Administrativia

- **Assignment 1 due thursday 9/25/2003 BEFORE midnight**
- **Midterm exam 10/09/2003 in class**

Last time: search strategies

Uninformed: Use only information available in the problem formulation

- Breadth-first
- Uniform-cost
- Depth-first
- Depth-limited
- Iterative deepening

Informed: Use heuristics to guide the search

- Best first:
- Greedy search – queue first nodes that maximize heuristic “desirability” based on estimated path cost from current node to goal;
- A* search – queue first nodes that maximize sum of path cost so far and estimated path cost to goal.
- Iterative improvement – keep no memory of path; work on a single current state and iteratively improve its “value.”
- Hill climbing – select as new current state the successor state which maximizes value.
- Simulated annealing – refinement on hill climbing by which “bad moves” are permitted, but with decreasing size and frequency. Will find global extremum.

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be <u>expanded</u> next by each of the following search algorithms?

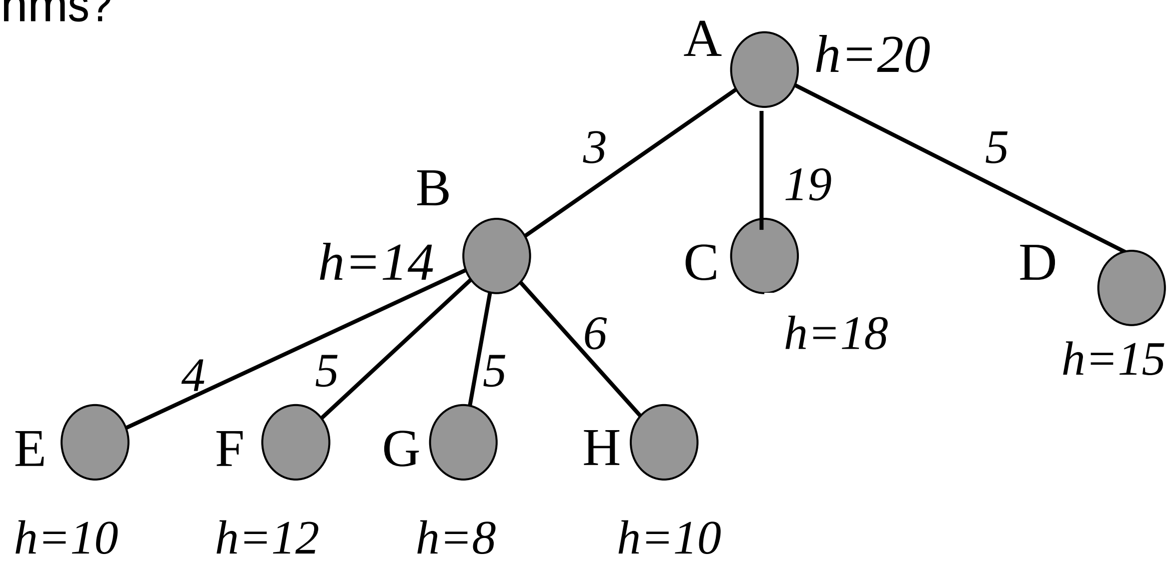
(a) Depth-first search

(b) Breadth-first search

(c) Uniform-cost search

(d) Greedy search

(e) A* search



Depth-first search

Node queue: initialization

#	state	depth	path cost	parent #
1	A	0	0	--

Depth-first search

Node queue: add successors to queue front; empty queue from top

#	state	depth	path cost	parent #
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1
1	A	0	0	--

Depth-first search

Node queue: add successors to queue front; empty queue from top

#	state	depth	path cost	parent #
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1
1	A	0	0	--

Depth-first search

Node queue: add successors to queue front; empty queue from top

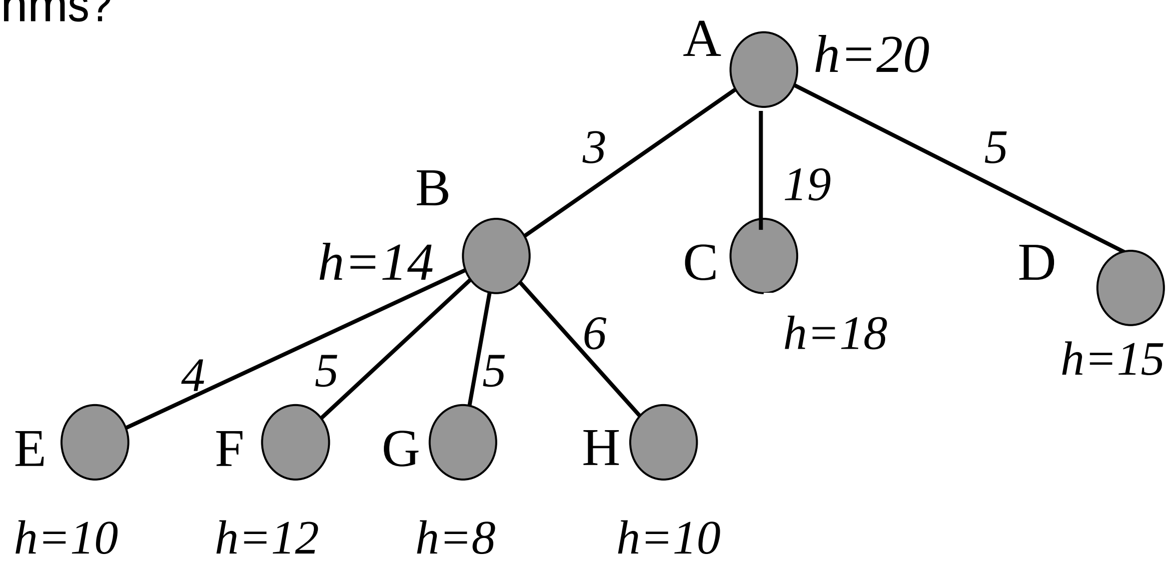
#	state	depth	path cost	parent #
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1
1	A	0	0	--

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be <u>expanded</u> next by each of the following search algorithms?

(a) Depth-first search
(b) Breadth-first search
(c) Uniform-cost search
(d) Greedy search
(e) A* search



Breadth-first search

Node queue: initialization

#	state	depth	path cost	parent #
1	A	0	0	--

Breadth-first search

Node queue: add successors to queue end; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1

Breadth-first search

Node queue: add successors to queue end; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2

Breadth-first search

Node queue: add successors to queue end; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	C	1	19	1
4	D	1	5	1
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be expanded next by each of the following search algorithms?

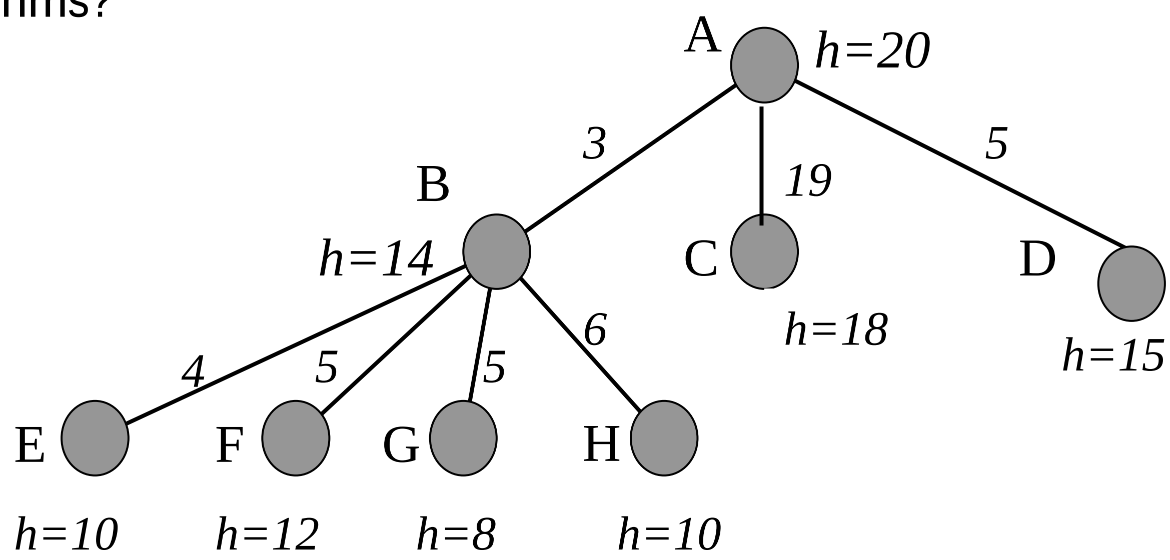
(a) Depth-first search

(b) Breadth-first search

(c) Uniform-cost search

(d) Greedy search

(e) A* search



Uniform-cost search

Node queue: initialization

#	state	depth	path cost	parent #
1	A	0	0	--

Uniform-cost search

Node queue: add successors to queue so that entire queue is sorted by path cost so far; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	D	1	5	1
4	C	1	19	1

Uniform-cost search

Node queue: add successors to queue so that entire queue is sorted by path cost so far; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	D	1	5	1
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2
4	C	1	19	1

Uniform-cost search

Node queue: add successors to queue so that entire queue is sorted by path cost so far; empty queue from top

#	state	depth	path cost	parent #
1	A	0	0	--
2	B	1	3	1
3	D	1	5	1
5	E	2	7	2
6	F	2	8	2
7	G	2	8	2
8	H	2	9	2
4	C	1	19	1

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be expanded next by each of the following search algorithms?

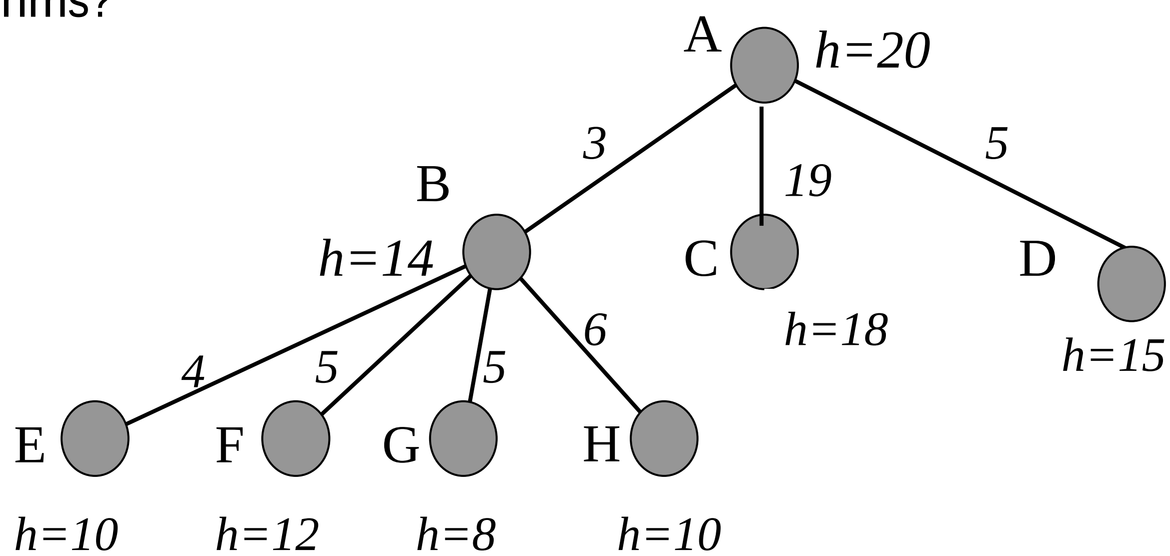
(a) Depth-first search

(b) Breadth-first search

(c) Uniform-cost search

(d) Greedy search

(e) A* search



Greedy search

Node queue: initialization

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--

Greedy search

Node queue: Add successors to queue, sorted by cost to goal.

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
3	D	1	5	15	20	1
4	C	1	19	18	37	1

Sort key

Greedy search

Node queue: Add successors to queue, sorted by cost to goal.

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
5	G	2	8	8	16	2
7	E	2	7	10	17	2
6	H	2	9	10	19	2
8	F	2	8	12	20	2
3	D	1	5	15	20	1
4	C	1	19	18	37	1

Greedy search

Node queue: Add successors to queue, sorted by cost to goal.

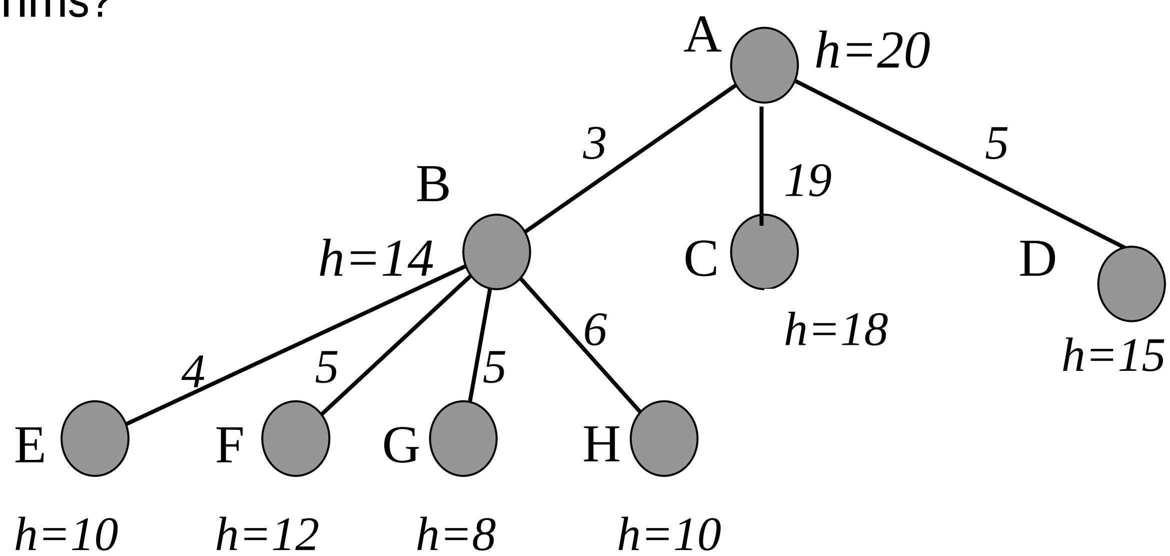
#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
5	G	2	8	8	16	2
7	E	2	7	10	17	2
6	H	2	9	10	19	2
8	F	2	8	12	20	2
3	D	1	5	15	20	1
4	C	1	19	18	37	1

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be <u>expanded</u> next by each of the following search algorithms?

(a) Depth-first search
(b) Breadth-first search
(c) Uniform-cost search
(d) Greedy search
(e) A* search



A* search

Node queue: initialization

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--

A* search

Node queue: Add successors to queue, sorted by total cost.

