Game Trees

Note: Some slides and/or pictures are adapted from Lecture slides / Books of

- Dr Zafar Alvi.
- Text Book - *Aritificial Intelligence Illuminated* by Ben Coppin, Narosa Publishers.
- Ref Books
  - *Artificial Intelligence- Structures & Strategies for Complex Problem Solving by* George F. Luger, 4th edition, Pearson Education.
  - *Artificial Intelligence A Modern Approach* by Stuart Russell & Peter Norvig.
  - Artificial Intelligence, Third Edition by Patrick Henry Winston

Game Trees

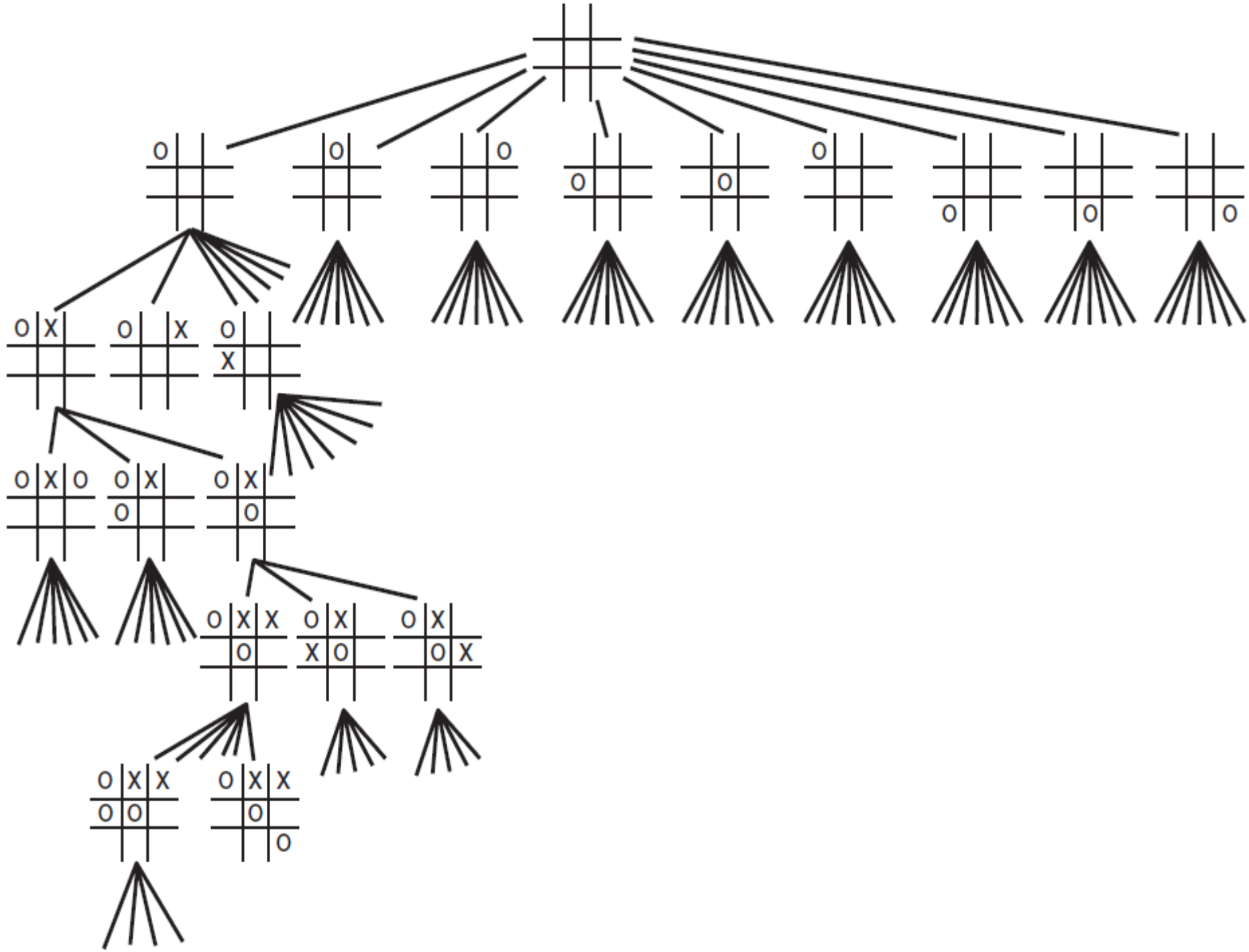
- Many two-player games can be efficiently represented using trees, called **game trees.**
- A game tree is an instance of a tree in which the root node represents the state before any moves have been made, the nodes in the tree represent possible states of the game (or positions), and arcs in the tree represent moves.

