Tree Search Algorithms - Practising C++

Finlay Barford

07/09/2024

Contents

1 Abstract

Having had about 2 years away from C++, I wanted to do a simple project to regain the basics. My last major coding project was creating a chess frontend, but I had always really wanted to eventually develop a fully fledged chess engine. As a getting back into things project, the full chess engine itself would be a mighty task.

Instead I've chosen to focus on something a bit more realistic, which would still be useful to that goal. With that said the project I've chosen is to create some tree search algorithms, more specifically a focus on the minimax algorithm for a 2 player zero sum game.

I'll then move onto trying out some Monte Carlo tree search, along with alpha-beta pruning - because these are the two most commonly used optimisations of the minimax algorithm with regards to chess. Going forward I'll make some assumptions on knowledge, but I'll be explaining the algorithms I use and how I designed them.

2 Prelim work

The main denominator of these search algorithms is that they are all designed for trees. So to begin I'll need to actually create the trees. The two main considerations for this will be the size of the tree - too big and testing could take a while on an inefficient algorithm design, too short and I may not notice how inefficient it is - and what value should it store - since I have no chess positions to evaluate, or really any data to take into account, I'll mimic a real use case by just randomly generating values between 10 and -10 to store.

To manage all this, I'll make a class that I will use to generate trees. Then I can create separate tree objects of differing sizes when it comes to testing.

Algorithm 1 RandTree

```
procedure RandTree(inpDepth,inpWidth)
    $depth \leftarrow$ inpDepth
    $width \leftarrow$ inpWidth
    tempDepth ← 0
    tempCount ← 0
    initTree(*tempDepth, *tempCount)
    generateData()
    if $i > stringlen$ then return false
    else test
```

RandTree	
Public: float[] nodeData Private: int depth int width vector<vector<int>> nodeRelation	nodeData - will hold the node random values depth - maximum distance from root node width - maximum number of children for each node nodeRelation - record of edges
Public: RandTree(inpDepth,inpWidth) Private: void initTree(int currentDepth) void generateData()	RandTree(inpDepth,inpWidth) - Constructor, will generate a full tree with depth inpDepth and width inpWi initTree(int currentDepth) - creates an empty tree with all input for recurrence, look at algorithm 2 for details generateData() - fills tree with randomised values between -10 and 10

Table 1: RandTree Class

Algorithm 2 initTree

```
procedure INITTREE(*currentDepth, *nodeCount)
    currentNodeWidth ← random integer between 1 and width
    Create temporary vector<int>
    for currentNodeWidth do
        if depthCount value+1 < OR = depth then
            *depthCount ← *depthCount + 1
            *nodeCount ← *nodeCount + 1
            vector <int> add nodeCount value
            initTree(*depthCount, *nodeCount)
        else*depthCount -= 1
    add created vector<int> to nodeRelation
    nodeData ← nodeData[nodeCount]
    for nodeCount do nodeData[i] ← 0
```

2.1 Setup

3 Proto 1 - Minimax

3.1 Theory

3.2 Implementation

3.3 Testing

3.4 Evaluation

4 Proto x - Minimax

4.1 Theory

4.2 Implementation

4.3 Testing

4.4 Evaluation