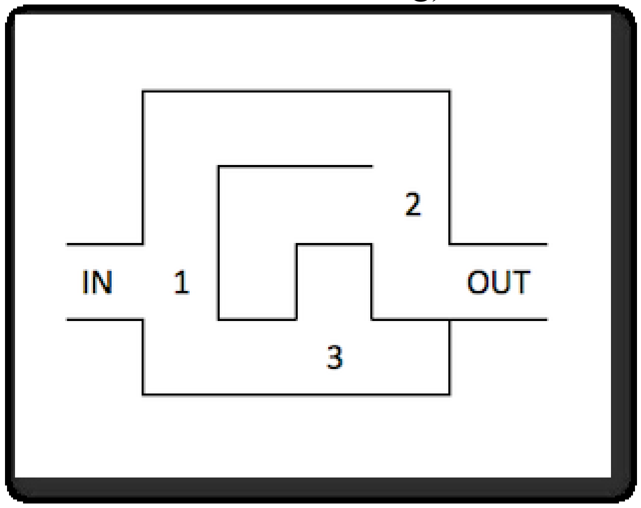
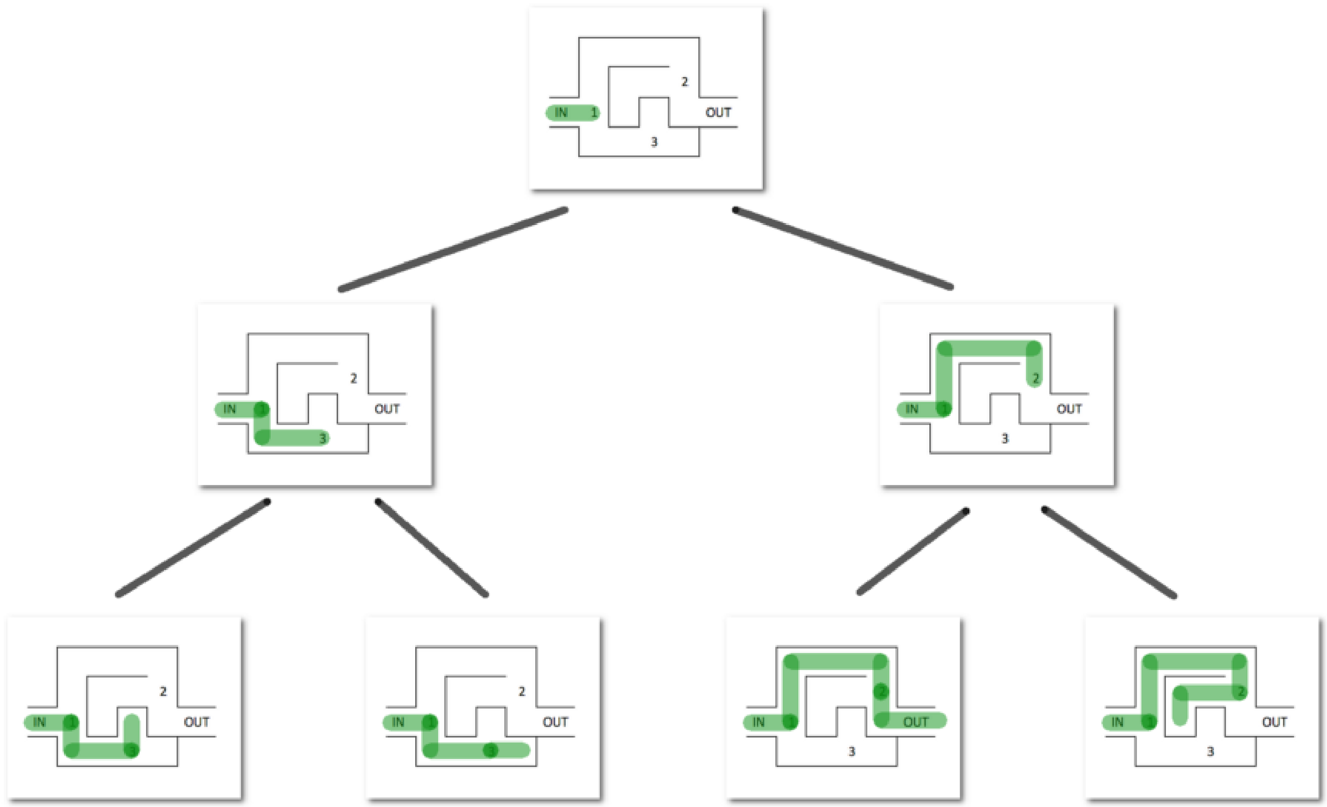
Backtracking Algorithms

