ADVERSARIAL SEARCH

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Eight (8)**	**Week No:**	**Nine(9)**	**Semester:**	
Lecturer:	*Dr. Abdus Salam*			*abdus.salam@aiub.edu*	

Lecture Outline

1. Game and Its Components
2. Game as a Search Problem
3. Perfect Decision in Two Players Games
4. MiniMax Algorithm
5. Imperfect Decision
6. Cutoff Search
7. Alpha-Beta Pruning

Game Playing: Introduction

Competitive environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

Mathematical **game theory**, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are **cooperative** or **competitive**

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).

This means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.

Games a Search Problem

- Some games can normally be defined in the form of a tree.

- Branching factor is usually an average of the possible number of moves at each node.

- This is a simple search problem: a player must search this search tree and reach a leaf node with a favorable outcome.

Tic-Tac-Toe

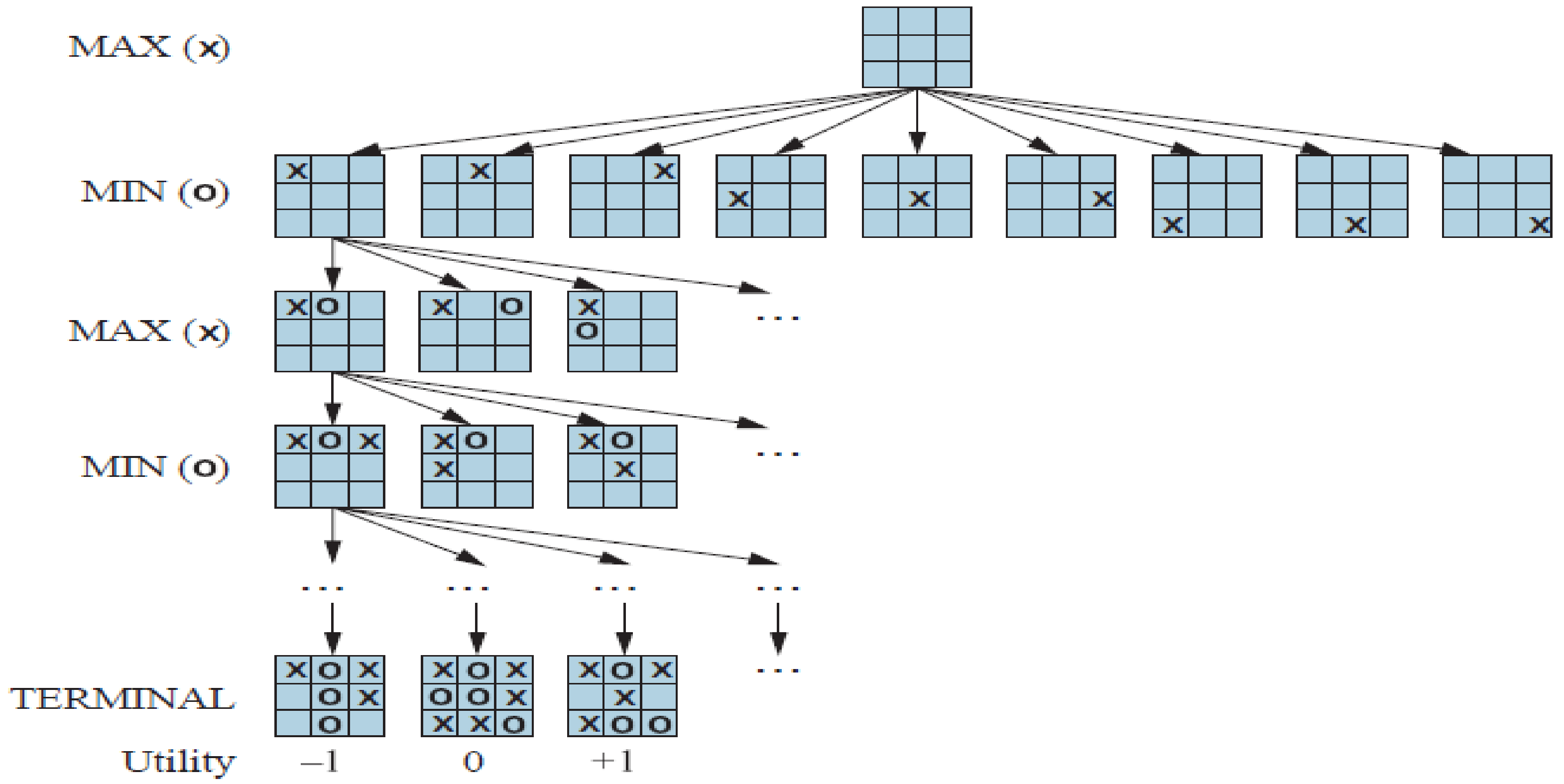


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Components of a Game

Initial state: Set-up specified by the rules, e.g., initial board configuration of chess.

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): Transition model defines the result of a move.

(2nd **ed.: Successor function:** list of (move , state) pairs specifying legal moves.)

Terminal-Test(s): Is the game finished? True if finished, false otherwise.

Utility function(s , p): Gives numerical value of terminal state s for player p.

E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.

E.g., win (+1), lose (0), and draw (1/2) in chess.

Two Player Game

Two players: Max and Min

Objective of both Max and Min to optimize winnings

Max must reach a terminal state with the highest utility

Min must reach a terminal state with the lowest utility

Game ends when either Max and Min have reached a terminal state

upon reaching a terminal state points maybe awarded or sometimes deducted

Search Problem Revisited

Simple problem is to reach a favorable terminal state

Problem Not so simple...

Max must reach a terminal state with as high a utility as possible regardless of Min’s moves

Max must develop a strategy that determines best possible move for each move Min makes.

