Game Playing

- The minimax algorithm
- Resource limitations
- alpha-beta pruning
- Elements of chance

What kind of games?

- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - …
- **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**

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 of the result.

2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.

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**)

Game vs. search problem

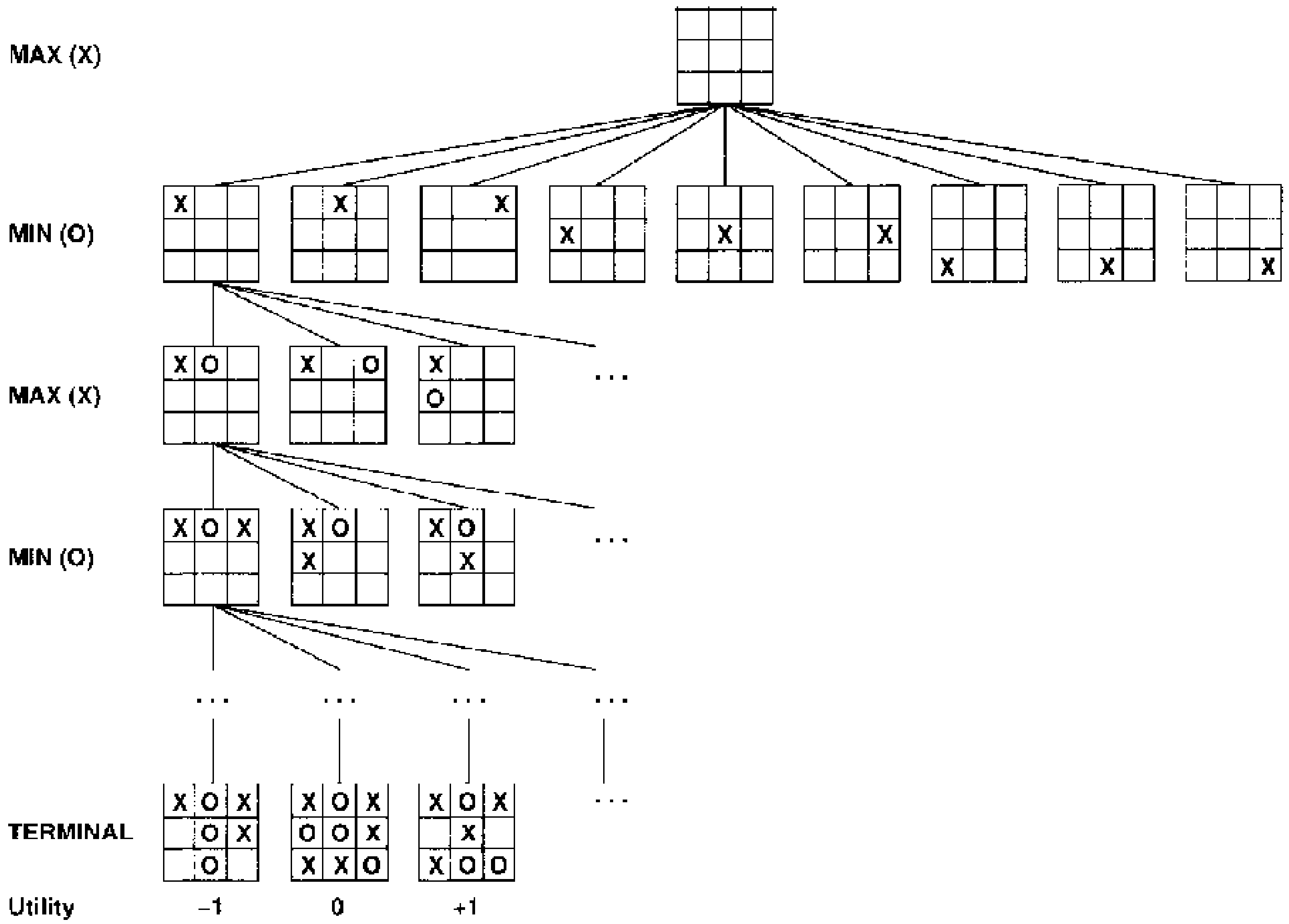
"Unpredictable" opponent $\Rightarrow$ solution is a contingency plan

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)

