Fundamentals of Artificial Intelligence

Adversarial Search
Game Playing

Games

- In **multiagent environments**, each agent needs to consider the actions of other agents and how they affect its own welfare.
- The agents can be **cooperative** or **competitive**.
- In **competitive environments**, the agents' goals are in conflict, giving rise to **Adversarial Search problems** — often known as **GAMES**.
- In mathematical **game theory**, a multiagent environment is treated as a **game**, the impact of each agent (economy) on the others is significant, regardless of whether the agents are ***cooperative*** or ***competitive***.
- In **AI**, the most common games are *deterministic*, *turn-taking*, *two-player*, **zero-sum games** of **perfect information** (such as chess).
  - In deterministic and fully observable environments in two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
  - If one player wins a game of chess, the other player necessarily loses.
  - It is this opposition between the agents' **utility functions** that makes the situation **adversarial**.

Games

- Games are too hard to solve.
  - Chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} nodes.
- Games require the ***ability to make decision even when calculating optimal decision is infeasible***.
- **Pruning** allows us to ignore portions of the search tree.
- **Heuristic evaluation functions** allow us to approximate **true utility of a state** without doing a complete search.
- Games such as backgammon includes elements of **imperfect information** because not all cards are visible to each player.
- Types of games

	deterministic	**chance**
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships,	bridge, poker, scrabble

