

Gaming Algorithms

Overview

Games hold an inexplicable fascination for many people and the notion that computers might play games has been there for a long long time. Two reasons that made the games appear to be a good domain to explore machine intelligence:

1. They provide a structured task in which it is very easy to measure the success or failure.
2. They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to the winning position.

The second point is true but not a simple thing. The second point helps only when we consider the simplest of the games. Otherwise, it just doesn't help to know the games can be played by just searching the sample space. For example in chess:

1. The average branching factor is around 35.
2. In an average game, each player might end up making around 50 moves.

Hence, in order to examine the complete game tree, we would have to examine a **whopping 35100 positions**.

The above makes it clear that a program that simply does a straightforward search of the game tree will not be able to select even it's first move during the lifetime of it's opponent.

Algorithms

For gaming algorithms, the search through the sample space is divided into two components:

1. **The generator:** The generator component generates the new states which are possible.
2. **The tester:** The component which tests each state and tells how good the state is.

It is important in gaming algorithms given the size of our sample that both of the above components are well optimized. The generator should be good so that only the good moves or paths are generated. The procedure should be improved so that the best moves are recognized quickly and explored first.

Hence, the game playing algorithm should be built from two components: a good plausible move generator and a good static evaluation function. Now, building these two components will involve incorporating a great deal of knowledge about the particular game being played. But since these two components will never be perfect, we need a good search technique to search as many nodes as possible and try to compute the next best possible move.

Backtracking

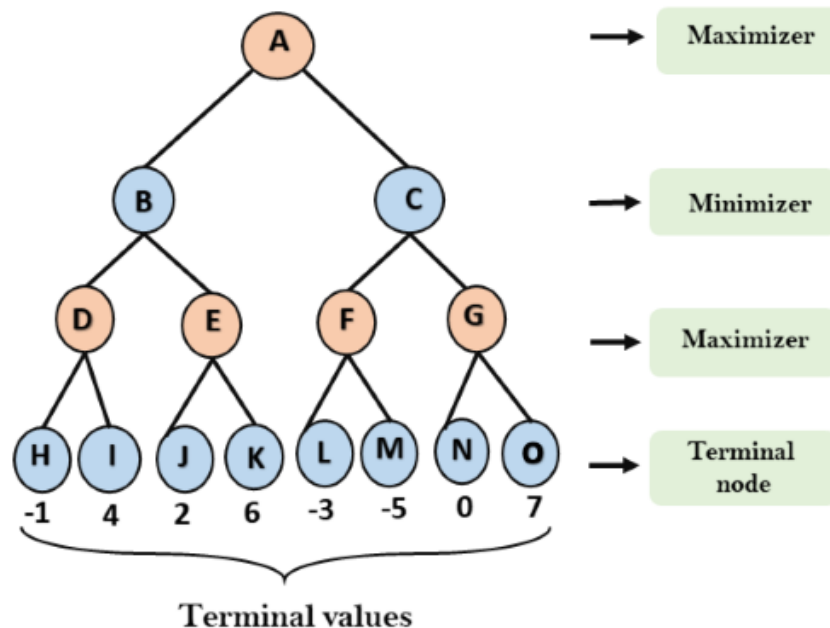
Backtracking is one of the techniques that is used for solving simple game problems. It is a technique where we keep searching the sample space using *Depth First Search*. We keep going deep and deep until we encounter a failure. Once a failure is encountered, that path is discarded and one step is taken back and we search for other possibilities. If all the possibilities give failure, we again take a step back. This process is repeated until we find success.

A very good example of this is to try to find a path in a maze. Imagine you have entered a maze and are trying to search for something within. You would in the normal course always take the left turn or the right turn consistently at every fork and continue. If you reach a dead end, you would return to the fork and try the right turn (or the left). Eventually, you would search the whole maze. **This process is called backtracking. Whenever there is a fork or alternate paths to be discovered a backtracking point is put up and the same is visited in the event of a failure.**

A good example of backtracking technique is a sudoku solver. We try filling digits one by one. Whenever we find that the current digit cannot lead to a solution, we remove it (backtrack) and try the next digit.

Minimax Algorithm

The minimax search is a depth-first and depth limited procedure. It's a smarter way to determine the best move in case of a 2 player game where one player tries to maximize and other player minimize.



The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions. Now we can apply the static evaluation function to those positions and simply choose the best one. After doing so, we can back that value up to the starting position to represent our evolution of it. Here we assume that the static evaluation function returns larger values to indicate good situations for us. So our goal is to maximize the value of the static evaluation function of the next board position. The opponents' goal is to minimize the value of the static evaluation function. **The alternation of maximizing and minimizing at alternate ply when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.**

```
function minimax(node, depth, maximizingPlayer) is
  if depth == 0 or node is a terminal node then
    return static evaluation of node


  if MaximizingPlayer then      // for Maximizer Player
    maxEva= -infinity
    for each child of node do
      eva= minimax(child, depth-1, false)
    maxEva= max(maxEva, eva)     //gives Maximum of the values
    return maxEva

  else                          // for Minimizer player
    minEva= +infinity
    for each child of node do
      eva= minimax(child, depth-1, true)
    minEva= min(minEva, eva)    //gives minimum of the values
    return minEva
```

Here is the link to my implementation of a *Sudoku Solver* using the backtracking algorithm

Algorithms-for-reals/Sudoku_solver at main · harshitagupta1512/Algorithms-for-reals

Contribute to harshitagupta1512/Algorithms-for-reals development by creating an account on GitHub.

 https://github.com/harshitagupta1512/Algorithms-for-reals/tree/main/Sudoku_solver

harshitagupta1512/
Algorithms-for-reals

1

Contributor

0

Issues

0

Stars

0

Forks

Sudoku Solver

Open index.html and you will see a 9X9 Sudoku grid, fill in some blocks and click on **solve**, if the sudoku is solvable, the grid is filled with a valid Sudoku solution else, you will get a prompt saying *Wrong Sudoku problem inputted.*

1								9
2								
3								
						8		
								8

1	4	5	2	3	6	7	8	9
2	6	7	1	8	9	3	4	5
3	8	9	4	5	7	1	2	6
4	1	2	3	6	8	5	9	7
5	3	8	7	9	1	2	6	4
7	9	6	5	2	4	8	1	3
6	2	1	9	7	3	4	5	8
8	5	3	6	4	2	9	7	1
9	7	4	8	1	5	6	3	2

Gaming Algorithms

4