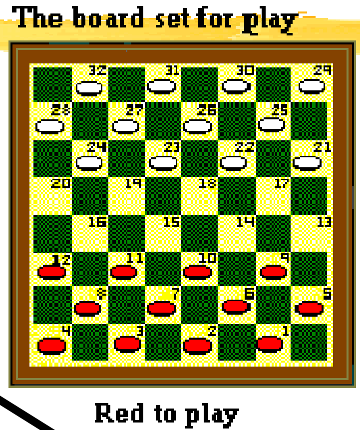
Example: Tic-Tac-Toe

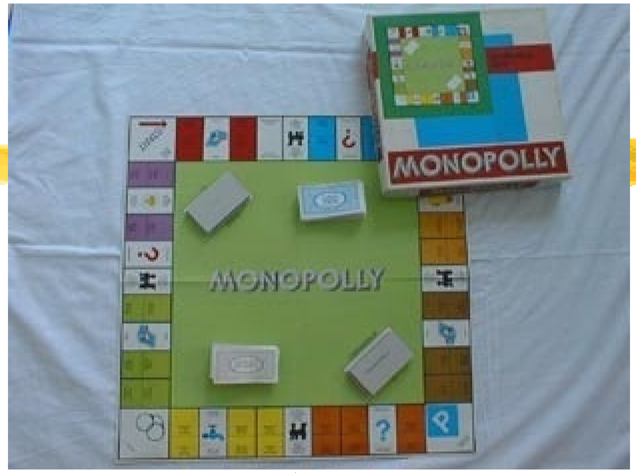


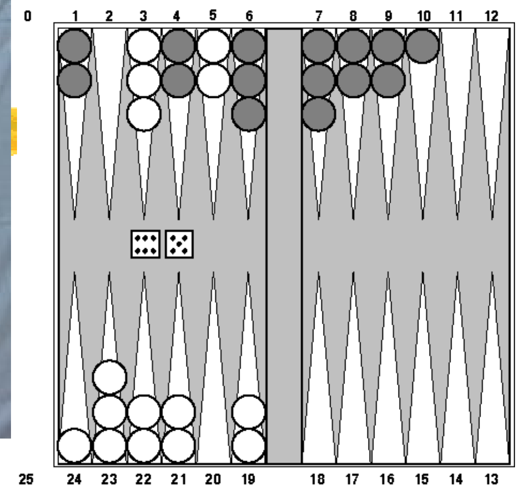
Type of games

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

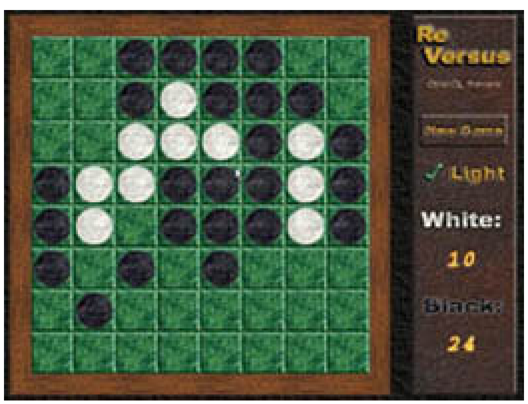
Type of games

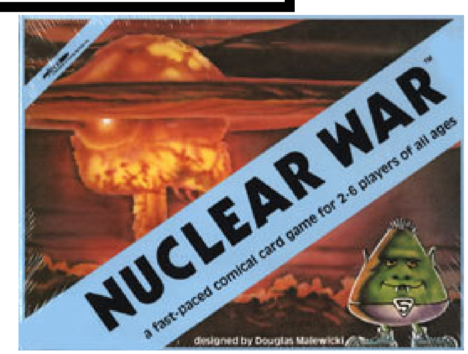






	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war



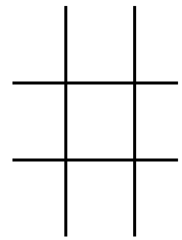


The minimax algorithm

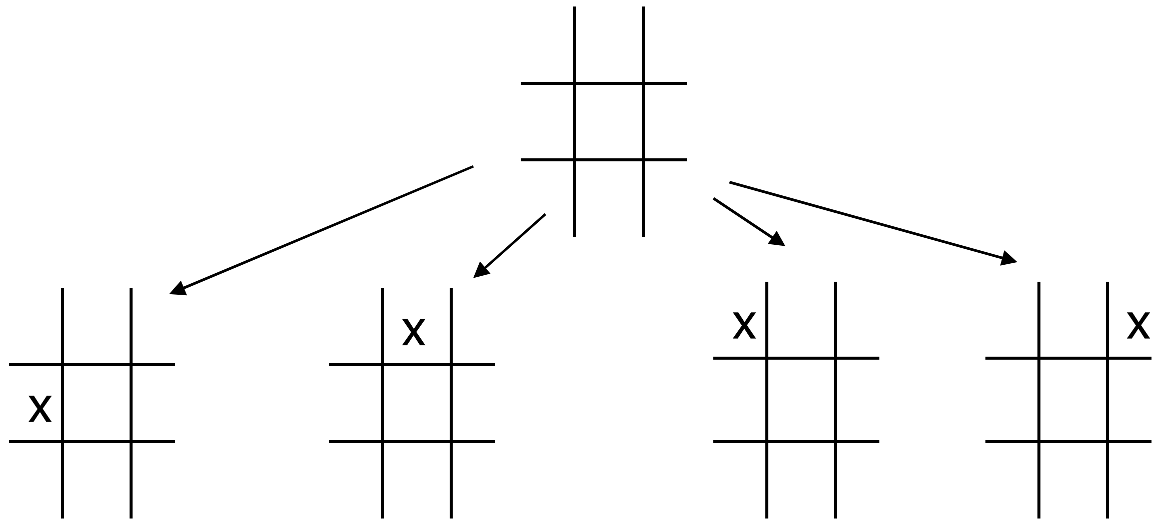
- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values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

