Artificial Intelligence (CS 3011)

CHAPTER 6: **Adversial search**

Sourav Kumar Giri
Asst. Professor

School of Computer Engineering
KIIT Deemed to be University

Chapter Outline

- ❑ Adversial search problems or Game playing problems
- ❑ Min-Max Algorithm
- ❑ Alpha-Beta Pruning

Adversarial search problems

- **Multi-agent environments**
  - ☐ Each agent needs to consider the actions of other agents and how they affect its own welfare
  - ☐ The unpredictability of these other agents can introduce possible contingencies into the agent’s problem-solving process
  - ☐ cooperative vs. competitive multi-agents

- **Adversarial search problems** (often known as **Games**)
  - ☐ Deals with competitive multi-agents which have conflicting goals
  - ☐ In AI, the most common games are of a rather specialized kind—what game theorists call **deterministic**, **turn-taking**, **two-player**, **zero-sum games** of perfect information (such as chess).

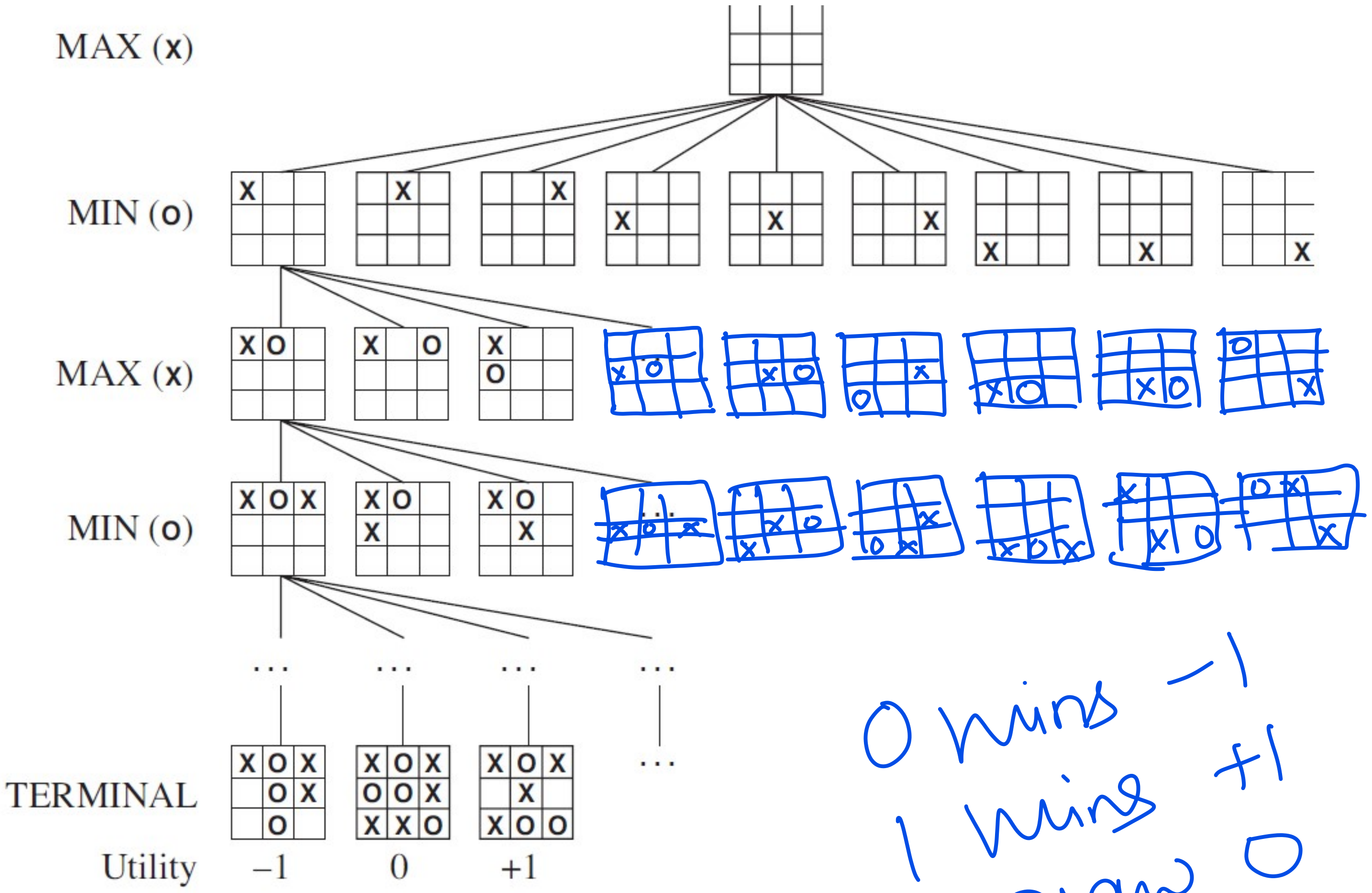
Definition of Game

- A game can be formally defined as a kind of search problem with the following elements:
  - **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.

Definition of Game

- **UTILITY(s, p):** A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2 . Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.
- A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has
- payoff of either 0 + 1, 1 + 0 or 1/2+1/2 . "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of 1/2.

