Minimax Algorithm in Game Theory

Backtracking

- ❖ for finding solutions to some computational problems,
- ❖ For notably constraint satisfaction problems,
- ❖ incrementally builds candidates to the solutions,
- ❖ abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.

It is used for decision making: find the best move out of a lot of possible moves.

Eight queens puzzle:

Asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

The game problem:

Given a matrix mat[][] of dimension N*M, a positive integer K and the source cell (X, Y), the task is to print all possible paths to move out of the matrix from the source cell (X, Y) by moving in all four directions in each move in at most K moves.

Input: N = 2, M = 2, X = 1, Y = 1, K = 2
Output:

```
(1 1)(1 0)
(1 1)(1 2)(1 3)
(1 1)(1 2)(0 2)
(1 1)(0 1)
(1 1)(2 1)(2 0)
(1 1)(2 1)(3 1)
```

Input: N = 1, M = 1, X = 1, Y = 1, K = 2
Output:

```
(1 1)(1 0)
(1 1)(1 2)
(1 1)(0 1)
(1 1)(2 1)
```

Programming Approach: The given problem can be solved by using Recursion and Backtracking:

Initialize an array, say, **arrayOfMoves[]** that stores all the moves moving from the source cell to the out of the matrix.
Define a recursive function, say **printAllmoves(N, M, moves, X, Y, arrayOfMoves)**, and perform the following steps:

- **Base Case:**
  - If the value of the **moves** is non-negative and the current cell **(X, Y)** is out of the matrix, then print all the moves stored in the **ArrayOfMoves[]**.
  - If the value of **moves** is less than **0** then return from the function.
- Insert the current cell **(X, Y)** in the array **arrayOfMoves[]**.
- Recursively call the function in all the four directions of the current cell **(X, Y)** by decrementing the value of **moves** by **1**
- If the size of the array **arrayOfMoves[]** is greater than 1, then remove the last cell inserted for the Backtracking steps.

Call the function **printAllmoves(N, M, moves, X, Y, arrayOfMoves)** to print all possible moves.

Minimax:

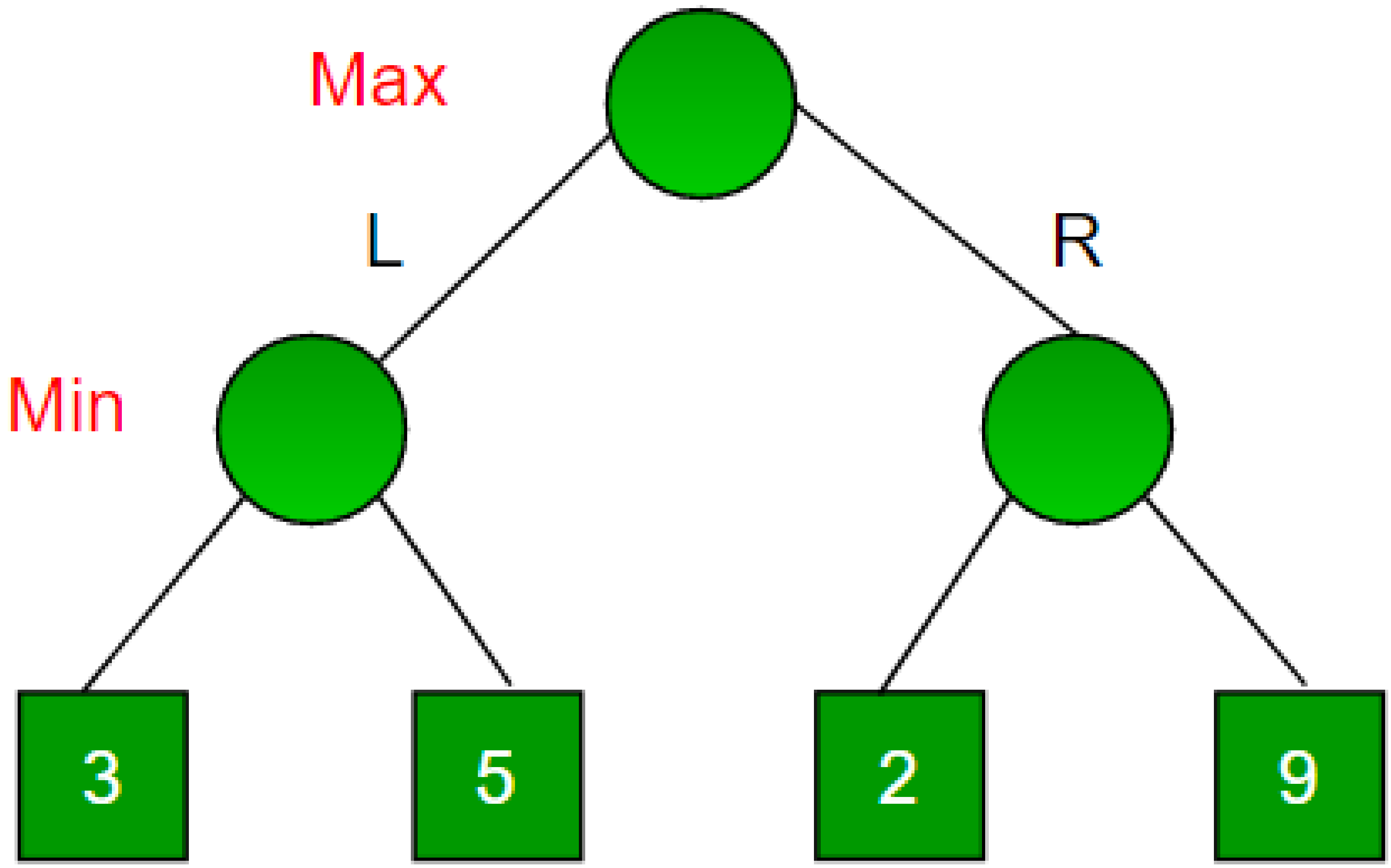
- A kind of backtracking algorithm
- used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally
- widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