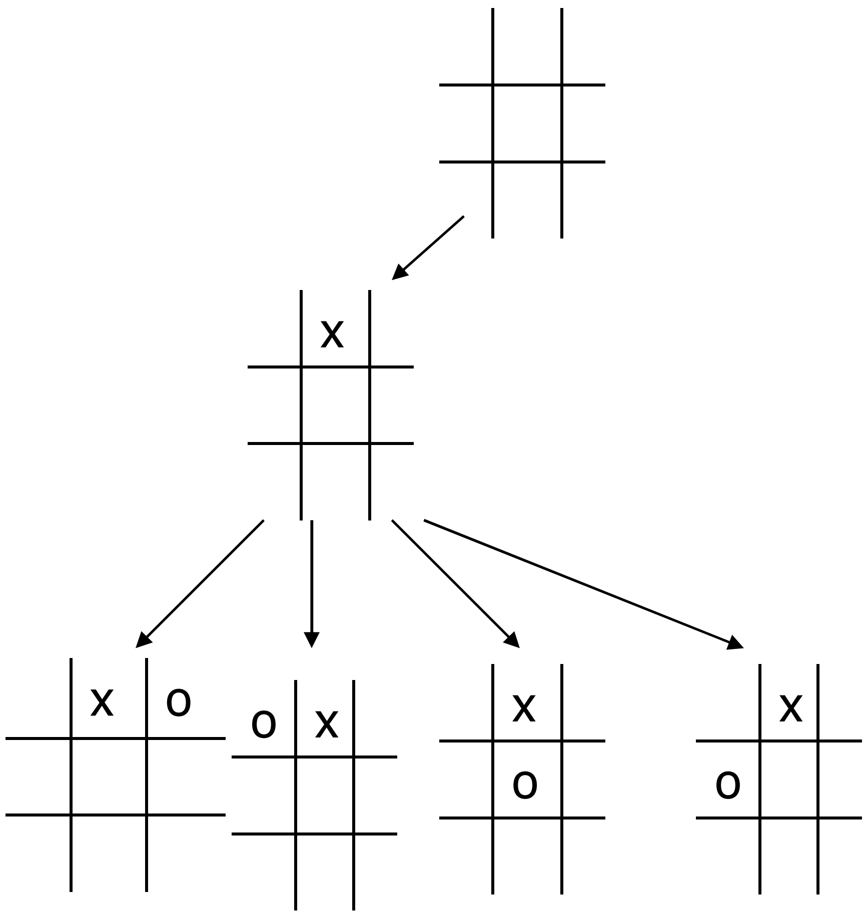
Generate Game Tree



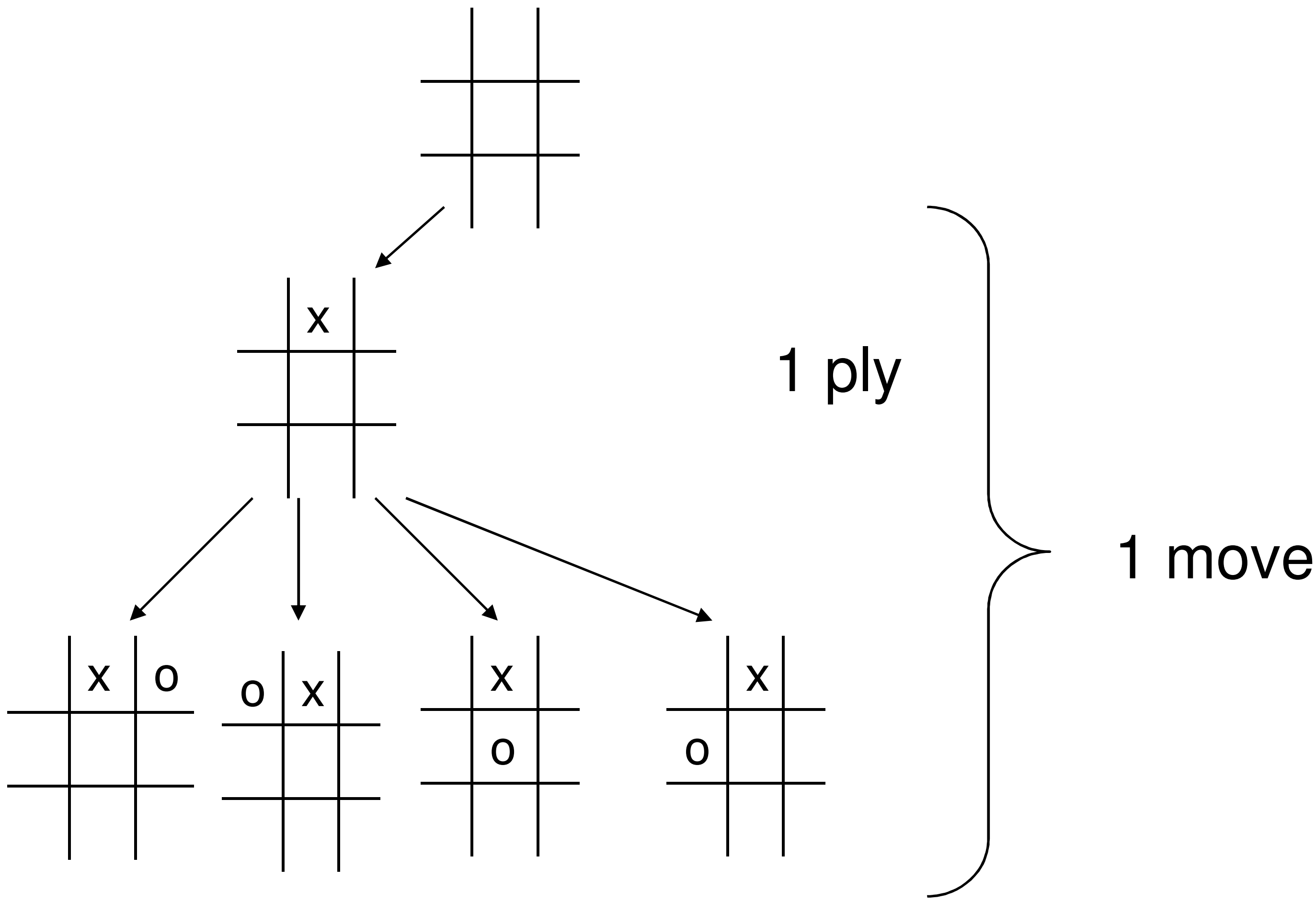
Generate Game Tree



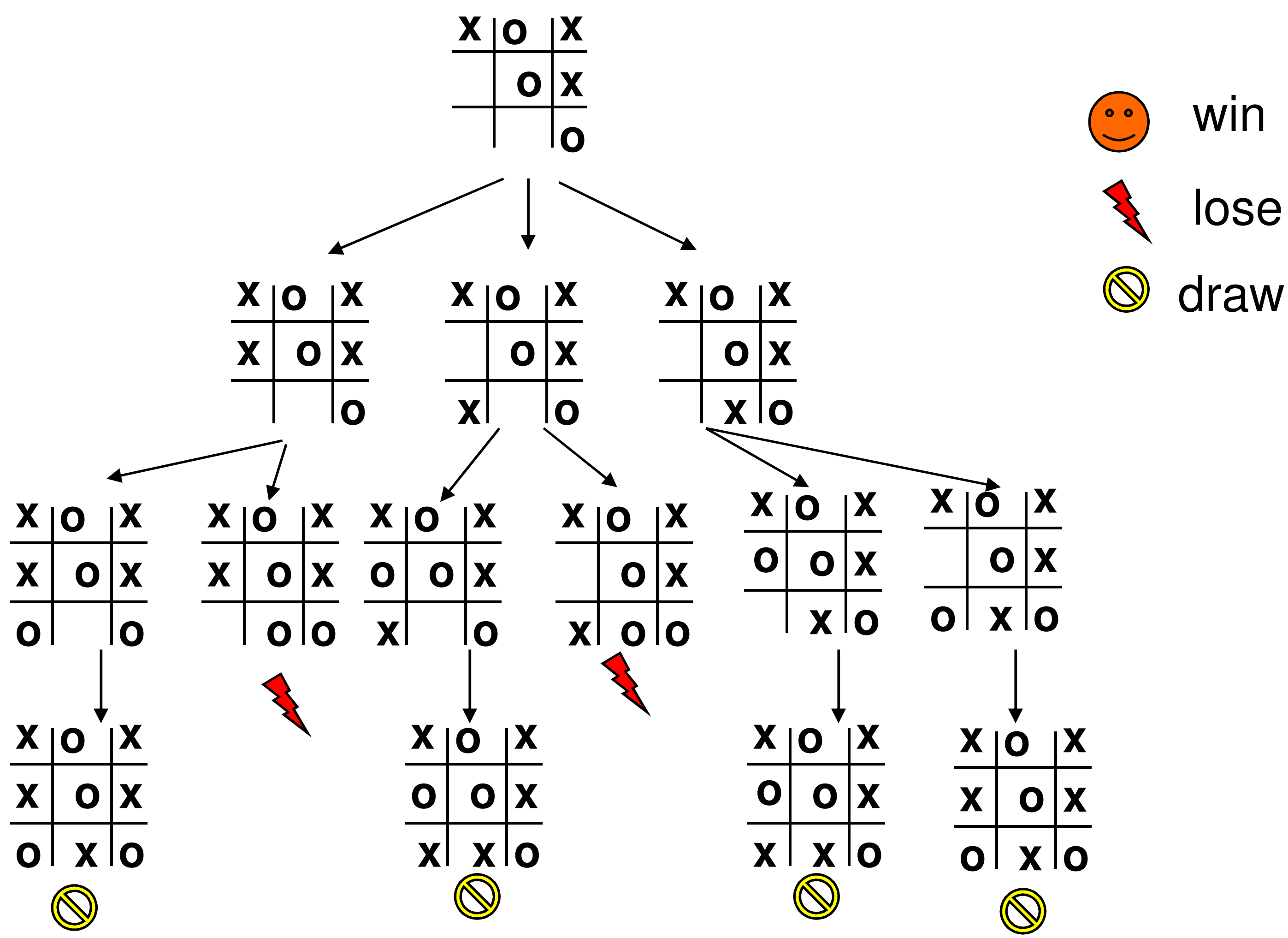
