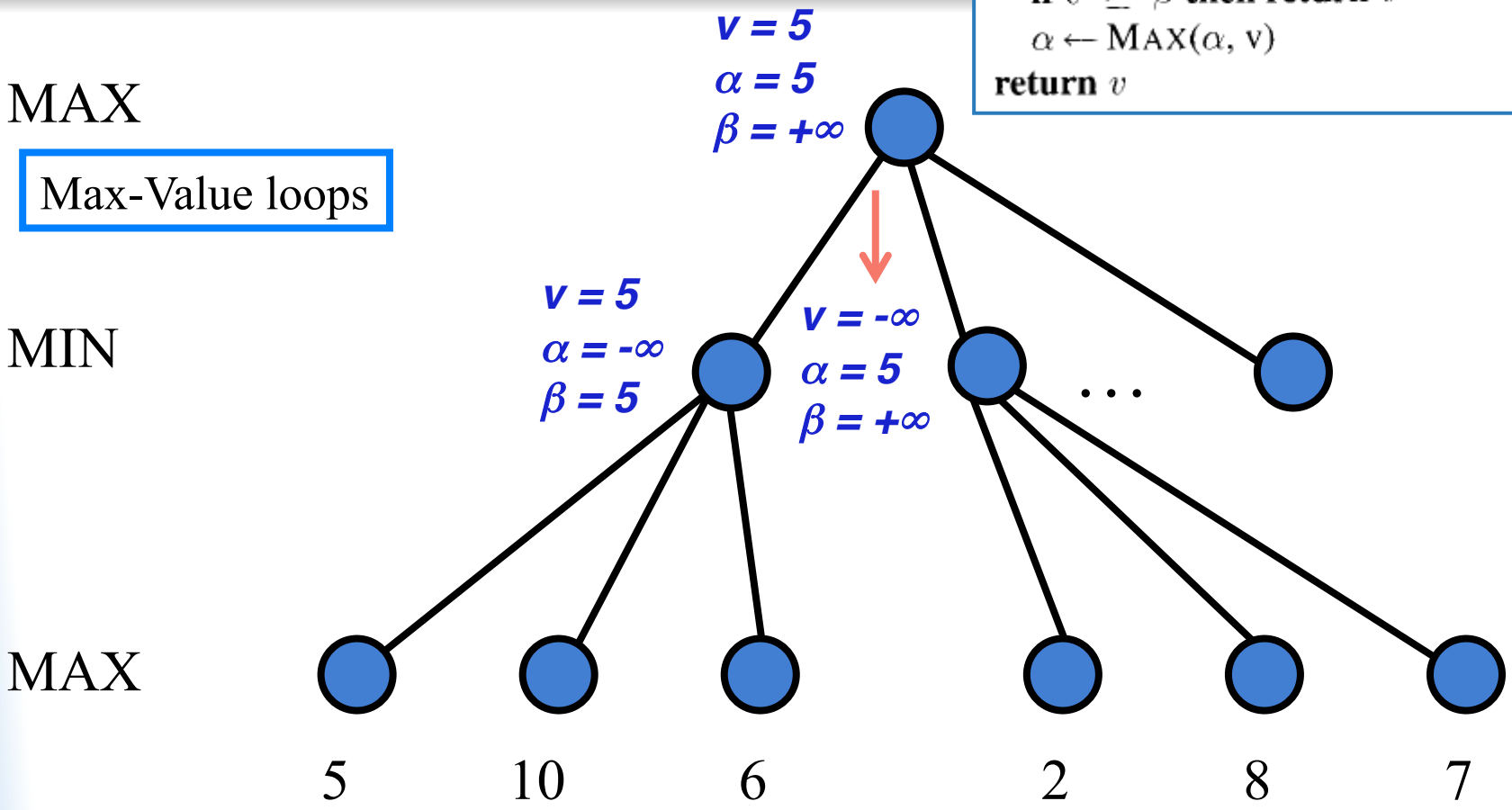
```
v ← −∞
for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
return v
```



More on the α–β algorithm

In Max-Value:

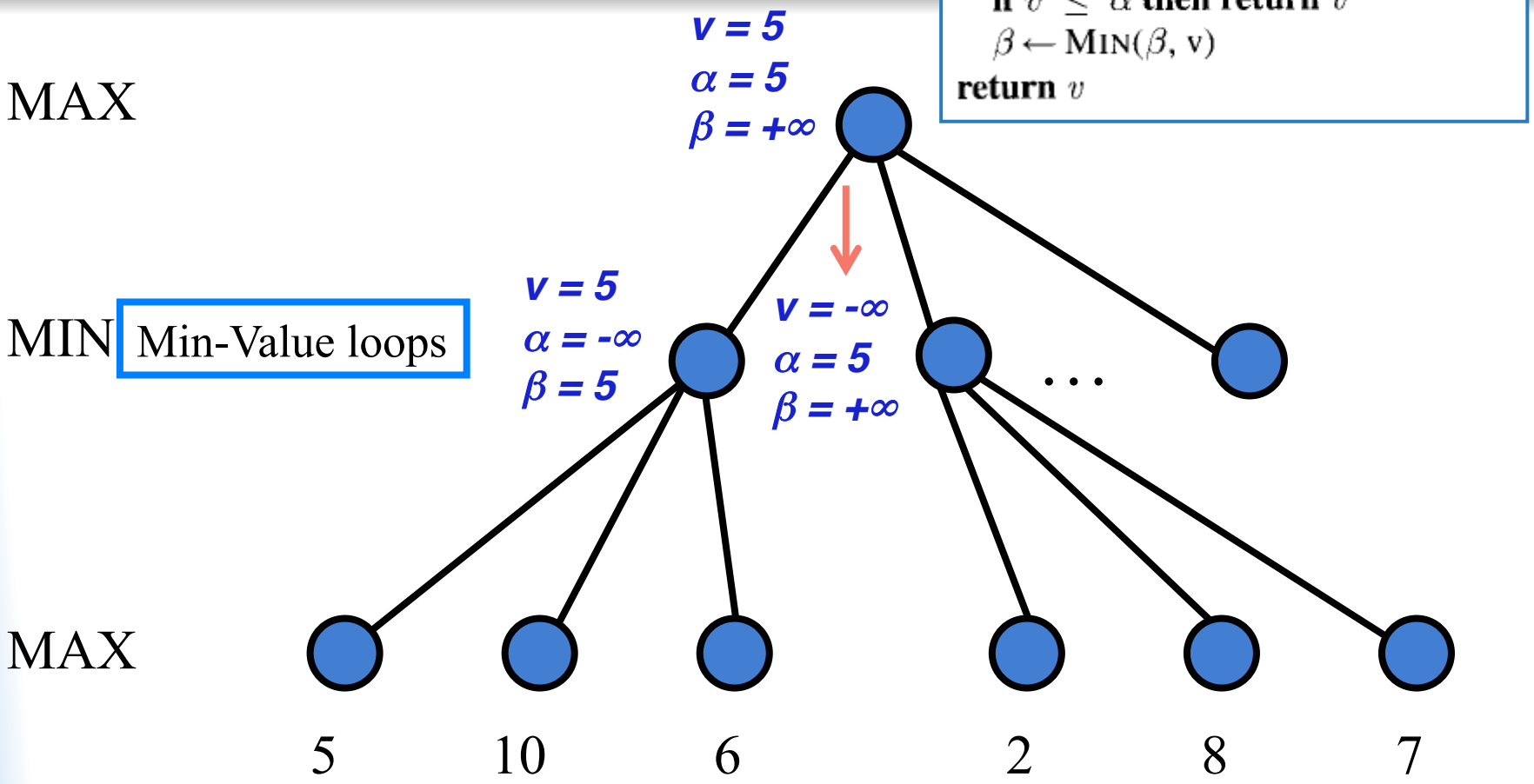
```
v ← −∞
for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
return v
```



More on the α–β algorithm

In Min-Value:

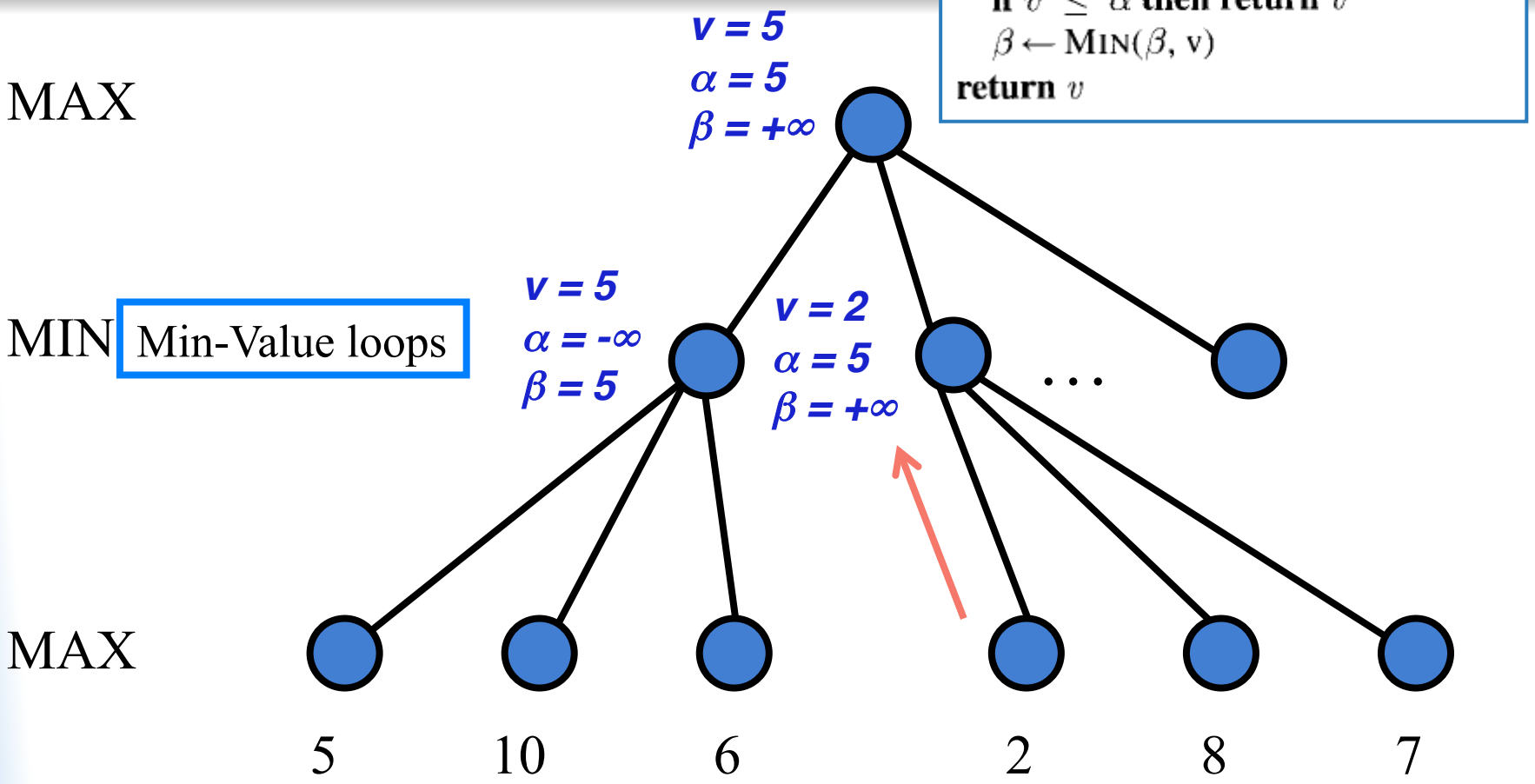
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Min-Value:

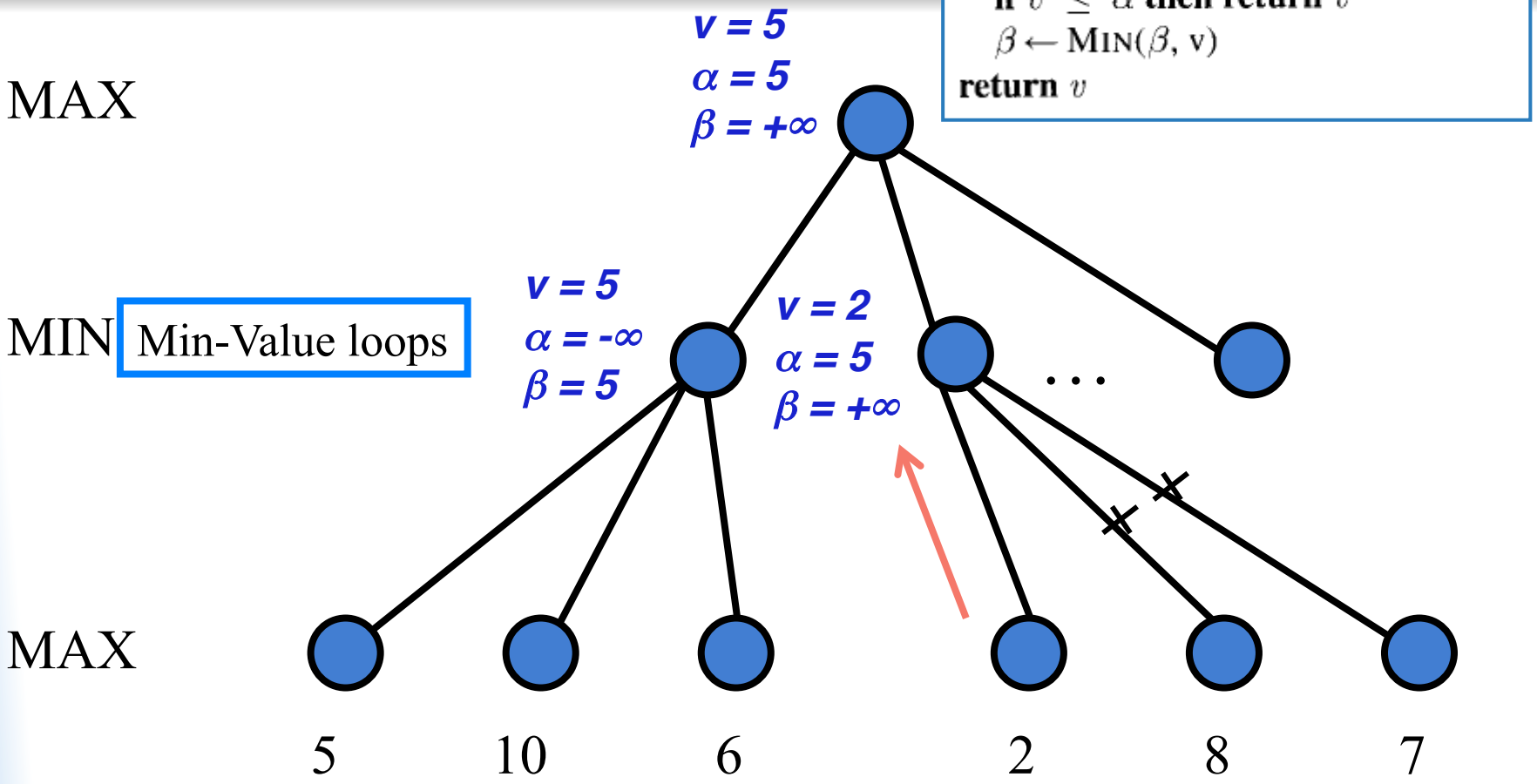
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Min-Value:

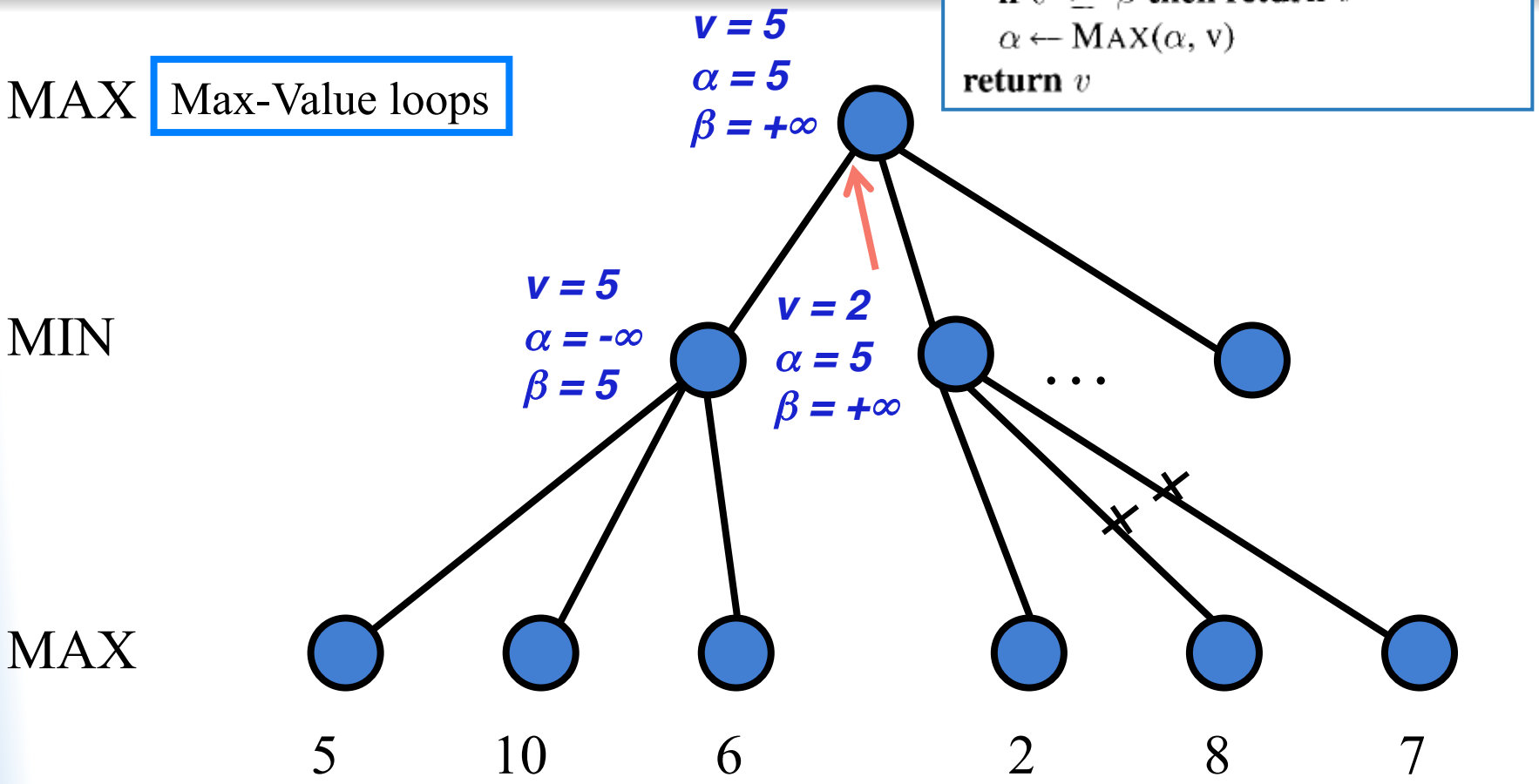
```
v ← +∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
return v
```



More on the α–β algorithm

In Max-Value:

```
v ← −∞
for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
return v
```



Properties of α–β

- Pruning does not affect the final result!!!
- Good move ordering improves effectiveness of pruning
- With *perfect ordering*, time complexity = $O(b^{m/2})$
  - doubles depth of search
  - need a heuristic how to order
  - can easily reach depth 8 => good chess
- A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)

2. Move evaluation without complete search

- The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it.
- Complete search is often too complex and impractical

- **Evaluation function:** evaluates value of state using heuristics and cuts off search

- **New MINIMAX:**
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation function

- The evaluation function should order the *terminal* states in the same way as the true utility function (a<b<c…).
- The computation must not take too long! Significant compared to Minimax?
- For nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Evaluation functions

- Most calculate features – e.g., number of pawns
- From that we can form categories, equivalence classes.
- Any category represent states that win, lose or result in draws.
- If we know 72% lead to win (+1), 20% to loss (-1), 8% drawn (0).
  Expected value:
- (0,72* +1) + (0,20* -1) + (0,08 * 0)= 0,52

