Artificial Intelligence

Adversarial Search

Last Lecture

- Informed Search
  - Greedy best-first search
  - A*
- Heuristic functions
  - Admissibility
  - Consistency

Today

- Adversarial Search
  - Minimax Search
    - Multiagent environment
  - Alpha-Beta prunning

Adversarial Search

- Recap
  - Multiagent environment
    - Agents must take into account the actions of other agents
  - Competitive environment
    - Agent's goals are in conflict (as opposed to cooperative)

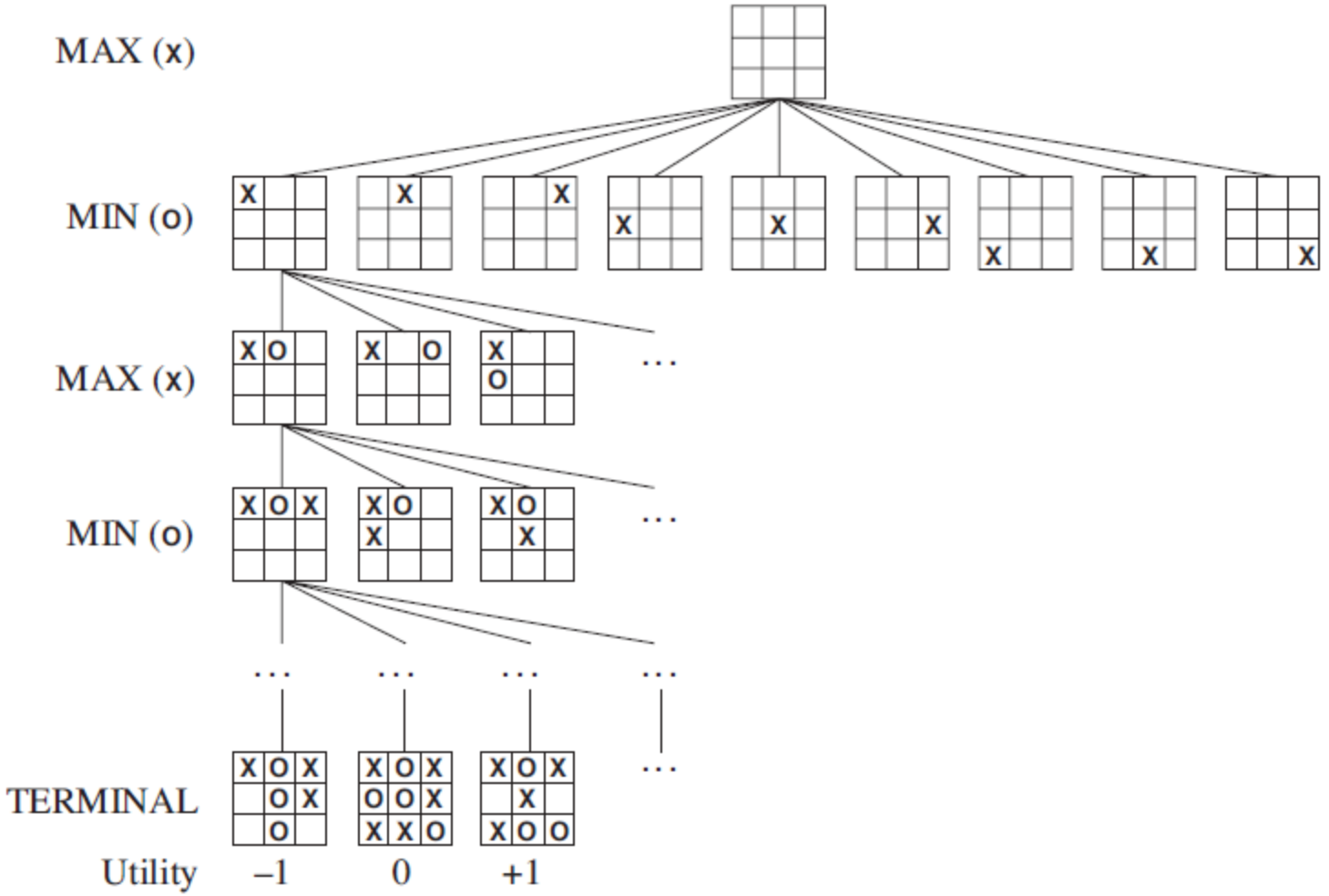
- Games = adversarial search problems
  - Determinisitic, turn-taking, two-player, zero-sum games
  - Zero-sum game: the payoff to all players is the same for every instance of the game
    - Chess / tic-tac-toe:
      - -1, 1 (second player wins)
      - 1, -1 (first player wins)
      - 0, 0 (draw)