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
3	D	1	5	15	20	1
4	C	1	19	18	37	1

Sort key

A* search

Node queue: Add successors to queue front, sorted by total cost.

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
5	G	2	8	8	16	2
6	E	2	7	10	17	2
7	H	2	9	10	19	2
3	D	1	5	15	20	1
8	F	2	8	12	20	2
4	C	1	19	18	37	1

A* search

Node queue: Add successors to queue front, sorted by total cost.

#	state	depth	path cost	cost to goal	total cost	parent #
1	A	0	0	20	20	--
2	B	1	3	14	17	1
5	G	2	8	8	16	2
6	E	2	7	10	17	2
7	H	2	9	10	19	2
3	D	1	5	15	20	1
8	F	2	8	12	20	2
4	C	1	19	18	37	1

Exercise: Search Algorithms

The following figure shows a portion of a partially expanded search tree. Each arc between nodes is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of the heuristic function, *h*.

Which node (use the node's letter) will be expanded next by each of the following search algorithms?

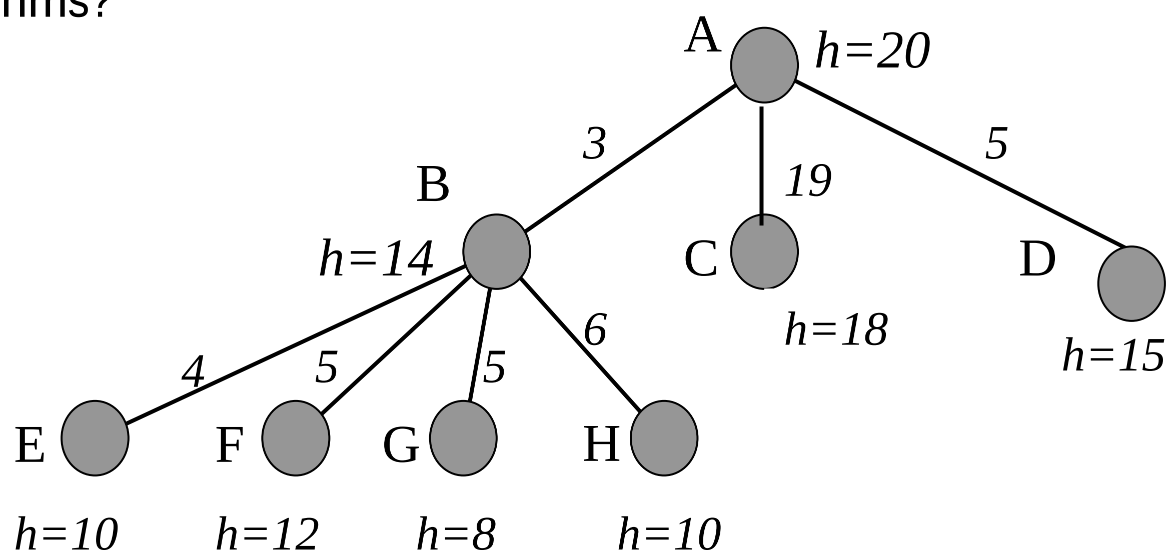
(a) Depth-first search

(b) Breadth-first search

(c) Uniform-cost search

(d) Greedy search

(e) A* search



Last time: Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to “temperature”
  local variables: *current*, a node
    next, a node
    T, a “temperature” controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] - VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Note: goal here is to maximize E.

Last time: Simulated annealing algorithm

- Idea: Escape local extrema by allowing "bad moves," but gradually decrease their size and frequency.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if T=0 **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] - VALUE[*current*]
    if $\Delta E < 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{-\Delta E/T}$

Algorithm when goal is to minimize E.

This time: Outline

- **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 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 <u>numeric</u> value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA **payoff function**)

Game vs. search problem

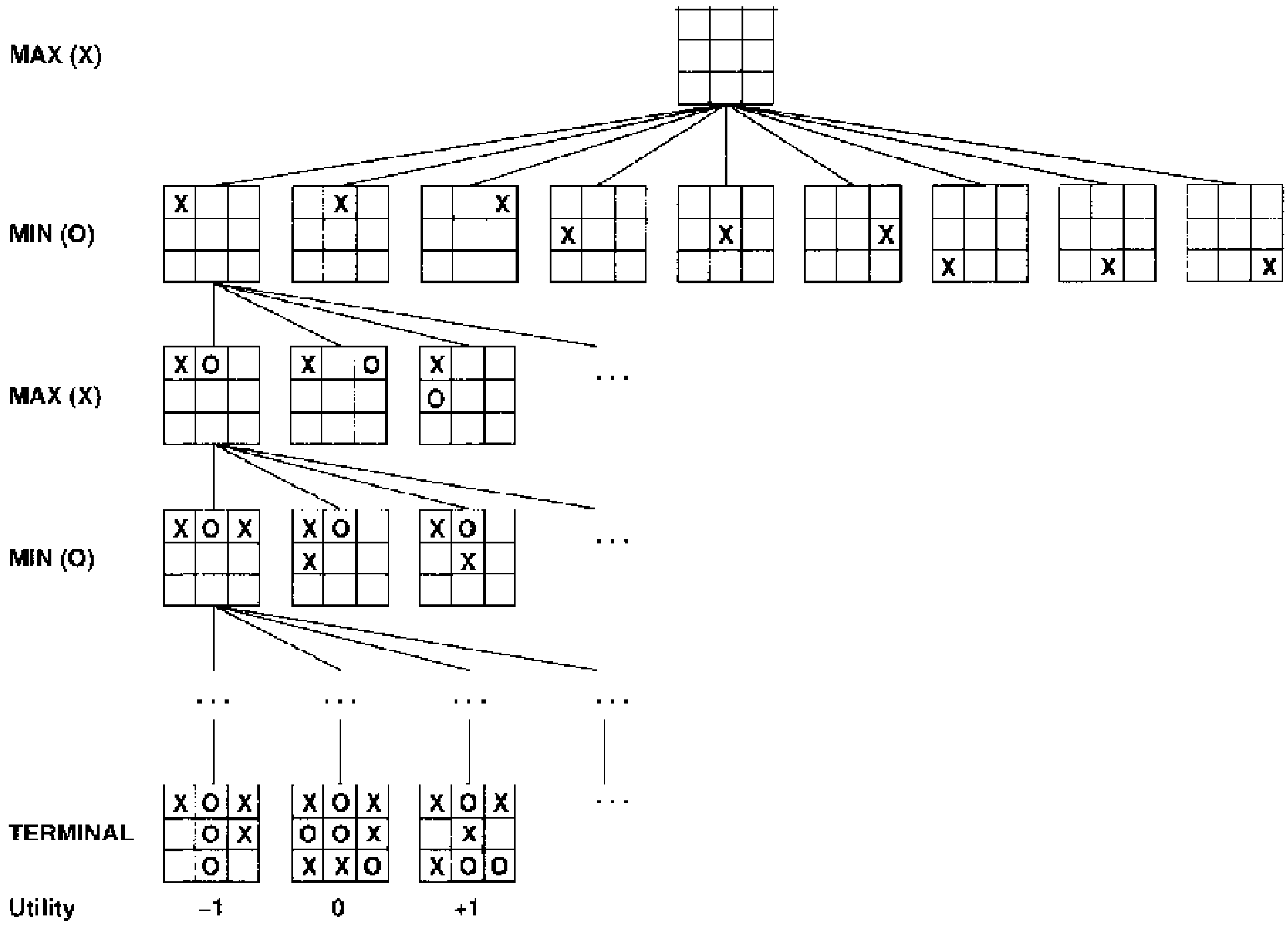
"Unpredictable" opponent $\Rightarrow$ solution is a contingency plan

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



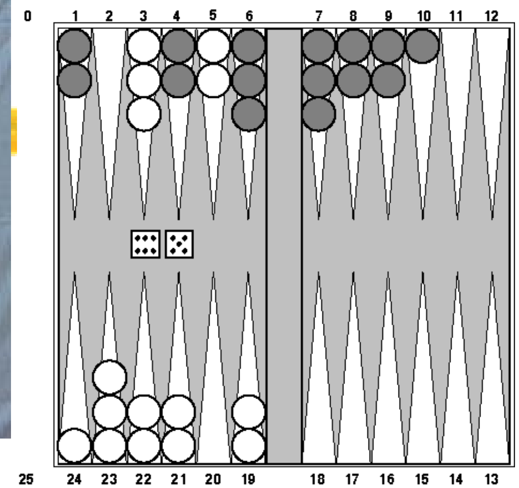
Type of games

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

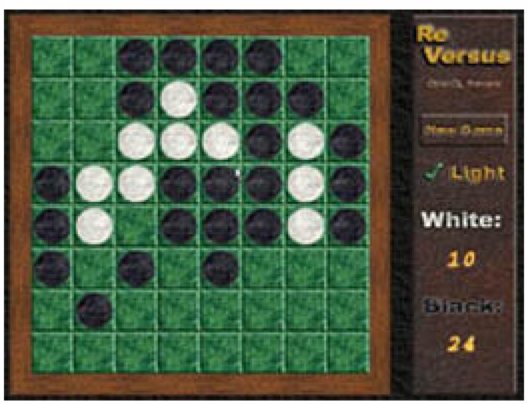
Type of games

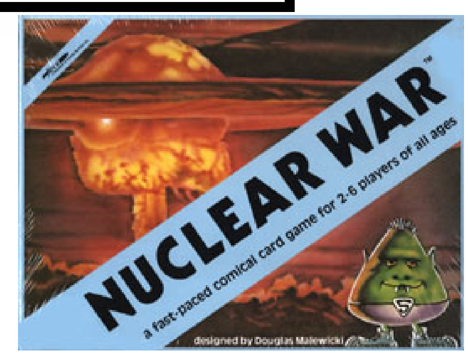
The board set for play

Red to play



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



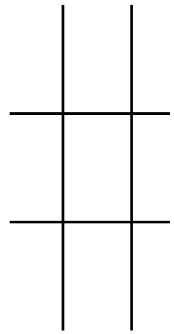


The minimax algorithm

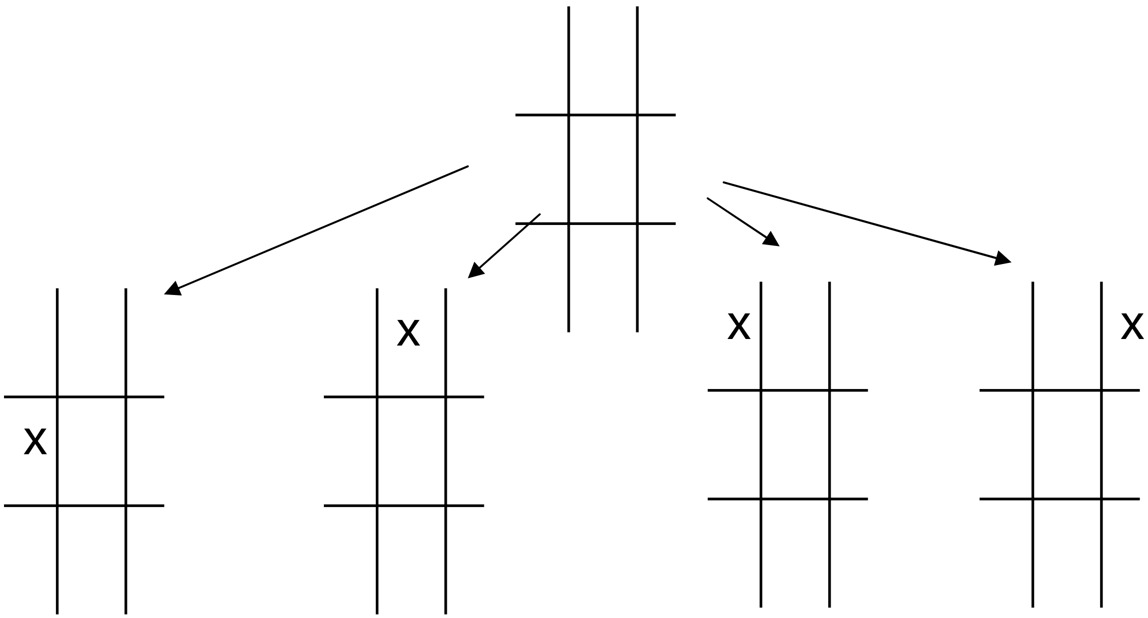
- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values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