Evaluation functions

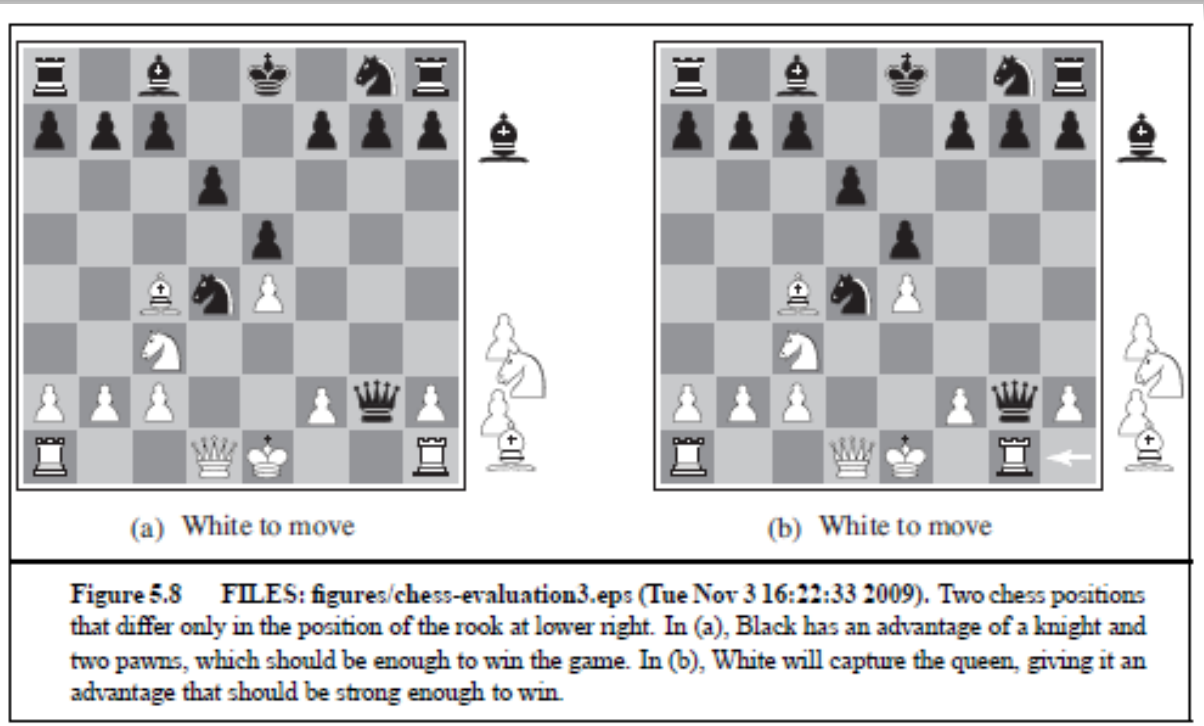


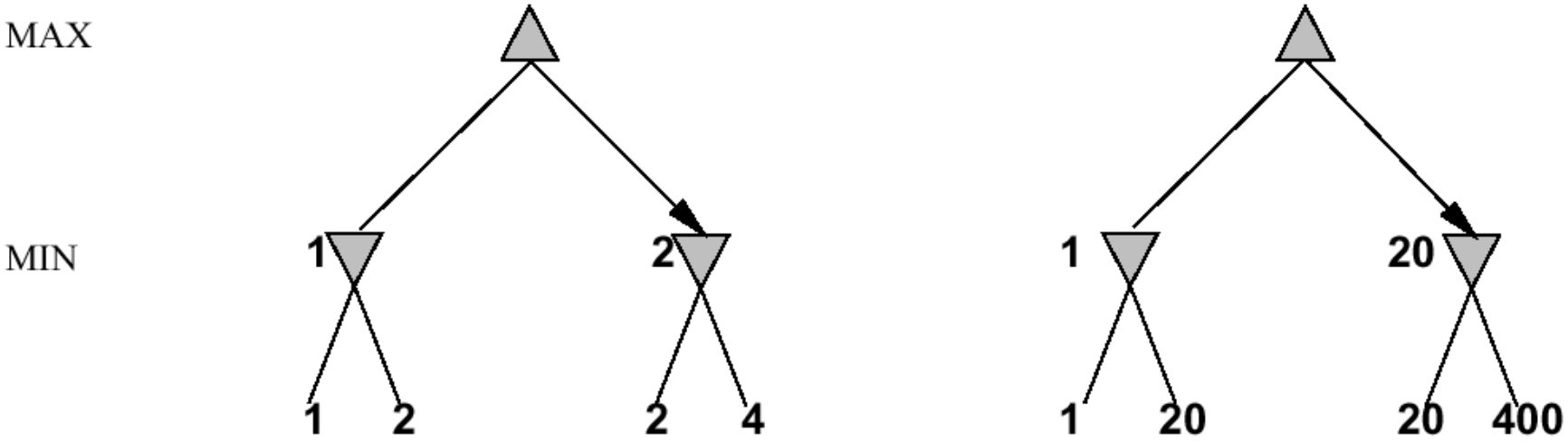
Figure 5.8 FILES: figures/chess-evaluation3.eps (Tue Nov 3 16:22:33 2009). Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

- **Weighted linear evaluation function:** to combine *n* heuristics

$$f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$$

E.g, *w's* could be the values of pieces (1 for prawn, 3 for bishop etc.)
f's could be the number of type of pieces on the board

Note: exact values do not matter



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:

payoff in deterministic games acts as an *ordinal utility* function

With cutoff and eval

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: $state$, current state in game
    α, the value of the best alternative for MAX along the path to $state$
    β, the value of the best alternative for MIN along the path to $state$

  if CUTOFF-TEST($state$, $depth$) **then return** EVAL($state$)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

Minimax with cutoff: viable algorithm?