Generate Game Tree



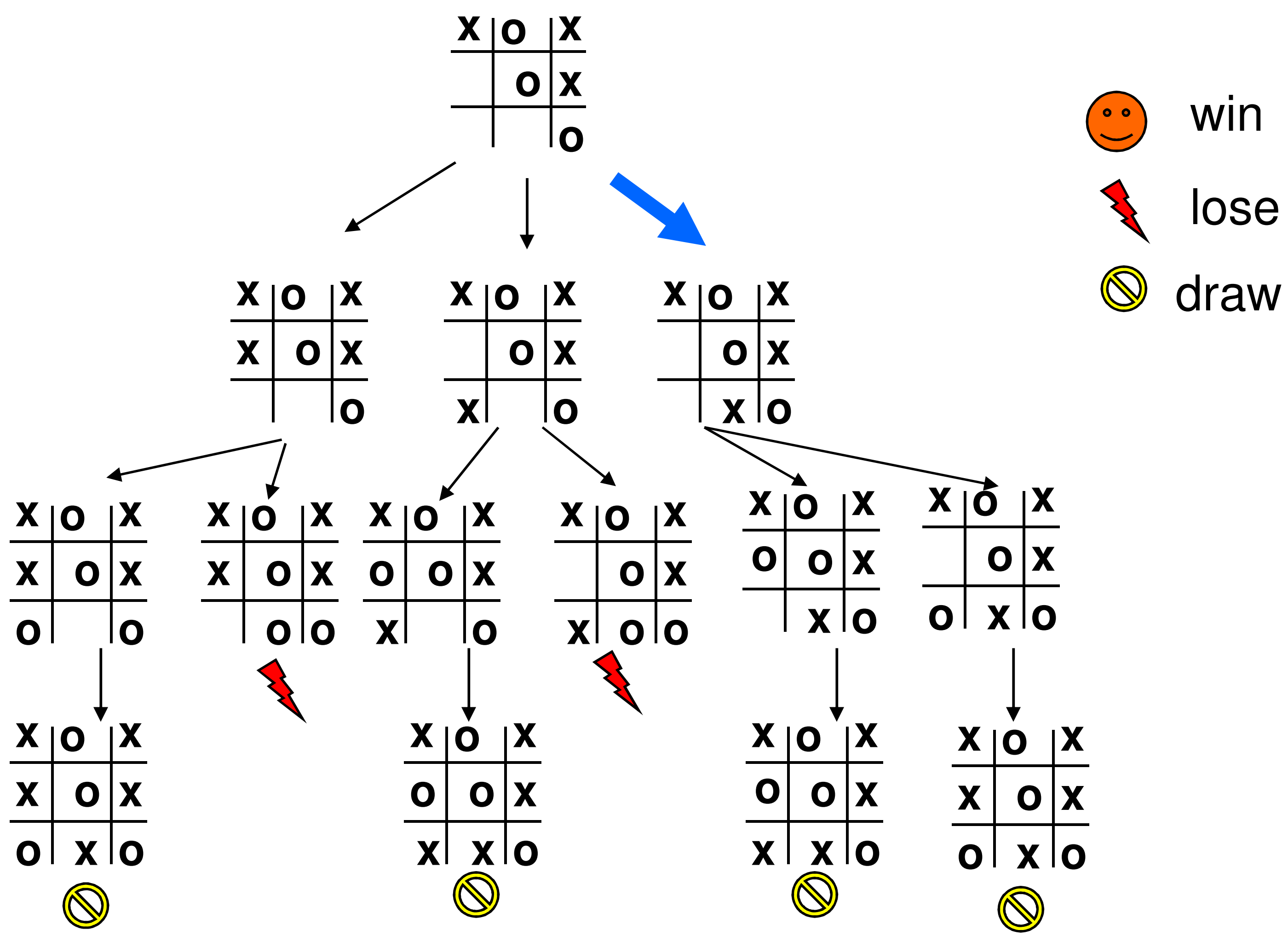
Generate Game Tree



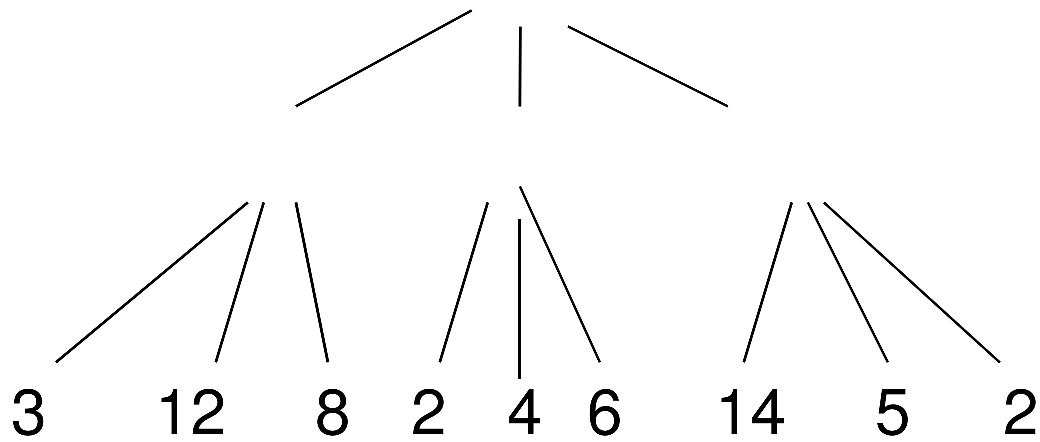
A subtree of the Gaming Tree



What is a good move?

