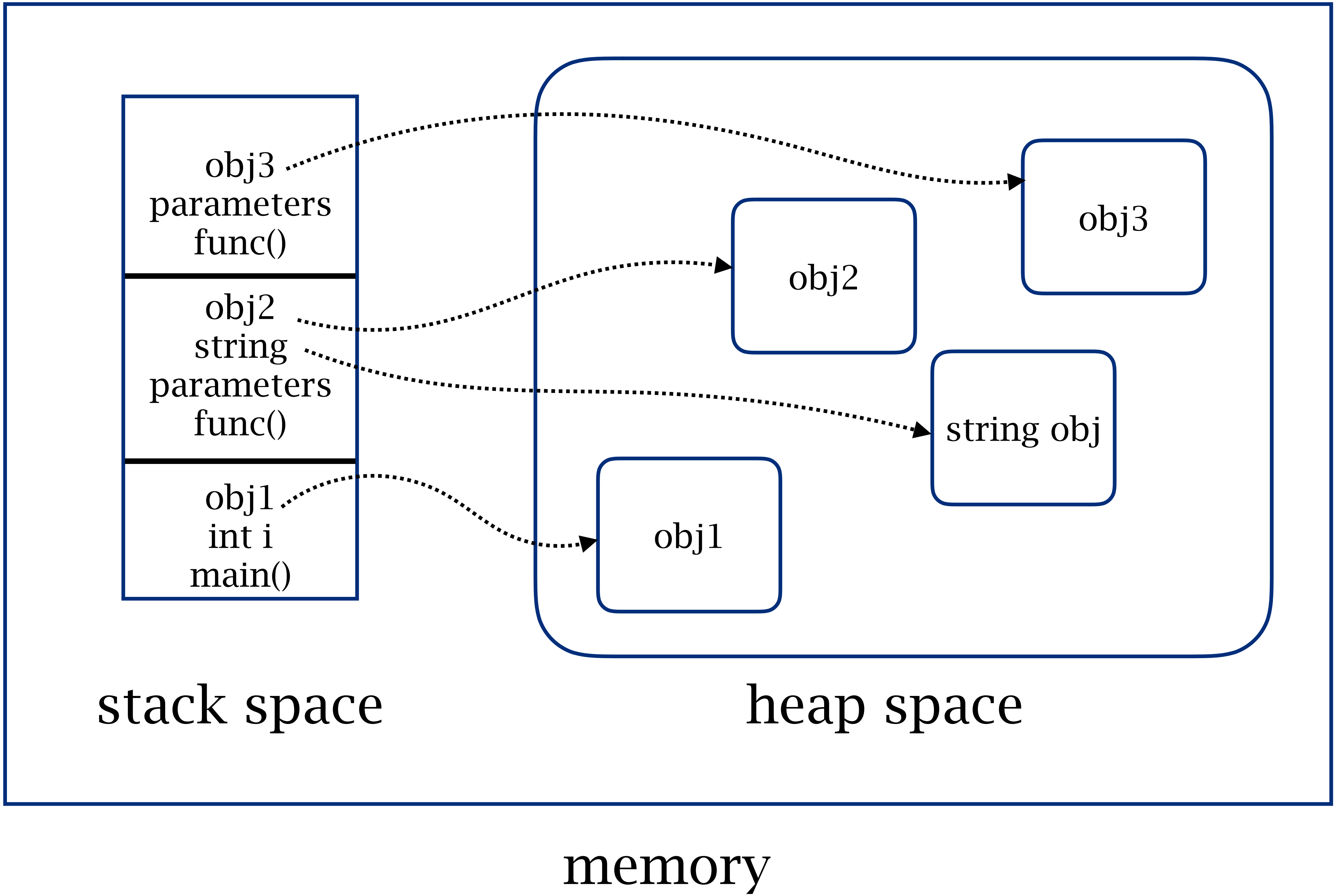
Lecture 4: Search 3

Previously...

Path-based search

- Uninformed search
  - Depth-first, breadth first, uniform-cost search
- Informed search
  - Best-first, **A* search**

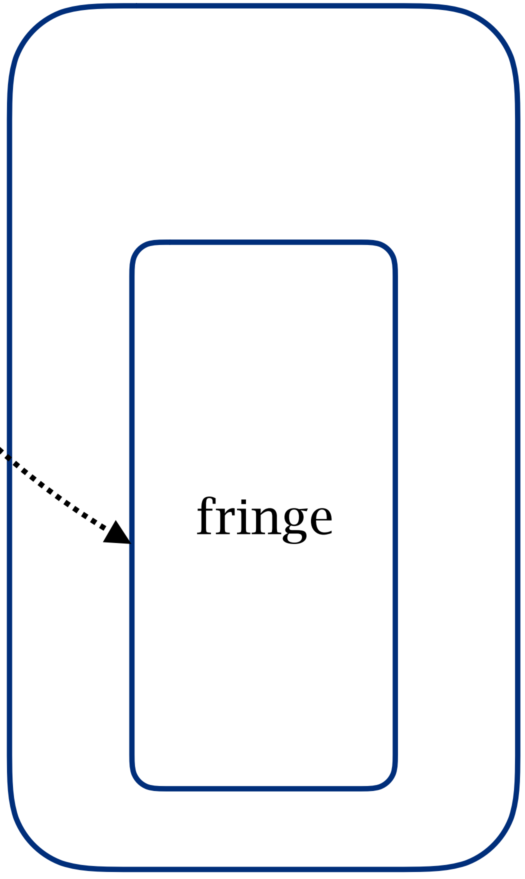
Stack and heap memory space



Deep-first search using heap

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```



heap space

flexible memory useage

Deep-first search using stack

```
function Tree-Search(node)
    if node has goal then return true
    for each action, result in Successor-Fn(problem, node) do
        s <- make Node from node
        hasgoal = Tree-Search(s)
        if hasgoal then return true
    end for
return false
```