Adversarial Search

- If game involves two players, we name players MAX and MIN
  - MAX moves first
  - then MIN
  - then MAX
  - ...
  - At the end, points are awarded to the winning player and penalties to the looser

- UTILITY(s, p): utility function (objective / payoff function)
  - defines the final numeric value for a terminal state *s* for player *p*
    - UTILITY(s, p1) vs. UTILITY(s, p2)

Game Tree – tic-tac-toe



Game Tree – tic-tac-toe

- States represent board configurations
- Players alternate: MAX(x) and MIN(o) [noughts and crosses]
- Tree grows until we reach a terminal state
  - All squares are filled
  - or one player has three in a row
- Values in terminal state indicate the utility function for MAX player

Adversarial Search

- Before *(normal* search)*:*
  - Optimal solution is a sequence of actions leading to a goal state

- Now (*adversarial* search):
  - MAX cannot find the best sequence of actions leading to a goal state (a win / maximum utility function) **without considering MIN's actions**
    - We need to specify MAX's move
      - in the initial state,
      - in the states resulting from every possible response by MIN,
      - in the states resulting from every possible response by MIN to those moves,
      - and so on

Minimax

- MINIMAX(n) = utility (for MAX) of being in state *n*

  assuming that both players **play optimally** until the end of the game
  - So we are assuming that the opponent (MIN) will act rationally

- UTILITY(n) is the utility for MAX

  If both MAX and MIN play optimally:
  - MAX wants to ..... UTILITY(n)
  - MIN wants to ..... UTILITY(n)

- [Example on (simple) game tree]