A (partial) game tree for the game of tic-tac-toe



O wins −1
1 wins +1
Draw 0

A two-ply game tree

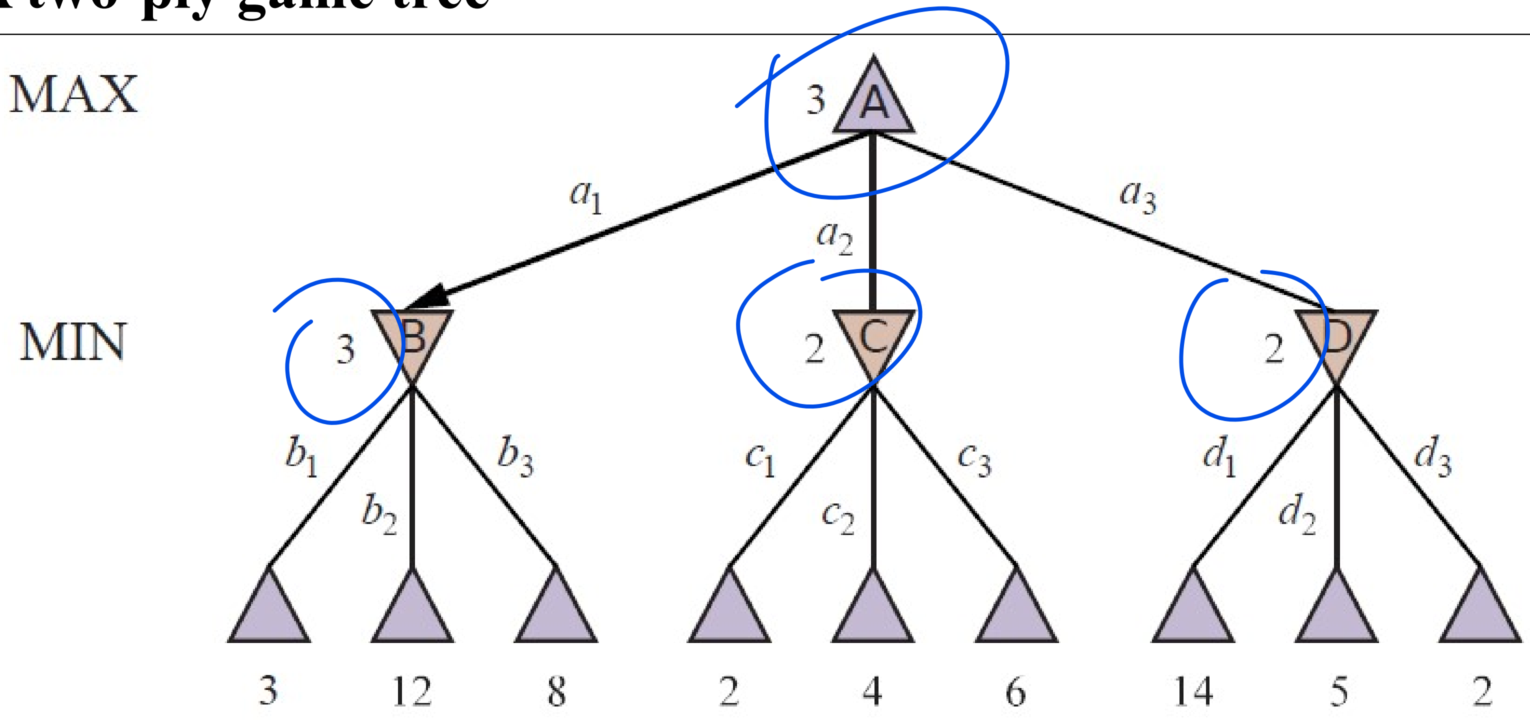


Figure. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values forMAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.

Min-Max strategy

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

Min-Max Algorithm

```
**function** MINIMAX-SEARCH($game$, $state$) **returns** $an\ action$
    player $\leftarrow$ $game$.TO-MOVE($state$)
    $value$, $move$ $\leftarrow$ MAX-VALUE($game$, $state$)
    **return** $move$

**function** MAX-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
    **if** $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
    $v \leftarrow -\infty$
    **for each** $a$ **in** $game$.ACTIONS($state$) **do**
        $v2$, $a2$ $\leftarrow$ MIN-VALUE($game$, $game$.RESULT($state$, $a$))
        **if** $v2 > v$ **then**
            $v$, $move \leftarrow v2$, $a$
    **return** $v$, $move$
```

Min-Max Algorithm

```
**function** MIN-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
  **if** $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow +\infty$
  **for each** $a$ **in** $game$.ACTIONS($state$) **do**
    $v2$, $a2 \leftarrow$ MAX-VALUE($game$, $game$.RESULT($state$, $a$))
    **if** $v2 < v$ **then**
      $v$, $move \leftarrow v2$, $a$
  **return** $v$, $move$
```

First three plies of a game tree with three players (A, B, C)