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply ≈ human novice
8-ply ≈ typical PC, human master
12-ply ≈ Deep Blue, Kasparov

Assume we have 100 seconds, evaluate 10^4 nodes/s; can evaluate 10^6 nodes/move

Other Results

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary 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.

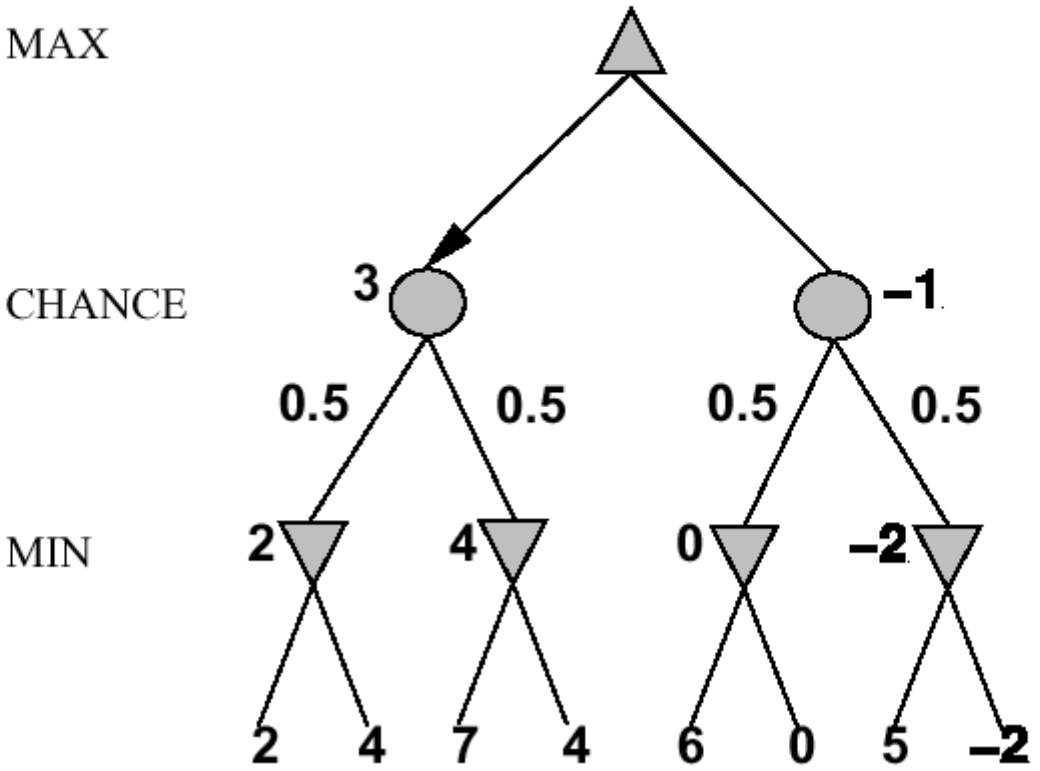
Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Nondeterministic games

E..g, in backgammon, the dice rolls determine the legal moves

Simplified example with coin-flipping instead of dice-rolling:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

```
...
if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
...
```