Tic-Tac-Toe Revisited

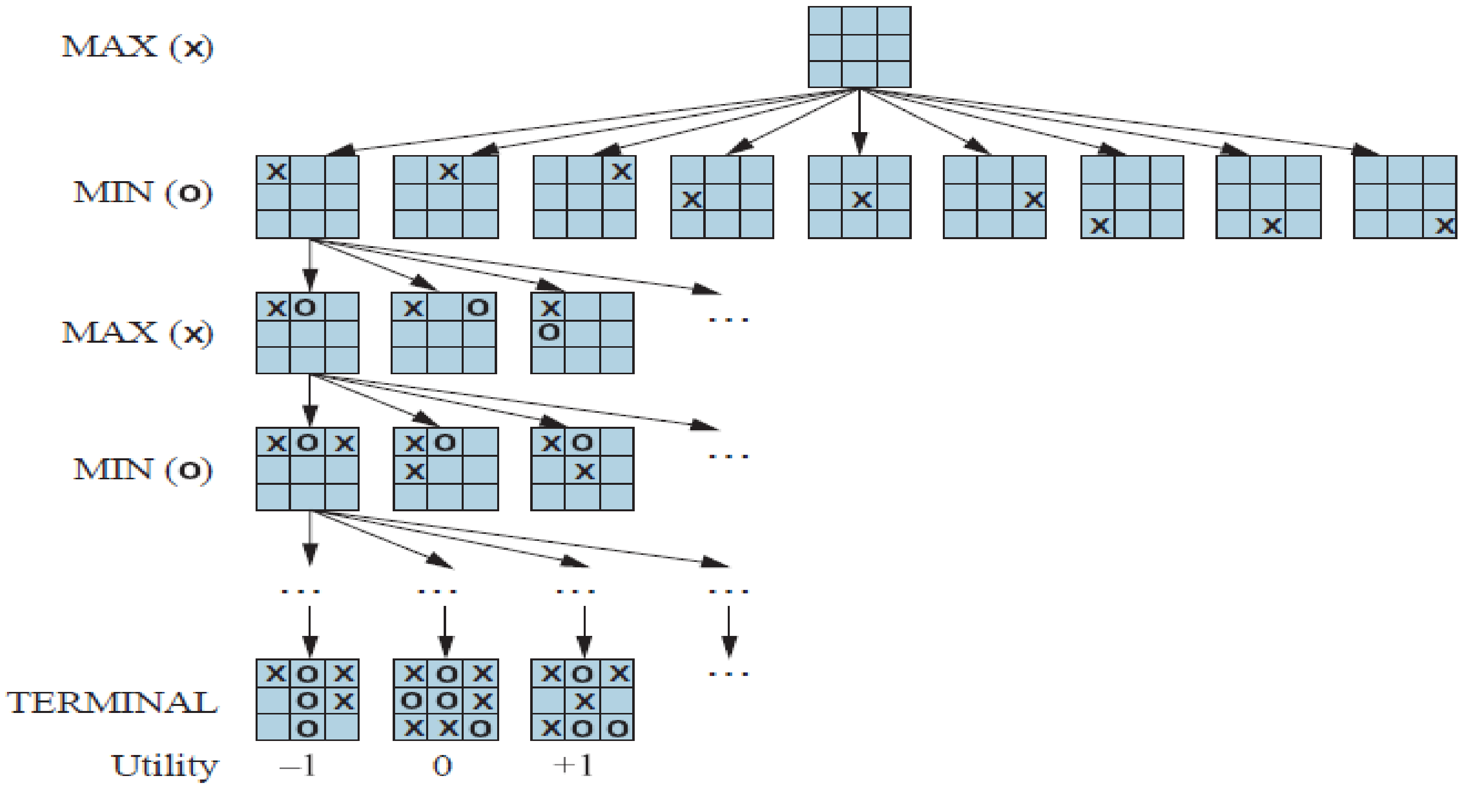
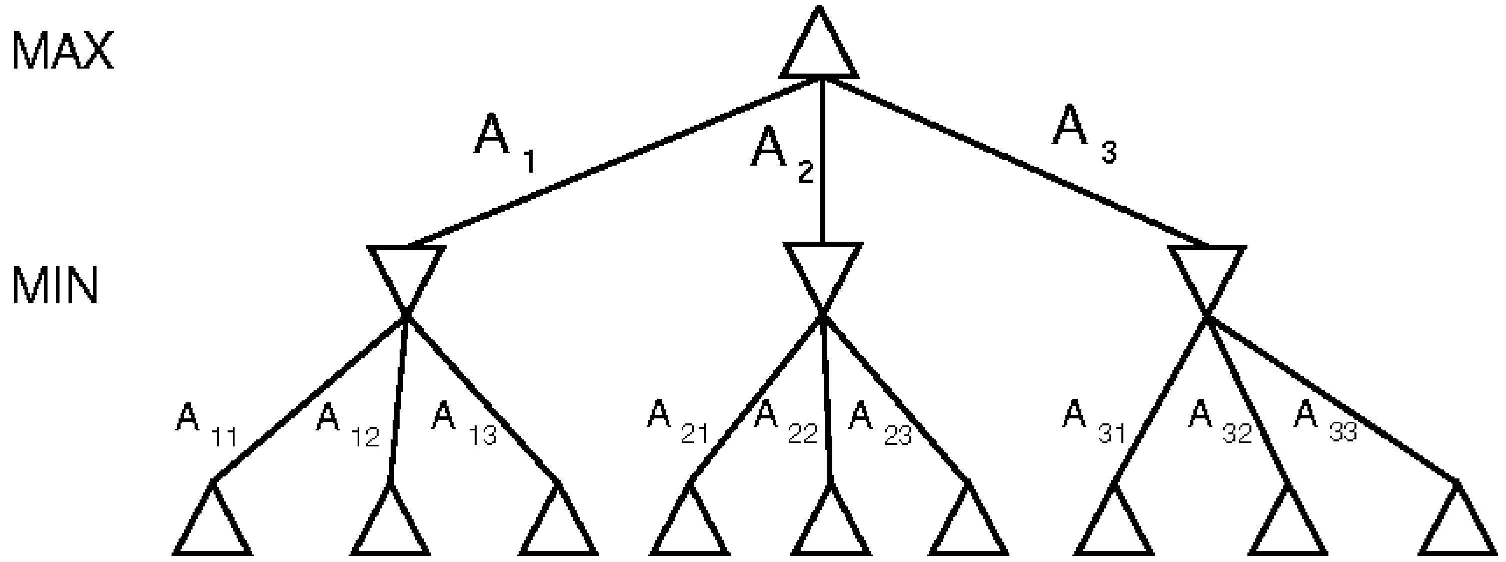


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Example: Two-Ply Game



Minimax Decision - maximizes the utility for Max based on the assumption that Min will attempt to Minimize this utility.

Minimax Algorithm

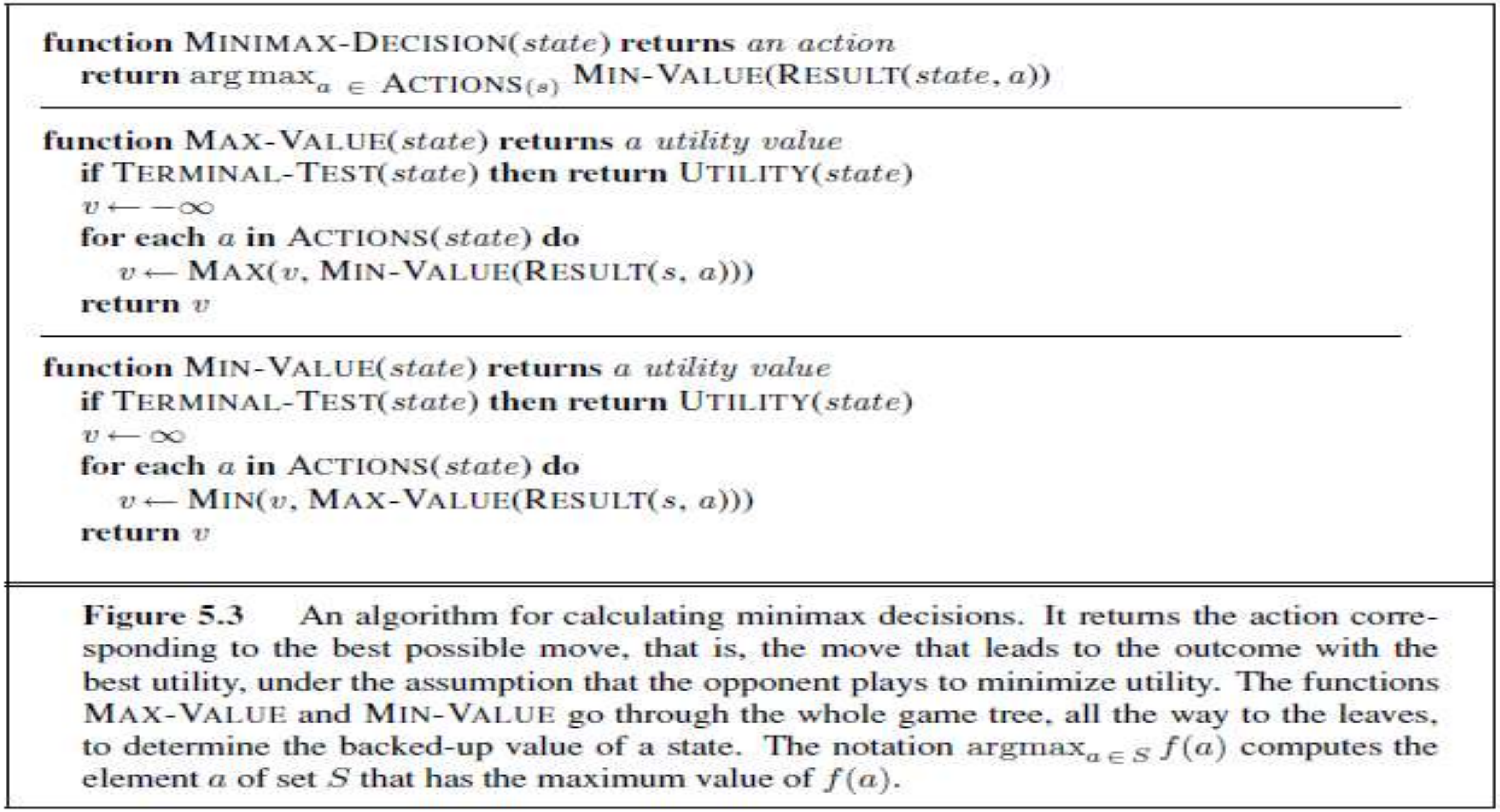
Minimax Algorithm determines optimum strategy for Max:

1. Generate the whole game tree, down to the leaves.
2. Apply utility (payoff) function to each leaf.
3. Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree. Back-up values from leaves through branch nodes:

   a Max node computes the Max of its child values

   a Min node computes the Min of its child values
4. At root: choose the move leading to the child of highest value.