Minimax – example

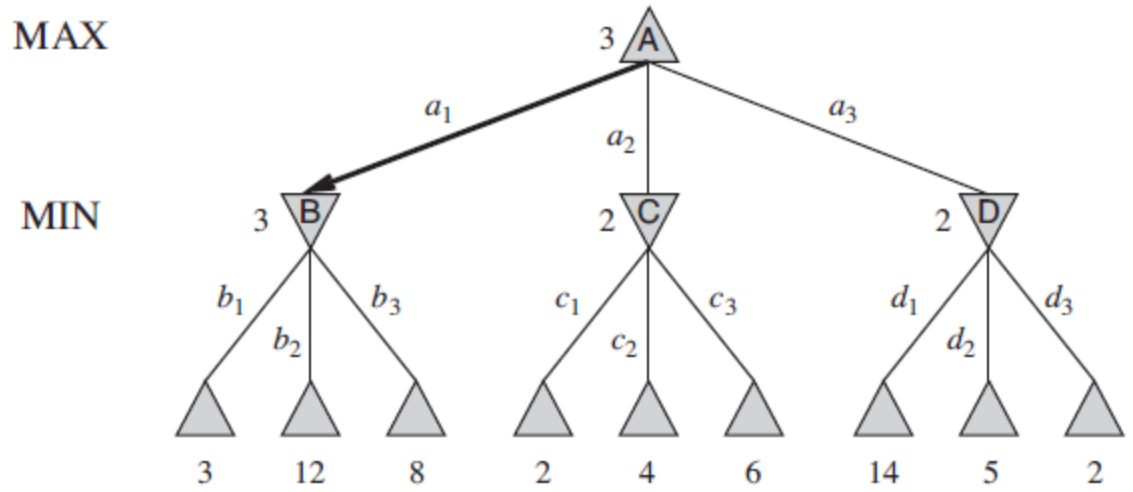


Figure 5.2 **FILES: figures/minimax.eps (Tue Nov 3 16:23:11 2009).** A two-ply game tree. The $\triangle$ nodes are "MAX nodes," in which it is MAX's turn to move, and the ∇ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if Terminal-test}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{Result}(s,a)) & \text{if Player}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{Result}(s,a)) & \text{if Player}(s) = \text{MIN} \end{cases}$$

- Recursive computation
  - Bottom up: from the leaves to the root
    - Minimax values are backed up through the tree

- Minimax performs a **complete depth-first search** exploration of the game tree

Minimax

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) 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)))
  return v

function MIN-VALUE(state) 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)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Minimax algorithm – more than two players

- If two players:
  - One value per node (utility function for MAX)
  - The utility function for MIN is just the opposite (implicit)
    - zero-sum games

- Three or more players
  - Must keep the utility functions for each player
    - an array
  - Each player chooses the action that maximizes its utility function

- [Example with simple game tree]

Minimax algorithm – more than two players

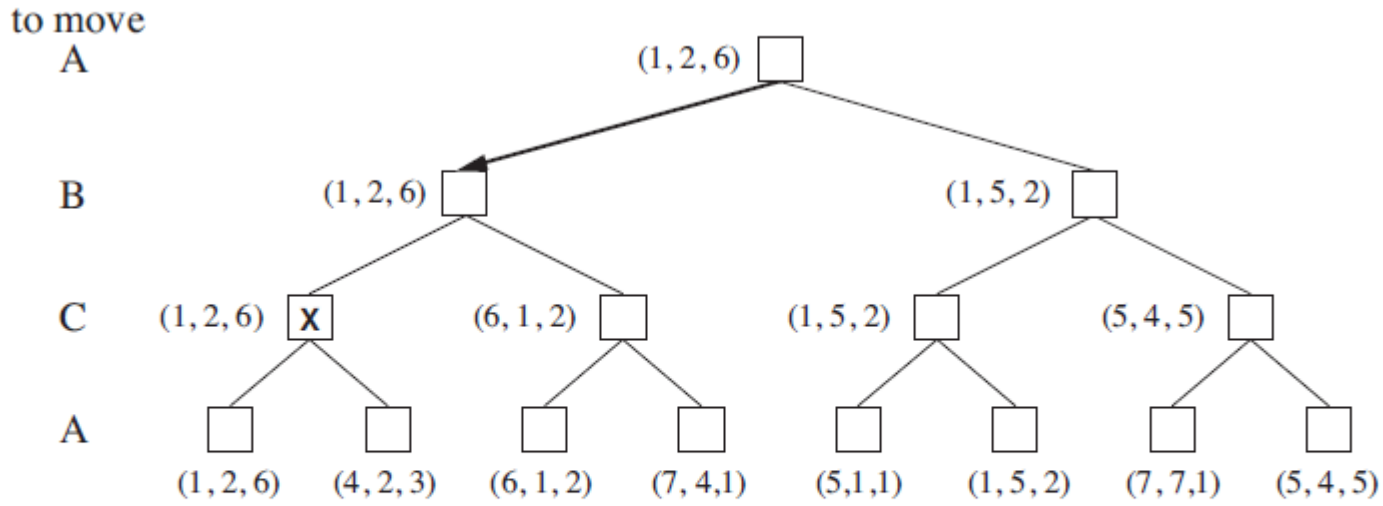


Figure 5.4 **FILES: figures/minimax3.eps (Tue Nov 3 16:23:11 2009).** The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha-Beta pruning

- Minimax examines an exponential number of nodes
  - Like depth-first search
  - Way too expensive for most games