Game Trees

- It is usual to represent the two players' moves on alternate levels of the game tree, so that all edges leading from the root node to the first level represent possible moves for the first player, and edges from the first level to the second represent moves for the second player, and so on.
- Leaf nodes in the tree represent final states, where the game has been **won, lost, or drawn**.

Game Trees

- One approach to playing a game might be for the computer to use a tree search algorithm such as depth-first or breadth-first search, looking for a goal state (i.e., a final state of the game where the computer has won).
- Unfortunately, this approach does not work because there is another intelligence involved in the game.
- Consider the partial game tree shown in next slide of tic-tac-toe



Game Trees

- The branching factor of the root node is 9 because there are nine squares in which the computer can place its first naught. The branching factor of the next level of the tree is 8, then 7 for the next level, and so on.
- For a computer to use this tree to make decisions about moves in a game of tic-tac-toe, it needs to use an **evaluation function**, which enables it to decide whether a given position in the game is good or bad.

Evaluation Functions

- Evaluation functions (also known as **static evaluators** because they are used to evaluate a game from just one static position) are vital to most game-playing computer programs.
- This is because it is almost never possible to search the game tree fully due to its size.
- Hence, a search will rarely reach a leaf node in the tree at which the game is either won, lost, or drawn, which means that the software needs to be able to **cut off** search and evaluate the position of the board at that node.
- Hence, an evaluation function is used to examine a particular position of the board and estimate how well the computer is doing, or how likely it is to win from this position.

Game Trees

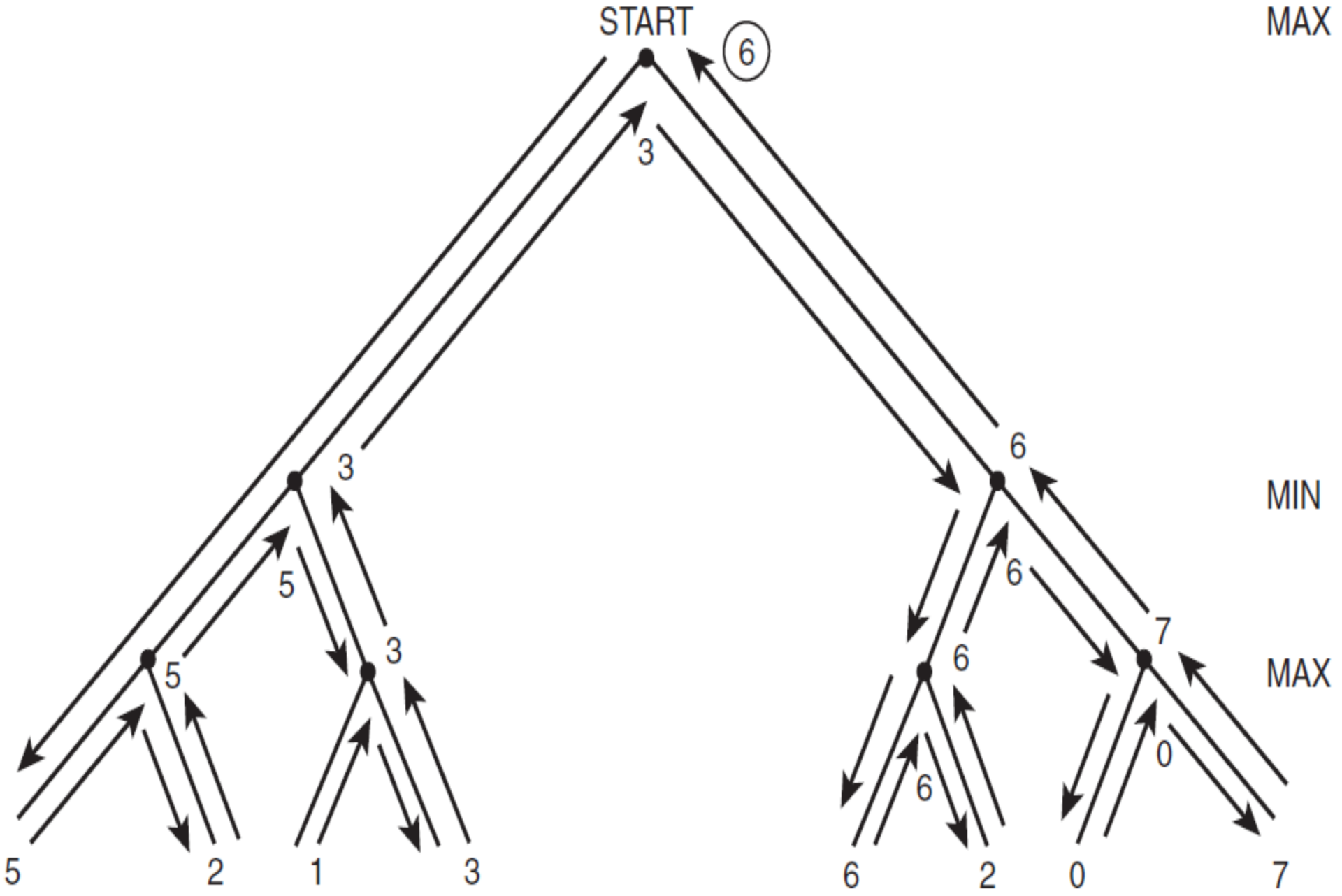
- One question is how the evaluation function will compare two positions.