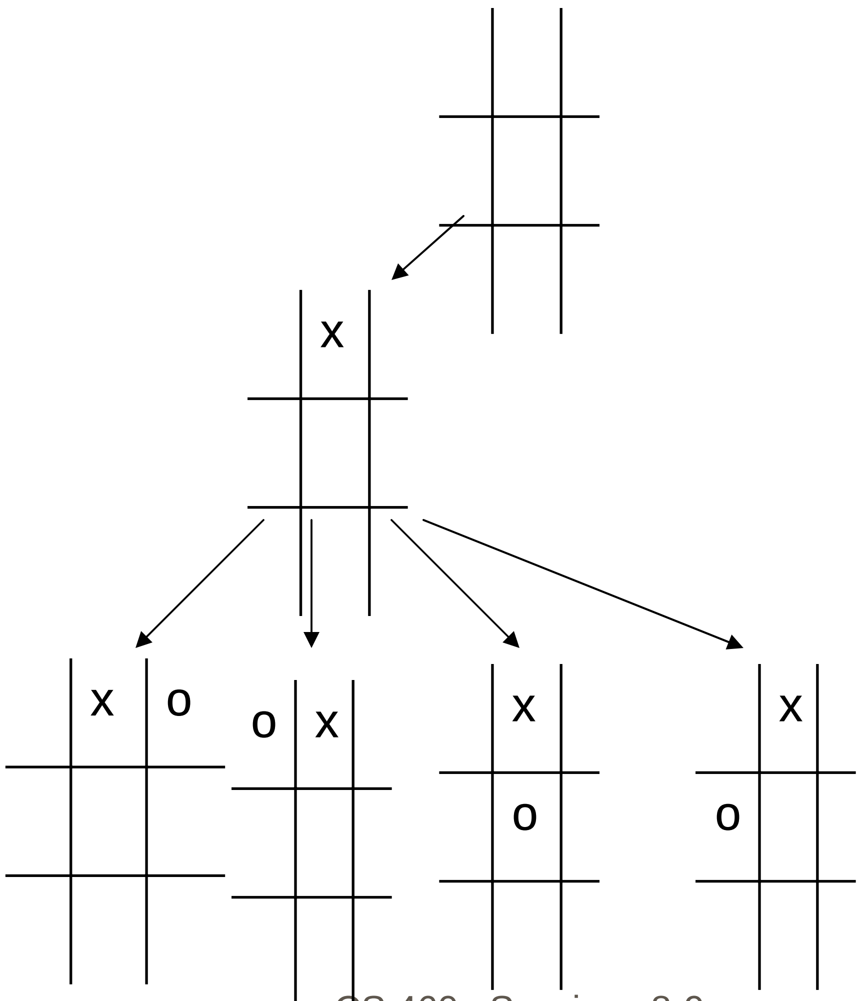
Generate Game Tree



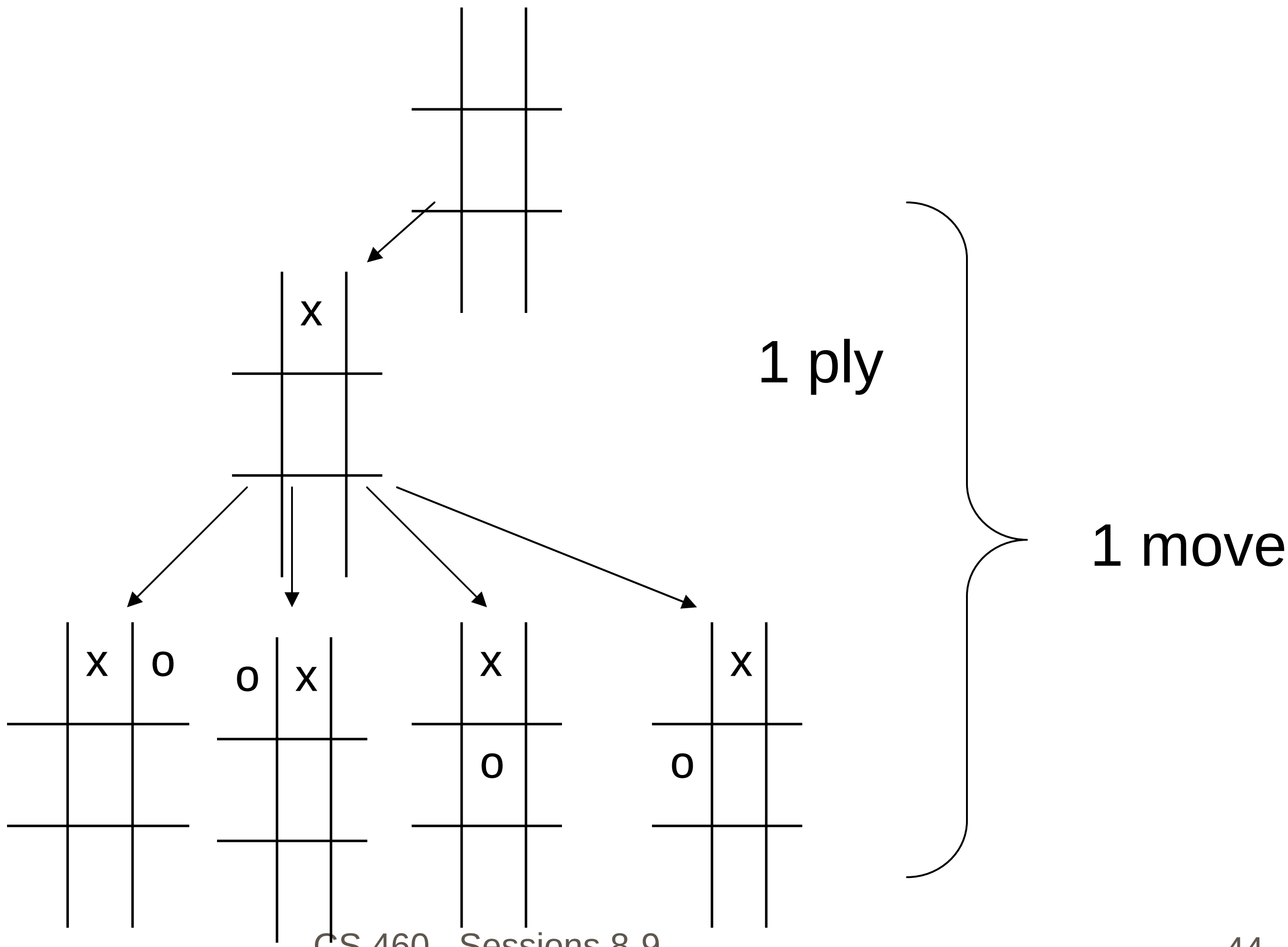
Generate Game Tree



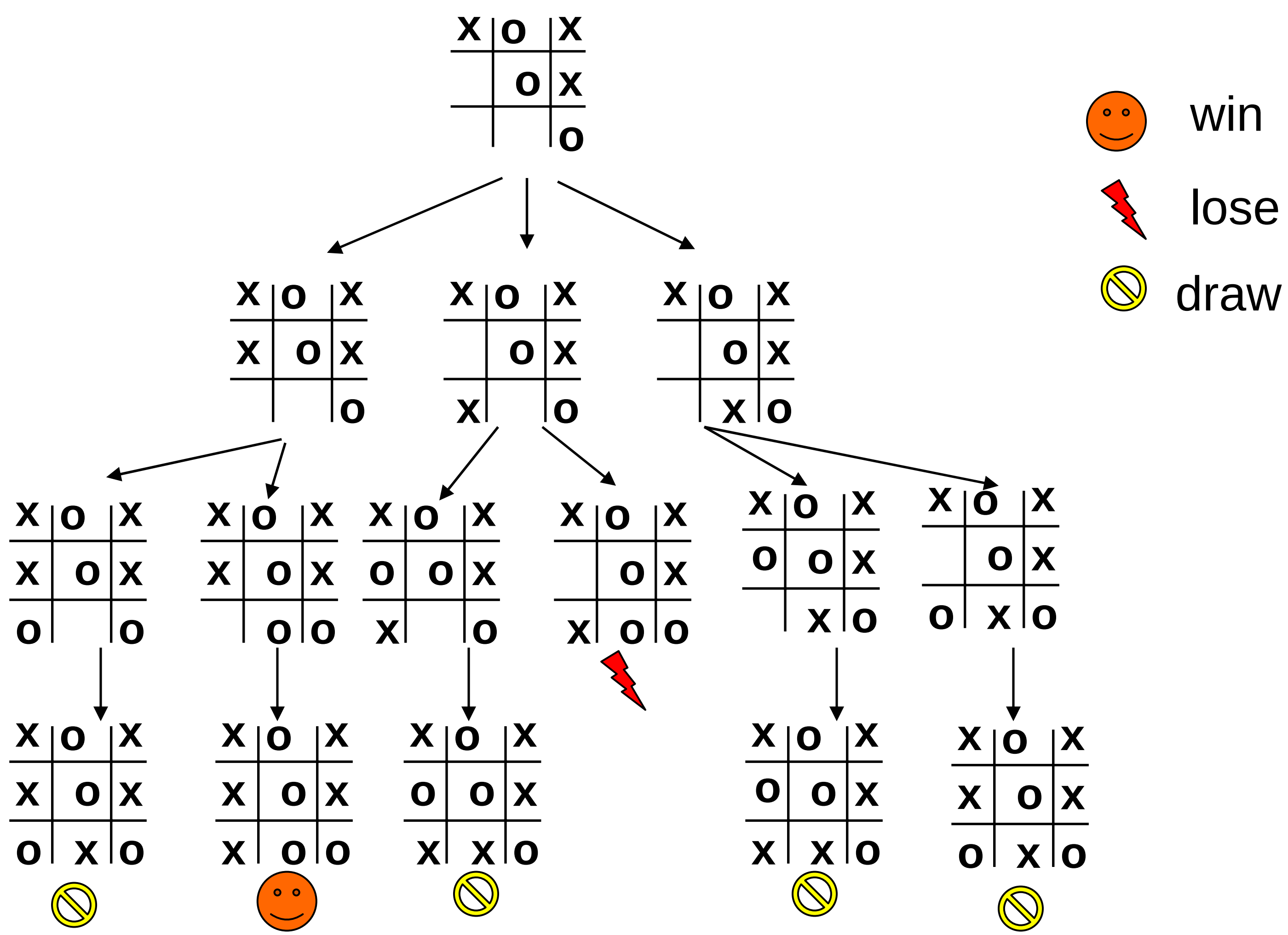
Generate Game Tree



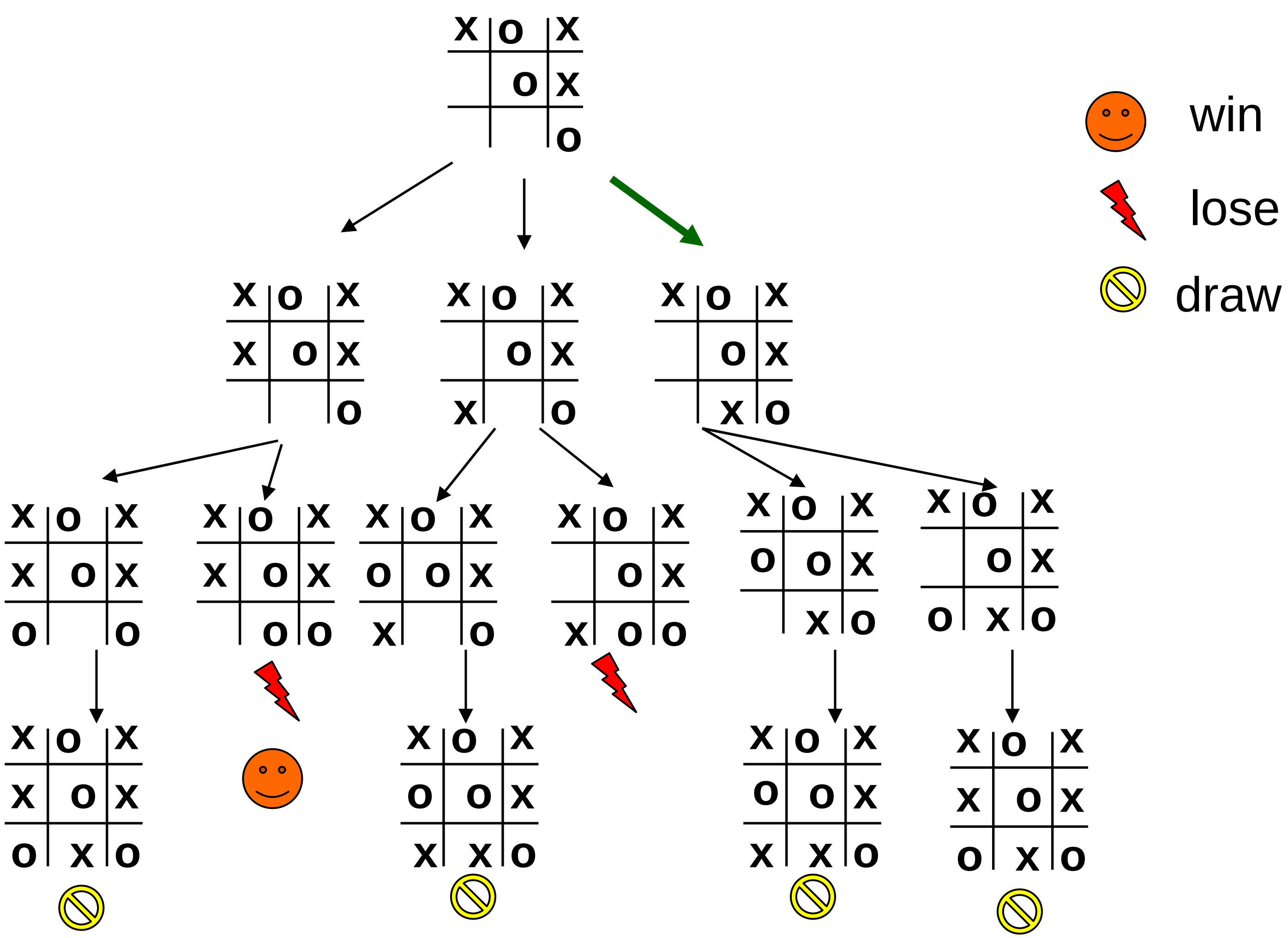
Generate Game Tree



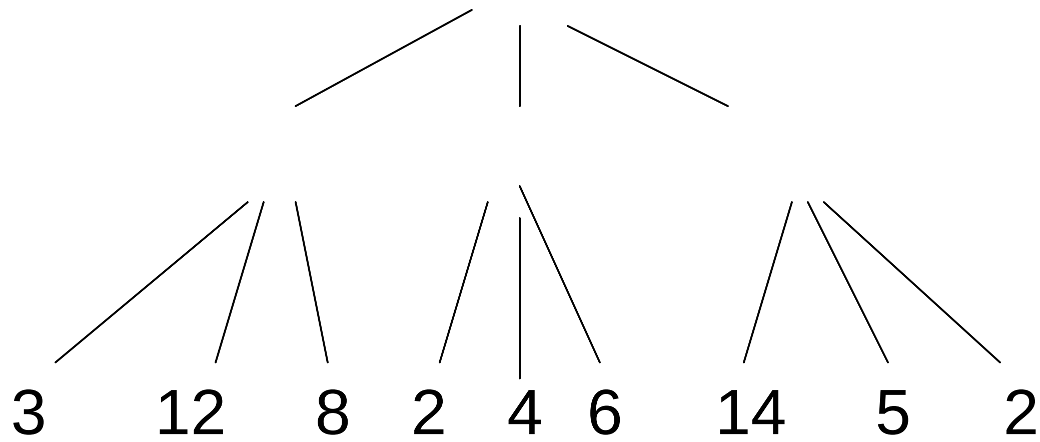
A subtree



What is a good move?