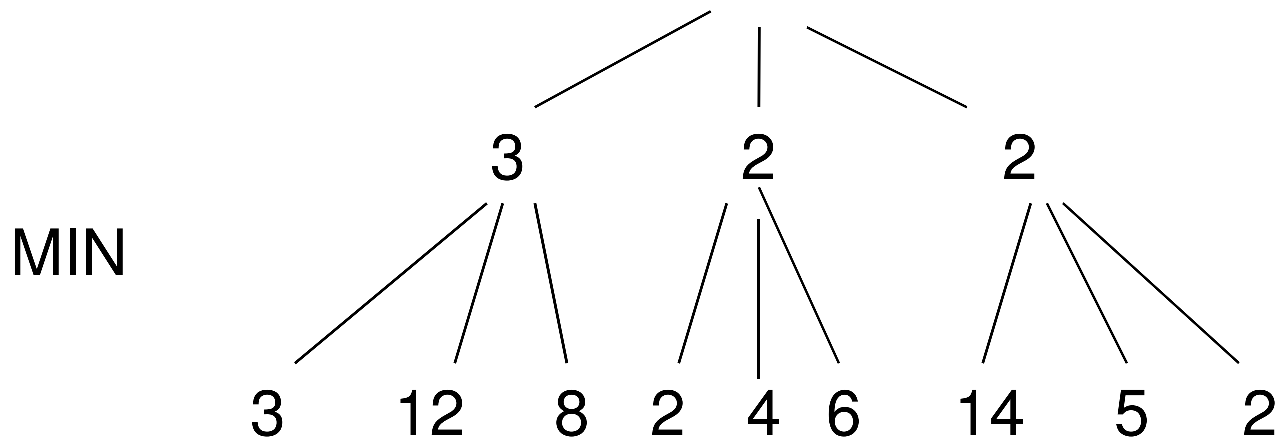


Minimax



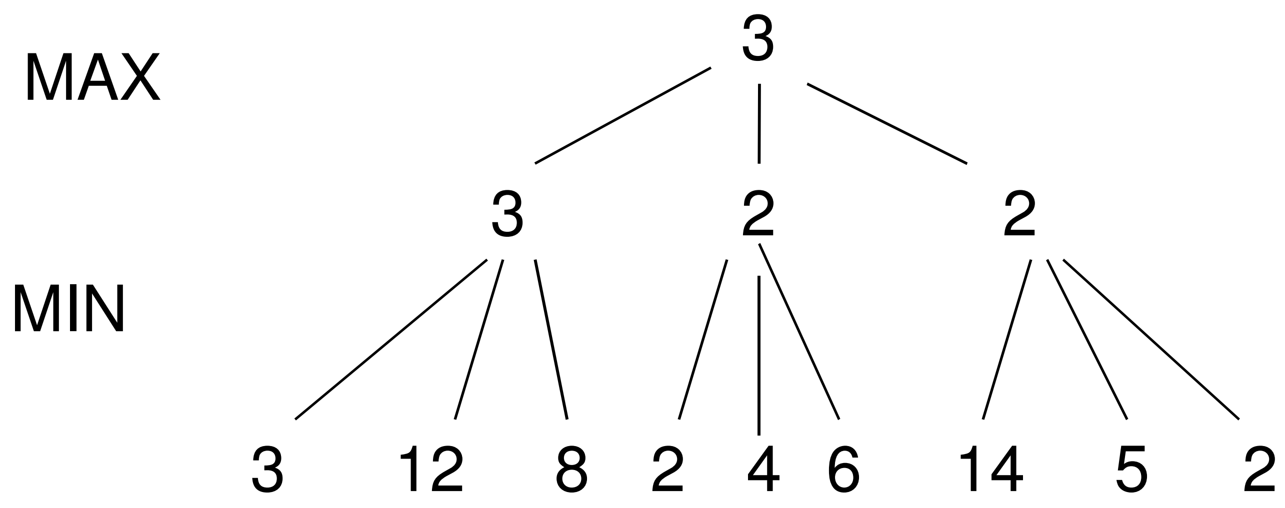
- Minimize opponent’s chance
- Maximize your chance

Minimax



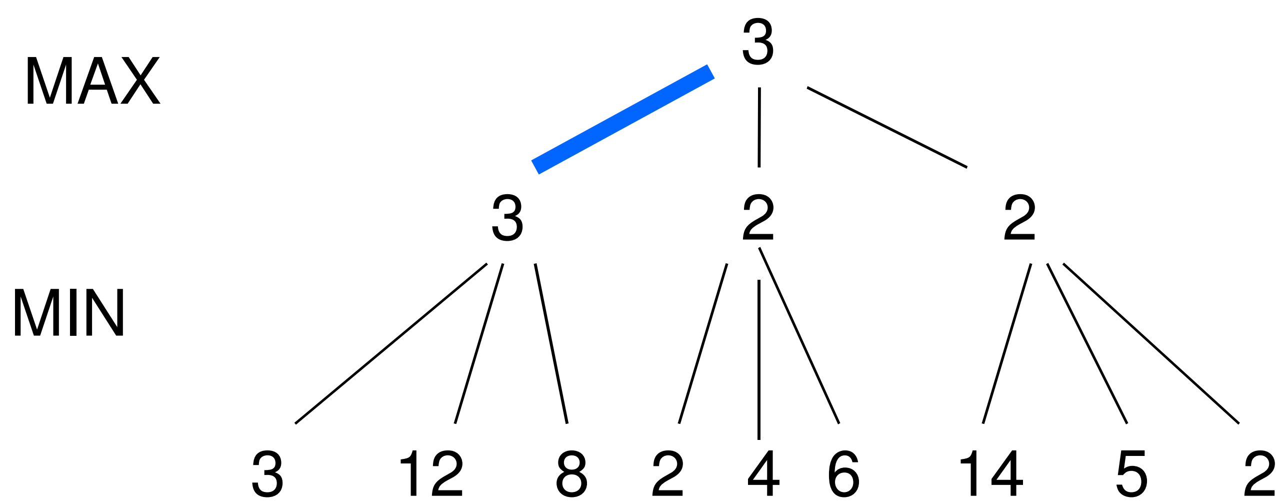
- Minimize opponent’s chance
- Maximize your chance

Minimax



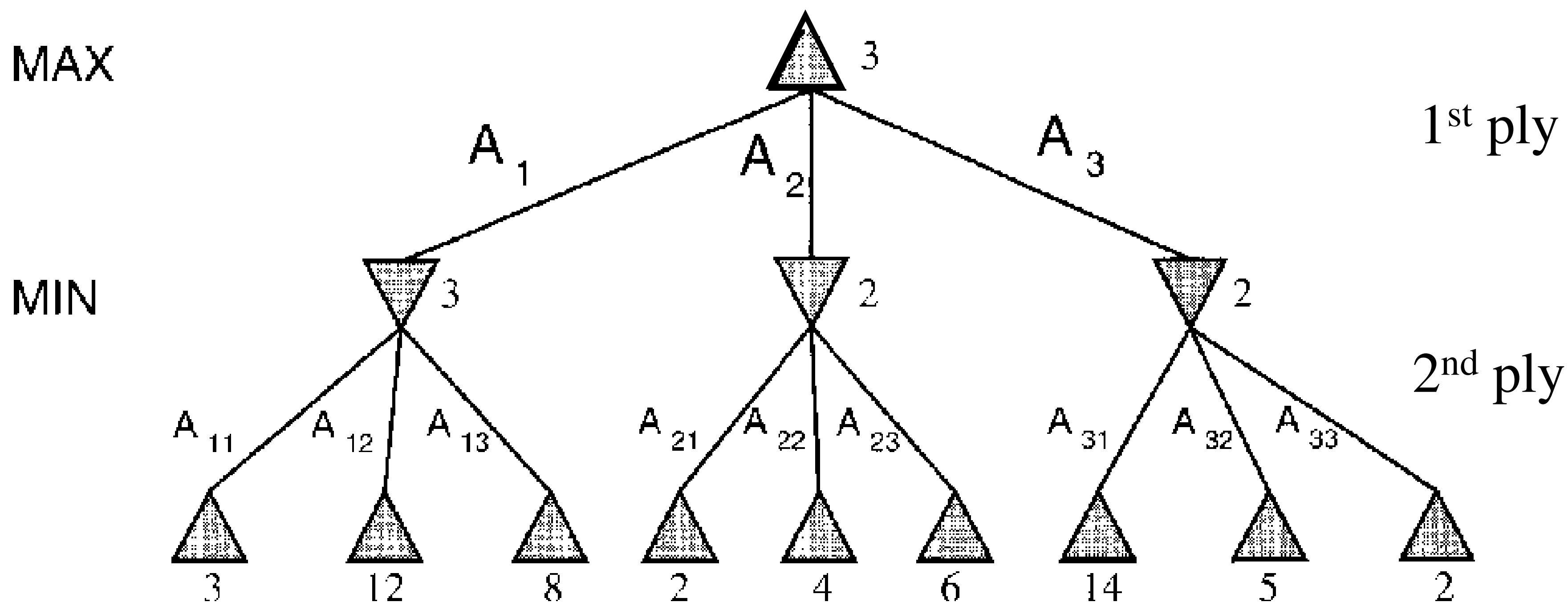
- Minimize opponent's chance
- Maximize your chance