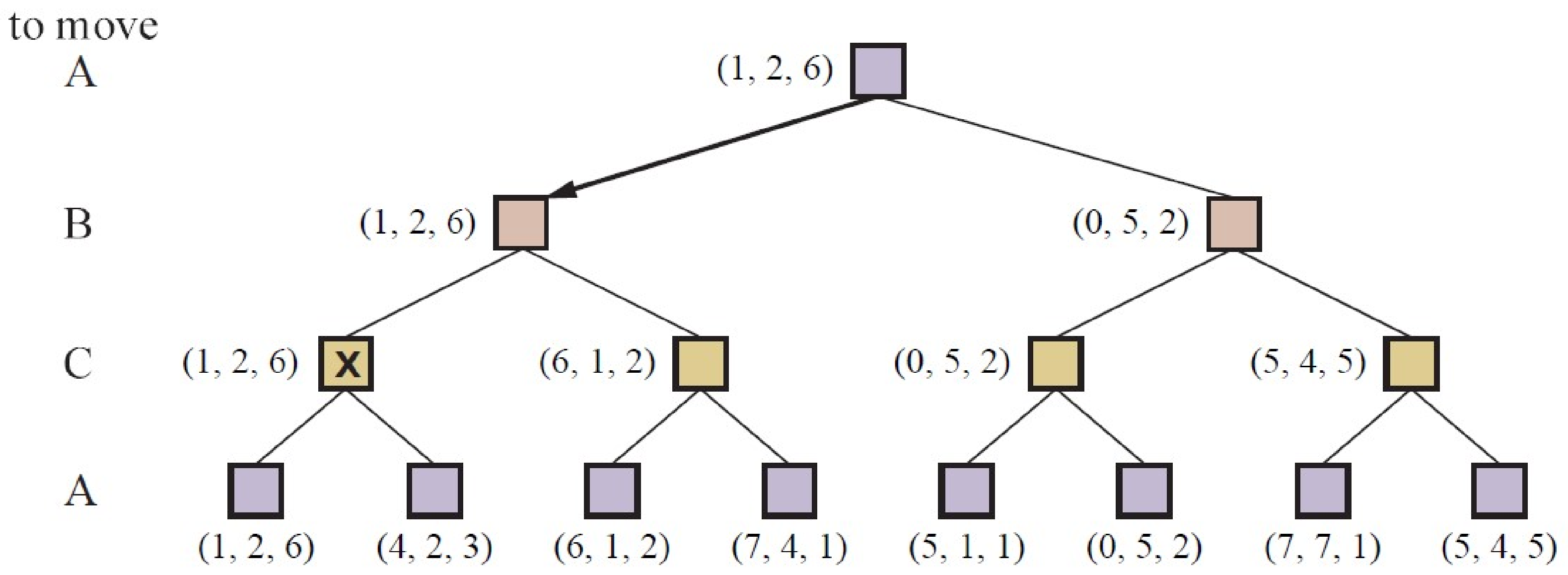
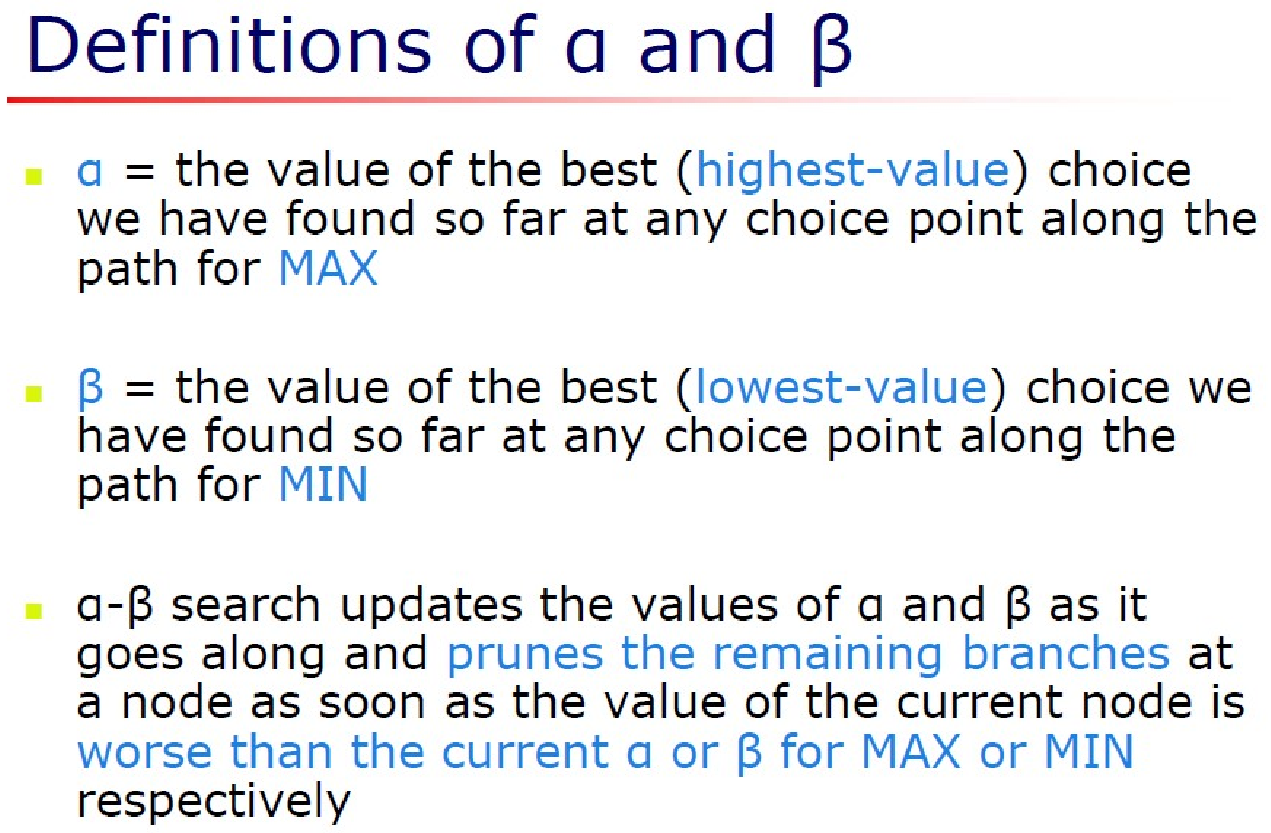


Figure. The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha-Beta Pruning

- ☐ The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves.
- ☐ Unfortunately we can’t eliminate the exponent, but we can effectively cut it in half.
- ☐ It is possible to compute the correct minimax decision without looking at every node in the game tree.
- ☐ A particular pruning technique called **alpha-beta pruning** can eliminate large part of the tree from consideration.
- ☐ When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that can not possibly influence the final decision.