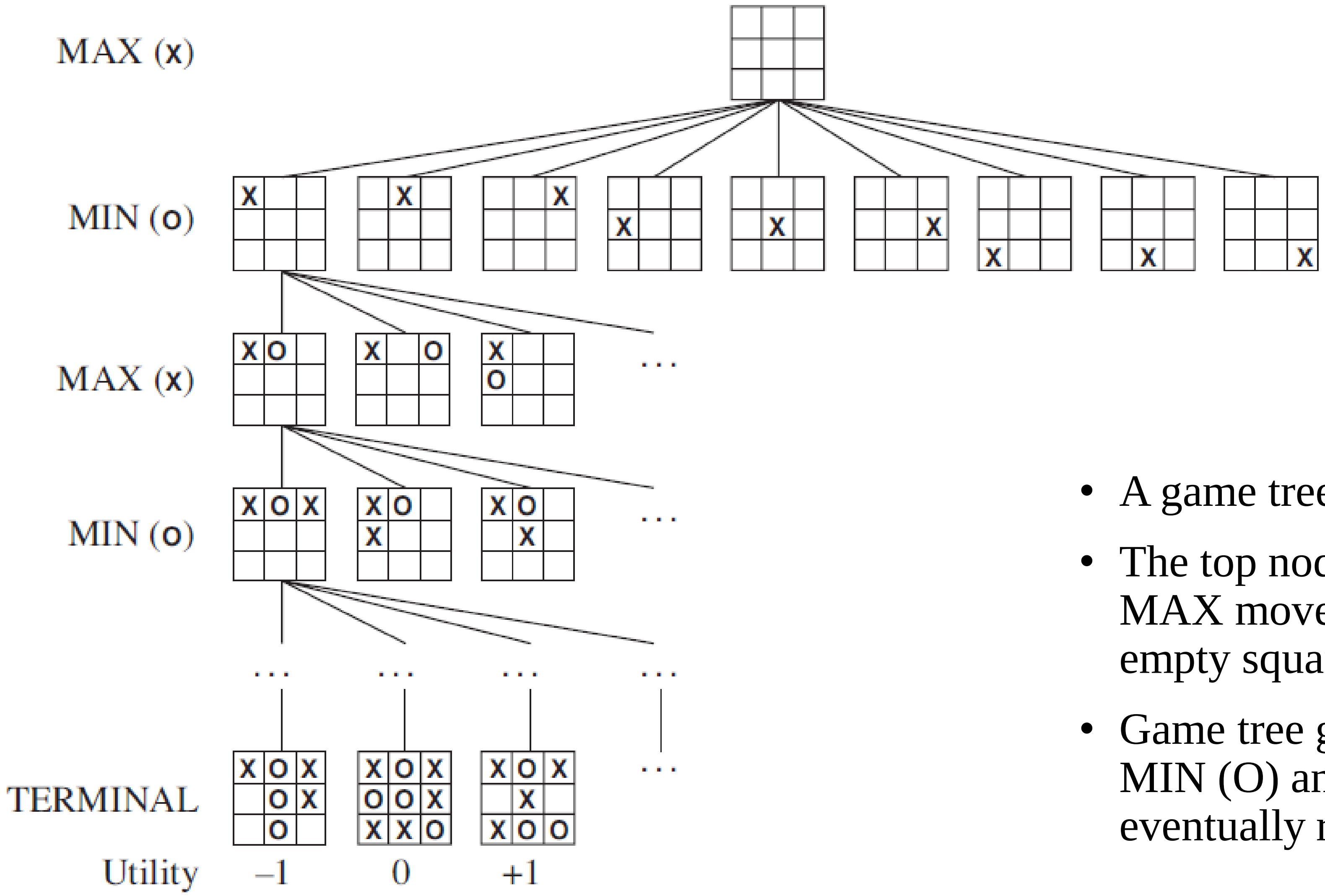
Games as Search Problem

- 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 **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 .
  - A **zero-sum game** is (constant-sum is a better term) 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.

Games as Search Problem

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the **nodes are game states** and the **edges are moves**.
  - For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.
  - But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.
- We have two players: MAX and MIN.
  - MAX (our player) moves first in the game.
- Regardless of size of game tree, MAX's (our player's) job to search for a good move in search tree.
  - A search algorithm does not explore the full game tree, and examines enough nodes to allow a player to determine what move to make.

Game Tree



- A game tree for the game of tic-tac-toe.
- The top node is the initial state, and MAX moves first, placing an X in an empty square.
- Game tree gives alternating moves by MIN (O) and MAX (X), until eventually reaching terminal states.

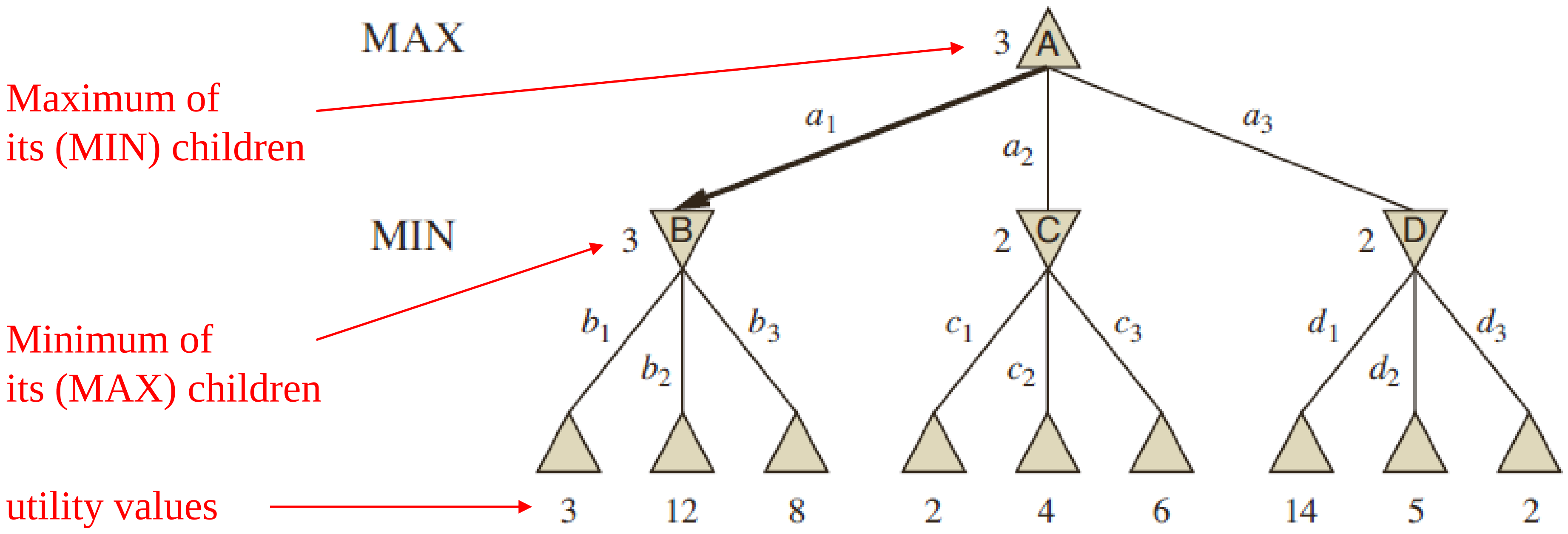
Optimal Decisions in Games

- **Optimal Solution** in a search problem is a sequence of actions leading to a ***goal state*** (***a terminal state that is a win).***
- MAX must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN.
- Given a game tree, **optimal strategy** can be determined from **minimax value** of each node n, and it is called as MINIMAX(n).
- **The minimax value of a node** ***is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game***.
- The **minimax value of a terminal state** is just its ***utility***.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

MINIMAX

- Δ nodes are "MAX nodes," in which it is MAX's turn to move, and ∇ nodes are "MIN nodes."
- Terminal nodes show the utility values for MAX; 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.