Minimax



- Minimize opponent's chance
- Maximize your chance

minimax = maximum of the minimum



Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

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)
```

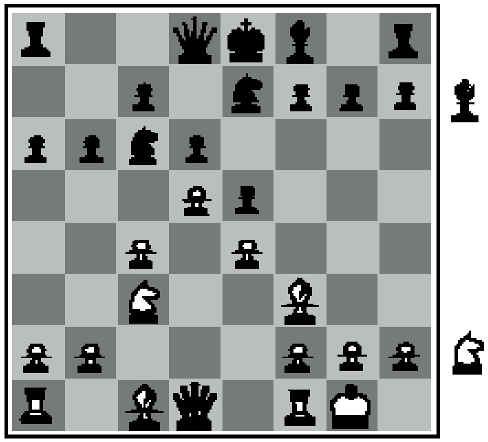
Complete: Yes, for finite state-space
Optimal: Yes

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

1. Move evaluation without complete search

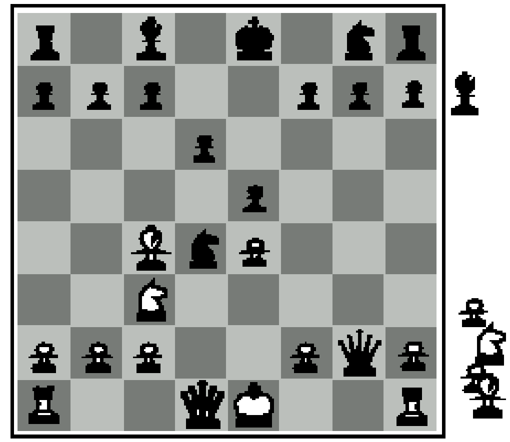
- Complete search is too complex and impractical
- **Evaluation function:** evaluates value of state using heuristics and cuts off search
- **New MINIMAX:**
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation functions



Black to move

White slightly better



White to move

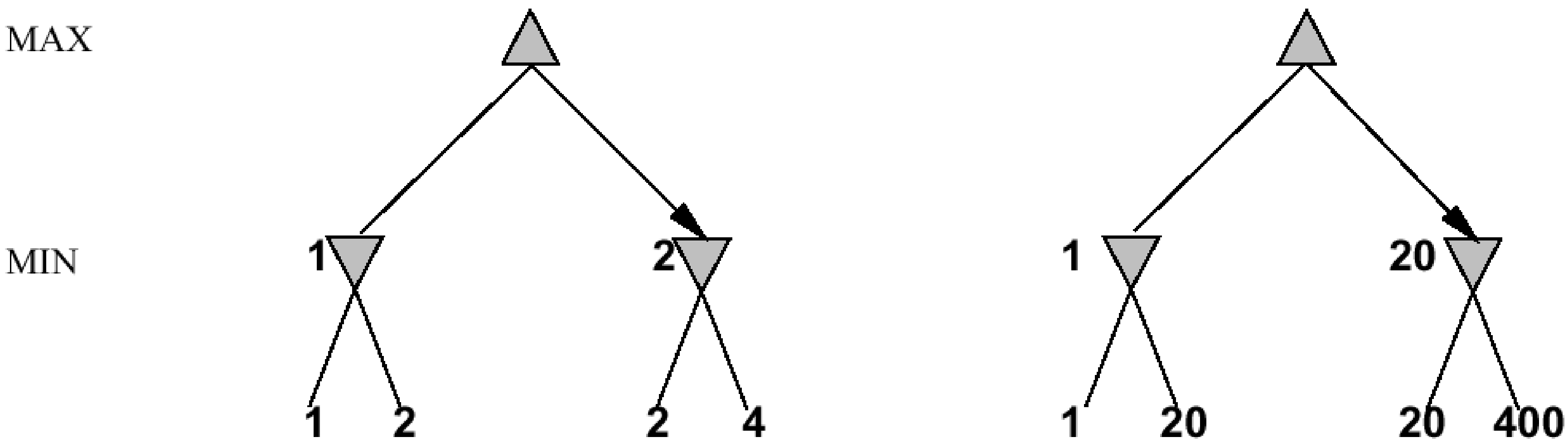
Black winning

- **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 pawn, 3 for bishop etc.)