Alpha-Beta Pruning



Definitions of α and β

- α = the value of the best (highest-value) choice we have found so far at any choice point along the path for MAX

- β = the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN

- α-β search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is worse than the current α or β for MAX or MIN respectively

Alpha-Beta Pruning

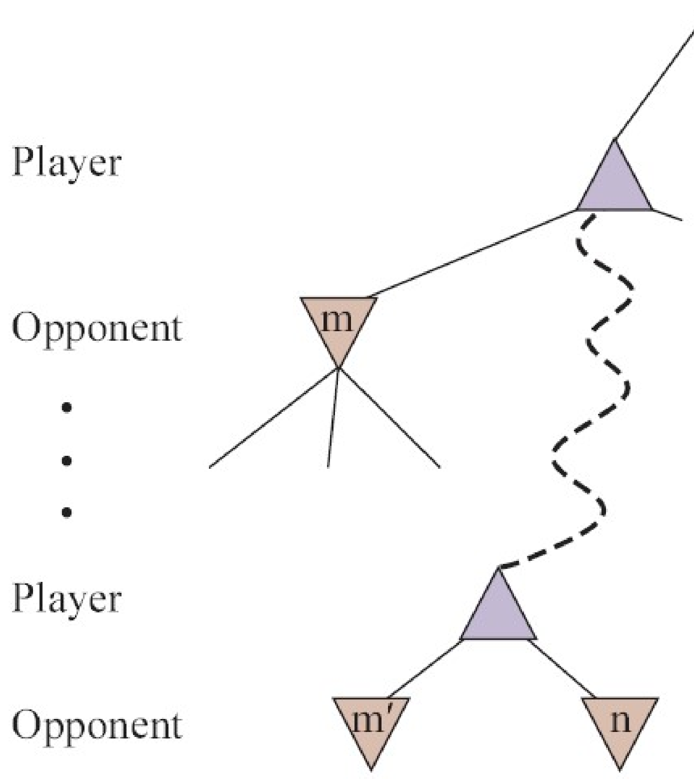
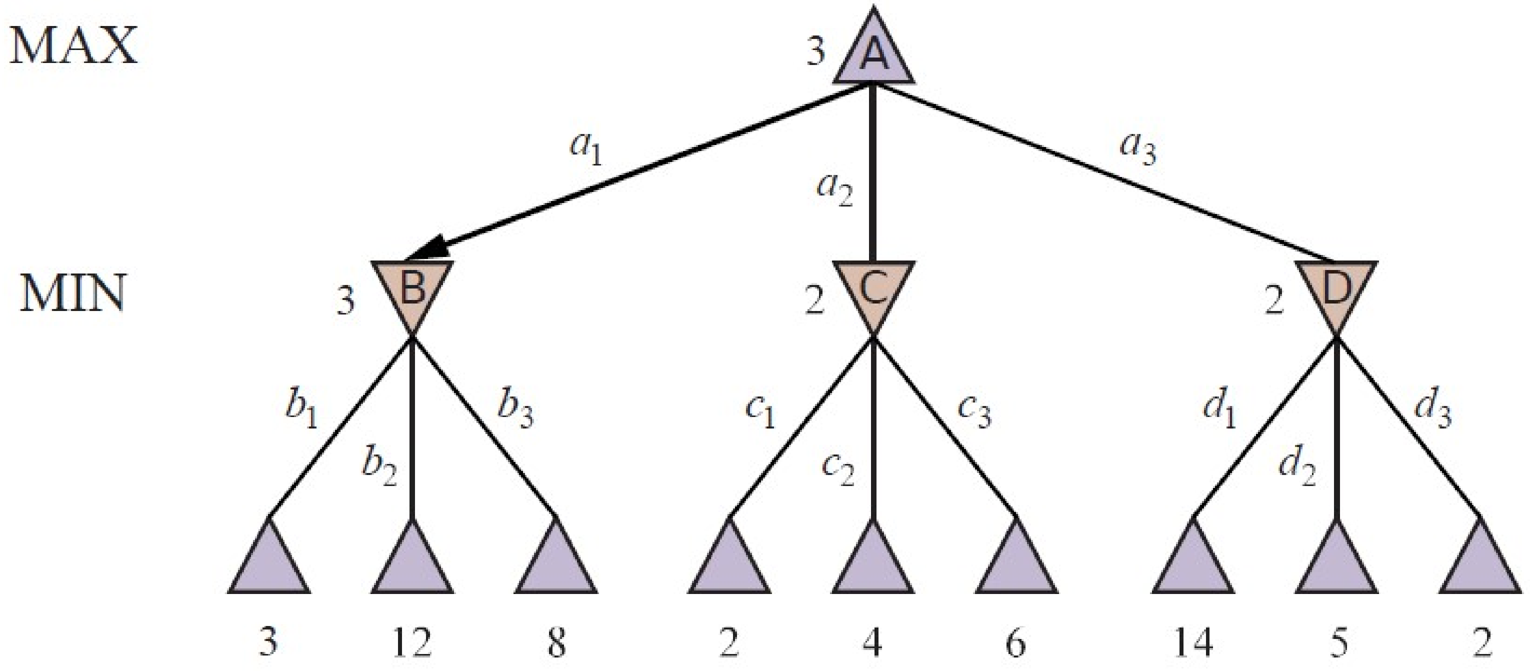


Figure. The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Alpha-Beta Pruning (Two-ply game tree)



The △ 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 forMAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.