Minimax Algorithm

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

```
function MAX-VALUE($state$) 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($s$, $a$)))
    return $v$
```

```
function MIN-VALUE($state$) 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($s$, $a$)))
    return $v$
```

- Minimax Algorithm 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.

Properties of Minimax Algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree.

Complete: **YES**, if tree is finite

Optimal: **YES**, against an optimal opponent.

Time complexity: **$O(b^m)$**

Space complexity: **$O(bm)$** (depth-first exploration) Algorithm generates all actions at once

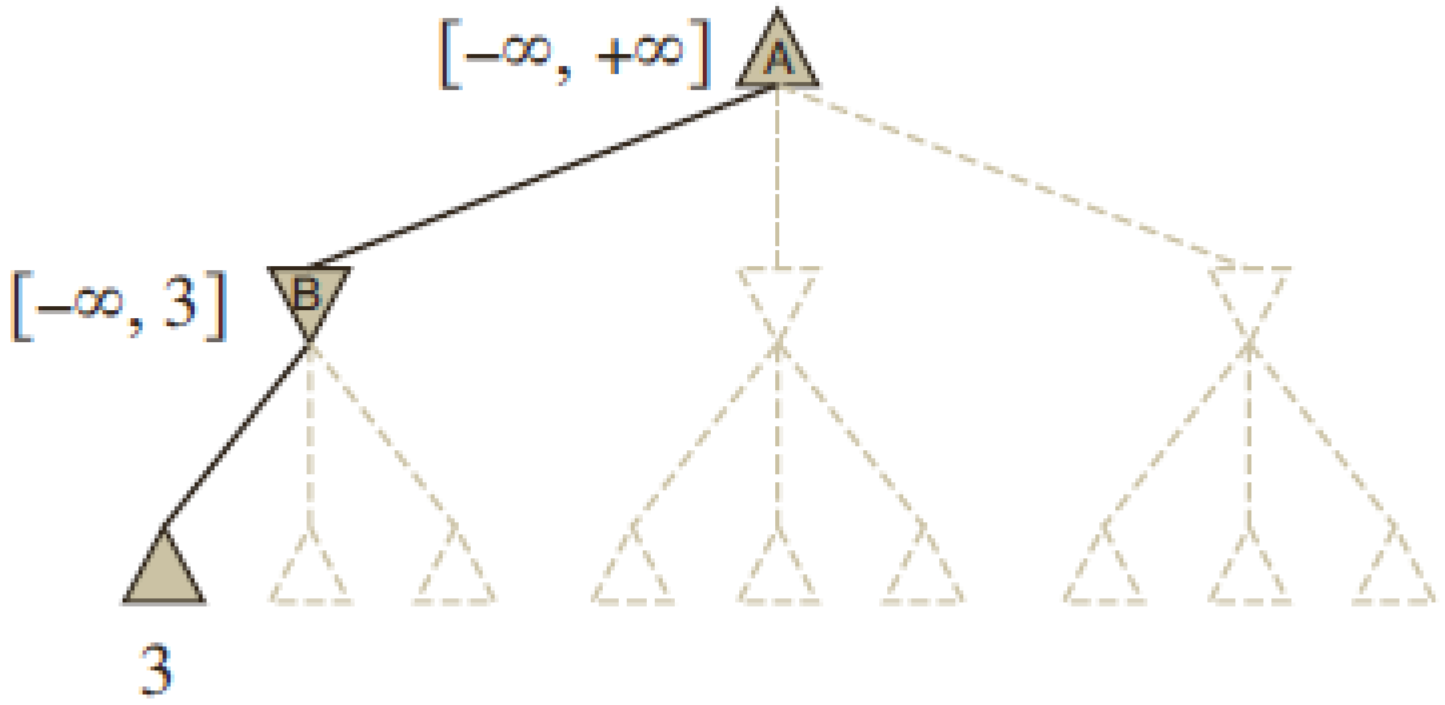
- For real games, the time cost is totally impractical, but minimax algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.
- For chess, $b \approx 35$, $m \approx 100$ for reasonable games ➔ exact solution completely infeasible