f's could be the number of type of pieces on the board

EVAL: exact values do not matter in <u>deterministic</u> case



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:
payoff in deterministic games acts as an *ordinal utility* function

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 $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

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

Minimax Algorithm using Cutoff & Evaluation Function

function MINIMAX-DECISION(*state*) **returns** *an action*

$v \leftarrow$ MAX-VALUE(*state*)
return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE(*state*) **returns** *a utility value*

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

---

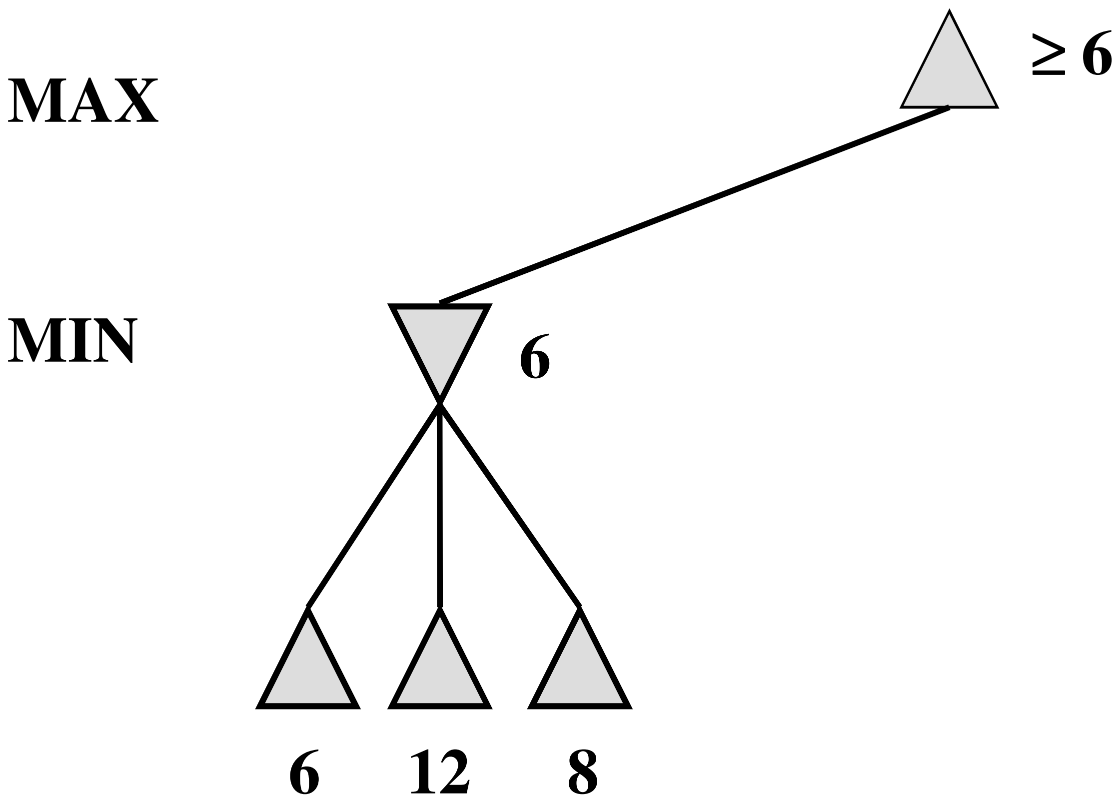
function MIN-VALUE(*state*) **returns** *a utility value*

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