- We want to prune parts of the tree
  - But still get the same result

- Alpha-Beta pruning main idea: keep two bounds
  - Alpha: lower bound of maximizing player
  - Beta: upper bound of minimizing player

... and prune whenever you can do so safely

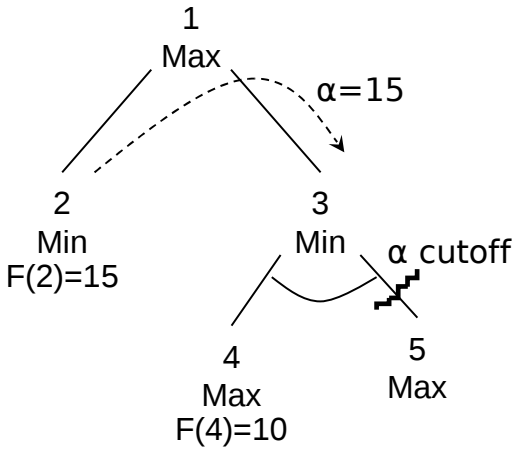
Alpha-Beta pruning

- Alpha: lower bound of maximizing player
  - Minimum value MAX will have at the end
- Beta: upper bound of minimizing player
  - Maximum value MIN will have at the end

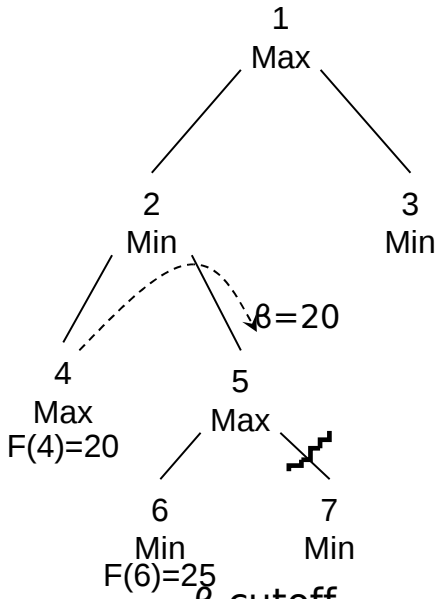
- When can we prune?
  - In a MAX node, prune if value of node is >= β
    - You are trying to maximize and current value is more or equal than the upper bound for MIN ...
      - So there is no point examining more nodes
  - In a MIN node, prune if value of node is <= α
    - You are trying to minimize and current value is less or equal than the lower bound for MAX ..
      - So there is no poing examining more nodes

Alpha-Beta Pruning Procedure

α—is the lower bound of maximizing player
β—is the upper bound of minimizing player



α cutoff
(pruning)



β cutoff
(pruning)

Alpha-Beta Pruning Procedure

In α-cutoff nodes 2 and 4 have been evaluated—either by the static function or by backing up from descendents (not shown here).

For player Max, the maximum of 2 is 15. However, node 3 can offer a maximum of 10. To understand consider two situations:

- Node 5 is smaller than 10, player Min will select node 5 but is irrelevant since Max player is sure to get 15 from node 2
- Node 5 is larger than 10, in which case player Min disregards it in favor of node 4.

So, node 3 which receives the lower bound $\alpha = 15$ does not need to evaluate node 5 and any subsequent nodes. Node 5 is α cutoff, or α pruning.

Alpha-Beta Pruning Procedure

In β-cutoff example, the minimum value of node 2 is 20 as given by node 4. Any other value at node 5 larger than 20 is not going to be selected by Min player at node 2.

The maximizer at node 5 gets the maximum of at least 25 from node 6.

This value is larger than 20, that already Minimizer player at node 2 can have, so node 7 is not explored, is a β cutoff.