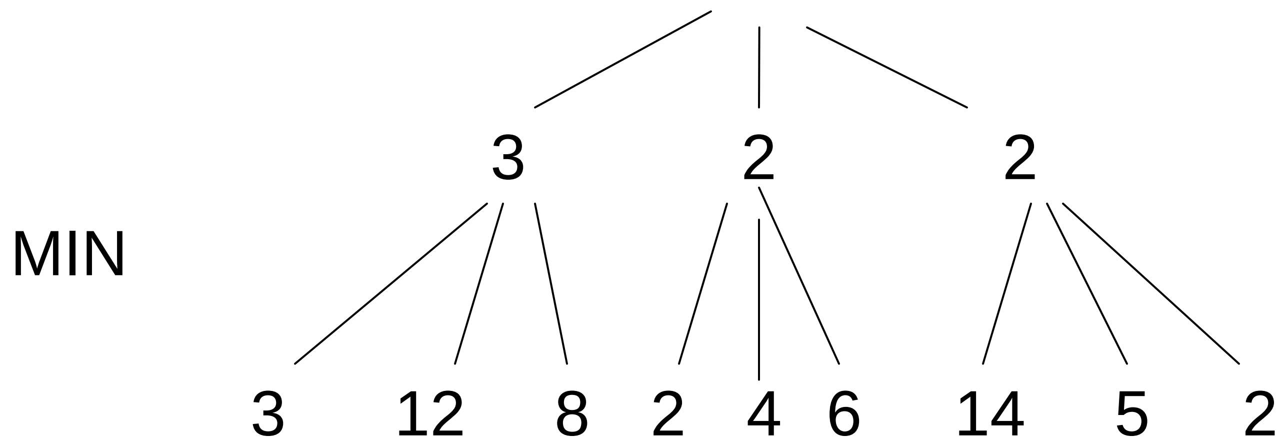


Minimax



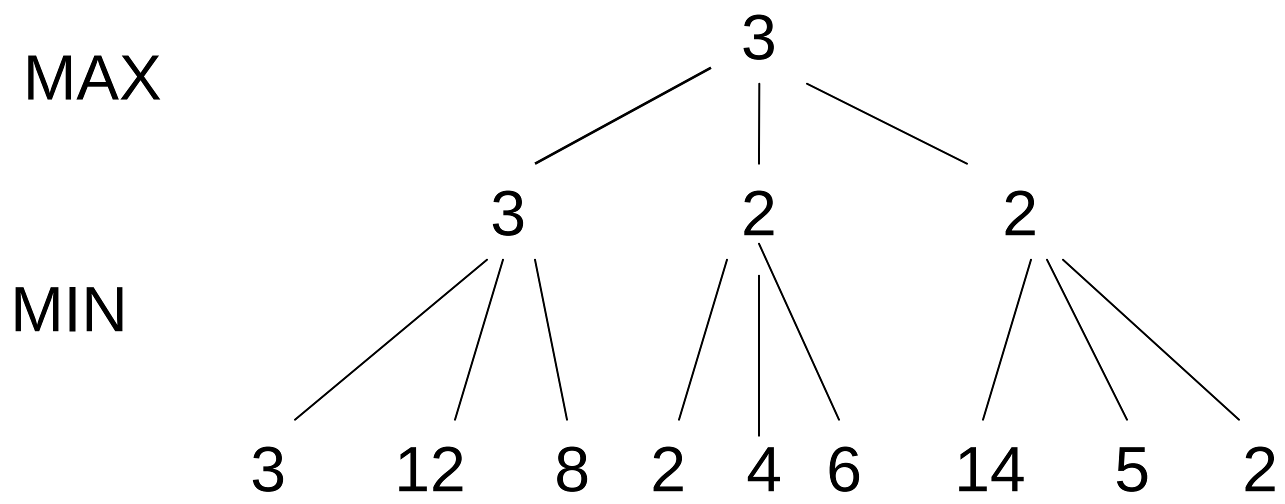
- Minimize opponent's chance
- Maximize your chance

Minimax



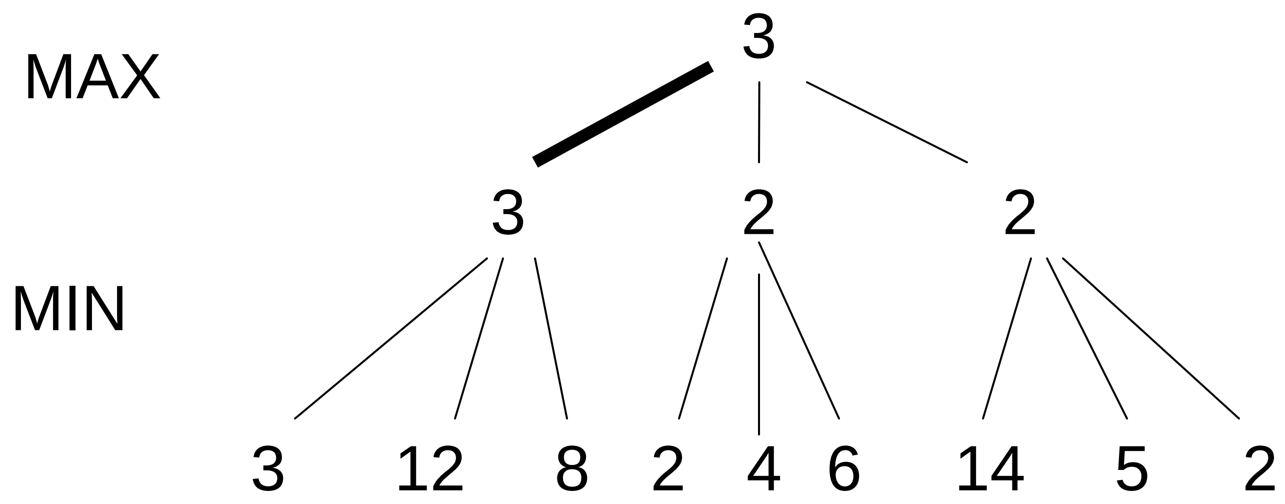
- Minimize opponent's chance
- Maximize your chance

Minimax



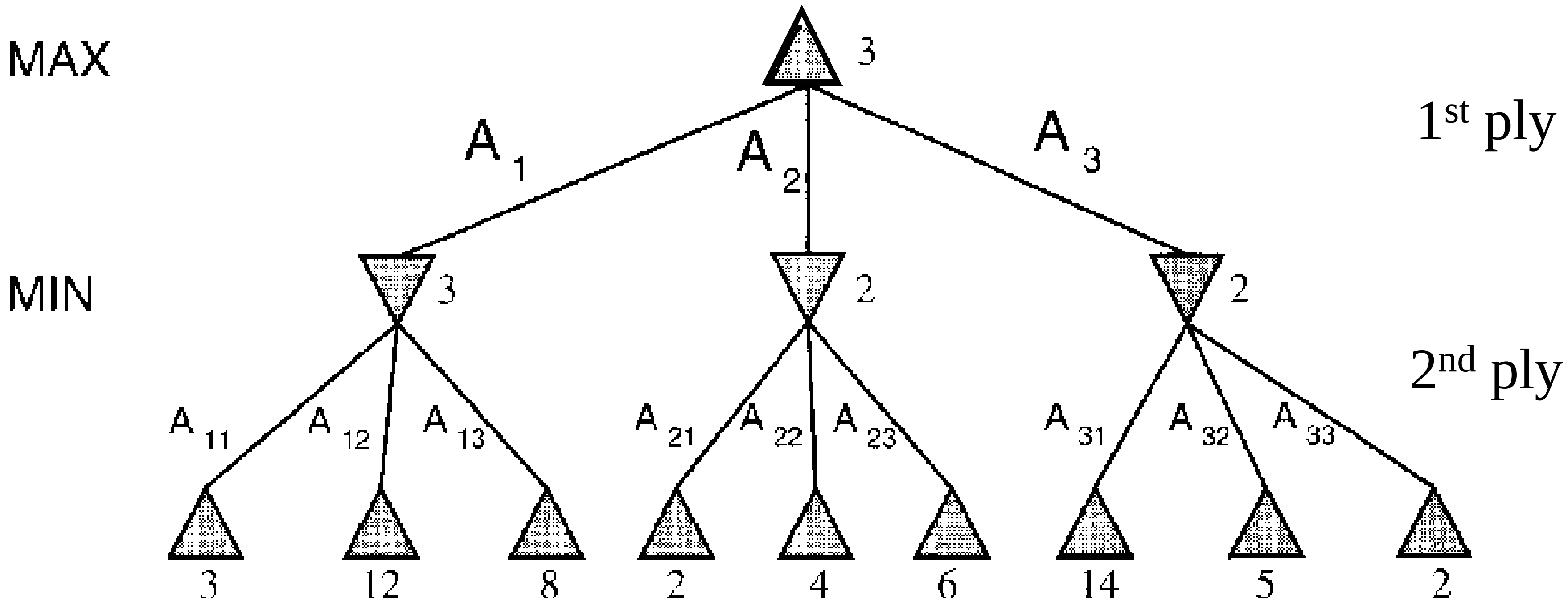
- Minimize opponent's chance
- Maximize your chance