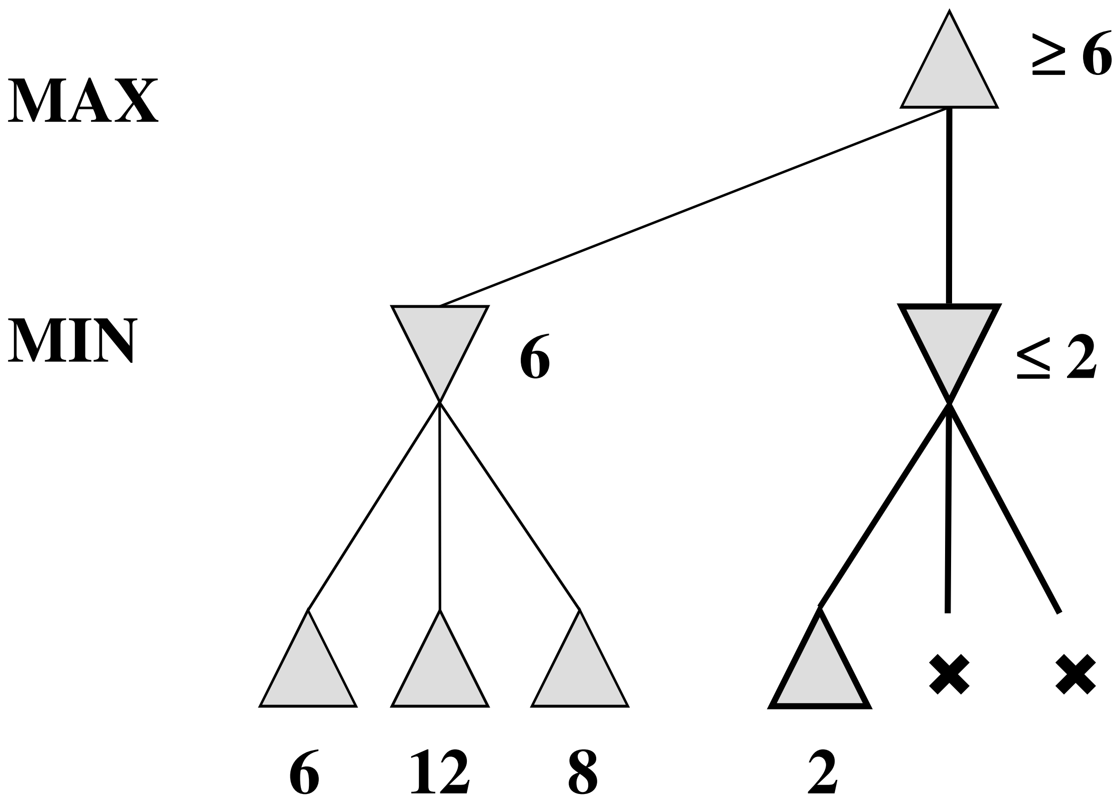
2. α-β pruning: search cutoff

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node
- **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

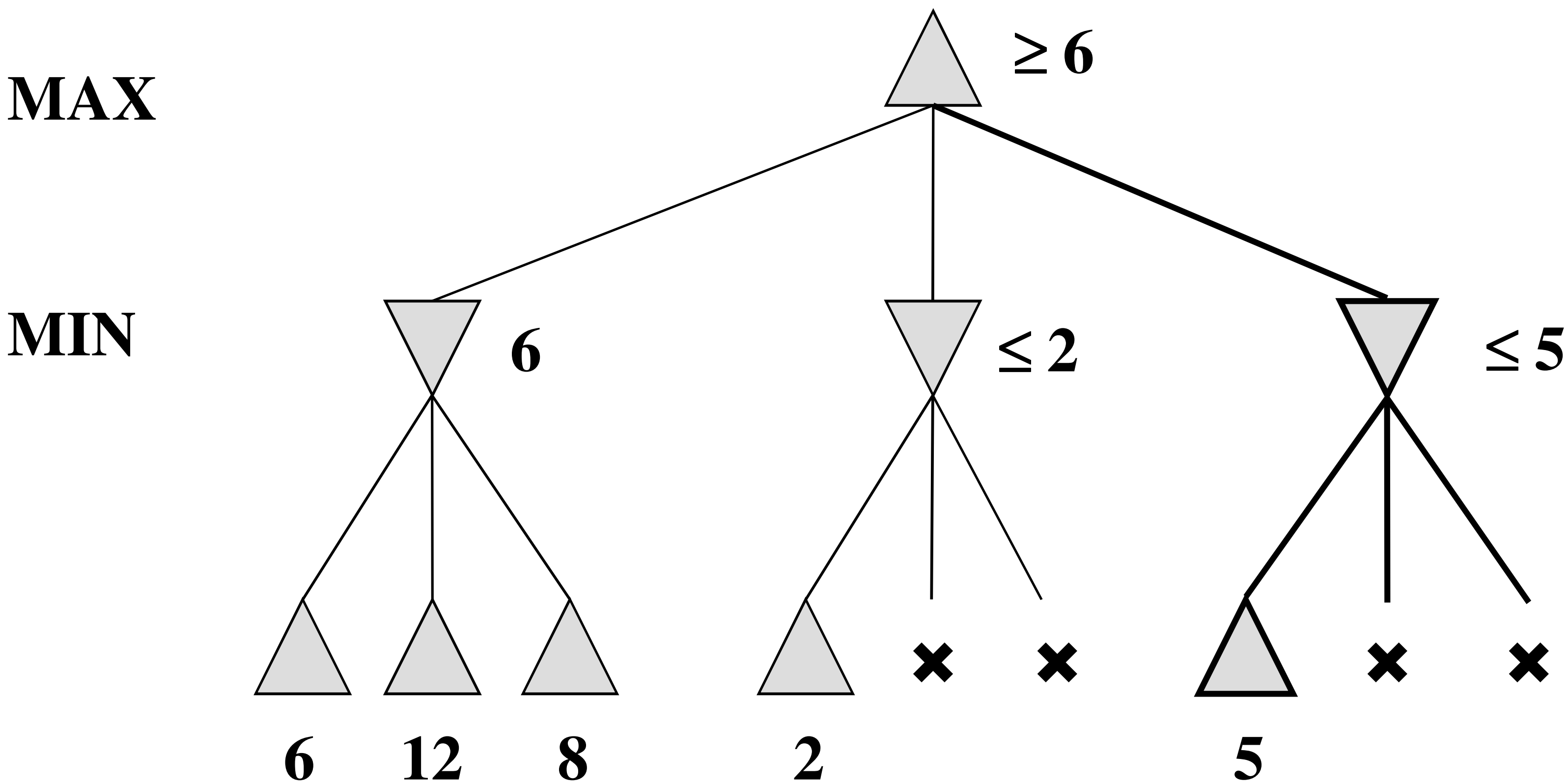
α-β pruning: example



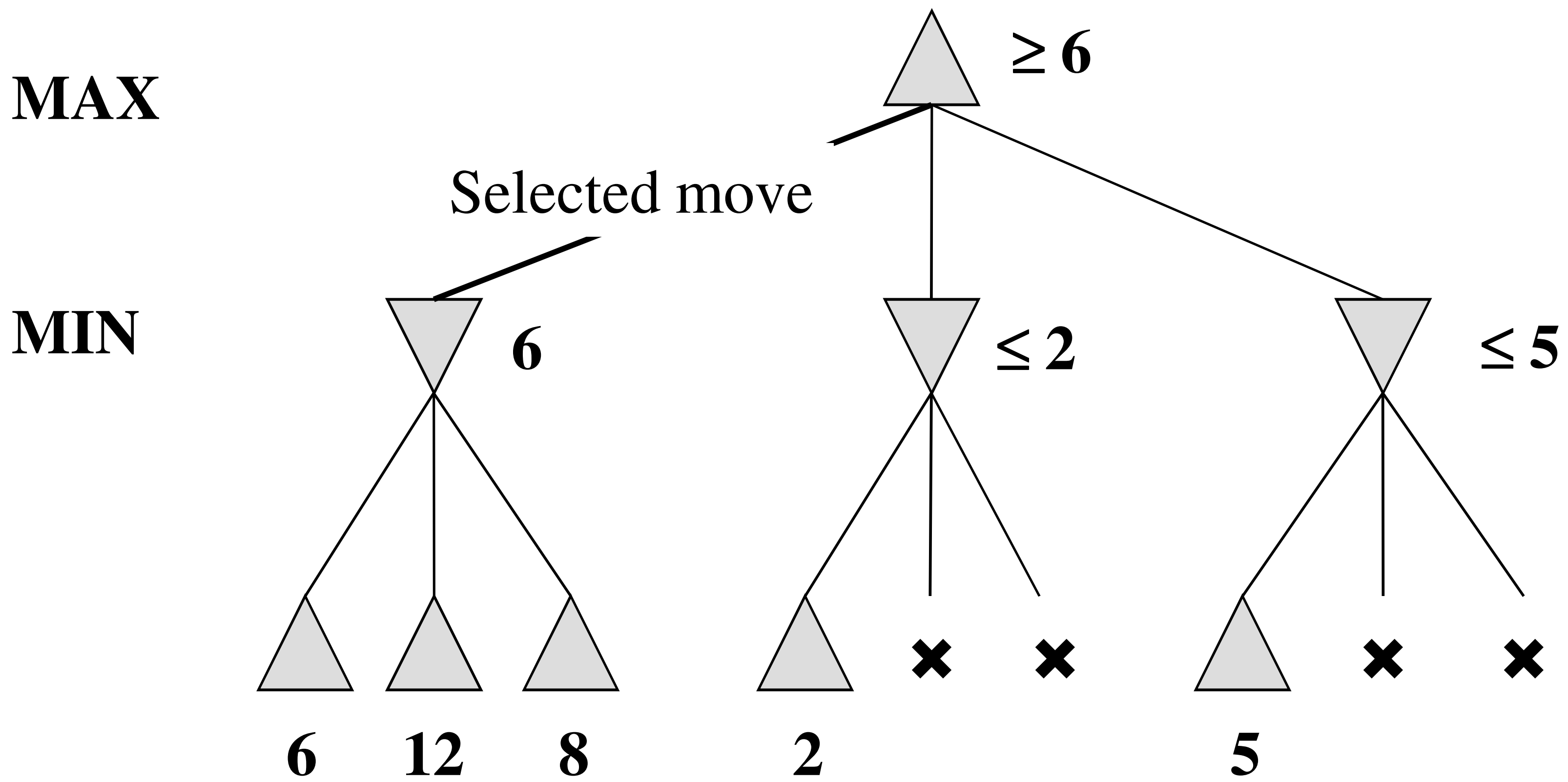
α-β pruning: example



α-β pruning: example



α-β pruning: example



Properties of α-β

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ *doubles* depth of search
$\Rightarrow$ can easily reach depth 8 and play good chess

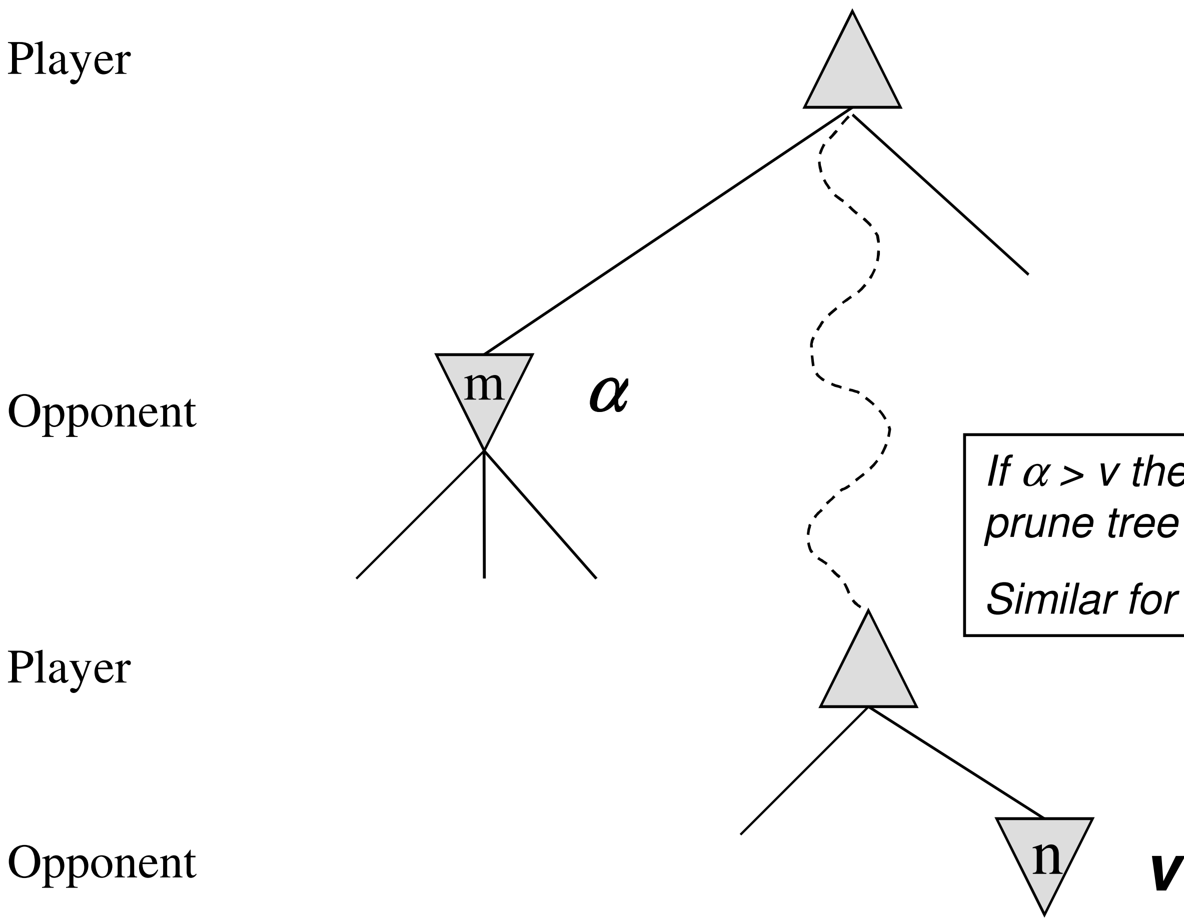
A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)

More on the α-β algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.

- 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

α-β pruning: general principle



If $\alpha > v$ then MAX will chose m so prune tree under n

Similar for β for MIN