return true node Tree-Search()
s node Tree-Search()
s node Tree-Search()

stack space

simple to code, risk of stack-overflow

Adversarial search

Competitive environments: Game

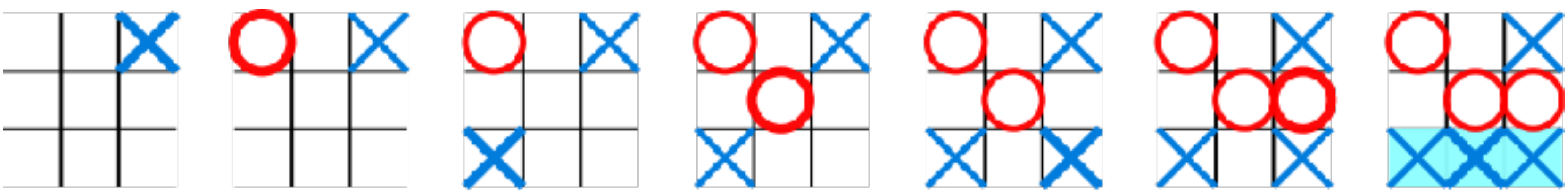
the agents' goals are in conflict

We consider:
* two players
* zero-sum games

Type of games:
* deterministic v.s. chance
* perfect v.s. partially observable information