MiniMax Pseudocode



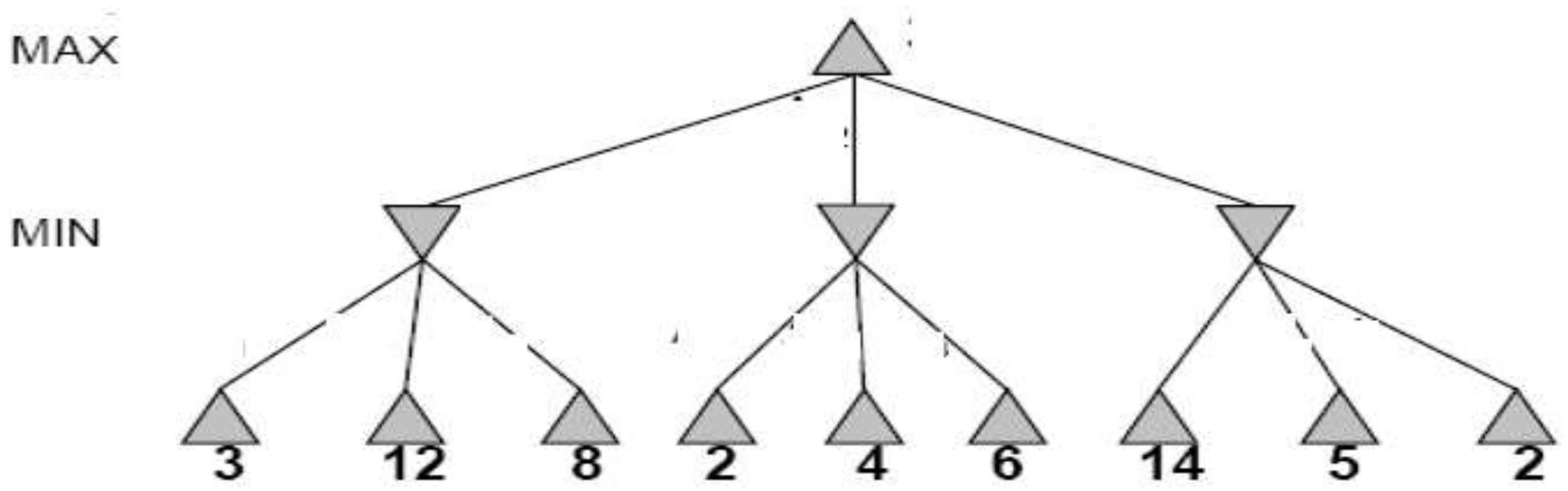
```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

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

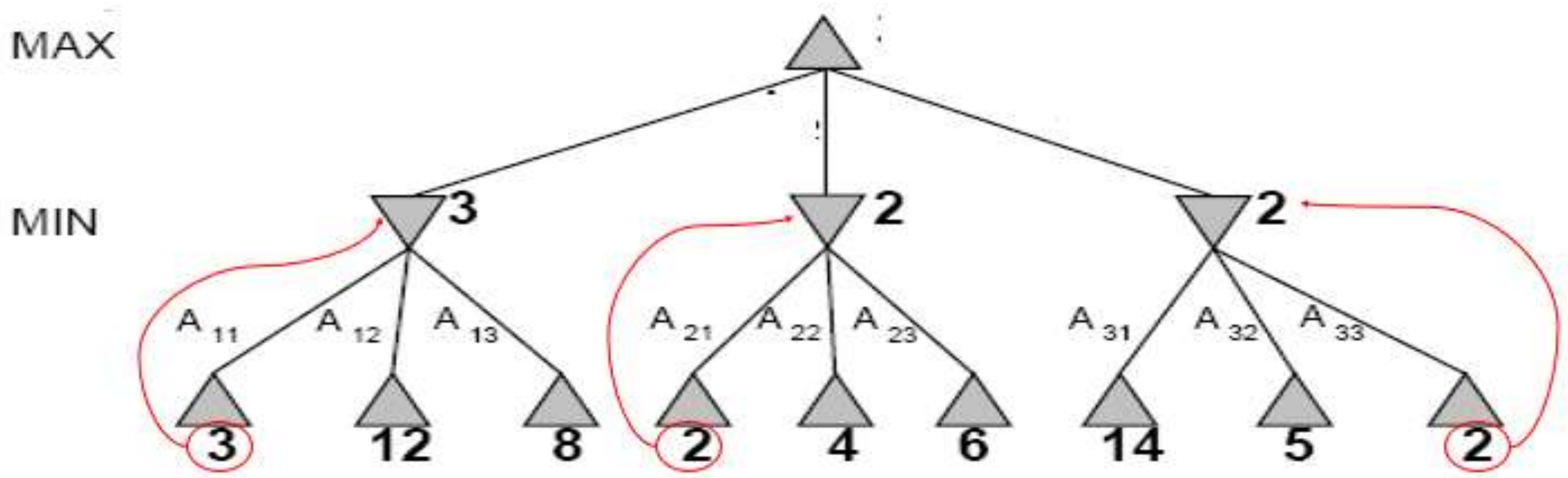
```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$
   for each a in ACTIONS(state) do
      $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

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 $f(a)$.