Backtracking is one of my favourite algorithms because of its simplicity and elegance; it doesn't always have great performance, but the branch cutting part is really exciting and gives you the idea of progress in performance while you code.
But let's first start with a simple explanation. According to Wikipedia:
Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.
Once you already have used backtracking, it's a pretty straightforward definition, but I realise that when you read it for the first time is not that clear (or—at least—it wasn't to me). A little example could help us. Imagine to have a maze and you want to find if it has an exit (for sake of precision, algorithms to get out of a maze using graphs are more efficient than backtracking). This is the maze:



A simple maze with only three junctions where we have labeled the junctions as 1, 2 and 3. If we want to check every possible path in the maze, we can have a look at the tree of paths, split for every junctions stop:



All the possible paths of the maze

Let’s see a pseudo code for traversing this maze and checking if there’s an exit:

```
function backtrack(junction):

  if is_exit:
    return true
  for each direction of junction:
    if backtrack(next_junction):
      return true

  return false
```

If we apply this pseudo code to the maze we saw above, we’ll see these calls:

```
- at junction 1 chooses down       (possible values: [down, up])
    - at junction 3 chooses right  (possible values: [right, up])
        no junctions/exit          (return false)
    - at junction 3 chooses up     (possible values: [right, up])
        no junctions/exit          (return false)
- at junction 1 chooses up         (possible values: [down, up])
    - at junction 2 chooses down   (possible values: [down, left])
        the exit was found!        (return true)
```

Please note that every time a line is indented, it means that there was a recursive call. So, when a no junctions/exit is found, the function returns a false value and goes back to the caller, that resumes to loop on the possible paths starting from the junction. If the loop arrives to the end, that means that from that junction on there’s no exit, and so it returns false. The idea is that we can build a solution step by step using recursion; if during the process we realise that is not going to be a valid solution, then we stop computing that solution and we return back to the step before (*backtrack*). In the case of the maze, when

we are in a dead-end we are forced to backtrack, but there are other cases in which we could realise that we're heading to a non valid (or not good) solution before having reached it. And that's exactly what we're going to see now.
Ref: https://medium.com/@andreaiacono/backtracking-explained-7450d6ef9e1a

MiniMax Algorithm

Minimax is a decision-making algorithm, **typically used in a turn-based, two player games**. The goal of the algorithm is to find the optimal next move.

In the algorithm, one player is called the maximizer, and the other player is a minimizer. If we assign an evaluation score to the game board, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score. In other words, **the maximizer works to get the highest score, while the minimizer tries get the lowest score by trying to counter moves.**

It is based on the zero-sum game concept. **In a zero-sum game, the total utility score is divided among the players. An increase in one player's score results into the decrease in another player's score.** So, the total score is always zero. For one player to win, the other one has to lose. Examples of such games are chess, poker, checkers, tic-tac-toe.

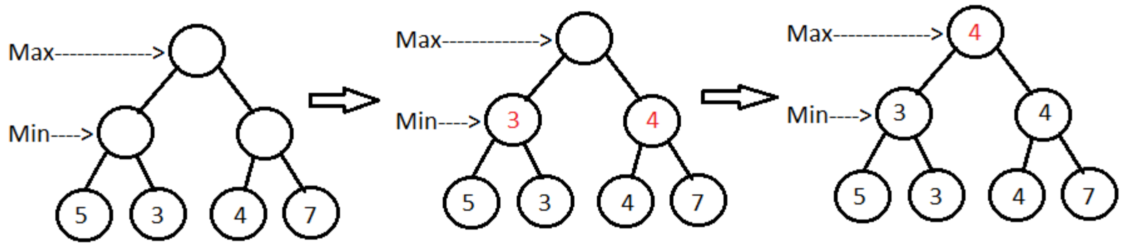
Minimax Algorithm

Our goal is to find the best move for the player. To do so, we can just choose the node with best evaluation score. To make the process smarter, we can also look ahead and evaluate potential opponent's moves.

For each move, we can look ahead as many moves as our computing power allows. The algorithm assumes that the opponent is playing optimally.

Technically, we start with the root node and choose the best possible node. We evaluate nodes based on their evaluation scores. In our case, evaluation function can assign scores to only result nodes (leaves). Therefore, we recursively reach leaves with scores and back propagate the scores.