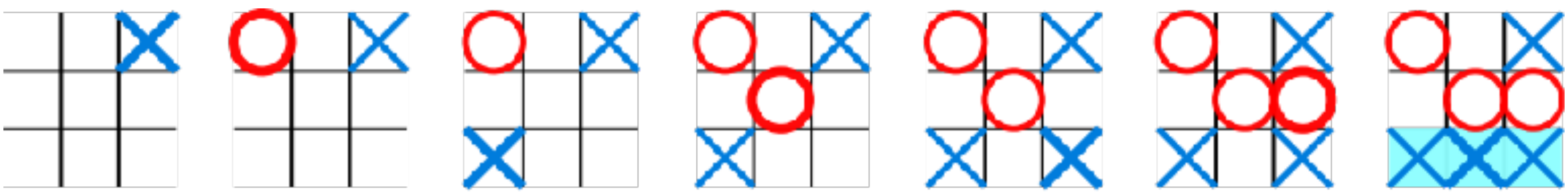
Example

两人轮流在一有九格方盘上划加字或圆圈，谁先把三个同一记号排成横线、直线、斜线，即是胜者



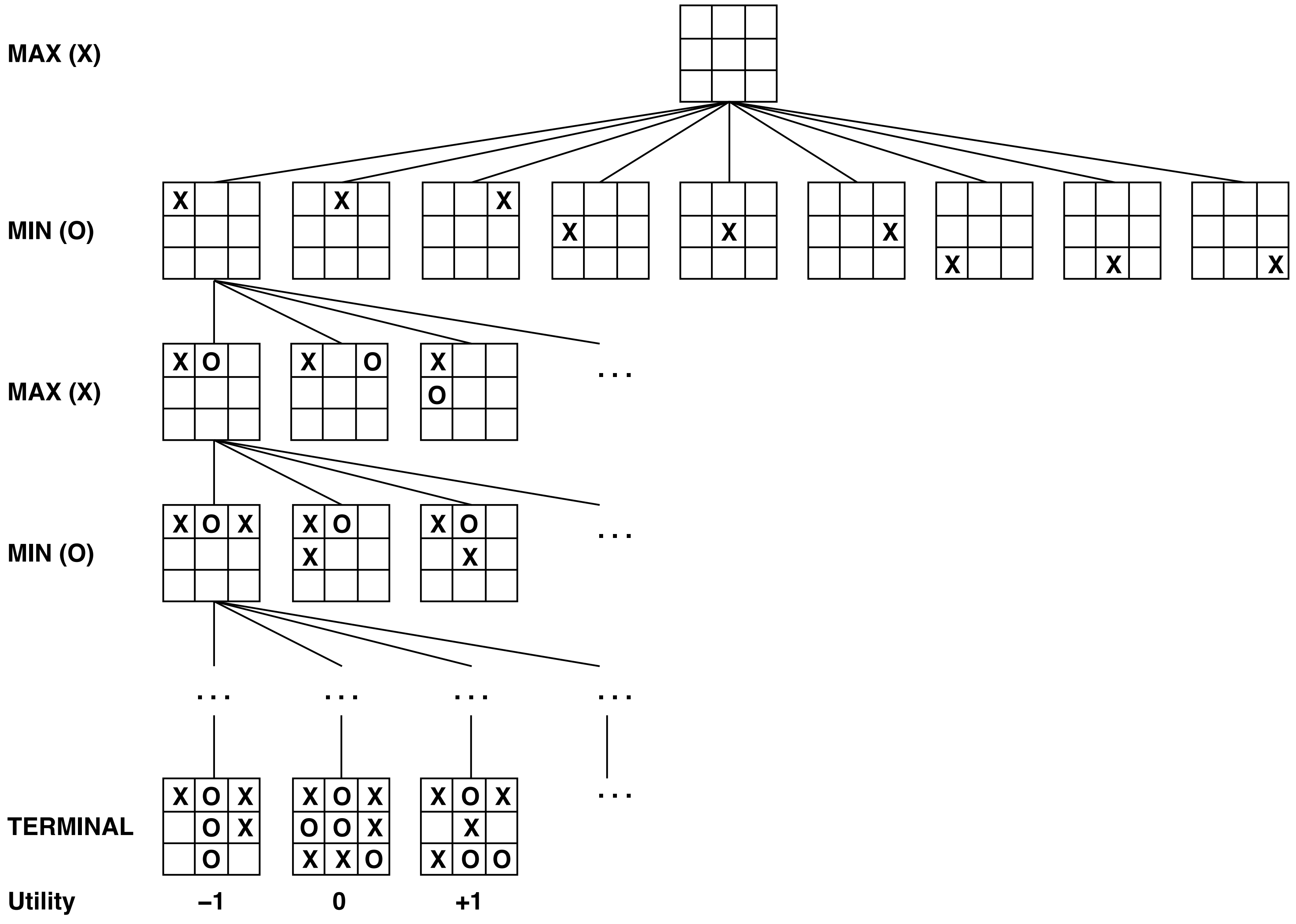
Definition of a game

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function),



two players: MAX and MIN

Tic-tac-toe search tree

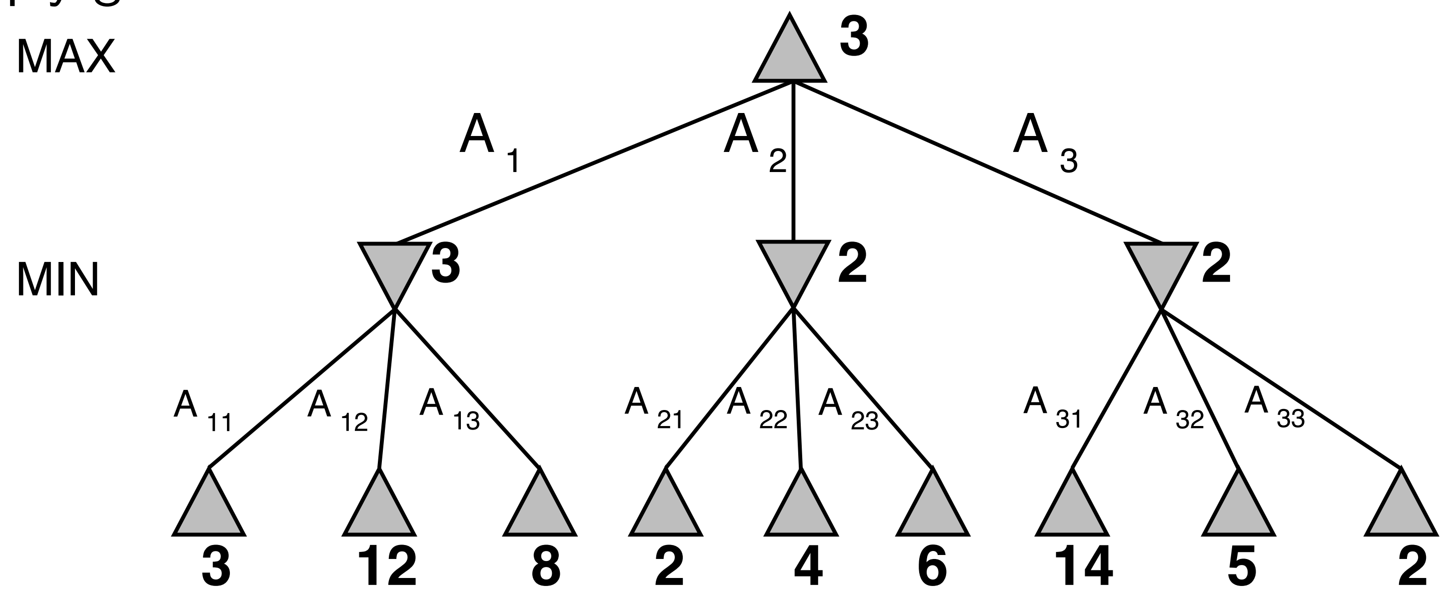


Optimal decision in games

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



$$
\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}
$$

Minimax algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game

  return the a in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(a, *state*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*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 TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do** $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