Alpha-Beta Pruning Example

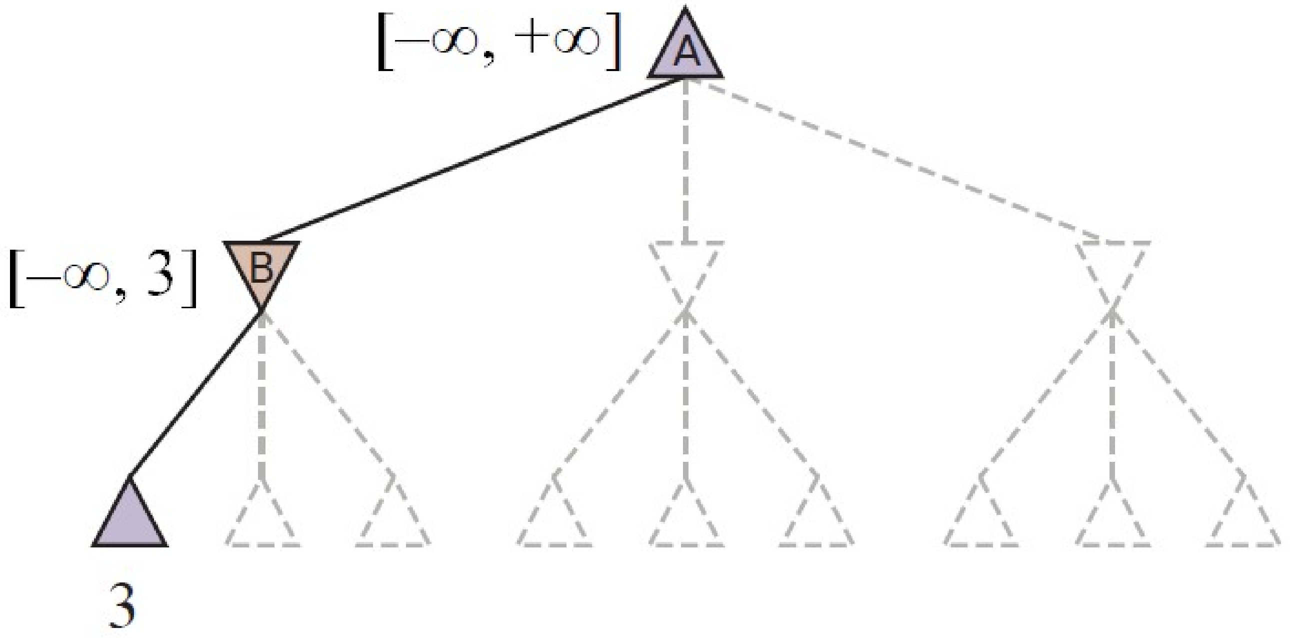


Figure. The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.

Alpha-Beta Pruning Example

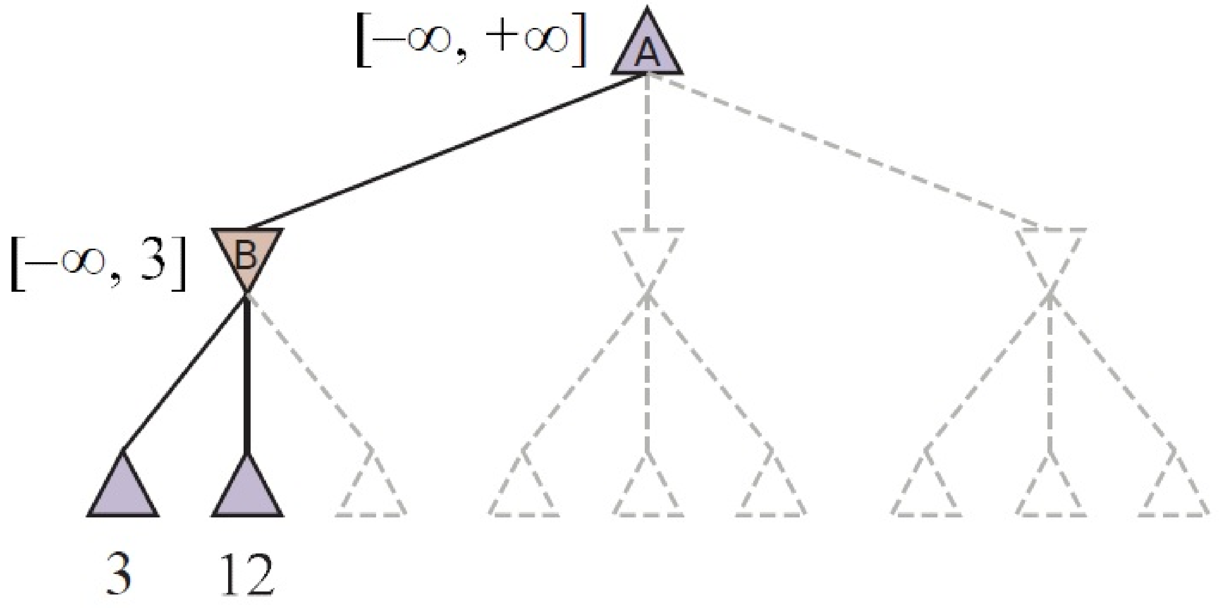


Figure. The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.