Consider the below game tree:



Maximizer starts with the root node and chooses the move with the maximum score. Unfortunately, only leaves have evaluation scores with them, and hence the algorithm has to reach leaf nodes recursively. In the given game tree, currently it's the minimizer's turn to **choose a move from the leaf nodes**, so the nodes with minimum scores (here, node 3 and 4) will get selected. It keeps picking the best nodes similarly, till it reaches the root node.

Now, let's formally define steps of the algorithm:

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
    - For max player, select the child with the maximum score
    - For min player, select the child with the minimum score
4. At the root node, choose the node with max value and perform the corresponding move

Ref: https://www.baeldung.com/java-minimax-algorithm

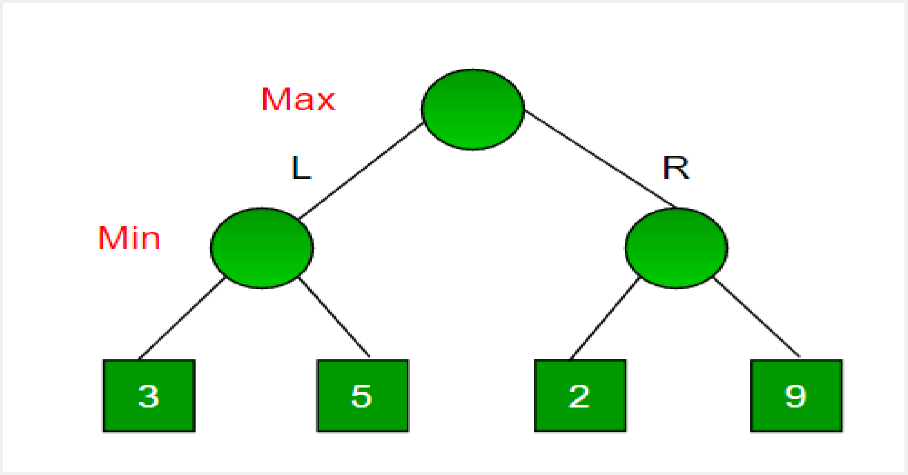
Minimax Algorithm in Game Theory | Set 1 (Introduction)

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

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state 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.

Example:
Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at next level. **Which move you would make as a maximizing player considering that your opponent also plays optimally?**



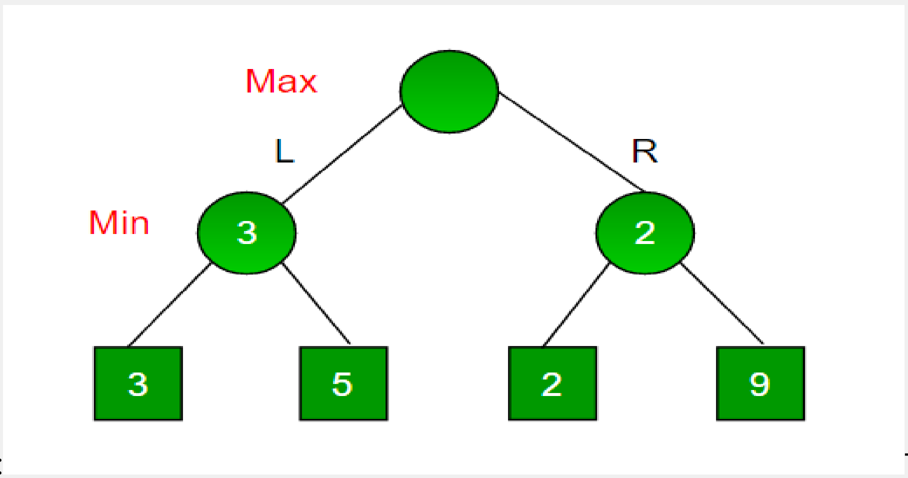
Since this is a backtracking based algorithm, it tries all possible moves, then backtracks and makes a decision.

- 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.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :



The above tree shows two possible scores when maximizer makes left and right moves.

Note: Even though there is a value of 9 on the right subtree, the minimizer will never pick that. We must always assume that our opponent plays optimally.

- In the above example, there are only two choices for a player. In general, there can be more choices. In that case, we need to recur for all possible moves and find the maximum/minimum. For example, in Tic-Tax-Toe, the first player can make 9 possible moves.
- In the above example, the scores (leaves of Game Tree) are given to us. For a typical game, we need to derive these values

Ref https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

Minimax Algorithm with Alpha-beta pruning

See https://www.hackerearth.com/blog/artificial-intelligence/minimax-algorithm-alpha-beta-pruning/