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
```

Properties of Minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

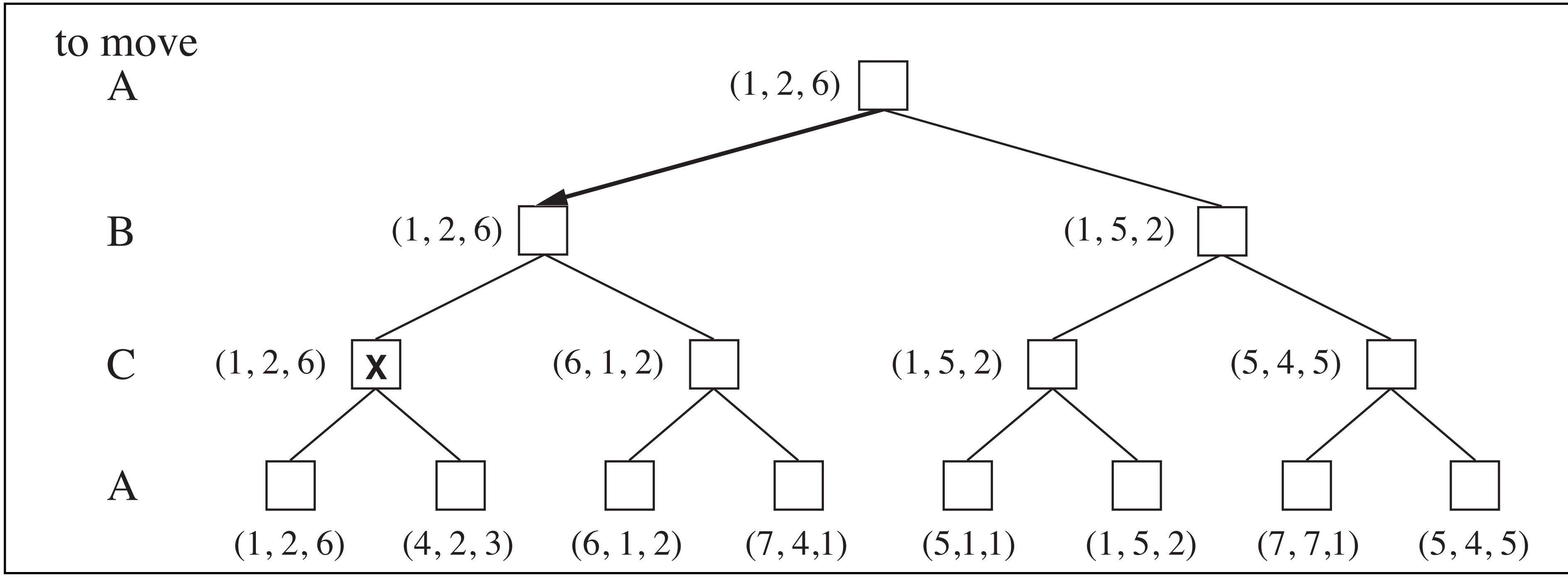
Time complexity?? $O(b^m)$

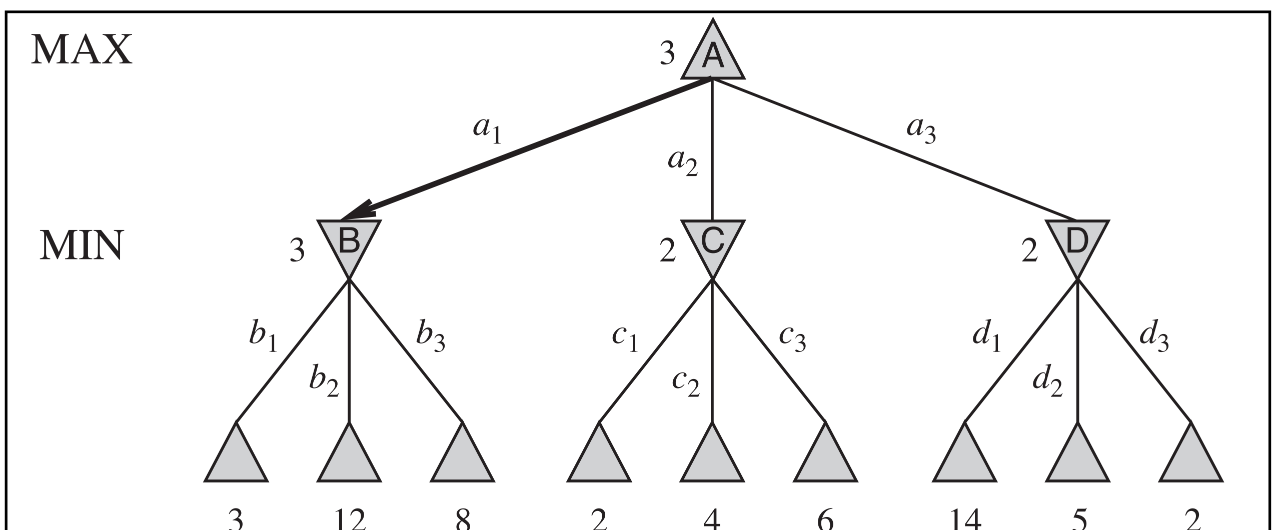
Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

Multiple players

a vector $\langle v_A, v_B, v_C \rangle$ is used for 3 players





Minimax algorithm — Redundancy

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game

  return the a in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(a, *state*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do** $v \leftarrow$ MAX(v, MIN-VALUE(s))