Two players: maximizer, minimizer

- The maximizer tries to get the highest score possible
- the minimizer tries to do the opposite and get the lowest score possible

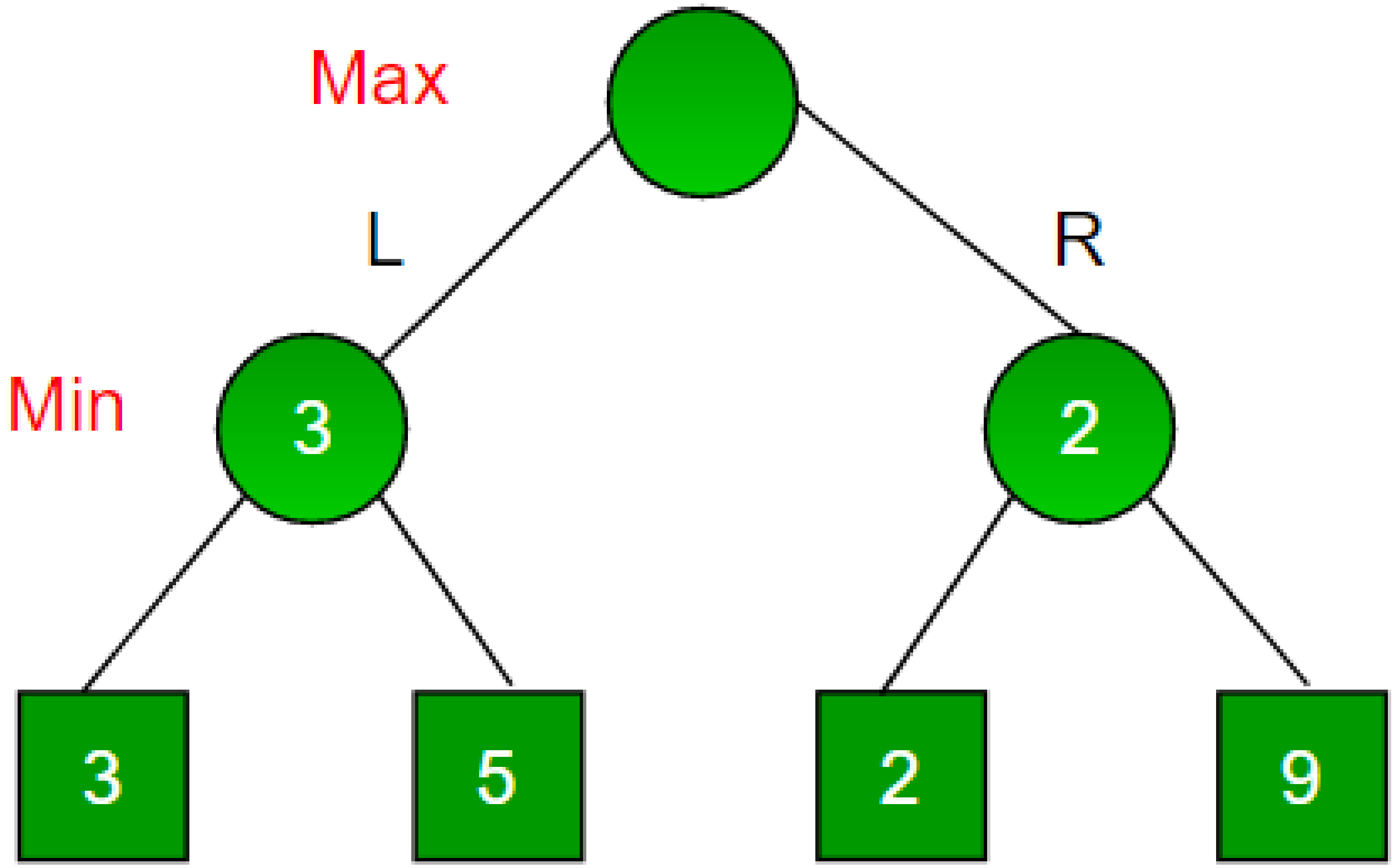
Board State:

- ❖Every board state has a value associated with it
- ❖If the maximizer has upper hand then, the score of the board will tend to be some positive value.
- ❖If the minimizer has the upper hand in that board state then it will tend to be some negative value.
- ❖The values of the board are calculated by some heuristics which are unique for every type of game.
- ❖The value of the board state is decided after both players each makes one move


Max
L
R
Min
3
5
2
9

Using Backtracking: it tries all possible moves, then backtracks and makes a decision.

- ❖Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3
- ❖Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.


Max
L
R
Min
3
2
3
5
2
9

```
# A simple Python3 program to find maximum score that maximizing player can get
import math

def minimax (curDepth, nodeIndex, maxTurn, scores, targetDepth):

    # base case : targetDepth reached
    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))

    else:
        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))

# This code is contributed by rootshadow
```

Homework for programming

Using python program to implement following algorithm:

Given a matrix mat[][] of dimension N*M, a positive integer K and the source cell (X, Y), the task is to print all possible paths to move out of the matrix from the source cell (X, Y) by moving in all four directions in each move in at most K moves.