Alpha–Beta Pruning

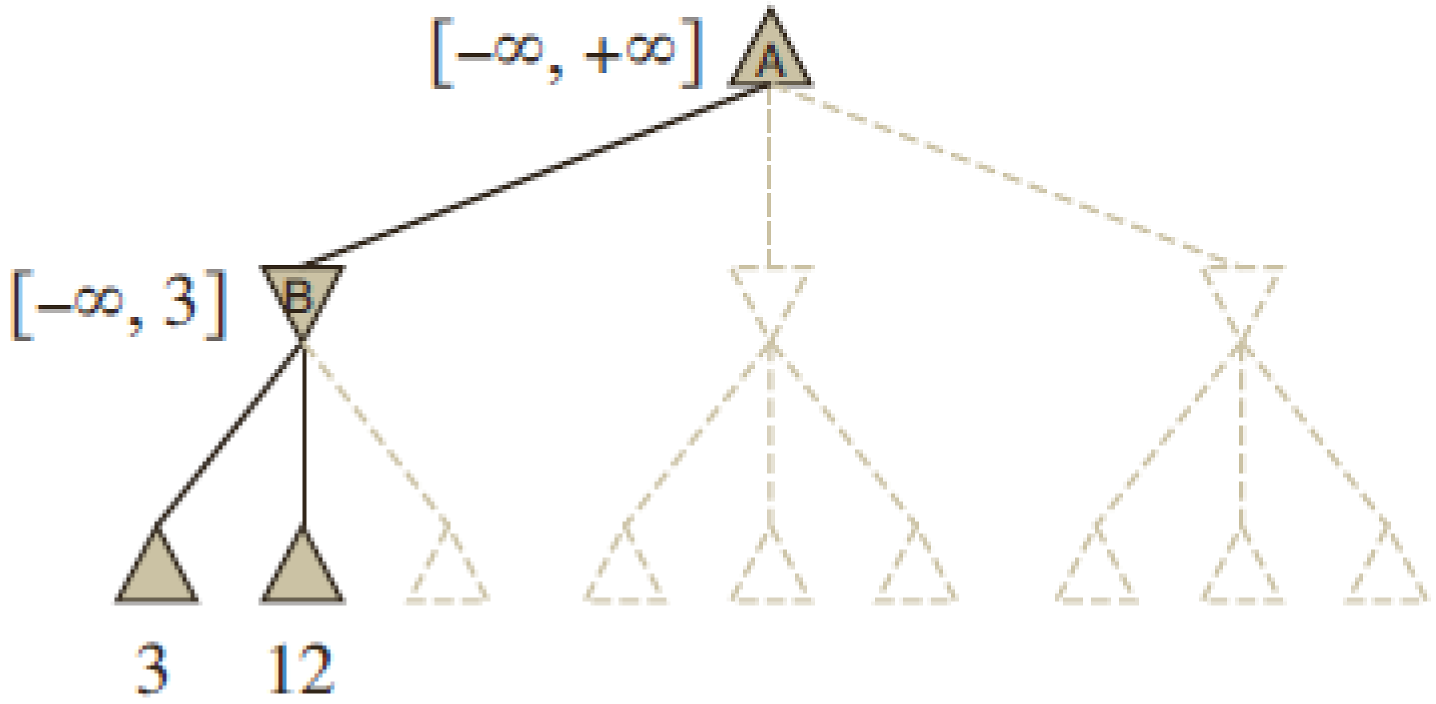
- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can’t eliminate the exponent, but we can effectively cut it in half.
- It is possible to compute **correct minimax decision** without looking at every node in game tree.
- **Alpha–Beta pruning** prunes away branches of the minimax tree that cannot possibly influence the final decision and it returns the same move as minimax algorithm would.