Alpha-Beta Pruning Example

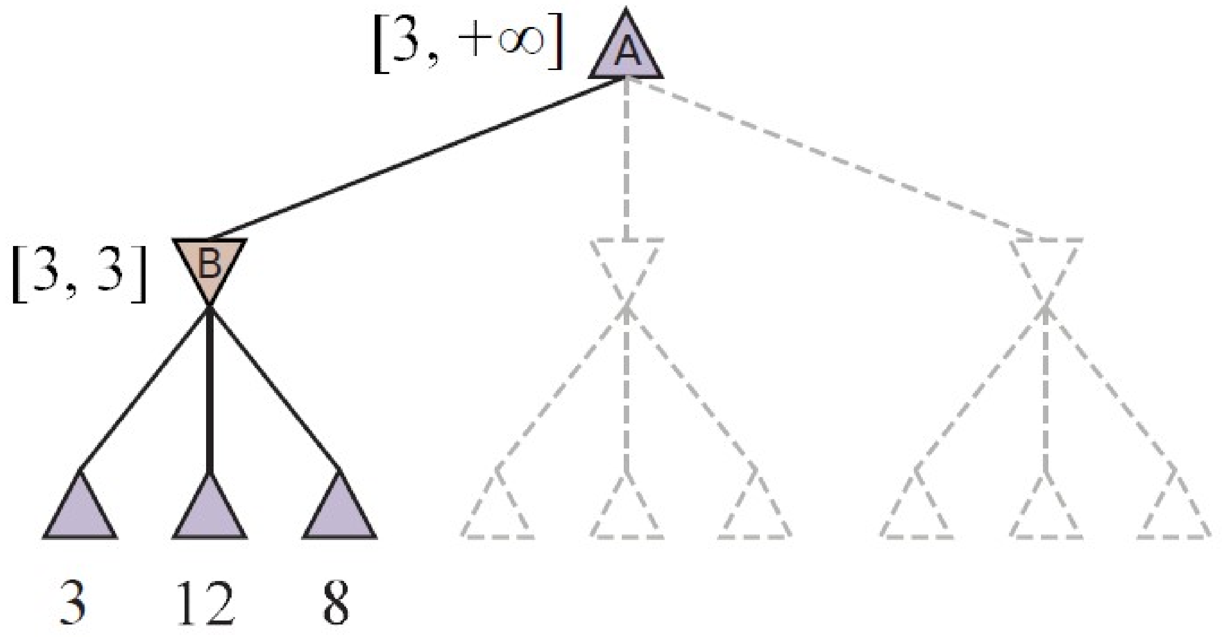


Figure. The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now we can infer that the value of the root is at least 3, becauseMAX has a choice worth 3 at the root.

Alpha-Beta Pruning Example

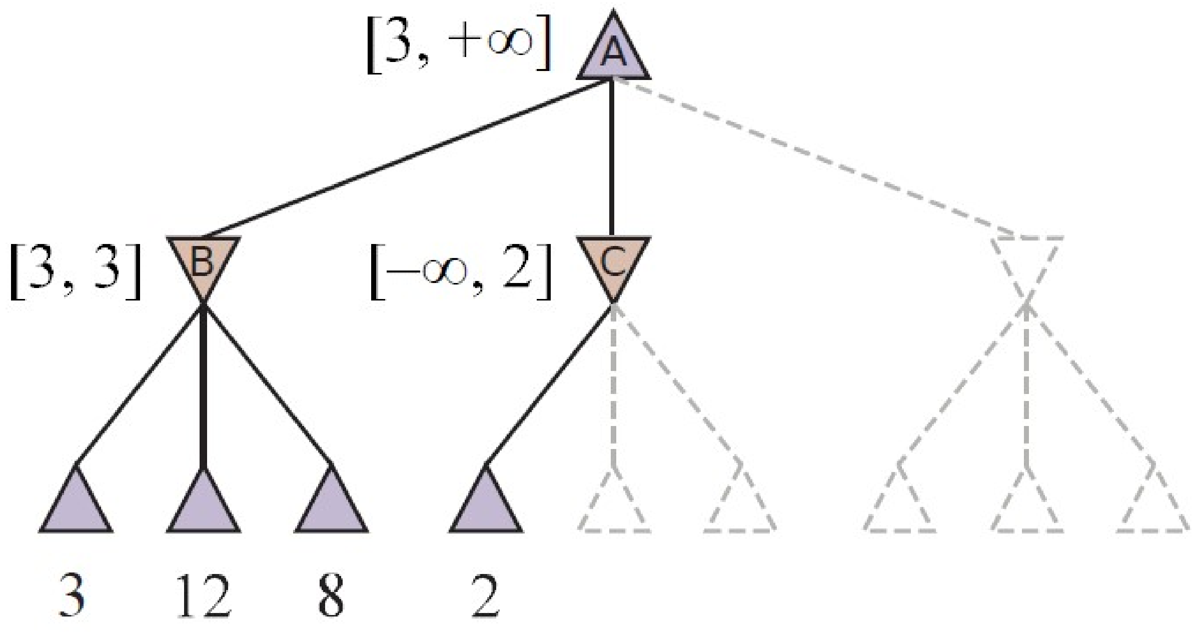


Figure. The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. **This is an example of alpha–beta pruning.**