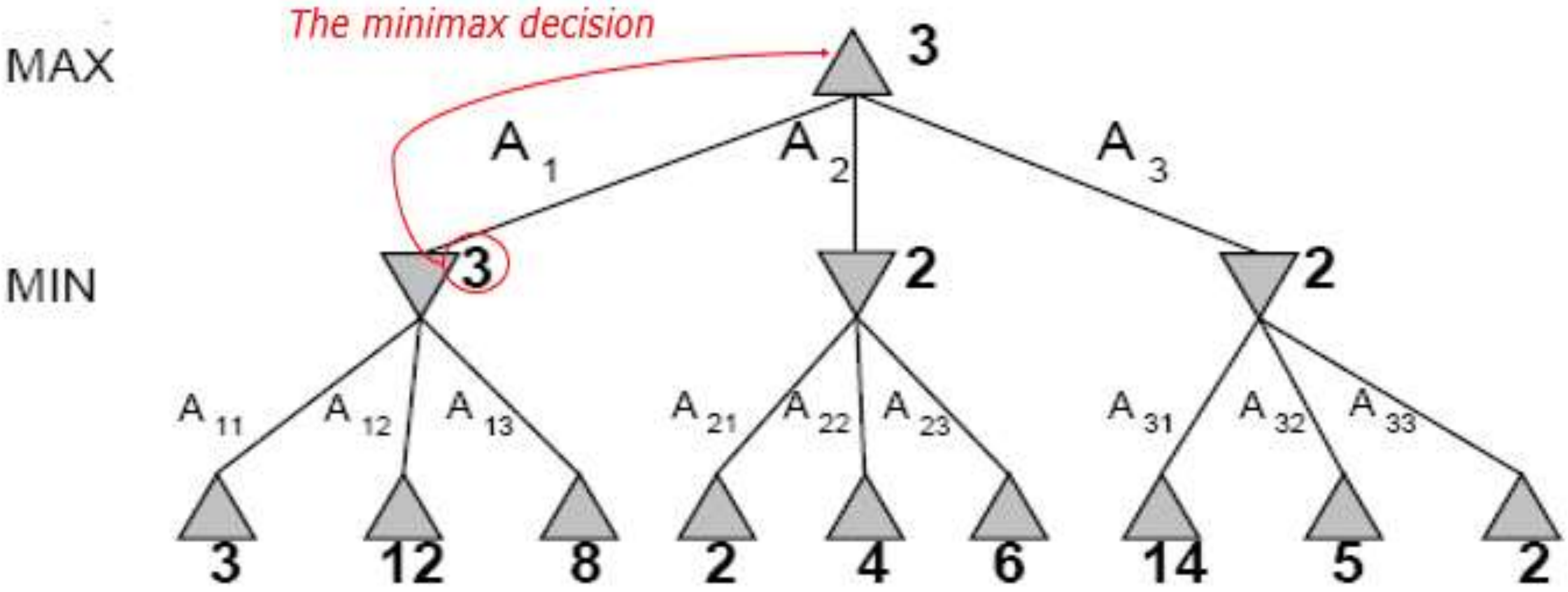
Two-Ply Game Tree



Two-Ply Game Tree



Two-Ply Game Tree



Two-Ply Game Tree

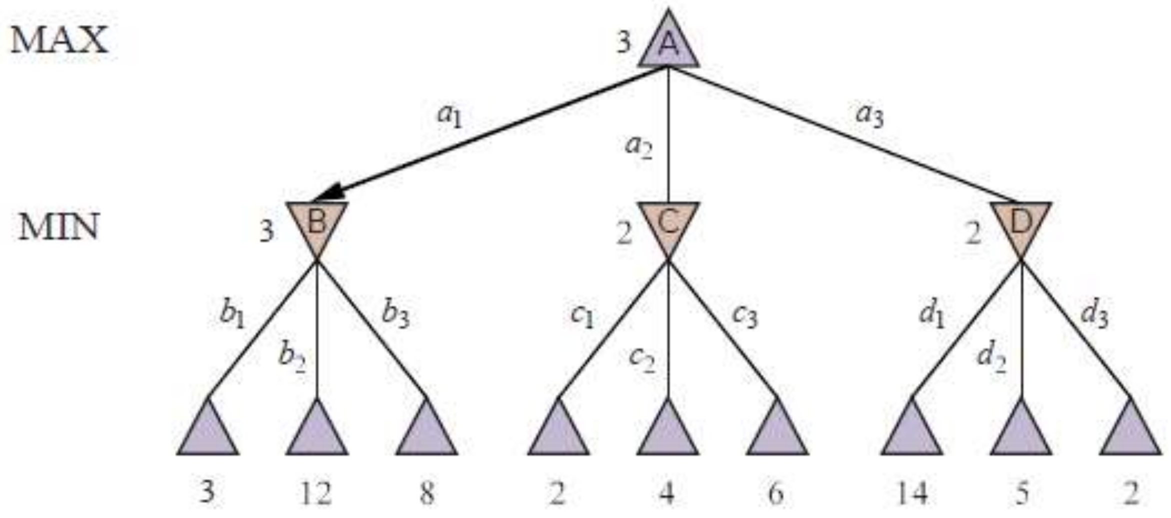


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Properties of minimax

Complete?

Yes (if tree is finite).

Optimal?

Yes (against an optimal opponent).

Time complexity?

$O(b^m)$

Space complexity?

$O(bm)$ (depth-first search, generate all actions at once)

$O(m)$ (backtracking search, generate actions one at a time)

Game Tree Size

Tic-Tac-Toe

- b ≈ 5 legal actions per state on average, total of 9 plies in game.
  - "ply" = one action by one player, "move" = two plies.
- $5^9 = 1,953,125$
- $9! = 362,880$
- → **exact solution quite reasonable**

Is There Another Way?

Take Chess on average has:

35 branches and

usually at least 100 moves

so game space is: 35^{100}

Is this a realistic game space to search?

Since time is important factor in gaming searching this game space is highly undesirable.

Imperfect Decisions

- Many game produce very large search trees.

- Cutoffs must be implemented due to time restrictions,

Evaluation Functions

A function that returns an estimate of the expected utility of the game from a given position.

Given the present situation give an estimate as to the value of the next move.

The performance of a game-playing program is dependant on the quality of the evaluation functions.

How to Judge Quality

Evaluation functions must agree with the utility functions on the terminal states.

It must not take too long (trade off between accuracy and time cost).

Should reflect actual chance of winning.

Design of Evaluation Function

Different evaluation functions must depend on the nature of the game.

Encode the quality of a position in a number that is representable within the framework of the given language.

Design a heuristic for value to the given position of any object in the game.

Material Advantage Evaluation Functions

Values of the pieces are judge independent of other pieces on the board. A value is returned based on the material value of the computer minus the material value of the player.

Weighted Linear Functions

$w_1f_1+w_2f_2+\ldots\ldots w_nf_n$

w's are weight of the pieces
f's are features of the particular positions

Heuristic Evaluation Functions

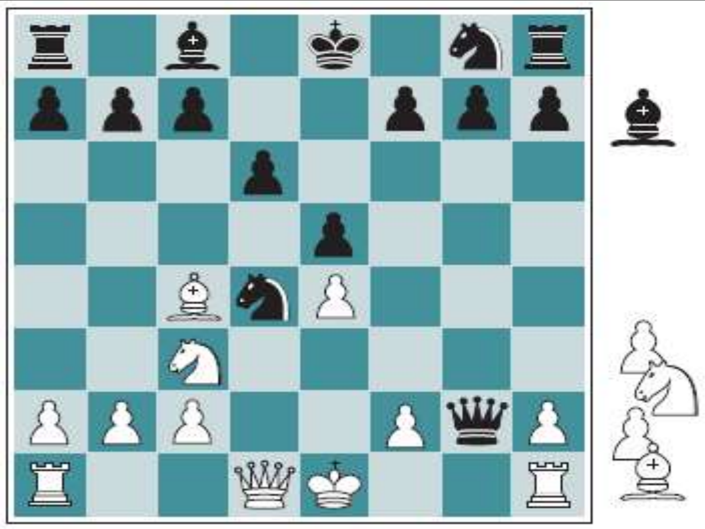
An Evaluation Function:

- Estimates how good the current board configuration is for a player.
- Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the players.
- Often called "static" because it is called on a static board position.
- Othello: Number of white pieces - Number of black pieces
- Chess: Value of all white pieces - Value of all black pieces

Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

If the board evaluation is X for a player, it's -X for the opponent "Zero-sum game"