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)
```

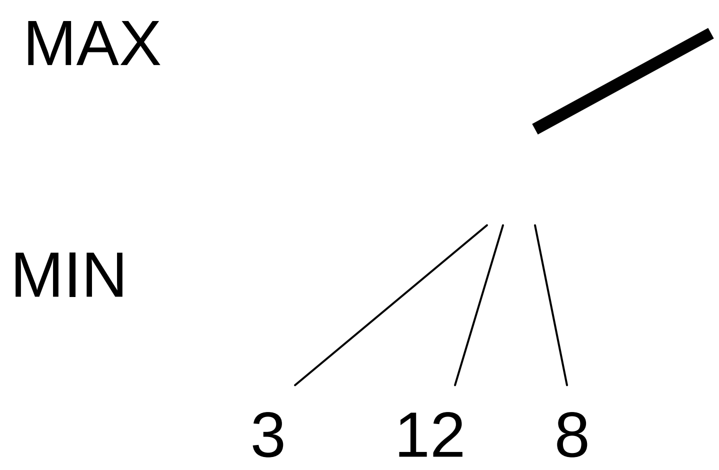
Complete: Yes, for finite state-space

Optimal: Yes

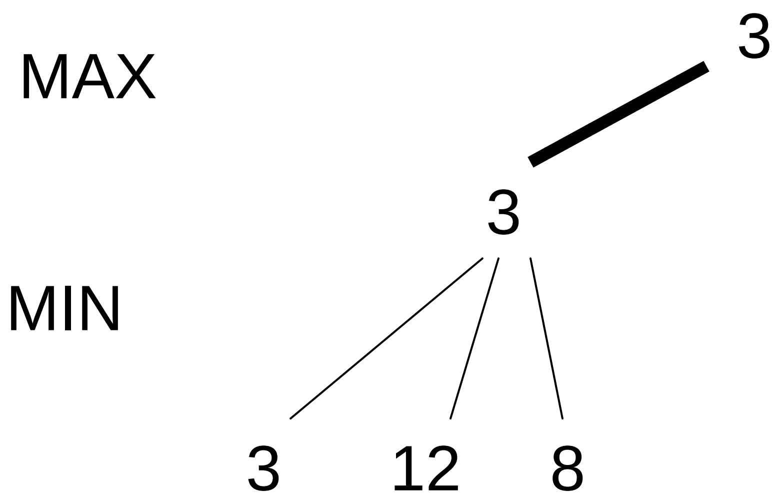
Time complexity: $O(b^m)$

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

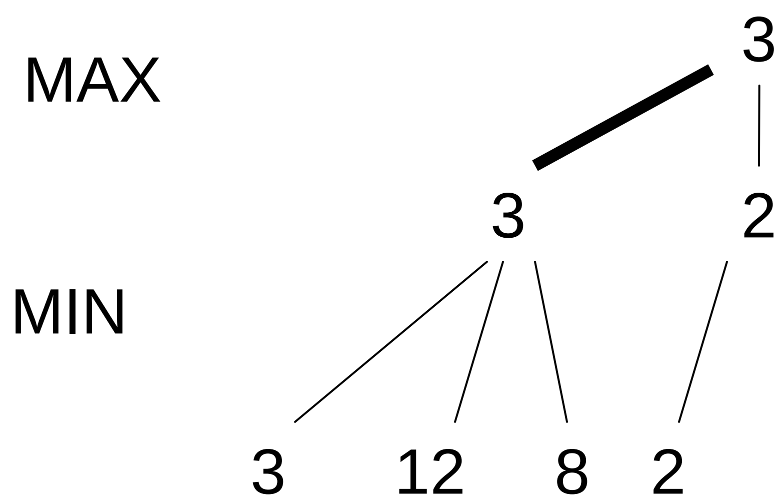
Do We Have To Do All That Work?



Do We Have To Do All That Work?

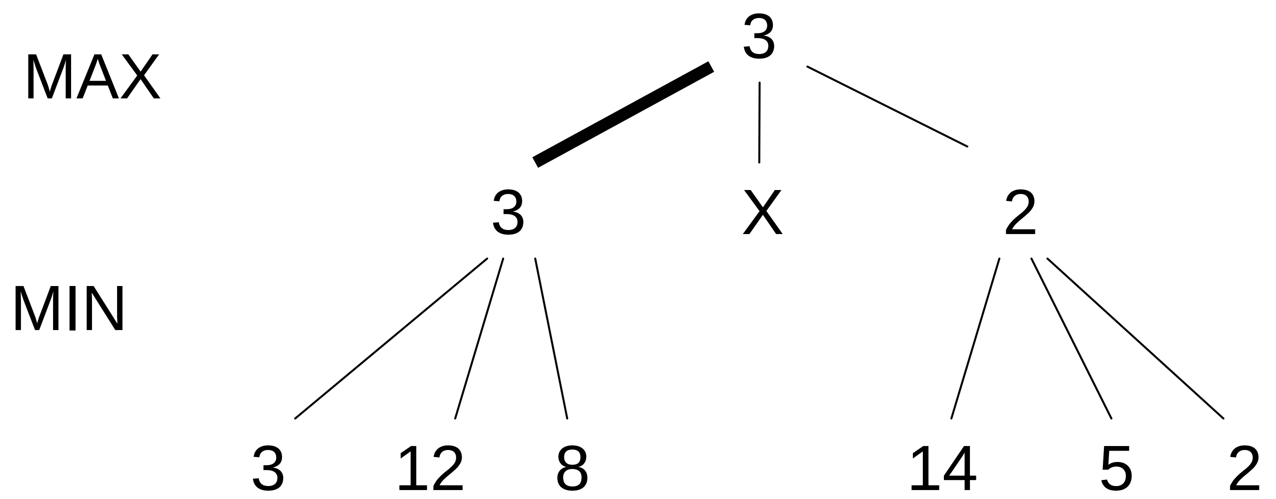


Do We Have To Do All That Work?



Since 2 is smaller than 3, then there is no need for further search

Do We Have To Do All That Work?

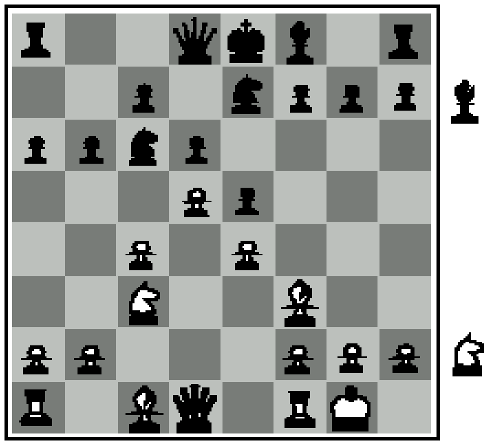


More on this next time: α-β pruning

1. Move evaluation without complete search

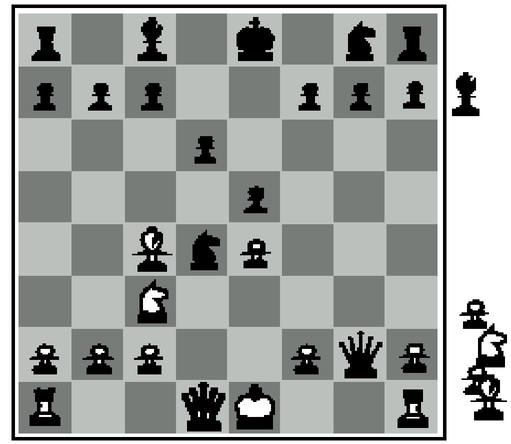
- Complete search is too complex and impractical
- **Evaluation function:** evaluates value of state using heuristics and cuts off search
- **New MINIMAX:**
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility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

Evaluation functions



Black to move

White slightly better



White to move

Black winning

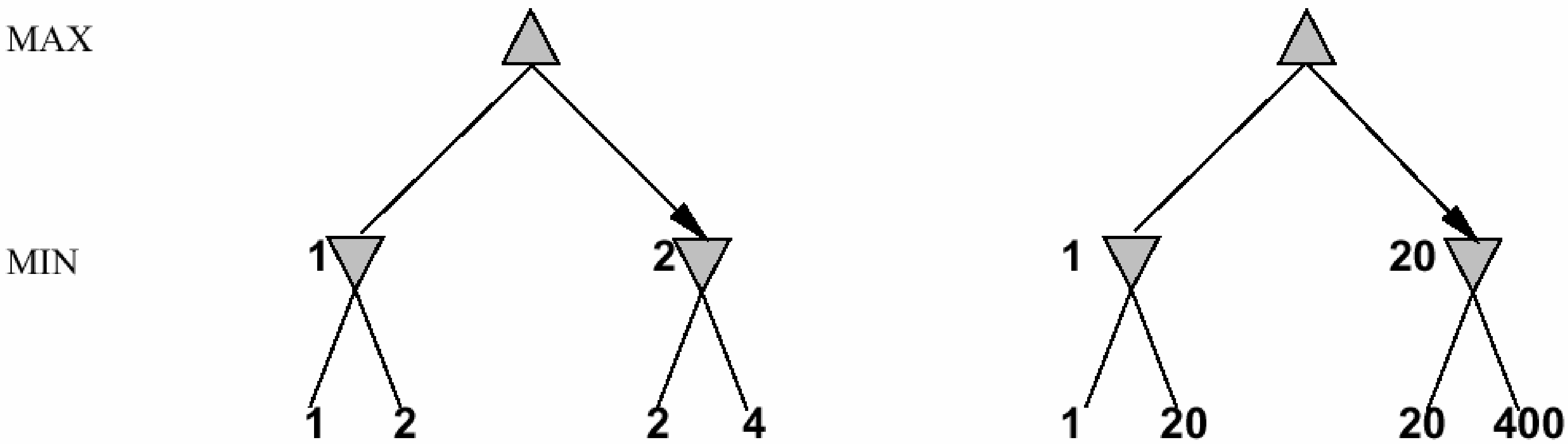
- **Weighted linear evaluation function:** to combine *n* heuristics

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

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

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

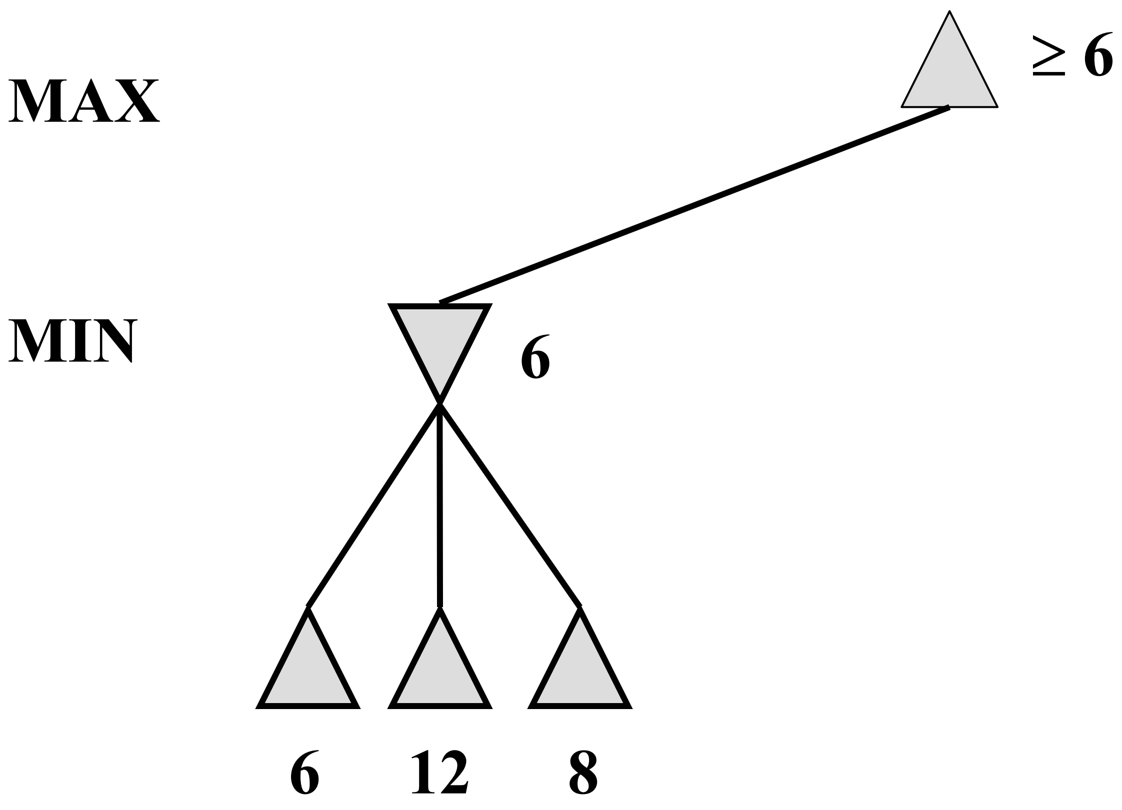
2. α-β pruning: search cutoff

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

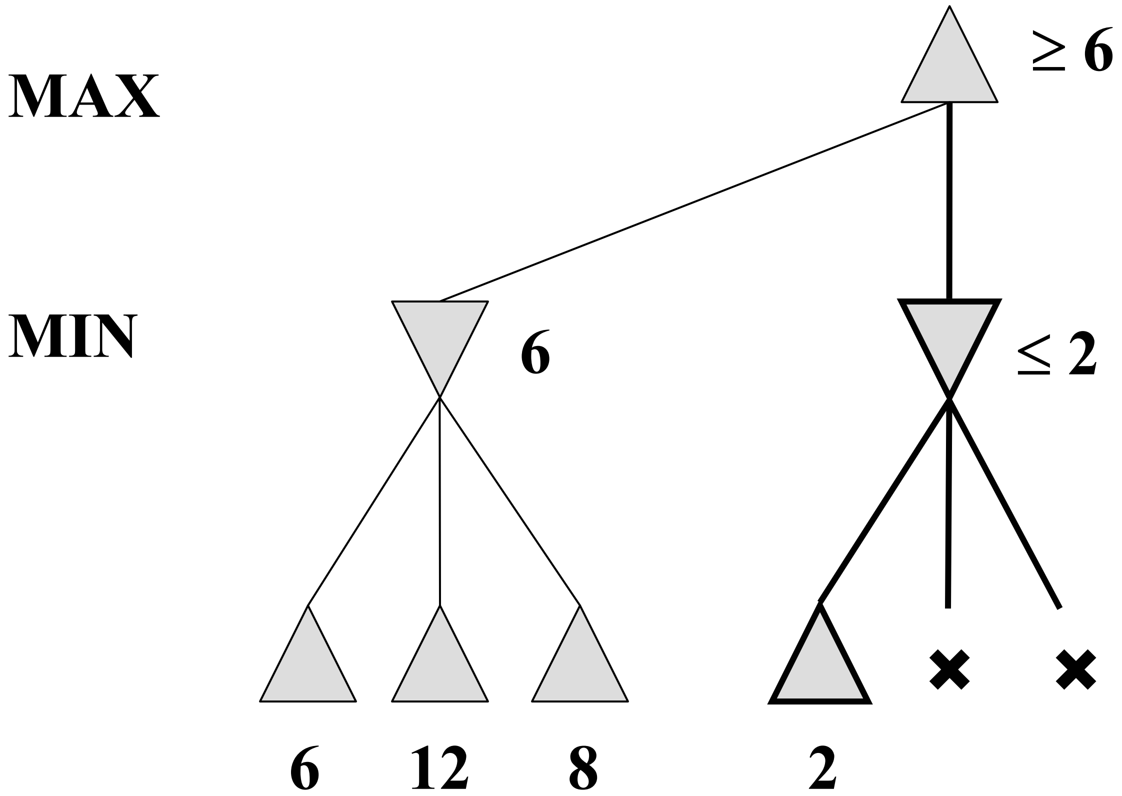
- ***α-β* pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.

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

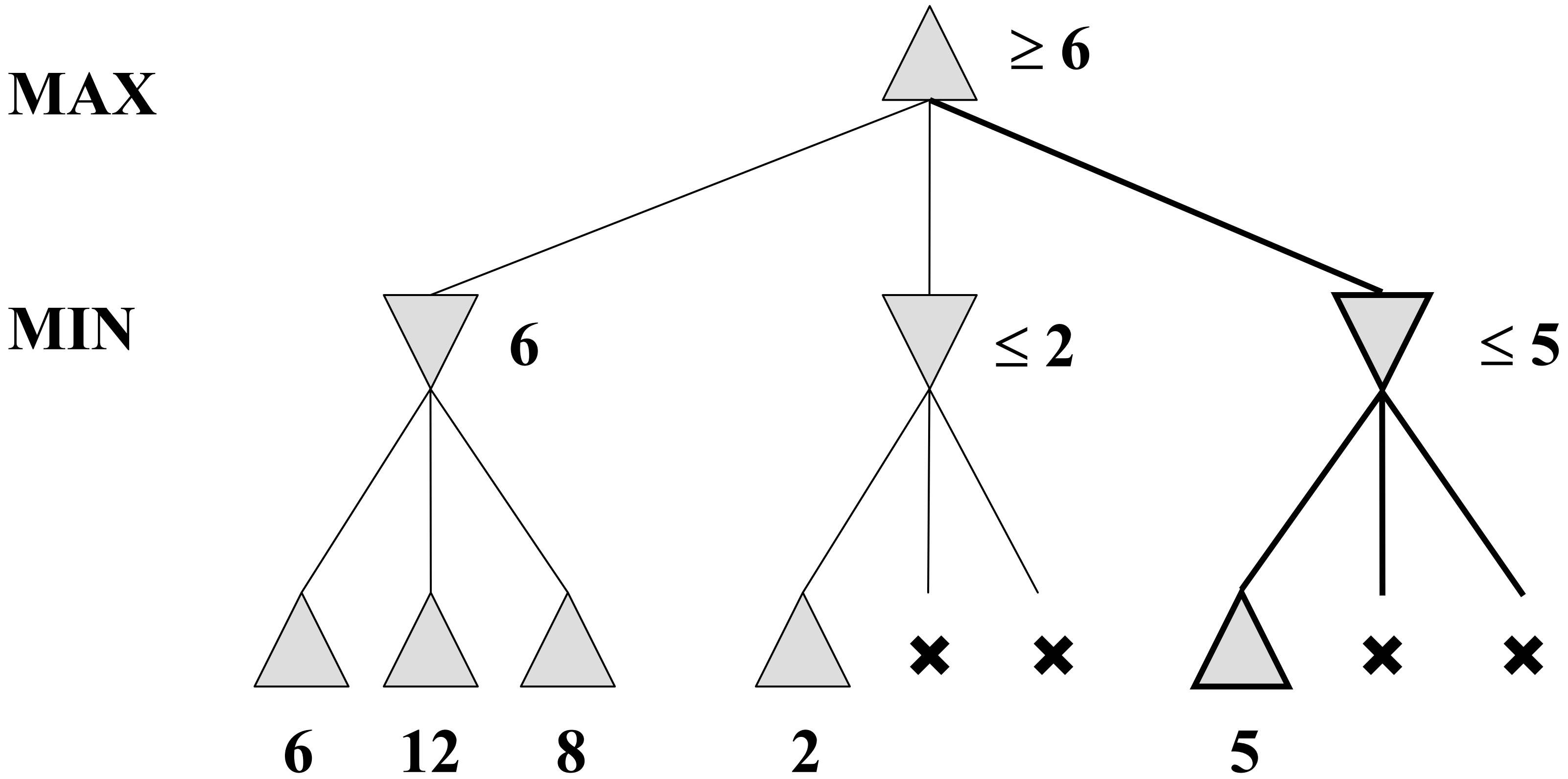
α-β pruning: example



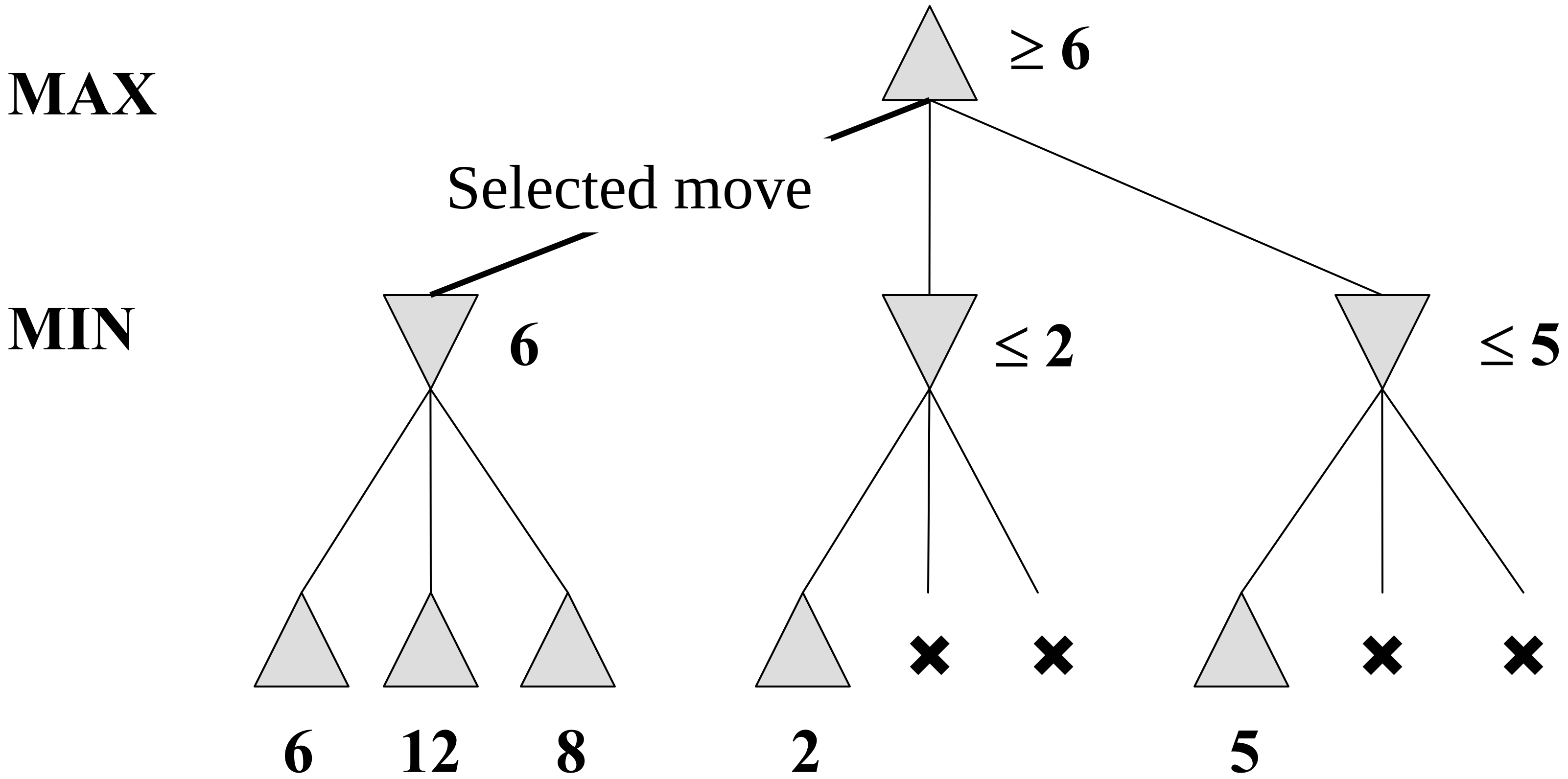
α-β pruning: example



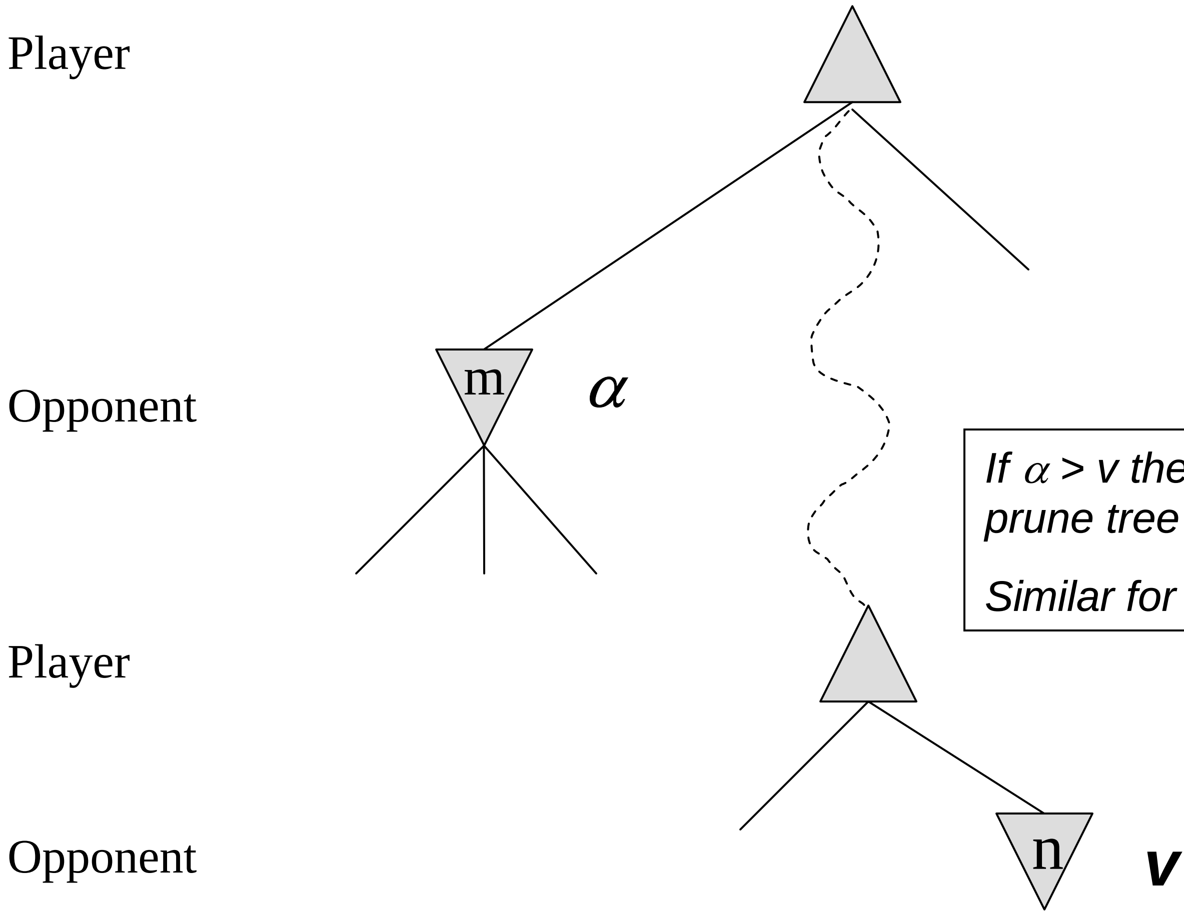
α-β pruning: example



α-β pruning: example



α-β pruning: general principle



If α > v then MAX will chose m so prune tree under n

Similar for β for MIN

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*)

The α-β algorithm:

Basically Minimax + keep track of α, β + prune

function MAX-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

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

---

function MIN-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each s **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(s, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

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.

More on the α-β algorithm: start from Minimax

Basically MINIMAX ~~+ keep track of α, β + prune~~

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
   inputs: state, current state in game
           game, game description
           ~~α, the best score for MAX along the path to state~~
           ~~β, the best score for MIN along the path to state~~

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       α ← MAX(α, MIN-VALUE(s, game, α, β))
       ~~if α ≥ β then return β~~
   end
   return α
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       β ← MIN(β, MAX-VALUE(s, game, α, β))
       ~~if β ≤ α then return α~~
   end
   return β
```