Alpha-Beta Pruning Example

MINIMAX(root)

$= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2))$

$= \max(3, \min(2, x, y), 2)$

$= \max(3, z, 2)$, where $z = \min(2, x, y) \leq 2$

$= 3.$

- ☐ *The value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y.*

Alpha-Beta Pruning Example

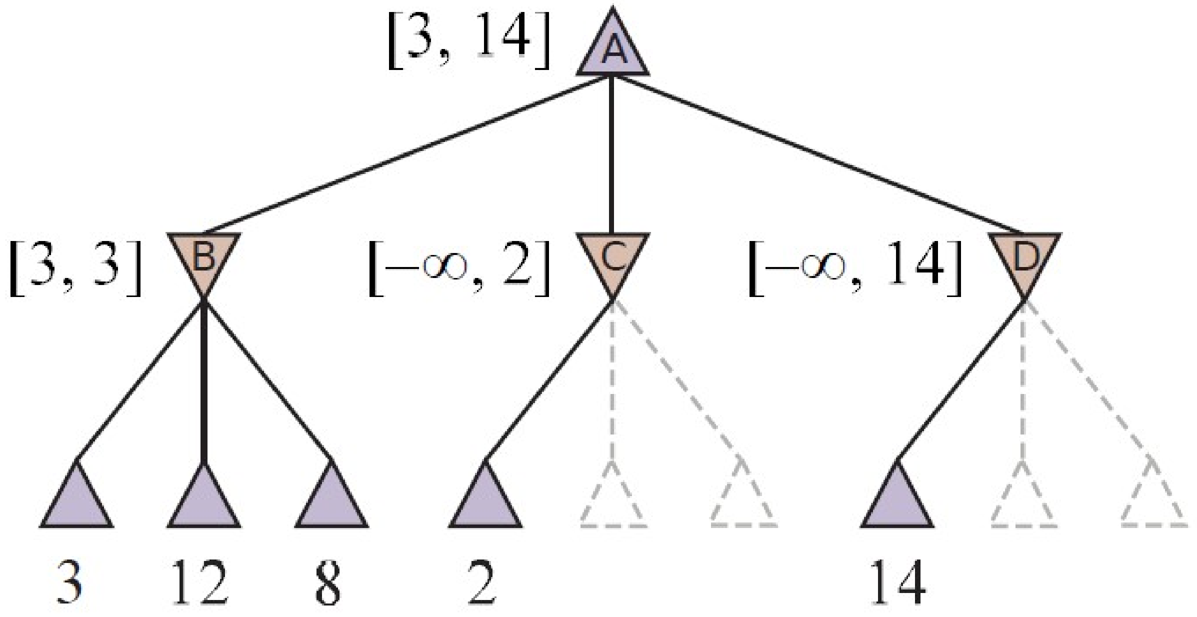


Figure. The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX’s best alternative (i.e., 3), so we need to keep exploring D’s successor states. Notice also that we now have bounds on all of the successors of the root, so the root’s value is also at most 14.