  return v

Vmax=5

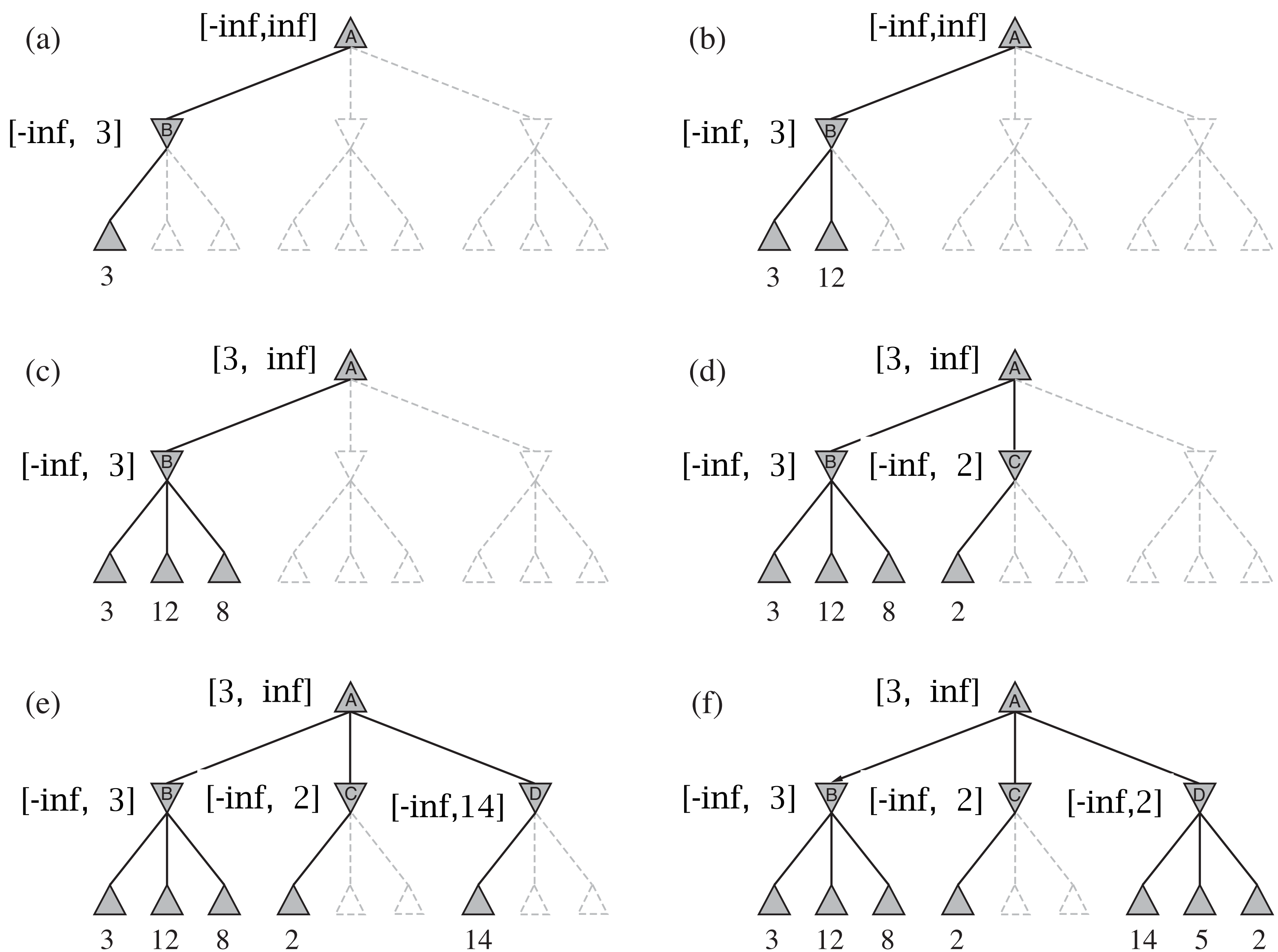
---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do** $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

Vmin=3

Minimax algorithm — Redundancy

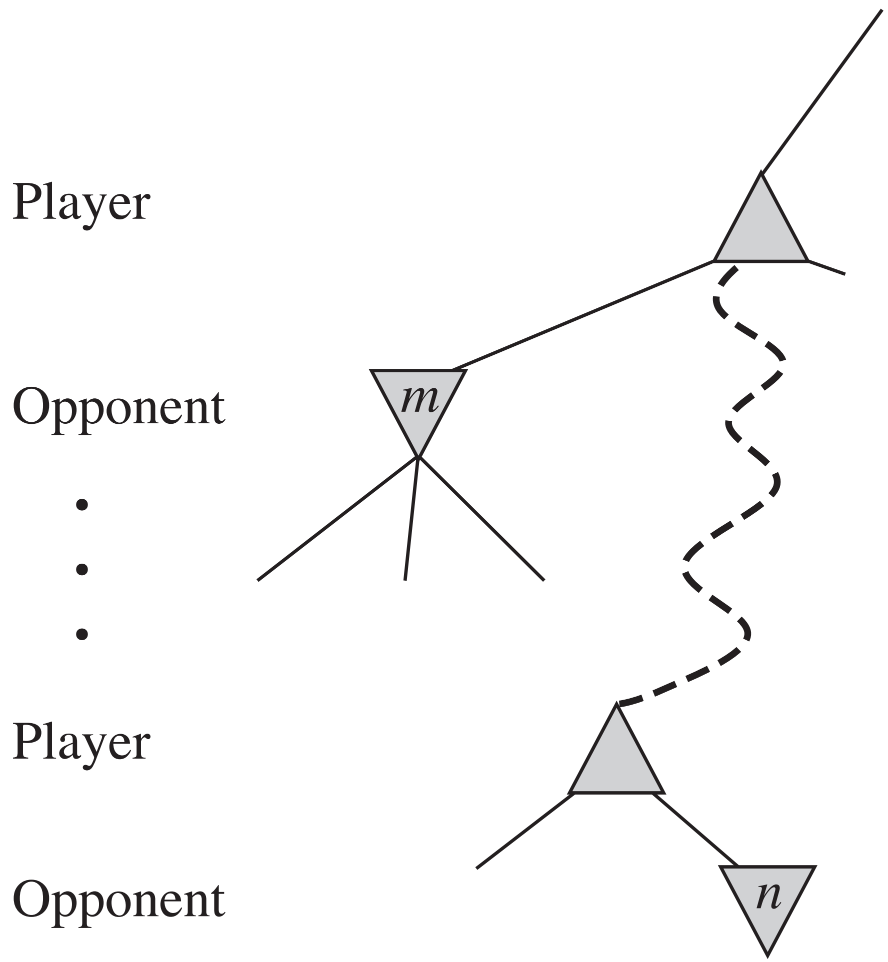
[Vmax,Vmin]



Alpha-Beta pruning

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.



Alpha-Beta pruning

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

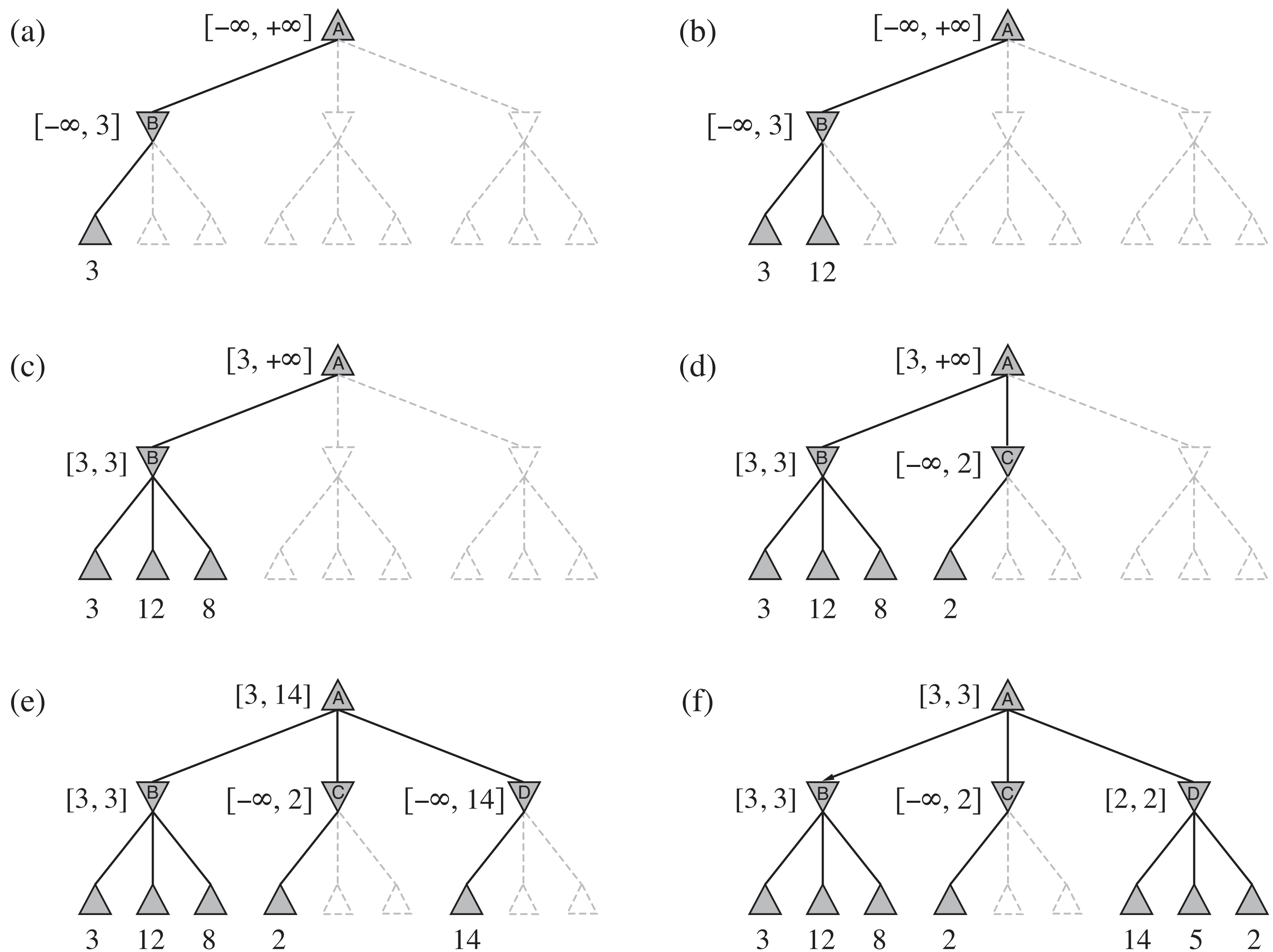
function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta pruning

down: [v_{max},v_{min}]   up: [alpha, beta]



Properties of alpha-beta

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

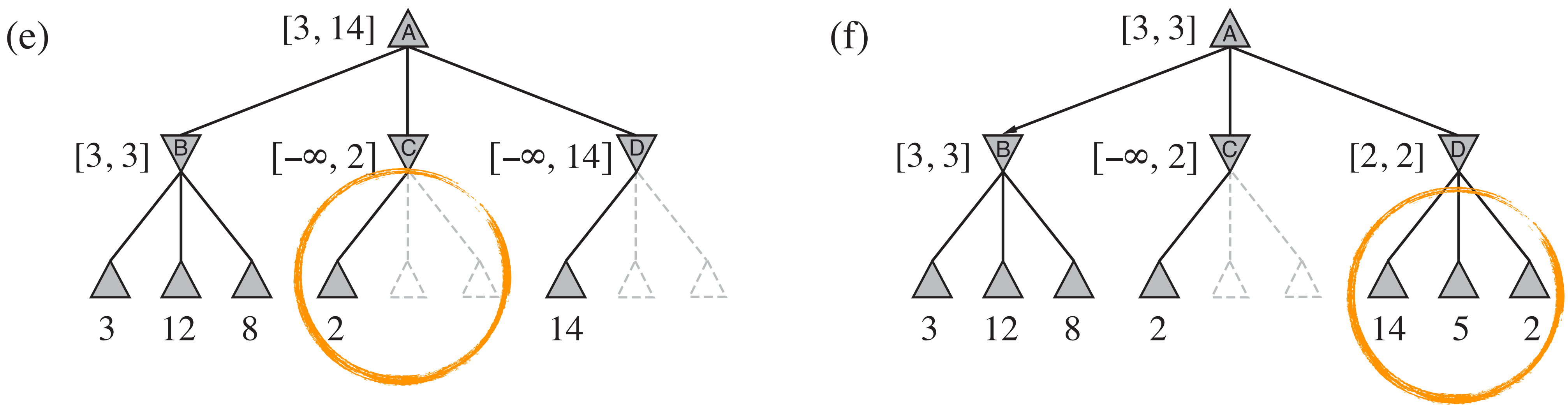