Minimax

- When evaluating game trees, it is usual to assume that the computer is attempting to maximize some score that the opponent is trying to minimize.
- Normally we would consider this score to be the result of the evaluation function for a given position, so we would usually have a high positive score mean a good position for the computer, a score of 0 mean a neutral position, and a high negative score mean a good position for the opponent.

Minimax

- The Minimax algorithm is used to choose good moves. It is assumed that a suitable static evaluation function is available, which is able to give an overall score to a given position.
- In applying Minimax, the static evaluator will only be used on leaf nodes, and the values of the leaf nodes will be filtered up through the tree, to pick out the best path that the computer can achieve.


START
6
MAX
3
3
6
MIN
5
5
3
6
7
6
MAX
6
0
5
2
1
3
6
2
0
7

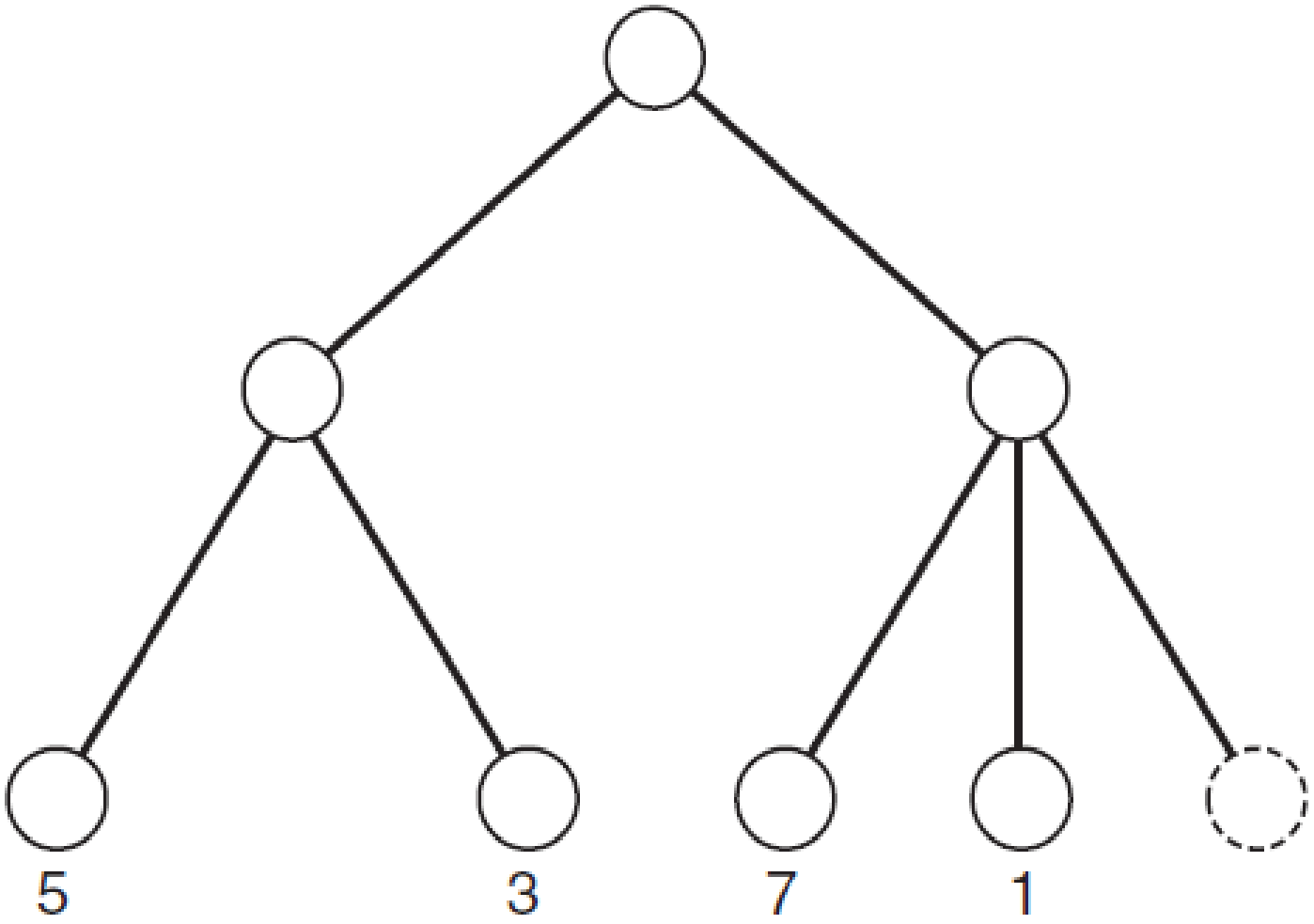
Bounded Lookahead

- Minimax, as we have defined it, is a very simple algorithm and is unsuitable for use in many games, such as chess where the game tree is extremely large.
- The problem is that in order to run Minimax, the entire game tree must be examined, and for games such as chess, this is not possible due to the potential depth of the tree and the large branching factor.
- In such cases, **bounded lookahead** is very commonly used and can be combined with Minimax.
- The idea of bounded lookahead is that the search tree is only examined to a particular depth.
- All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function.

Alpha–Beta Pruning

- Bounded lookahead can help to make smaller the part of the game tree that needs to be examined.
- In some cases, it is extremely useful to be able to **prune** sections of the game tree.
- Using alpha–beta pruning, it is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient.
- The principle behind alpha–beta pruning is that if a move is determined to be worse than another move that has already been examined, then further examining the possible consequences of that worse move is pointless.