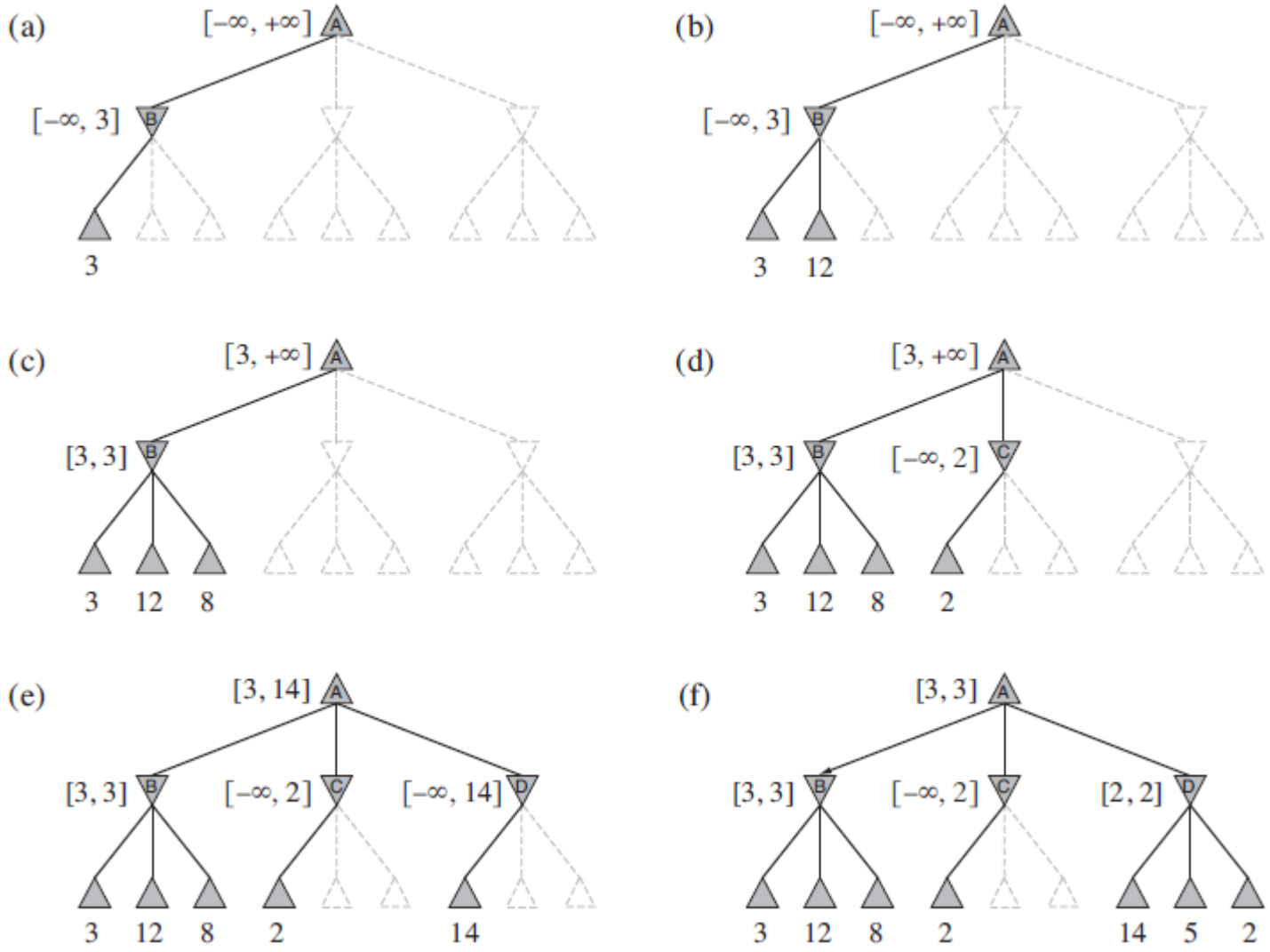
The Alpha-Beta procedure starts from node 1 where two parameters are set

$\alpha = -\infty$

$\beta = +\infty$

As nodes are visited, α and β may be increased or decreased respectively, and branches cutoff.

Alpha-Beta pruning – example



Alpha-Beta Procedure

```
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
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).