Evaluation Function


(a) White to move

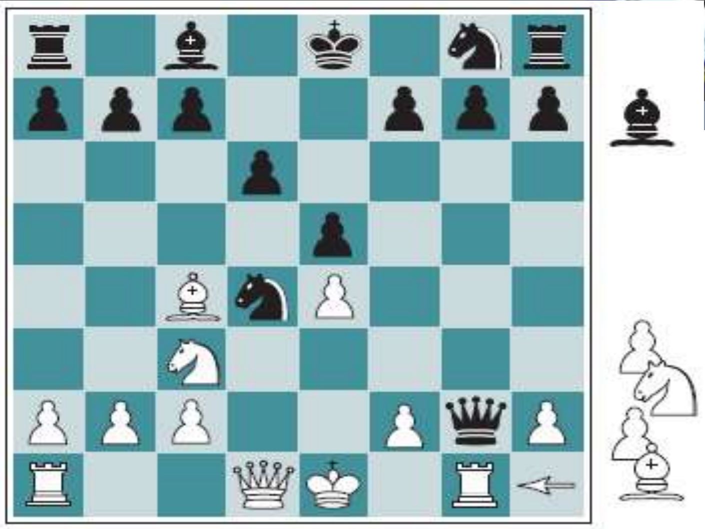

(b) White to move

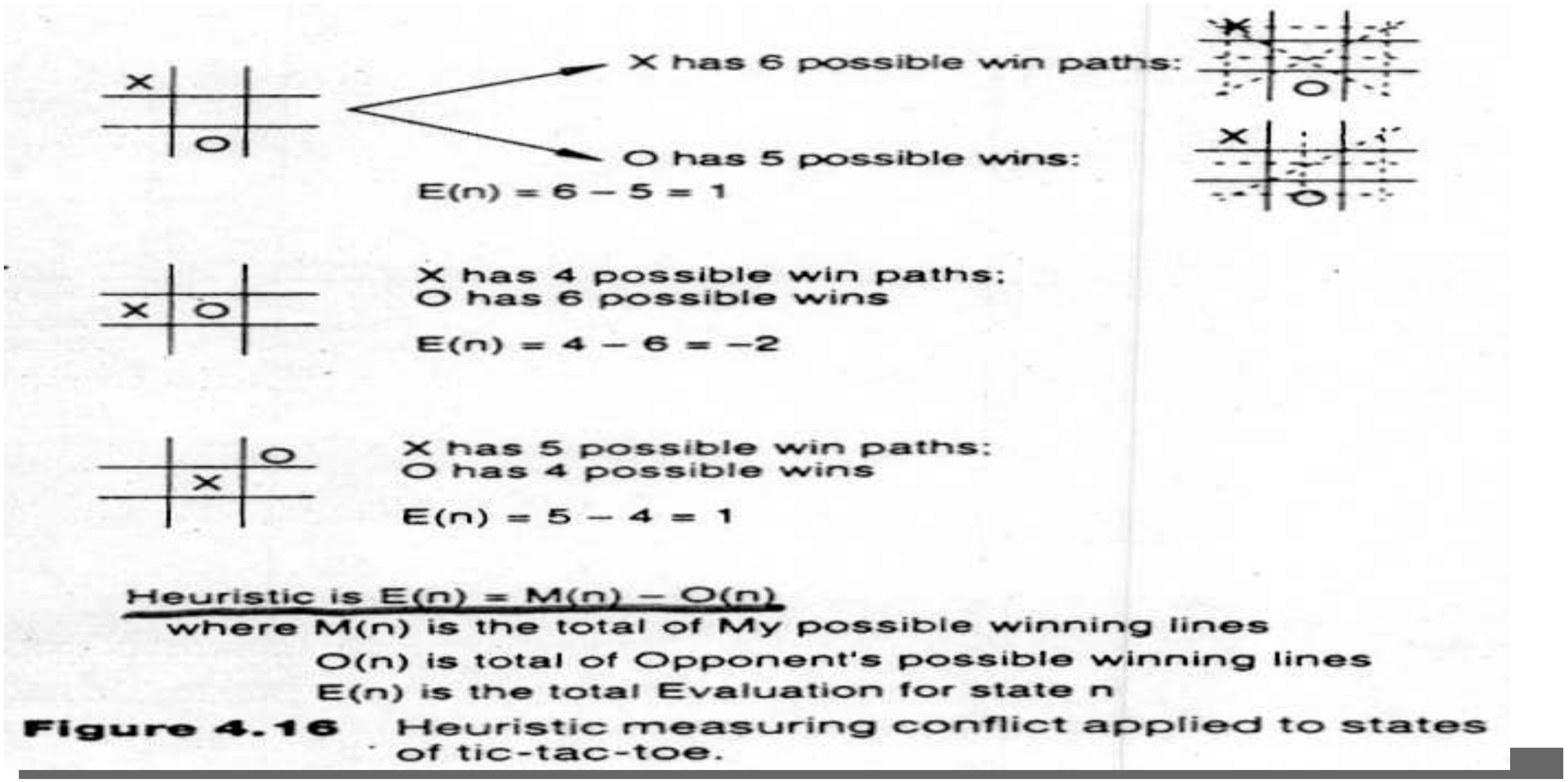
Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

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.

Heuristic evaluation function for tic-tac-toe



X has 6 possible win paths:

O has 5 possible wins:

E(n) = 6 – 5 = 1

X has 4 possible win paths;
O has 6 possible wins

E(n) = 4 – 6 = –2

X has 5 possible win paths;
O has 4 possible wins

E(n) = 5 – 4 = 1

Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

Figure 4.16 Heuristic measuring conflict applied to states of tic-tac-toe.

Cutoff Search

Cutting of searches at a fixed depth dependent on time

The deeper the search the more information is available to the program the more accurate the evaluation functions

Iterative deepening – when time runs out the program returns the deepest completed search.

Is searching a node deeper better than searching more nodes?

Consequences

Evaluation function might return an incorrect value.

If the search in cutoff and the next move results involves a capture, then the value that is return maybe incorrect.

Horizon problem:
Moves that are pushed deeper into the search trees may result in an oversight by the evaluation function.

Pruning

What is pruning?

The process of eliminating a branch of the search tree from consideration without examining it.

Why prune?

To eliminate searching nodes that are potentially unreachable.

To speedup the search process.

Alpha-Beta Pruning

A technique to find the optimal solution according to a limited depth search using evaluation functions.

Returns the same choice as minimax cutoff decisions but examines fewer nodes.

Gets its name from the two variables that are passed along during the search which restrict the set of possible solutions.

Alpha-beta: Definitions

Alpha –
the value of the best choice so far along the path for MAX.

Beta –
the value of the best choice (lowest value) so far along the path for MIN.

Implementation

Set root node alpha to negative infinity and beta to positive infinity.

Search depth first, propagating alpha and beta values down to all nodes visited until reaching desired depth.

Apply evaluation function to get the utility of this node.

Implementation (Cont'd)

The Max player will only update the value of alpha.

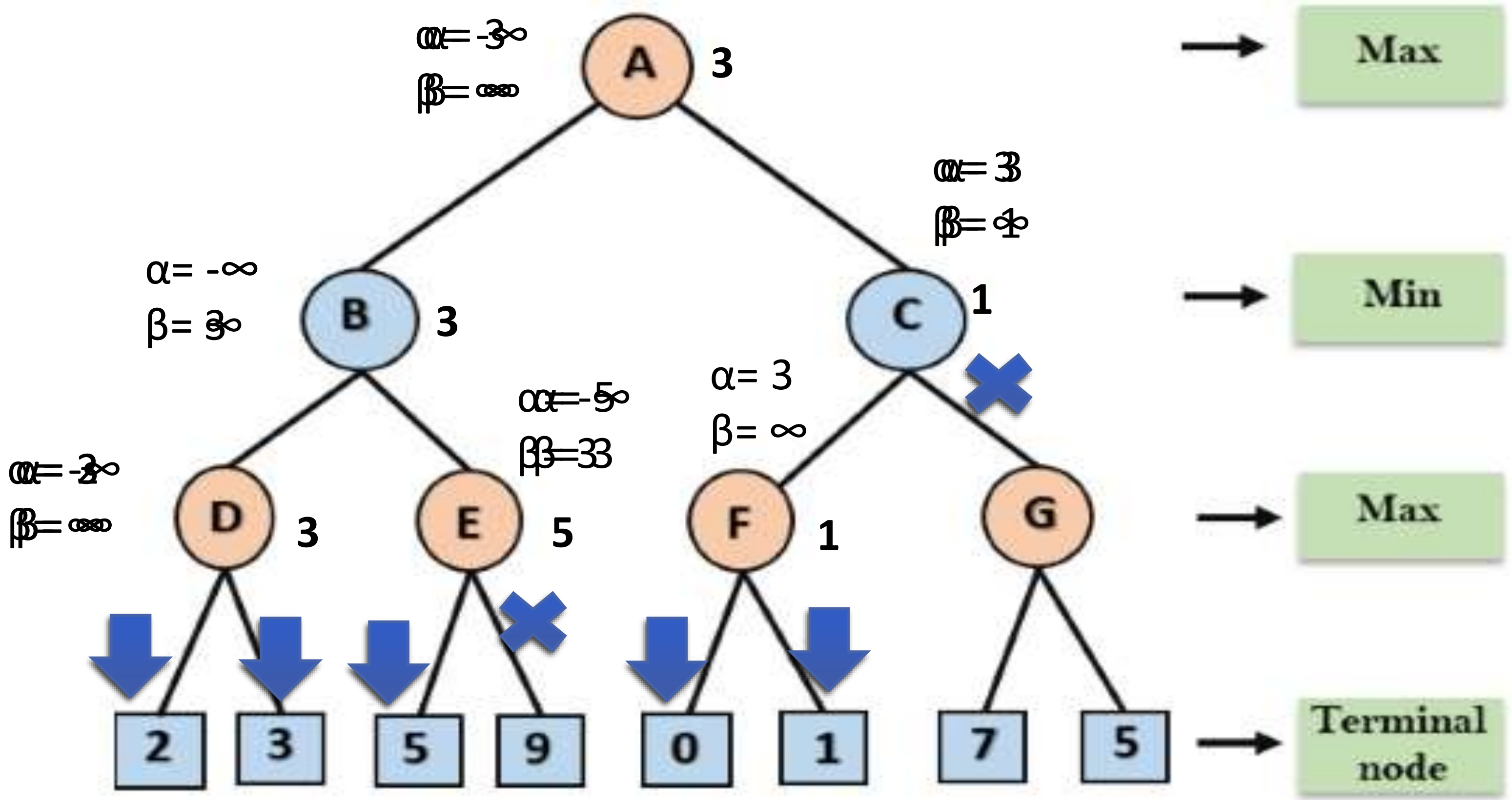
The Min player will only update the value of beta.