Remember: 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.)

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

More on the α-β algorithm: start from Minimax

Basically MINIMAX + keep track of α, β + prune

function MAX-VALUE(*state, game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

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

---

function MIN-VALUE(*state, game*, α, β) **returns** the minimax value of *state*

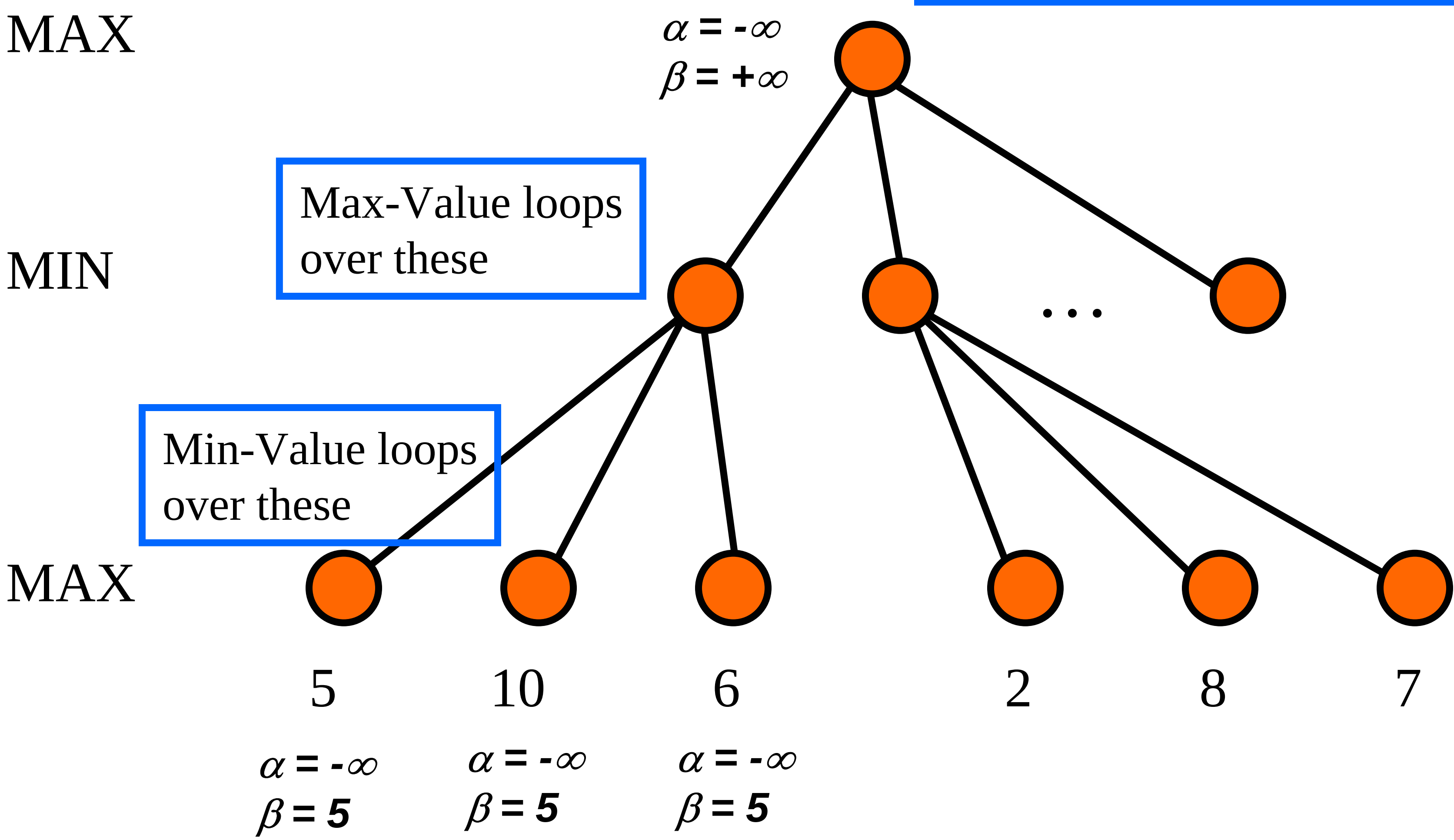
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each s **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(s, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

Note: These are both Local variables. At the Start of the algorithm, We initialize them to ***$\alpha = -\infty$ and $\beta = +\infty$***

More on the α-β algorithm

In Min-Value:

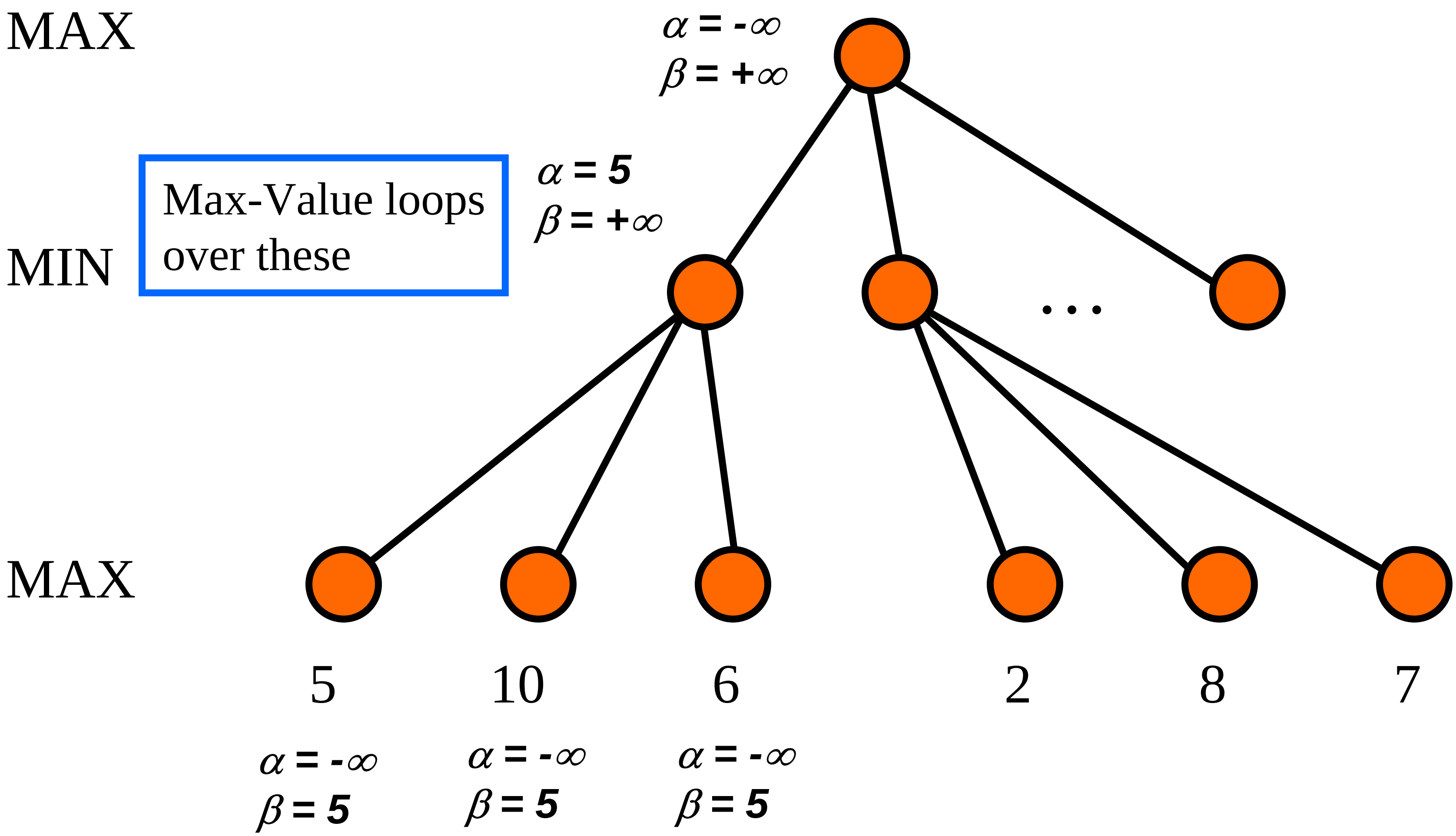
> **for each** s **in** SUCCESSORS($state$) **do**
> $\beta \leftarrow$ MIN(β, MAX-VALUE($s, game, \alpha, \beta$))
>   **if** $\beta \leq \alpha$ **then return** α
> **end**
> **return** β