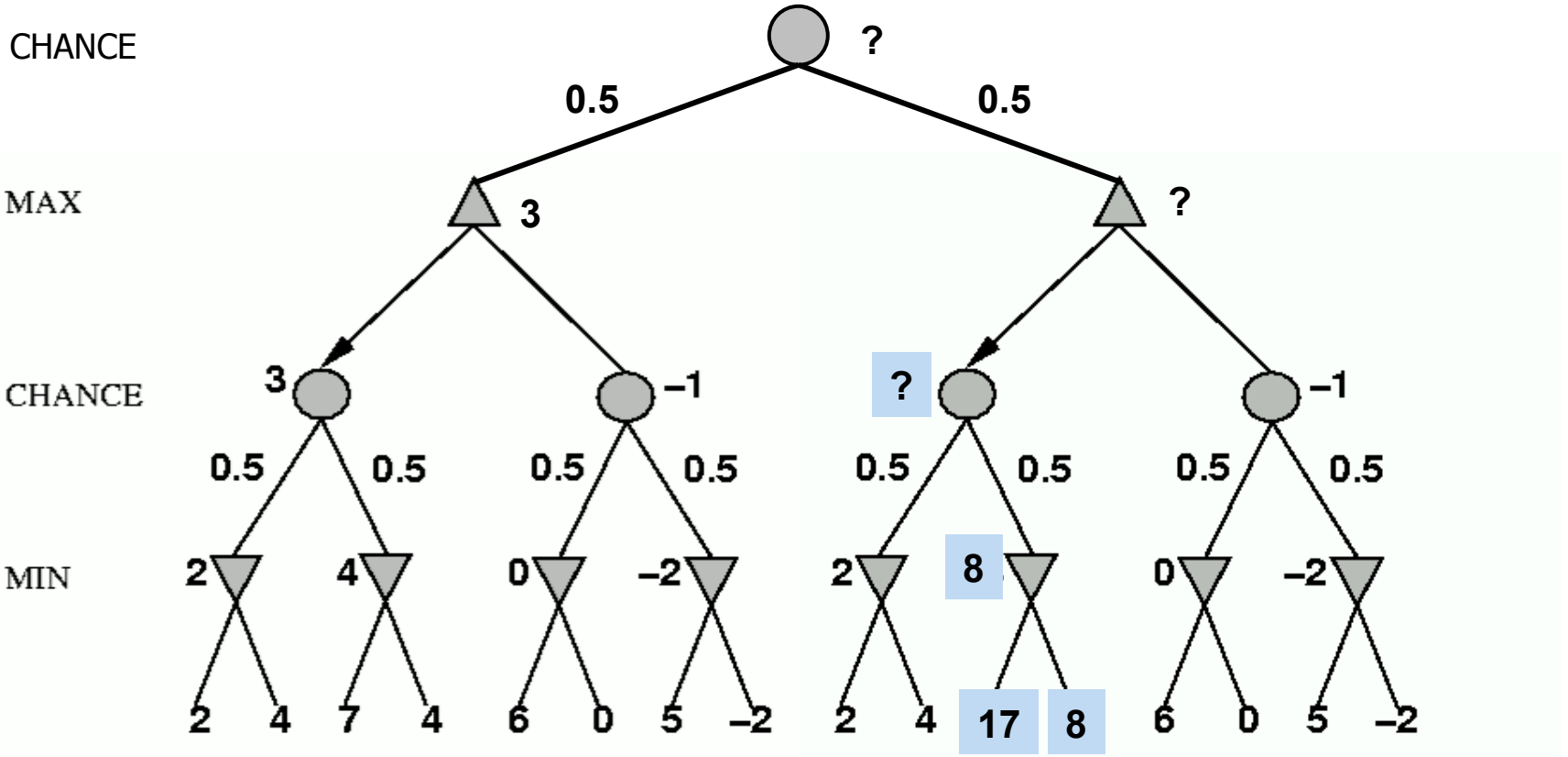
A version of α–β pruning is possible
but only if the leaf values are bounded. Why??

Remember: Minimax algorithm

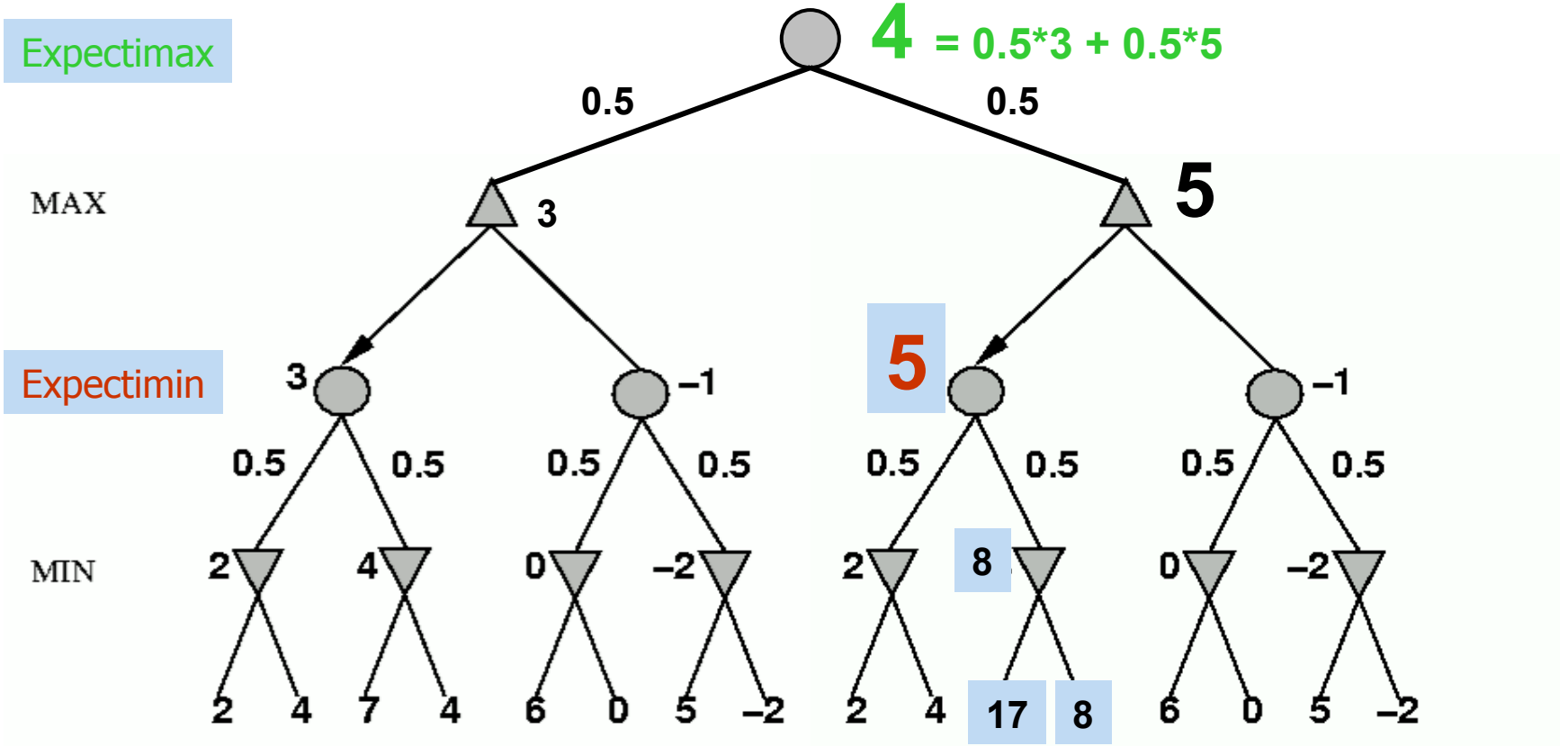
```
function MINIMAX-DECISION(game) returns an operator
    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]

function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
```