While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.

We will only pass the alpha, beta values to the child nodes.

Prune whenever $\alpha \geq \beta$.

Alpha-Beta Example



General alpha-beta pruning

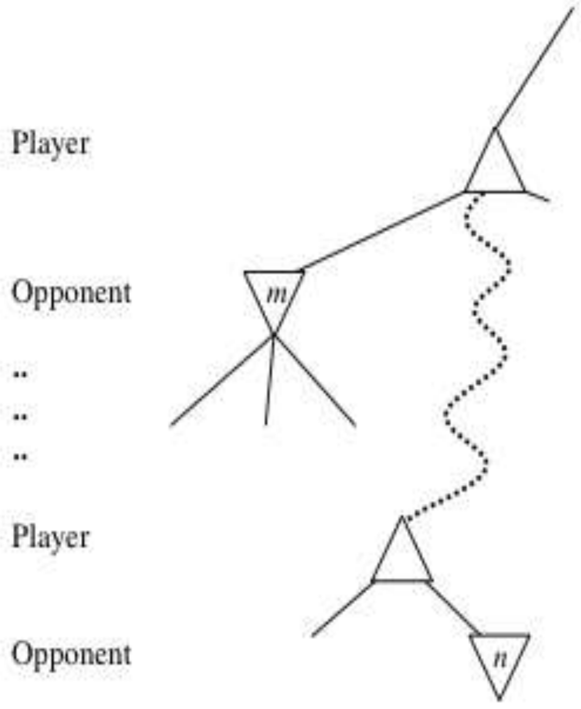
Consider a node *n* in the tree ---

If player has a better choice at:

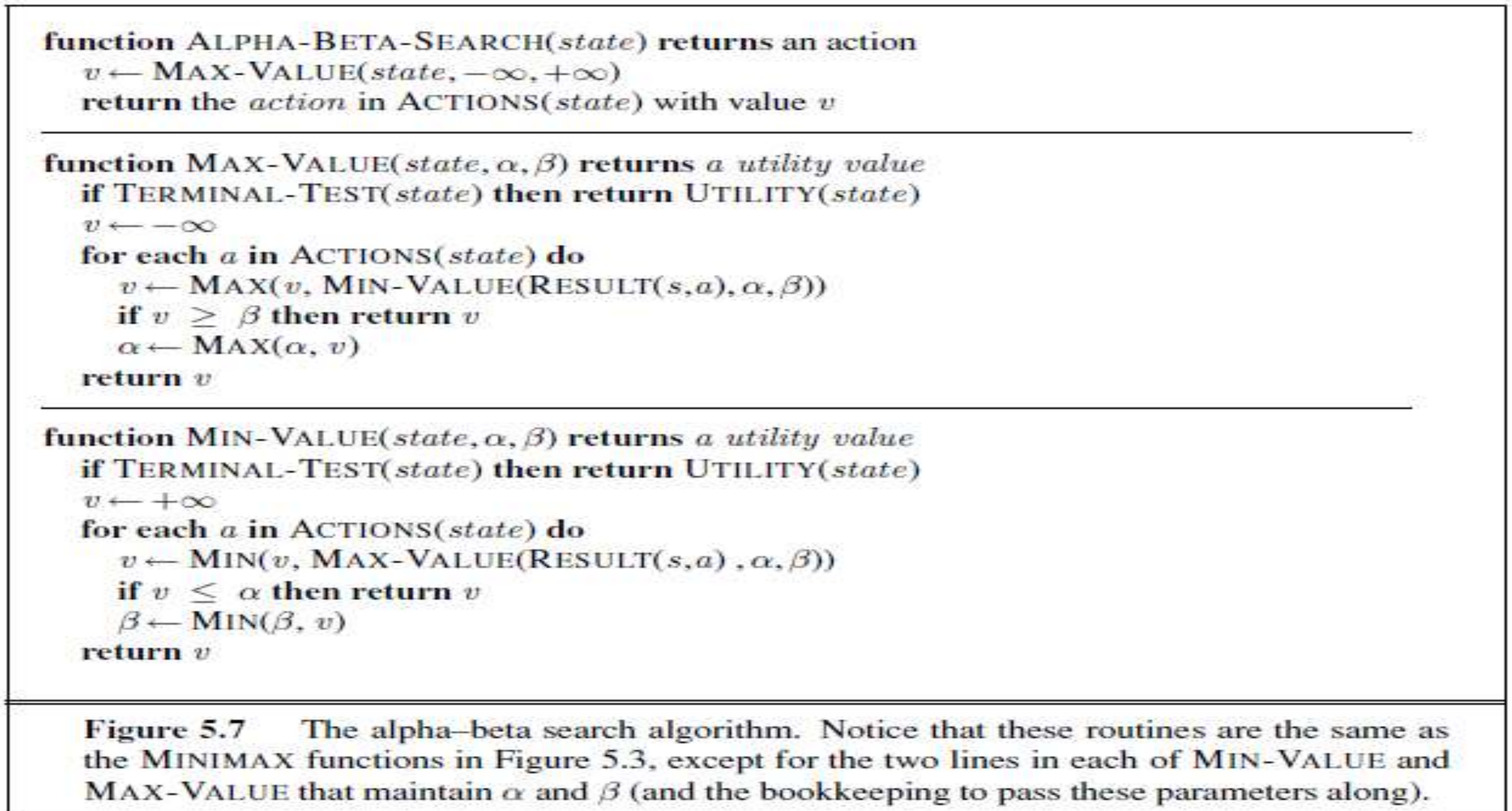
Parent node of n
Or any choice point further up

Then *n* will never be reached in play.

Hence, when that much is known about *n*, it can be pruned.