More on the α-β algorithm

In Max-Value:

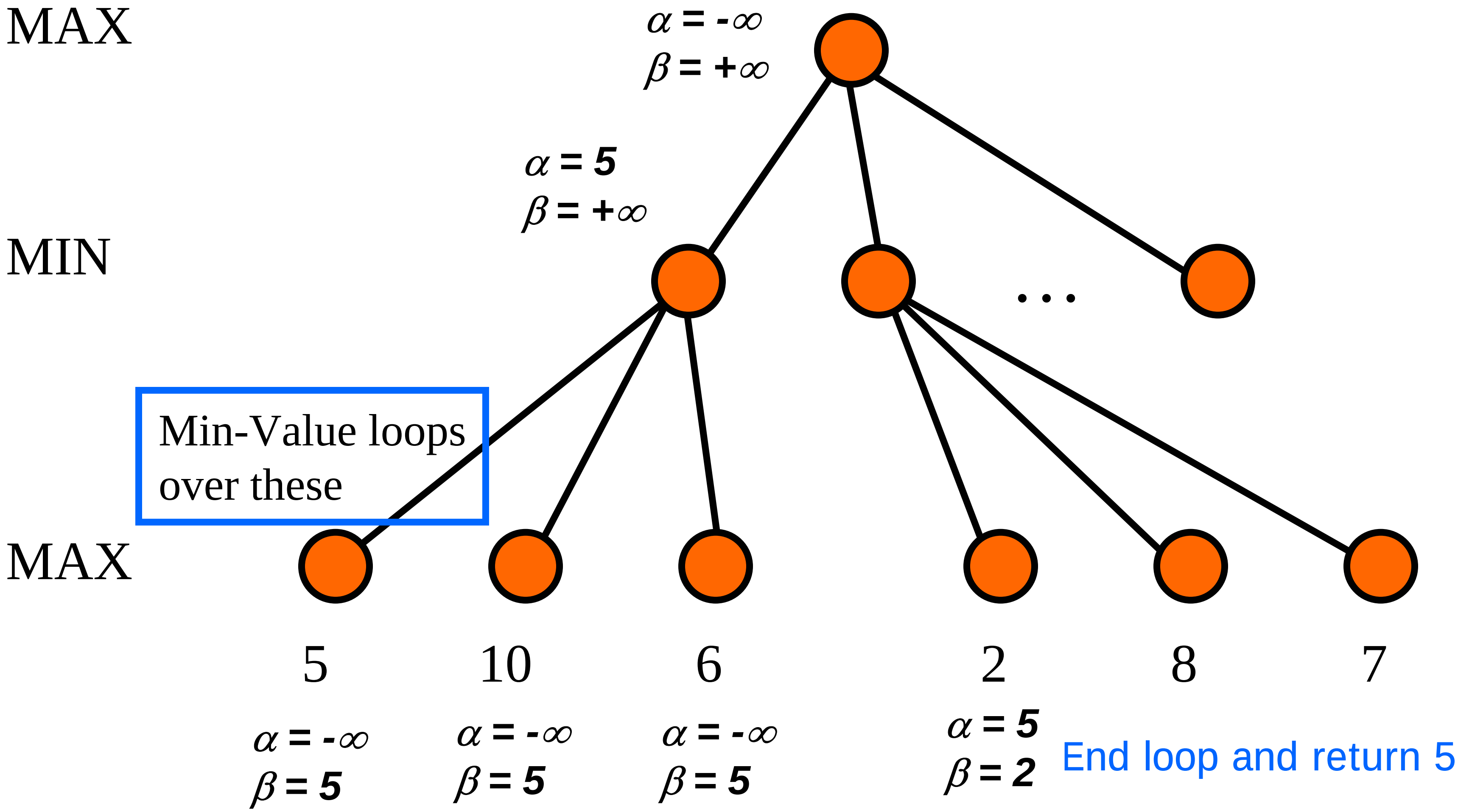
```
for each s in SUCCESSORS(state) do
    α ← MAX(α, MIN-VALUE(s, game, α, β))
    if α ≥ β then return β
end
return α
```



More on the α-β algorithm

In Min-Value:

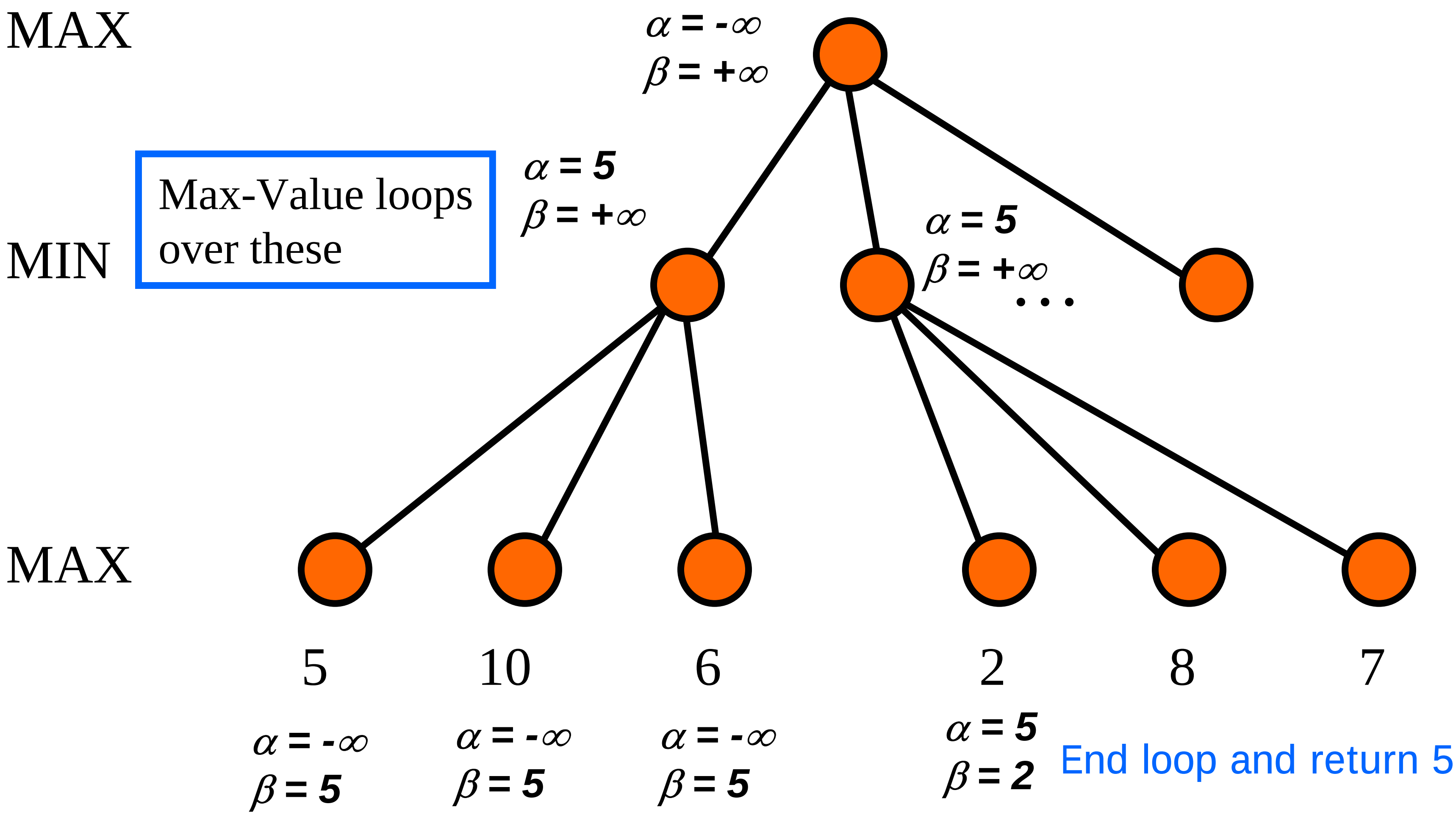
for each s **in** SUCCESSORS($state$) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE($s, game, \alpha, \beta$))
    if $\beta \le \alpha$ **then return** α
end
return β



More on the α-β algorithm

In Max-Value:

```
for each s in SUCCESSORS(state) do
    α ← MAX(α, MIN-VALUE(s, game, α, β))
    if α ≥ β then return β
end
return α
```



Another way to understand the algorithm

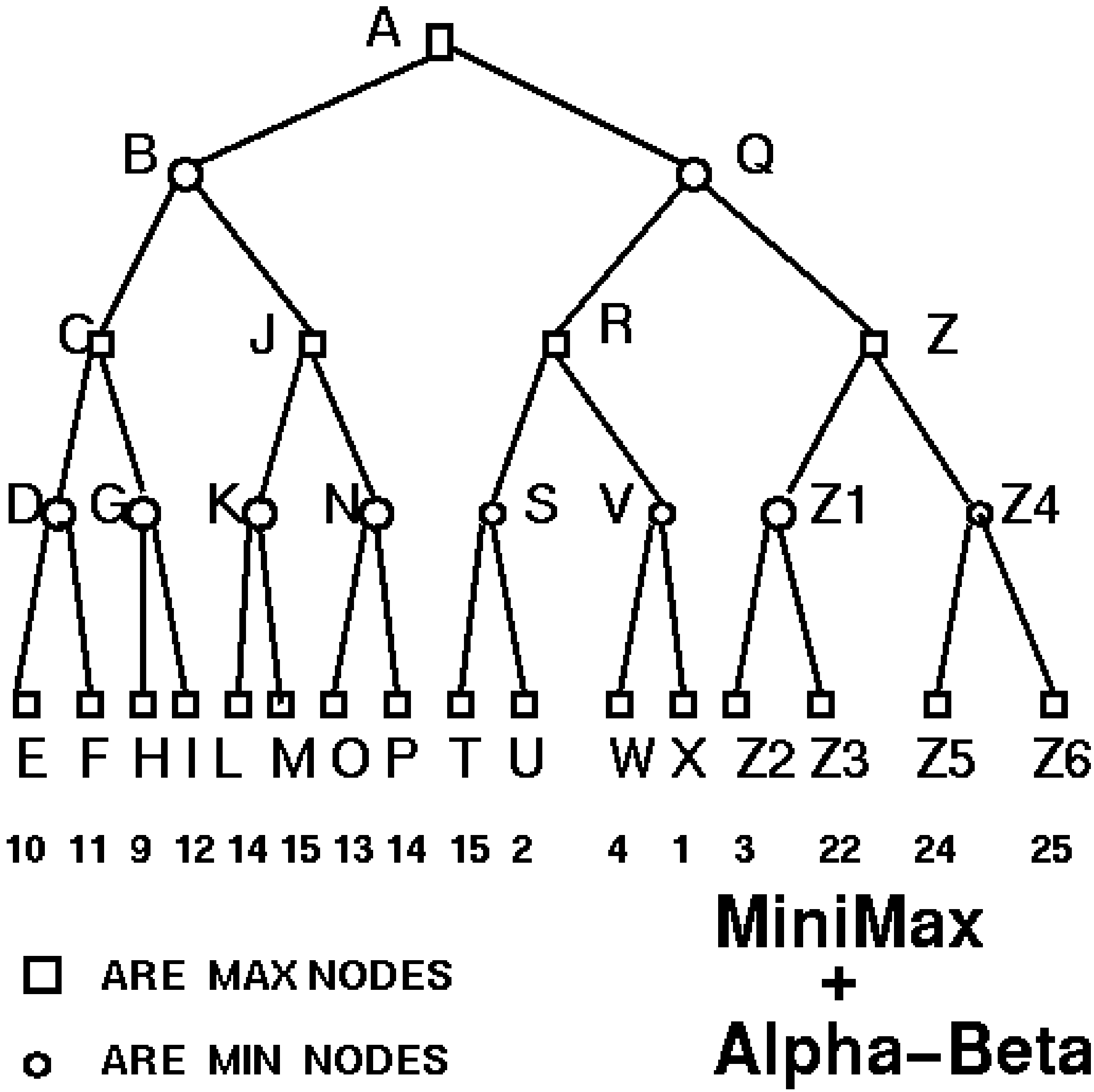
- From:

http://yoda.cis.temple.edu:8080/UGAIWWW/lectures95/search/alpha-beta.html

- For a given node N,

α is the value of N to MAX
β is the value of N to MIN

Example



MiniMax
+
Alpha–Beta

α-β algorithm:

Basically MINIMAX + keep track of α, β + prune

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
   inputs: state, current state in game
           game, game description
           α, the best score for MAX along the path to state
           β, the best score for MIN along the path to state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       α ← MAX(α, MIN-VALUE(s, game, α, β))
       if α ≥ β then return β
   end
   return α
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       β ← MIN(β, MAX-VALUE(s, game, α, β))
       if β ≤ α then return α
   end
   return β
```

Solution

NODE	TYPE	ALPHA	BETA	SCORE
A	Max	-I	+I	
B	Min	-I	+I	
C	Max	-I	+I	
D	Min	-I	+I	
E	Max	10	10	10
D	Min	-I	10	
F	Max	11	11	11
D	Min	-I	10	10
C	Max	10	+I	
G	Min	10	+I	
H	Max	9	9	9
G	Min	10	9	9
C	Max	10	+I	10
B	Min	-I	10	
J	Max	-I	10	
K	Min	-I	10	
L	Max	14	14	14
K	Min	-I	10	10
…				

NODE	TYPE	ALPHA	BETA	SCORE
…				
J	Max	10	10	10
B	Min	-I	10	10
A	Max	10	+I	
Q	Min	10	+I	
R	Max	10	+I	
S	Min	10	+I	
T	Max	5	5	5
S	Min	10	5	5
R	Max	10	+I	
V	Min	10	+I	
W	Max	4	4	4
V	Min	10	4	4
R	Max	10	+I	10
Q	Min	10	10	10
A	Max	10	10	10

State-of-the-art for deterministic games

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.

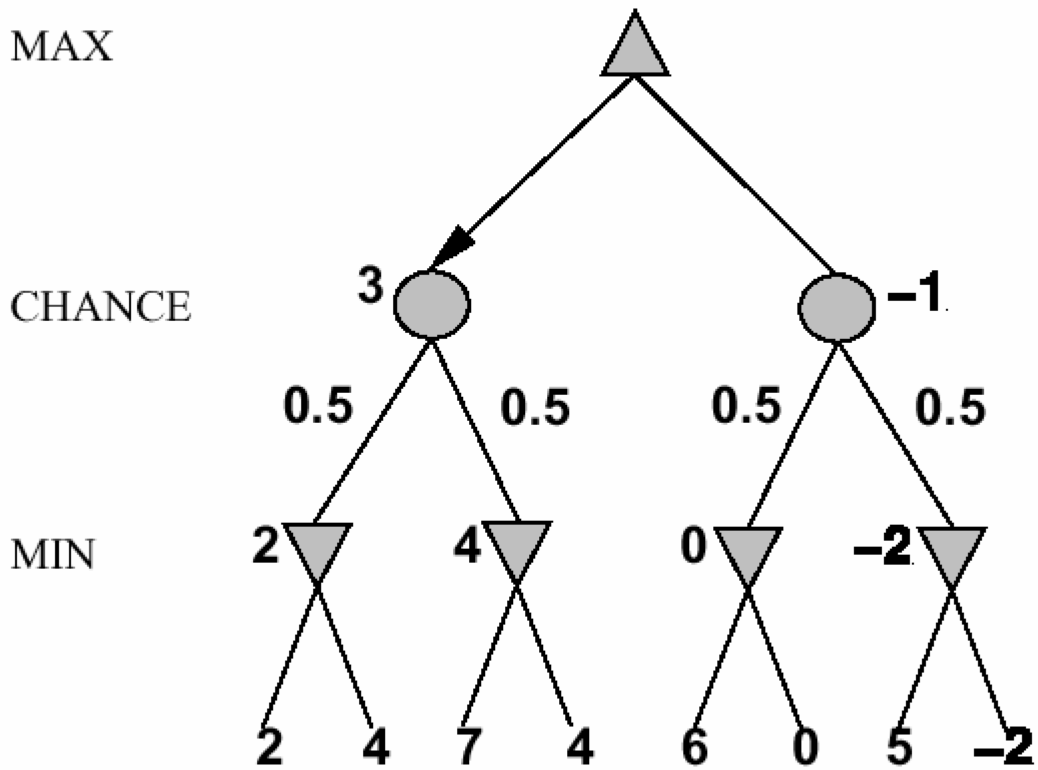
Othello: human champions refuse to compete against computers, who are too good.

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 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??

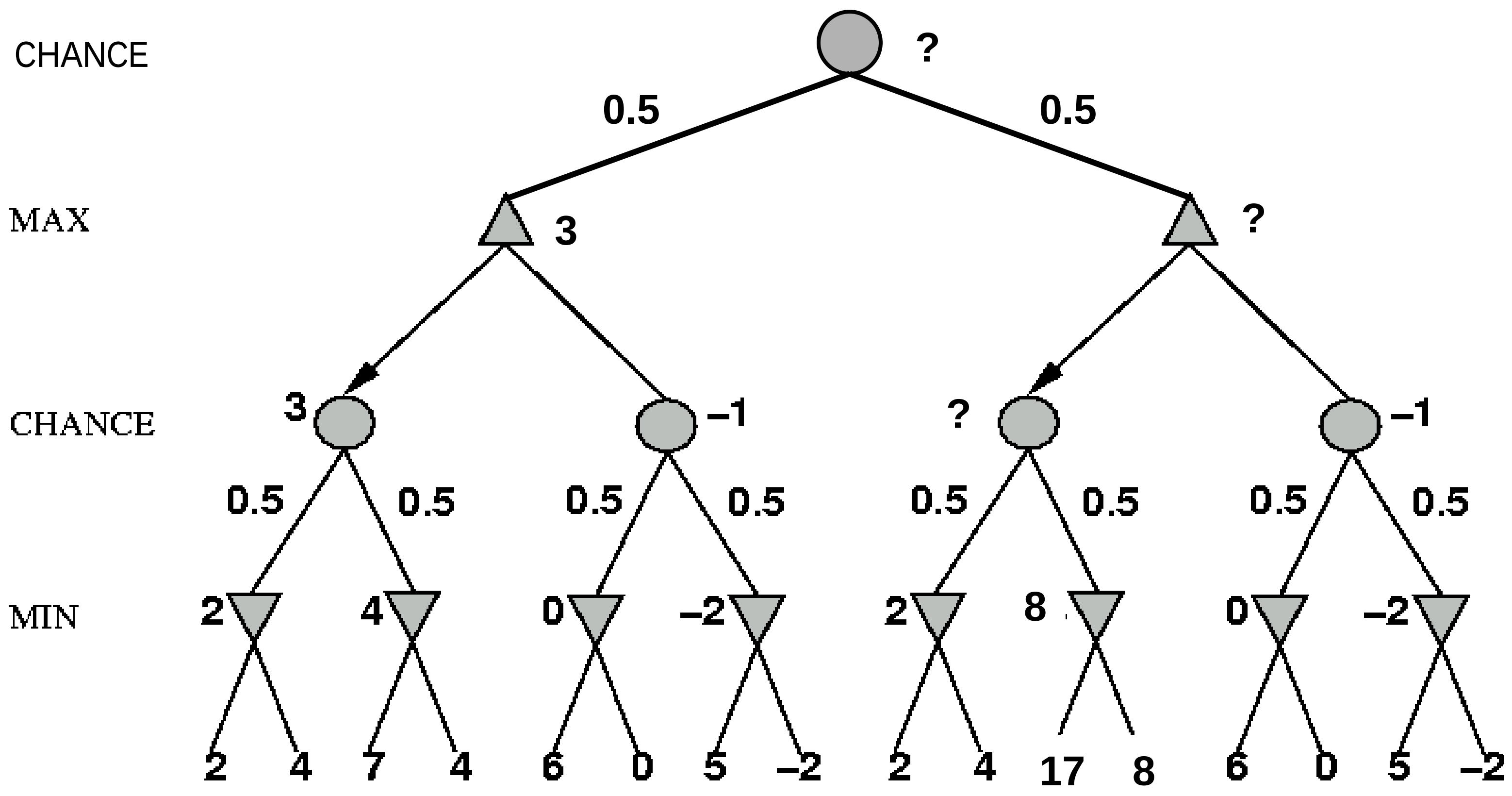
Remember: Minimax algorithm

```
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]
```

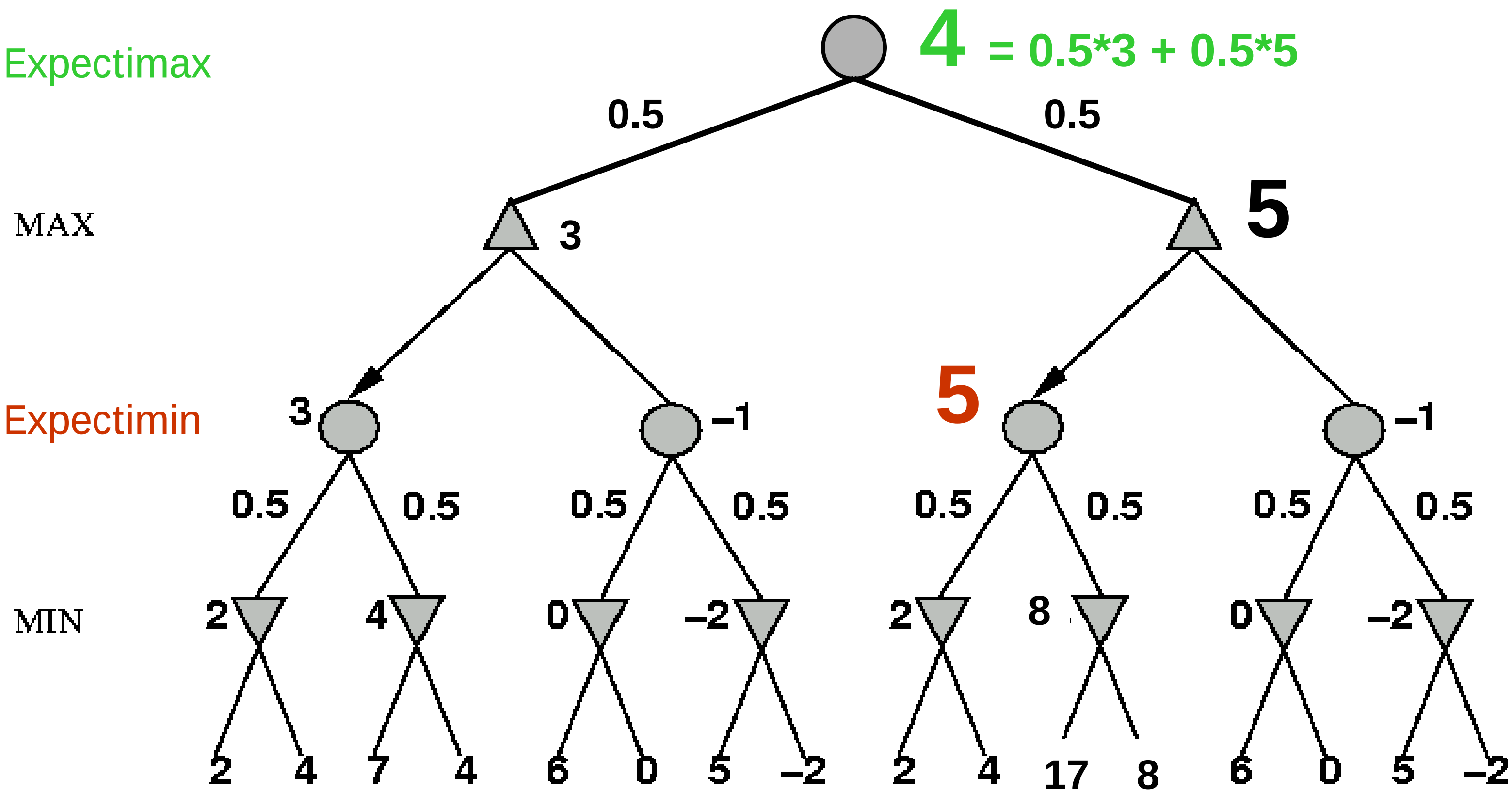
```
function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)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

Nondeterministic games: the element of chance

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

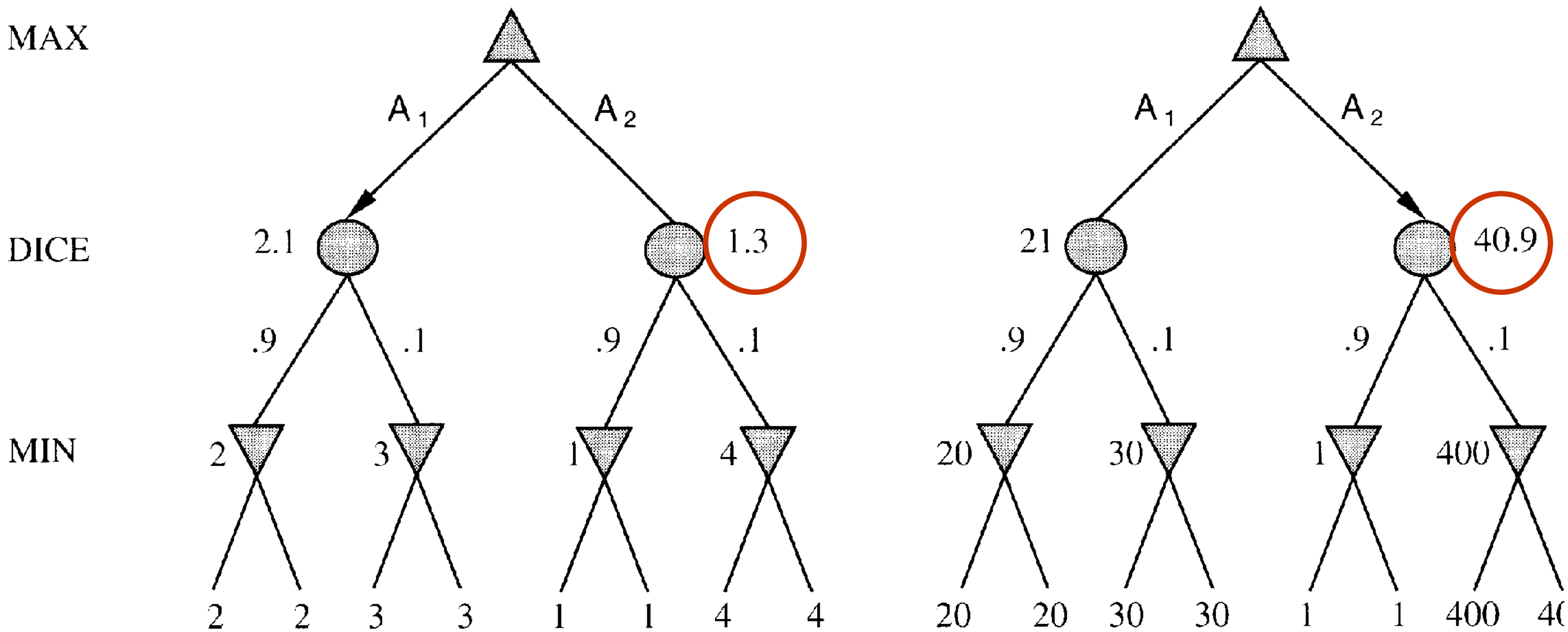


Nondeterministic games: the element of chance



Evaluation functions: Exact values DO matter

Order-preserving transformation do not necessarily behave the same!



State-of-the-art for nondeterministic games

Dice rolls increase b: 21 possible rolls with 2 dice
Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
$\Rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

Summary

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

- ♢ perfection is unattainable ⇒ must approximate
- ♢ good idea to think about what to think about
- ♢ uncertainty constrains the assignment of values to states

Games are to AI as grand prix racing is to automobile design

Exercise: Game Playing

Consider the following game tree in which the evaluation function values are shown below each leaf node. Assume that the root node corresponds to the maximizing player. Assume the search always visits children left-to-right.

(a) Compute the backed-up values computed by the minimax algorithm. Show your answer by writing values at the appropriate nodes in the above tree.

(b) Compute the backed-up values computed by the alpha-beta algorithm. What nodes will not be examined by the alpha-beta pruning algorithm?

(c) What move should Max choose once the values have been backed-up all the way?