Alpha-Beta Pruning Example

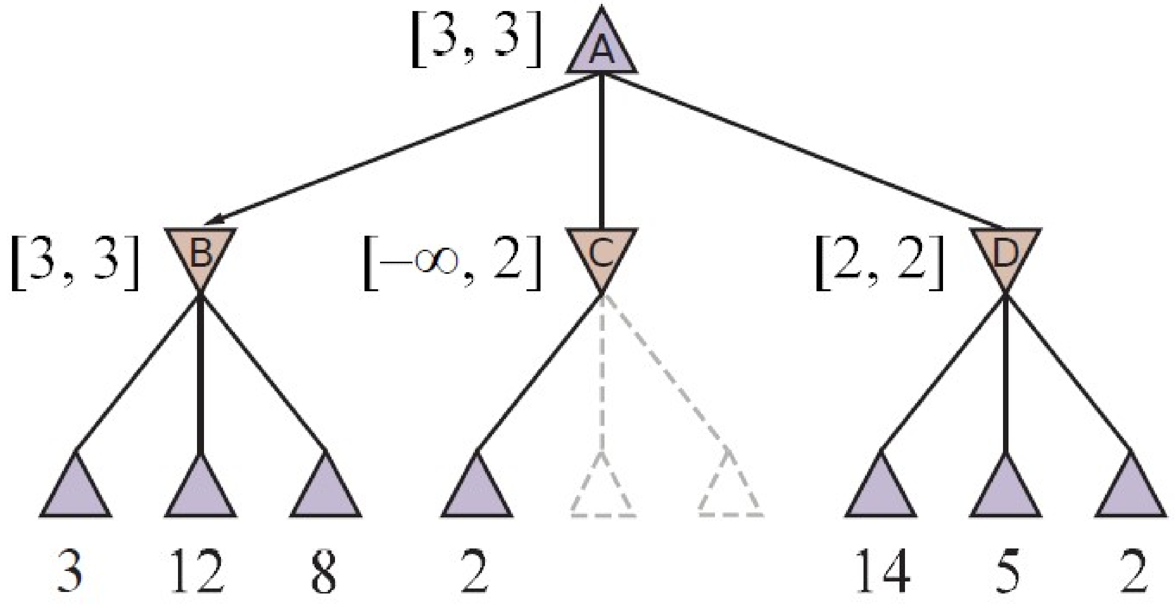
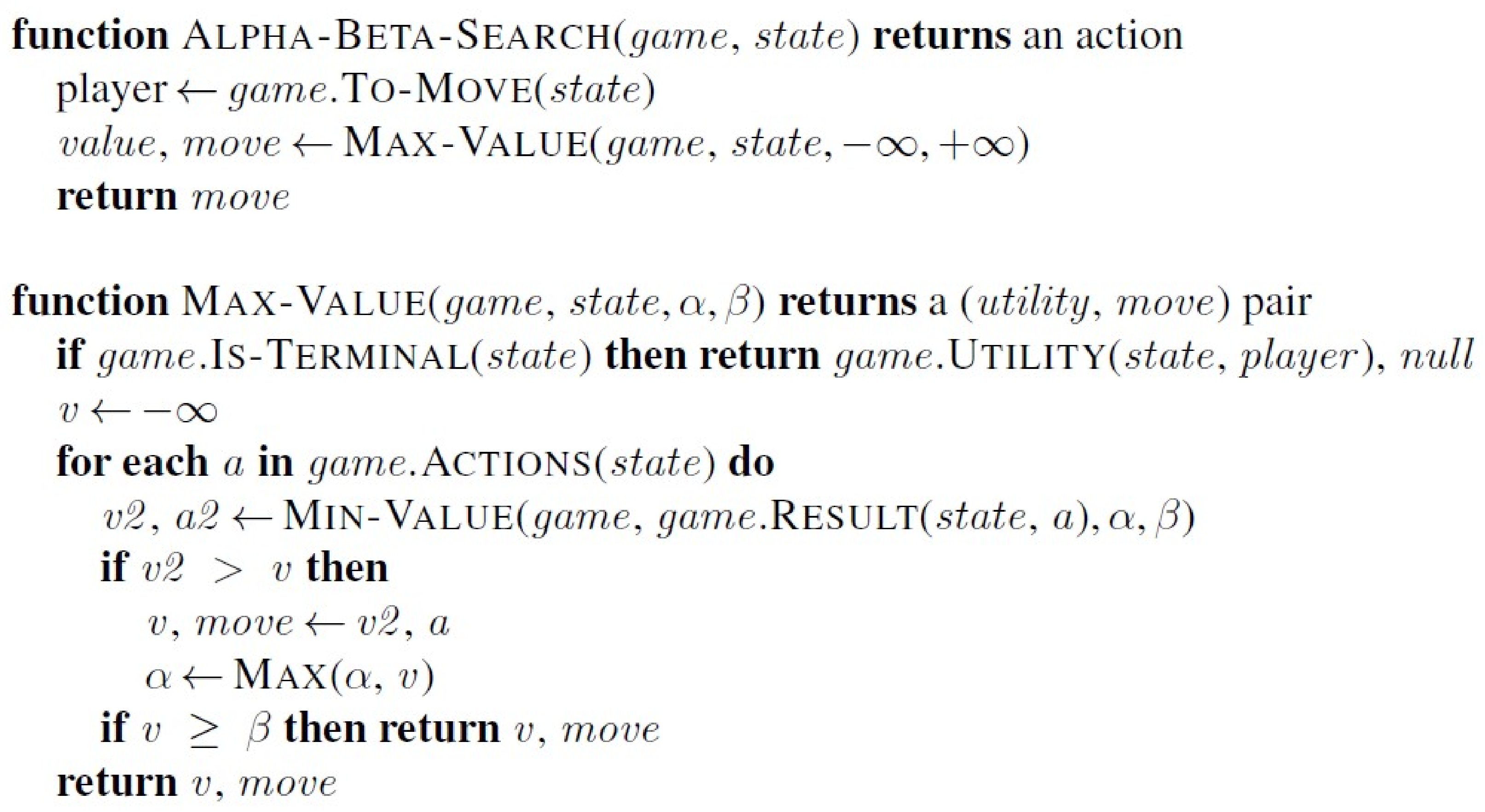


Figure. The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX’s decision at the root is to move to B, giving a value of 3.

Alpha-Beta Pruning Algorithm



```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
    return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    $v \leftarrow -\infty$
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
        if $v2 > v$ then
            v, move ← v2, a
            $\alpha \leftarrow$ MAX($\alpha$, v)
        if $v \geq \beta$ then return v, move
    return v, move
```

Alpha-Beta Pruning Algorithm

```
function MIN-VALUE($game$, $state$, $\alpha$, $\beta$) returns a ($utility$, $move$) pair
  if $game$.IS-TERMINAL($state$) then return $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow +\infty$
  for each $a$ in $game$.ACTIONS($state$) do
    $v2$, $a2 \leftarrow$ MAX-VALUE($game$, $game$.RESULT($state$, $a$), $\alpha$, $\beta$)
    if $v2 < v$ then
      $v$, $move \leftarrow v2$, $a$
      $\beta \leftarrow$ MIN($\beta$, $v$)
    if $v \leq \alpha$ then return $v$, $move$
  return $v$, $move$
```