Alpha-Beta Search Algorithm



```
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($s,a$), $\alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    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($s,a$) , $\alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta \leftarrow$ MIN($\beta$, $v$)
    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).

Effectiveness of Alpha-Beta Search

Worst-Case

- branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search

Best-Case

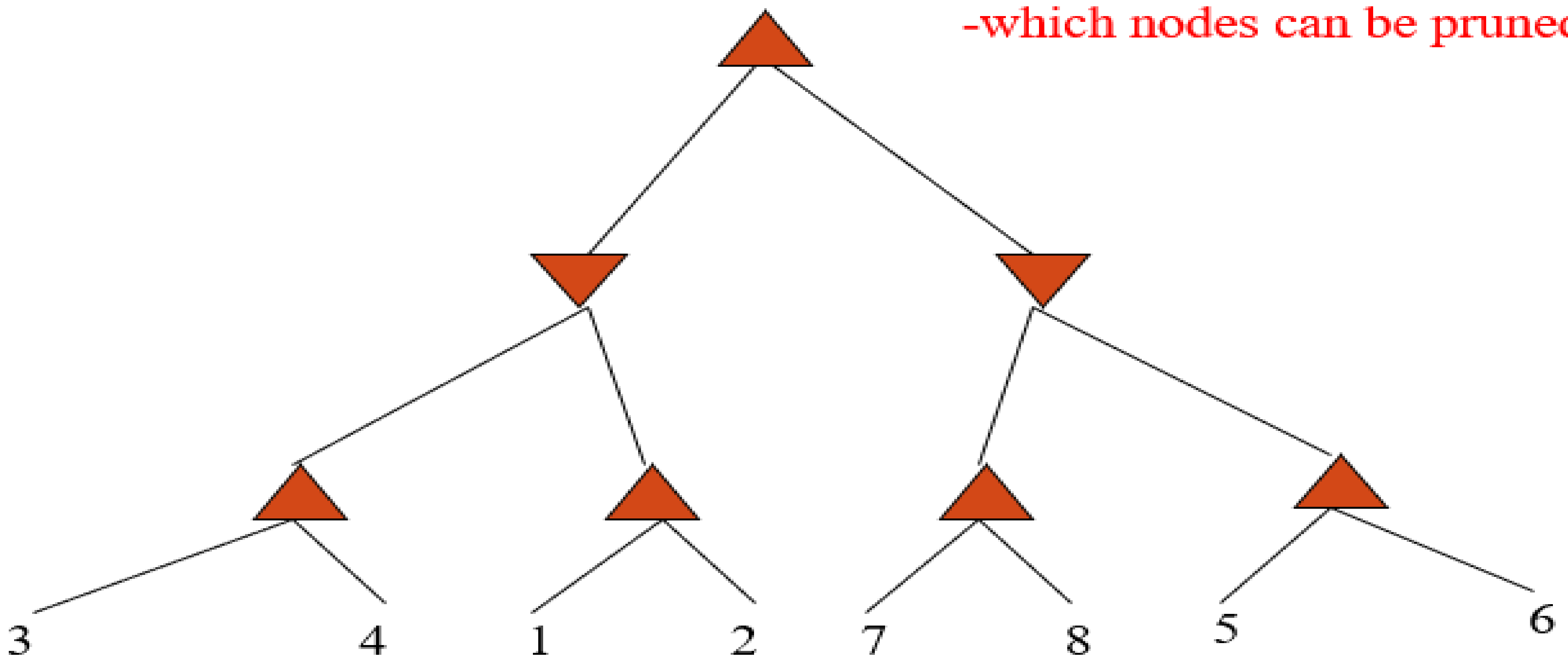
- each player's best move is the left-most child (i.e., evaluated first)
- in practice, performance is closer to best rather than worst-case
- E.g., sort moves by the remembered move values found last time.
- E.g., expand captures first, then threats, then forward moves, etc.
- E.g., run Iterative Deepening search, sort by value last iteration.

In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$

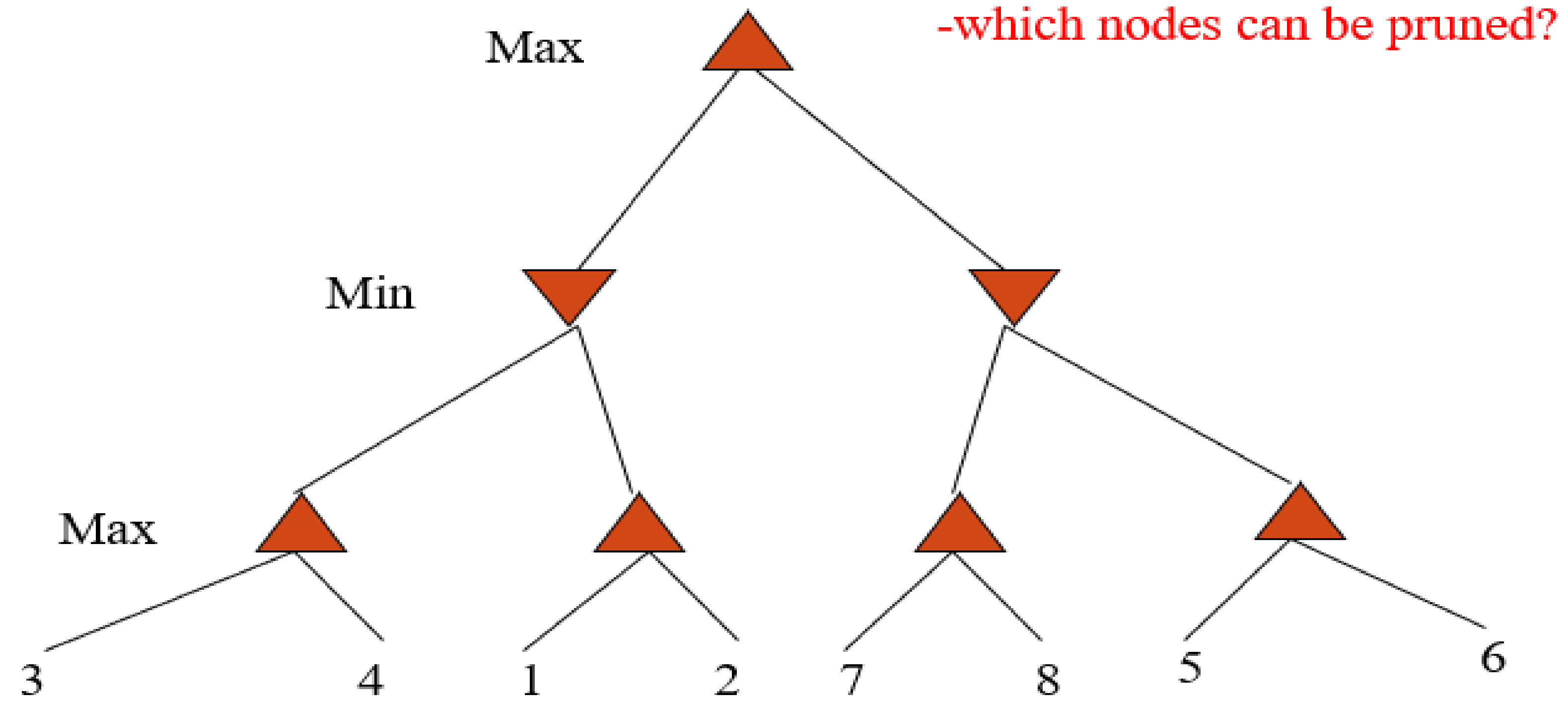
- this is the same as having a branching factor of sqrt(b),
  - $(\text{sqrt}(b))^d = b^{(d/2)}$,i.e., we effectively go from b to square root of b
- e.g., in chess go from b ~ 35 to b ~ 6
  - this permits much deeper search in the same amount of time

Example

-which nodes can be pruned?



Answer to Example

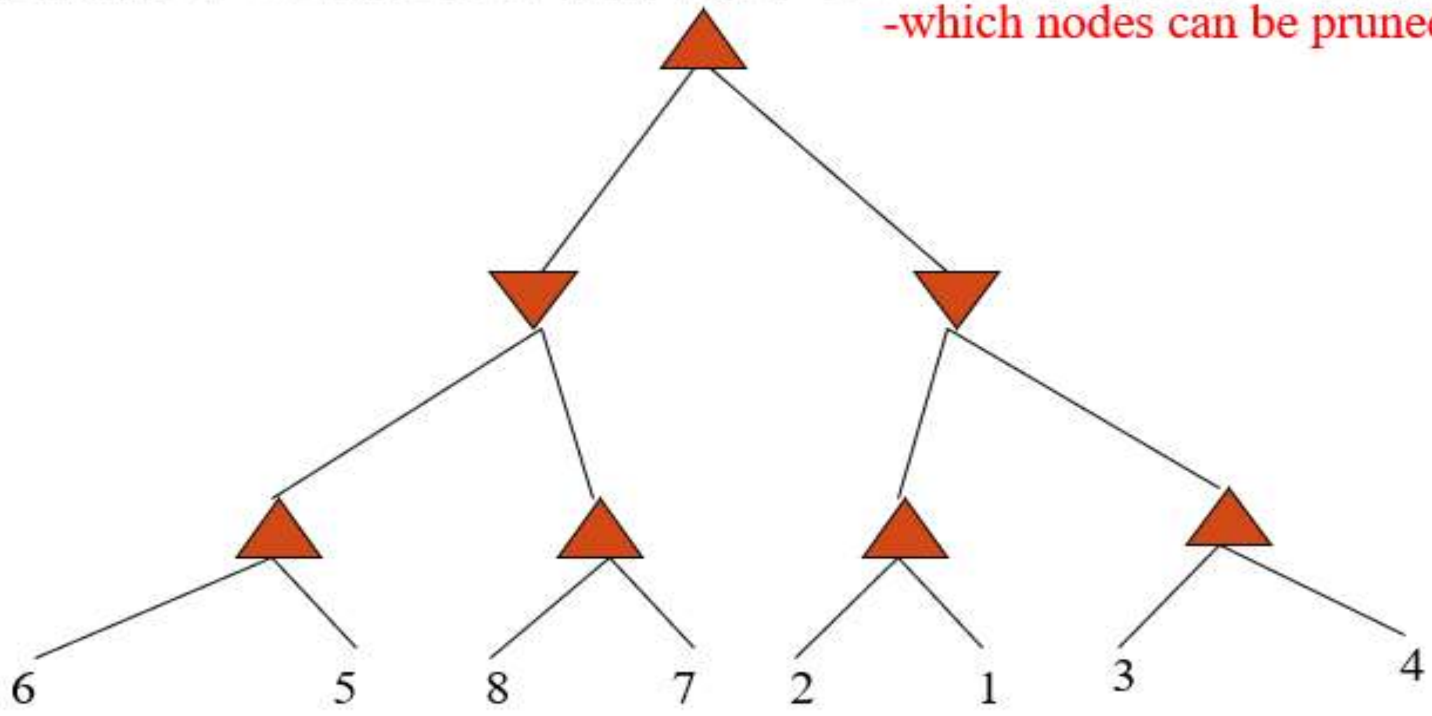


Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).

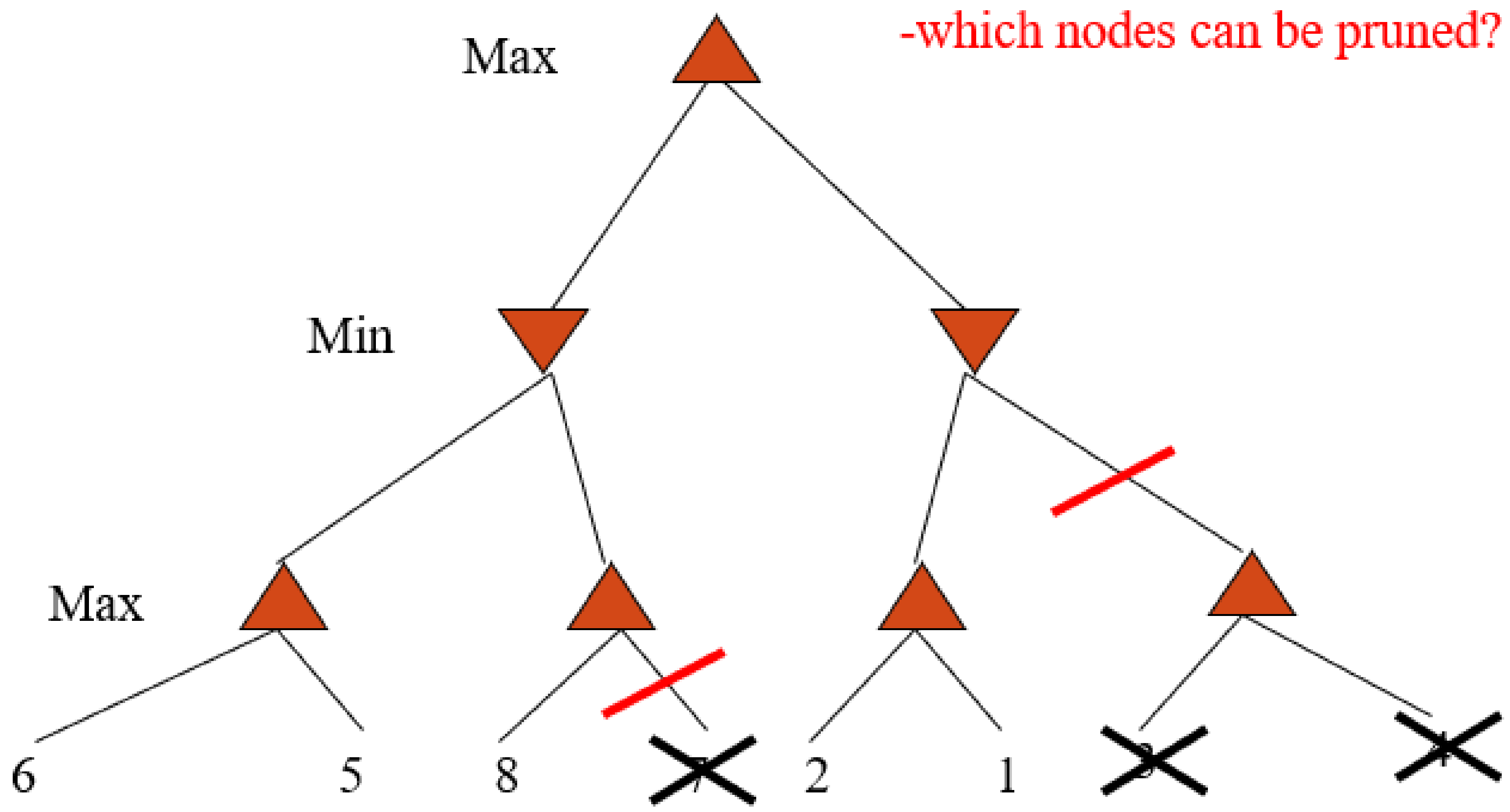
Second Example

(THE EXACT MIRROR IMAGE OF THE FIRST EXAMPLE)

-which nodes can be pruned?



Answer



Answer: LOTS! Because the most favorable nodes for both are explored first (i.e., in the diagram, are on the left-hand side).