With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

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

Unfortunately, 35^{50} is still impossible!

The search order is important

it might be worthwhile to try to examine first the successors that are likely to be best



Resource limits

Standard approach:

- Use CUTOFF-TEST instead of TERMINAL-TEST

  e.g., depth limit (perhaps add quiescence search)
- Use EVAL instead of UTILITY

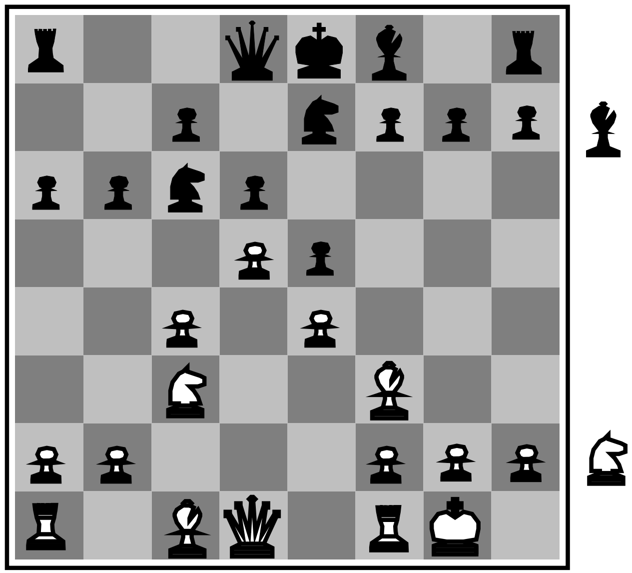
  i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds, explore 10^4 nodes/second

$\Rightarrow$ 10^6 nodes per move $\approx 35^{8/2}$

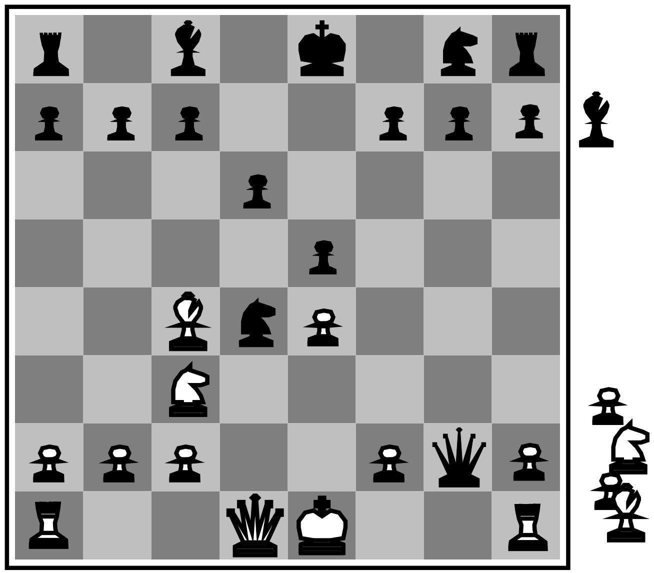
$\Rightarrow$ α–β reaches depth 8 $\Rightarrow$ pretty good chess program