- ***α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.***
- ***β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.***

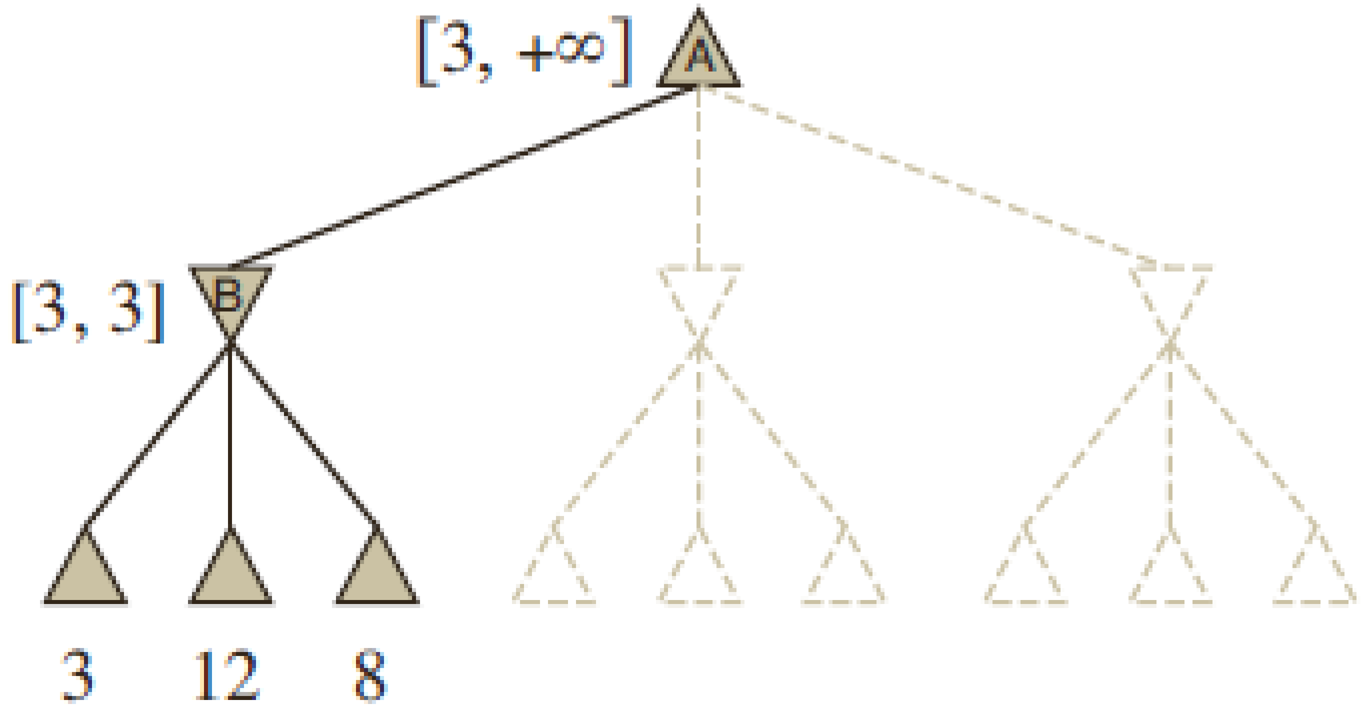
Alpha–Beta Pruning Example



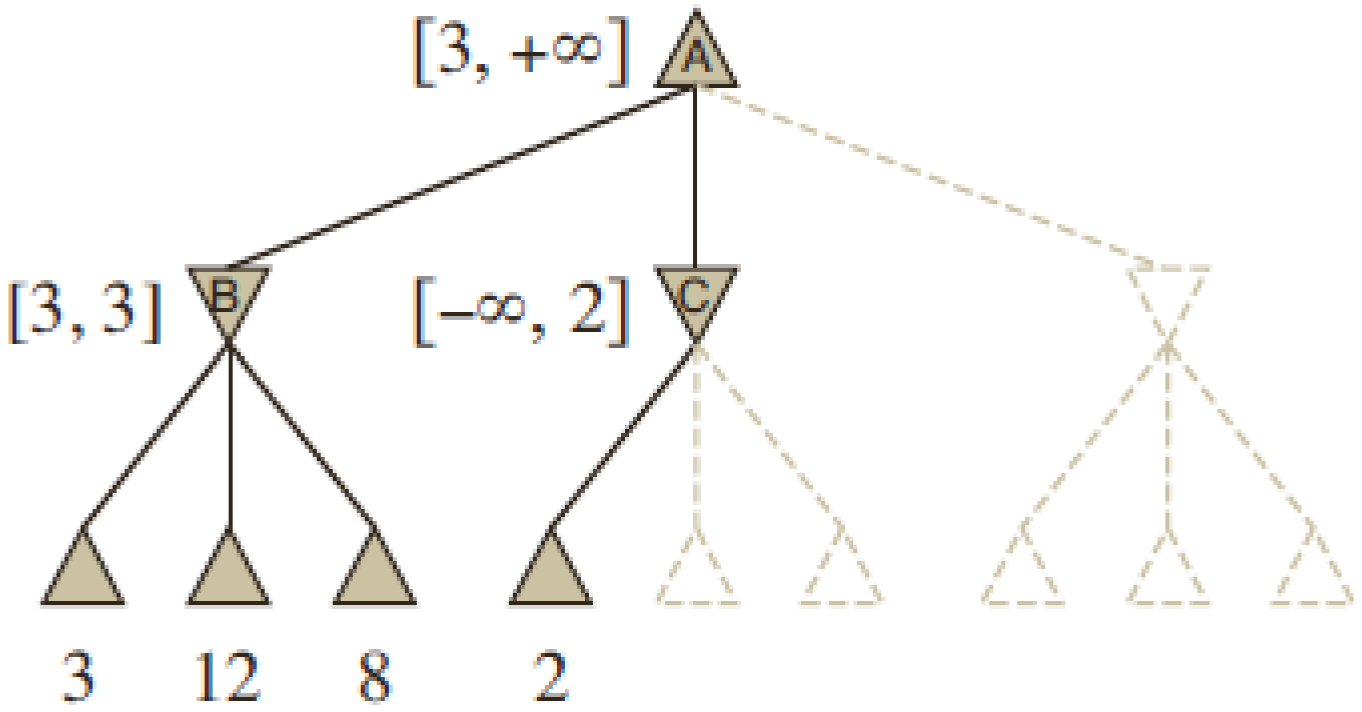
Alpha–Beta Pruning Example



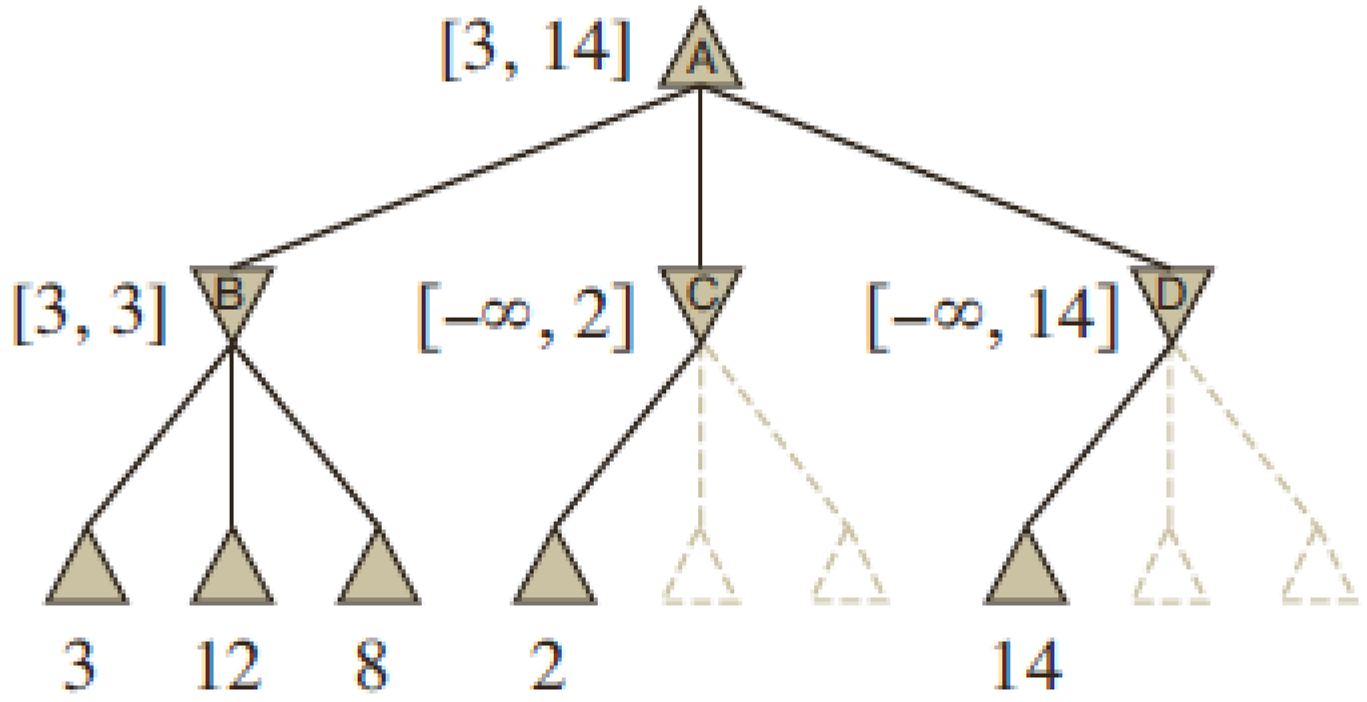
Alpha–Beta Pruning Example



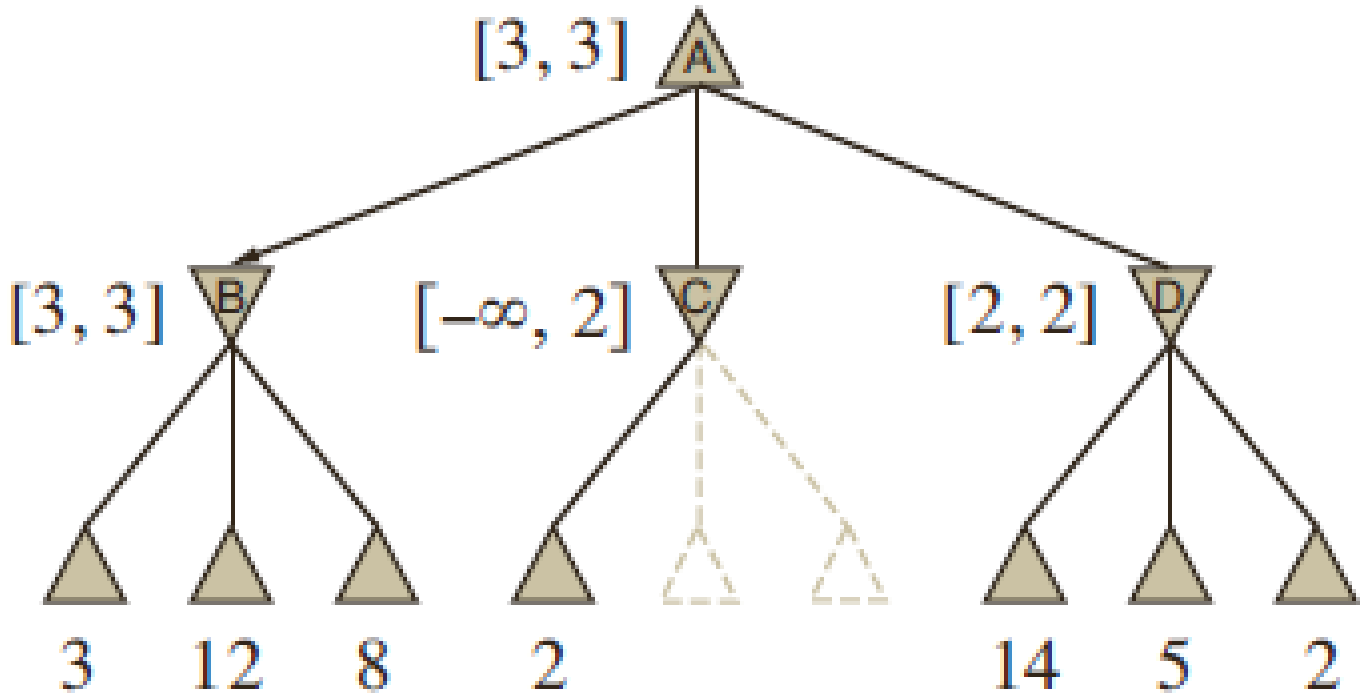
Alpha–Beta Pruning Example



Alpha–Beta Pruning Example



Alpha–Beta Pruning Example



Summary

- A game can be defined by
  - the **initial state** (how the board is set up),
  - the **legal actions** in each state,
  - the **result** of each action,
  - a **terminal test** (which says when the game is over), and
  - a **utility function** that applies to terminal states.
- In *two-player zero-sum games* with **perfect information**, the **minimax algorithm** can select optimal moves by a depth-first enumeration of the game tree.
- The **alpha–beta search algorithm** computes the same optimal move as **minimax**, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree so we need to cut the search off at some point and apply a heuristic evaluation function that estimates the utility of a state.