Nondeterministic games: the element of chance

expectimax and **expectimin**, expected values over all possible outcomes

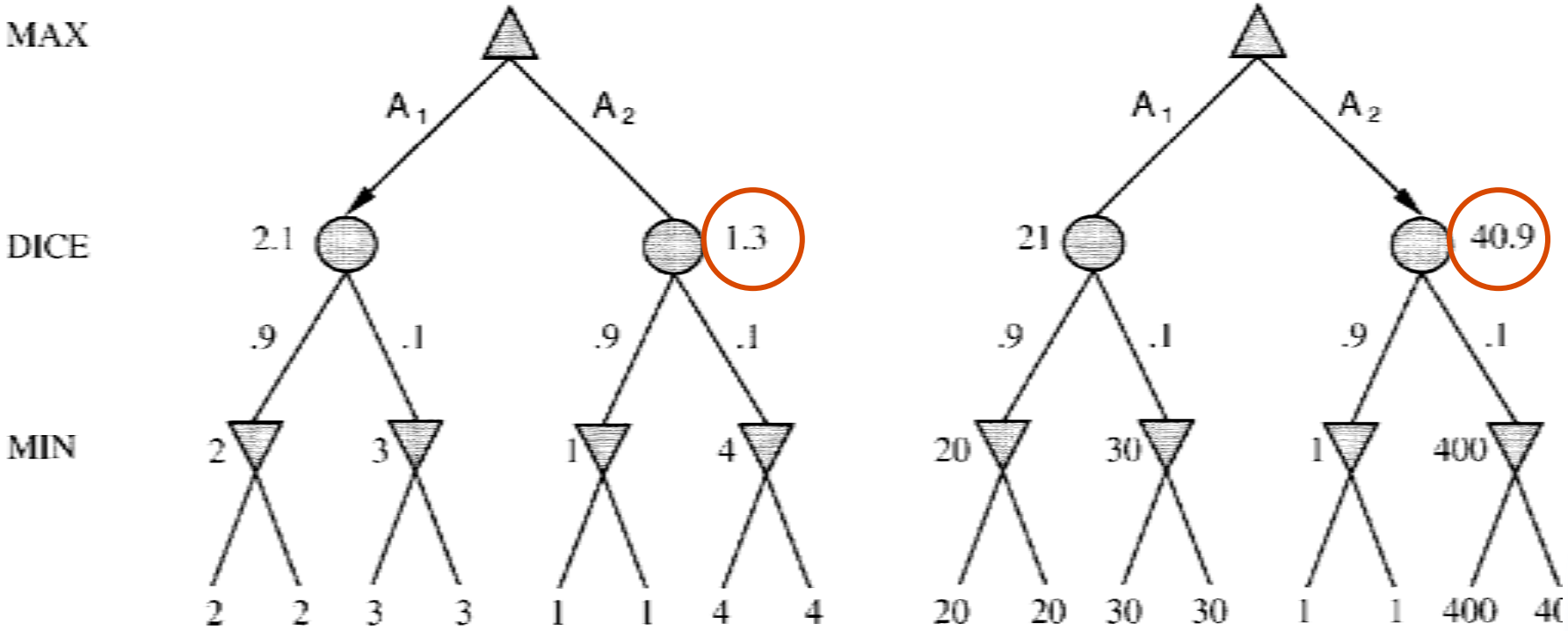


Nondeterministic games: the element of chance



Evaluation functions: Exact values DO matter

Order-preserving transformations do not necessarily behave the same!



Games of imperfect information

- E.g., card games, where opponent's initial cards are unknown
- Typically we can calculate a probability for each possible deal
- Seems just like having one big dice roll at the beginning of the game
- Idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals
- Special case: if an action is optimal for all deals, it's optimal.

- GIB, current best bridge program, approximates this idea by
  - generating 100 deals consistent with bidding information
  - picking the action that wins most tricks on average

Summary

- Games are fun to work on!
- They illustrate several important points about agent interation
  - perfection is unattainable ➔ must approximate
  - good idea to think about what to think about
  - uncertainty constrains the assignment of values to states
  - optimal decisions depend on information state, not real state
- Games are to multi-agent systems as grand prix racing is to automobile design