Evaluation functions



Black to move

White slightly better



White to move

Black winning

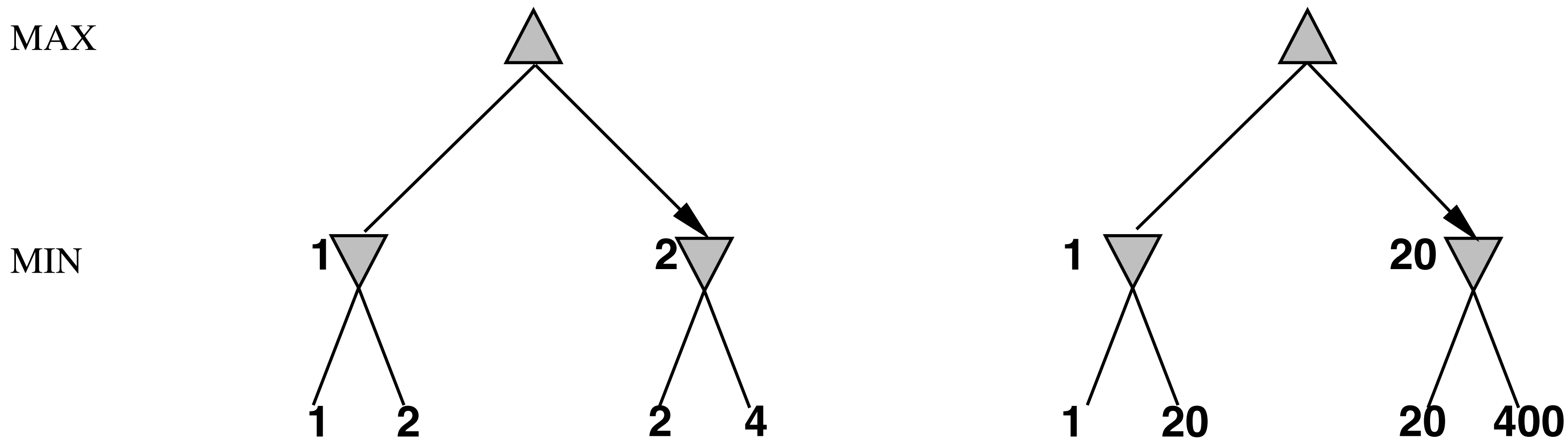
For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

H-Minimax

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$



Behaviour is preserved under any **monotonic** transformation of EVAL

Only the order matters:

payoff in deterministic games acts as an ordinal utility function

Deterministic games in practice

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.

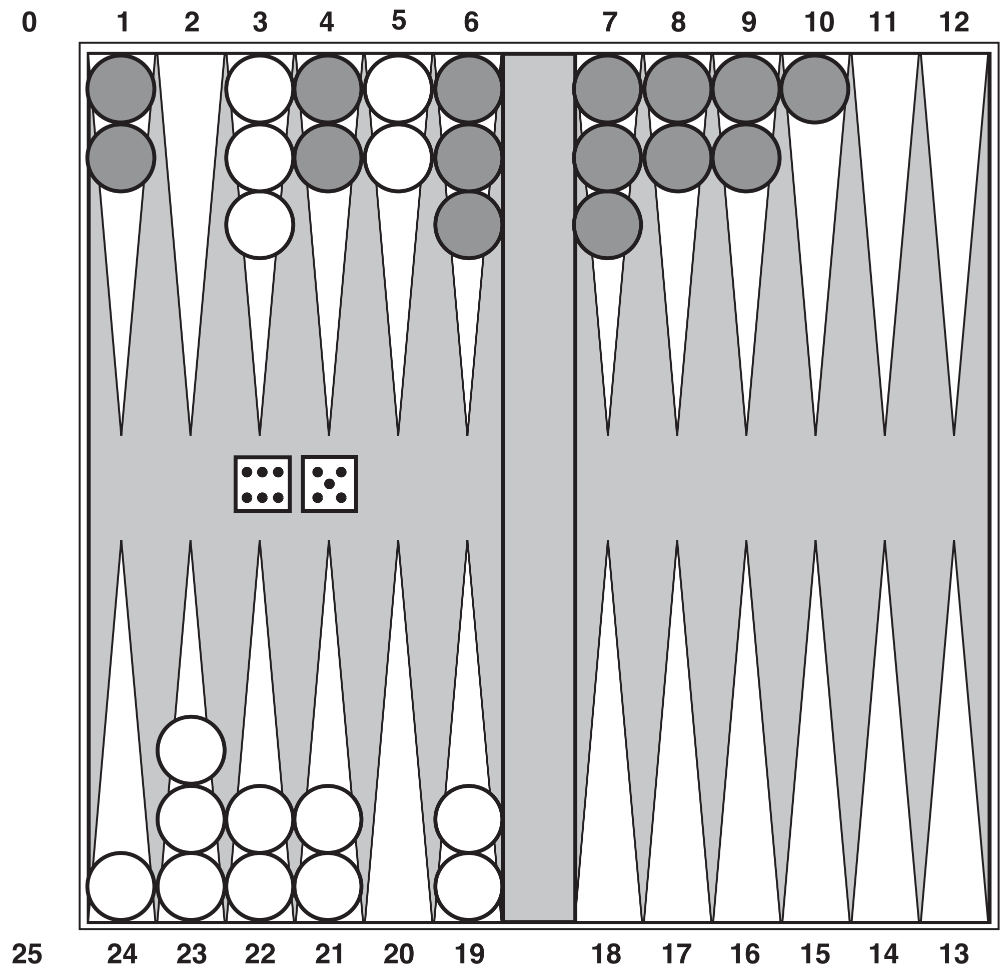
Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

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.~~

Stochastic games

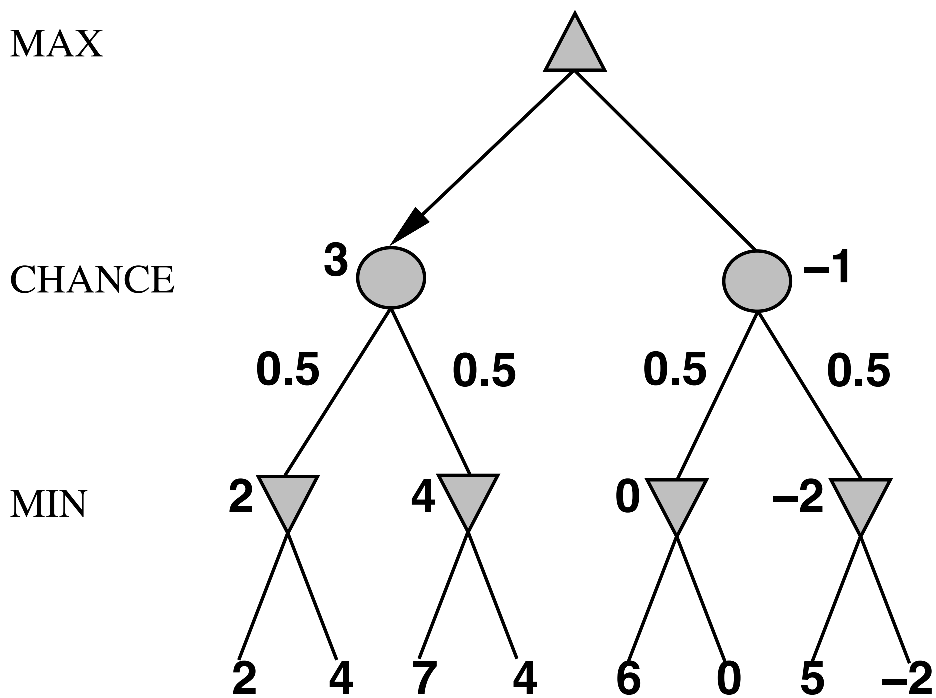
backgammon:



Expect-minimax

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



$$
\textsc{Expectiminimax}(s) =
\begin{cases}
\textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\
\max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\
\min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\
\sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance}
\end{cases}
$$

Nondeterministic games in practice

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

$\textsc{TDGammon}$ uses depth-2 search + very good $\textsc{Eval}$
$\approx$ world-champion level

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
1) generating 100 deals consistent with bidding information
2) picking the action that wins most tricks on average