Final Comments about Alpha-Beta Pruning

Pruning does not affect final results

Entire subtrees can be pruned.

Good move *ordering* improves effectiveness of pruning

Repeated states are again possible.

Store them in memory = transposition table

Problems

If there is only one legal move, this algorithm will still generate an entire search tree.

Designed to identify a "best" move, not to differentiate between other moves.

Overlooks moves that forfeit something early for a better position later.

Evaluation of utility usually not exact.

Assumes opponent will always choose the best possible move.

References

1. Chapter 5: Adversarial Search , Pages 161-176
“Artificial Intelligence: A Modern Approach,” by Stuart J. Russell and Peter Norvig,

Books

1. “Artificial Intelligence: A Modern Approach,” by Stuart J. Russell and Peter Norvig.
2. "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", by George F. Luger, (2002)
3. "Artificial Intelligence: Theory and Practice", by Thomas Dean.
4. "AI: A New Synthesis", by Nils J. Nilsson.
5. “Programming for machine learning,” by J. Ross Quinlan,
6. “Neural Computing Theory and Practice,” by Philip D. Wasserman, .
7. “Neural Network Design,” by Martin T. Hagan, Howard B. Demuth, Mark H. Beale, .
8. “Practical Genetic Algorithms,” by Randy L. Haupt and Sue Ellen Haupt.
9. “Genetic Algorithms in Search, optimization and Machine learning,” by David E. Goldberg.
10. "Computational Intelligence: A Logical Approach", by David Poole, Alan Mackworth, and Randy Goebel.
11. “Introduction to Turbo Prolog”, by Carl Townsend.