Consider the partial game tree



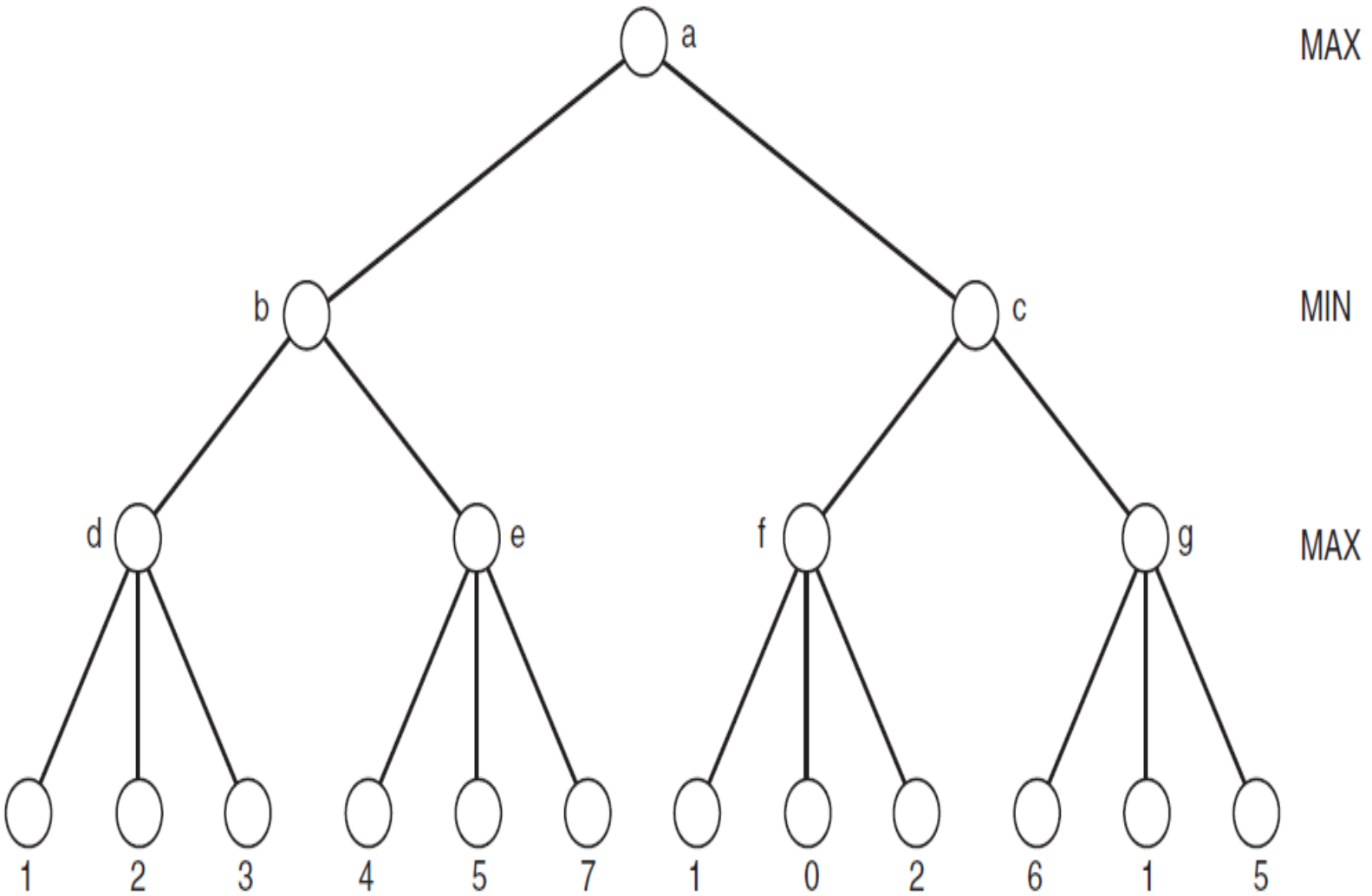
Alpha–Beta Pruning

- This very simple game tree has five leaf nodes.
- The top arc represents a choice by the computer, and so is a **maximizing level** (in other words, the top node is a max node).
- After calculating the static evaluation function for the first four leaf nodes, it becomes unnecessary to evaluate the score for the fifth.
- The reason for this can be understood as follows: In choosing the left-hand path from the root node, it is possible to achieve a score of 3 or 5.
- Because this level is a minimizing level, the opponent can be expected to choose the move that leads to a score of 3.

Alpha–Beta Pruning

- So, by choosing the left-hand arc from the root node, the computer can achieve a score of 3.
- By choosing the right-hand arc, the computer can achieve a score of 7 or 1, or a mystery value. Because the opponent is aiming to minimize the score, he or she could choose the position with a score of 1, which is worse than the value the computer could achieve by choosing the left-hand path.
- So, the value of the rightmost leaf node doesn't matter—the computer must not choose the right-hand arc because it definitely leads to a score of **at best** 1 (assuming the opponent does not irrationally choose the 7 option).

Example



Find

- *Some Studies in Machine Learning Using the Game of Checkers, Arthur Samuel*