Game Tree

A **game tree** is a type of recursive search function that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move. They are very useful for Artificial Intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play. The most commonly-cited example is chess, but they are applicable to many situations.

Tip: This is an advanced topic and is difficult to implement. It is best to read the entire article and gain a lot of experience before trying to write the script for a game tree, which is a hard and time-consuming task.

Note: Please note, there is no working Scratch code here, because game-tree implementation is so unique to the problem/game it is trying to solve/win.

Contents [hide]

Limitations

There are a few major limitations to game trees.

Time
As mentioned above, game trees are rarely used in real-time scenarios (when the computer isn't given very much time to think). The reason for this will soon become apparent, but to put it brief: the method requires a lot of processing by the computer, and that takes time.

For the above reason (and others) they work best in turn-based games.

They require complete knowledge of how to move.
Games with uncertainty generally do not mix well with game trees. You can try to implement them in such games, but it will be very difficult, and the results may prove less than satisfactory.

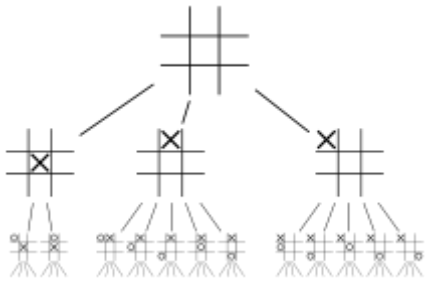
They are ineffective at accurately ascertaining the best choices in scenarios with many possible choices

General Concept

Game trees are generally used in board games to determine the best possible move. For the purpose of this article, Tic-Tac-Toe will be used as an example.

The idea is to start at the current board position, and check all the possible moves the computer can make. Then, from each of those possible moves, to look at what moves the opponent may make. Then to look back at the computer's. Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion. It will do this for every possible outcome (see image below), effectively playing thousands (often more) of games. From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.

If it does not make sense, just think of how you think during a board game. You think "if I make this move, he/she could make this one, and I could counter with this, etc." Game trees do just that. They examine every possible move and every opponent move, and then try to see if they are winning after as many moves as they can think through.



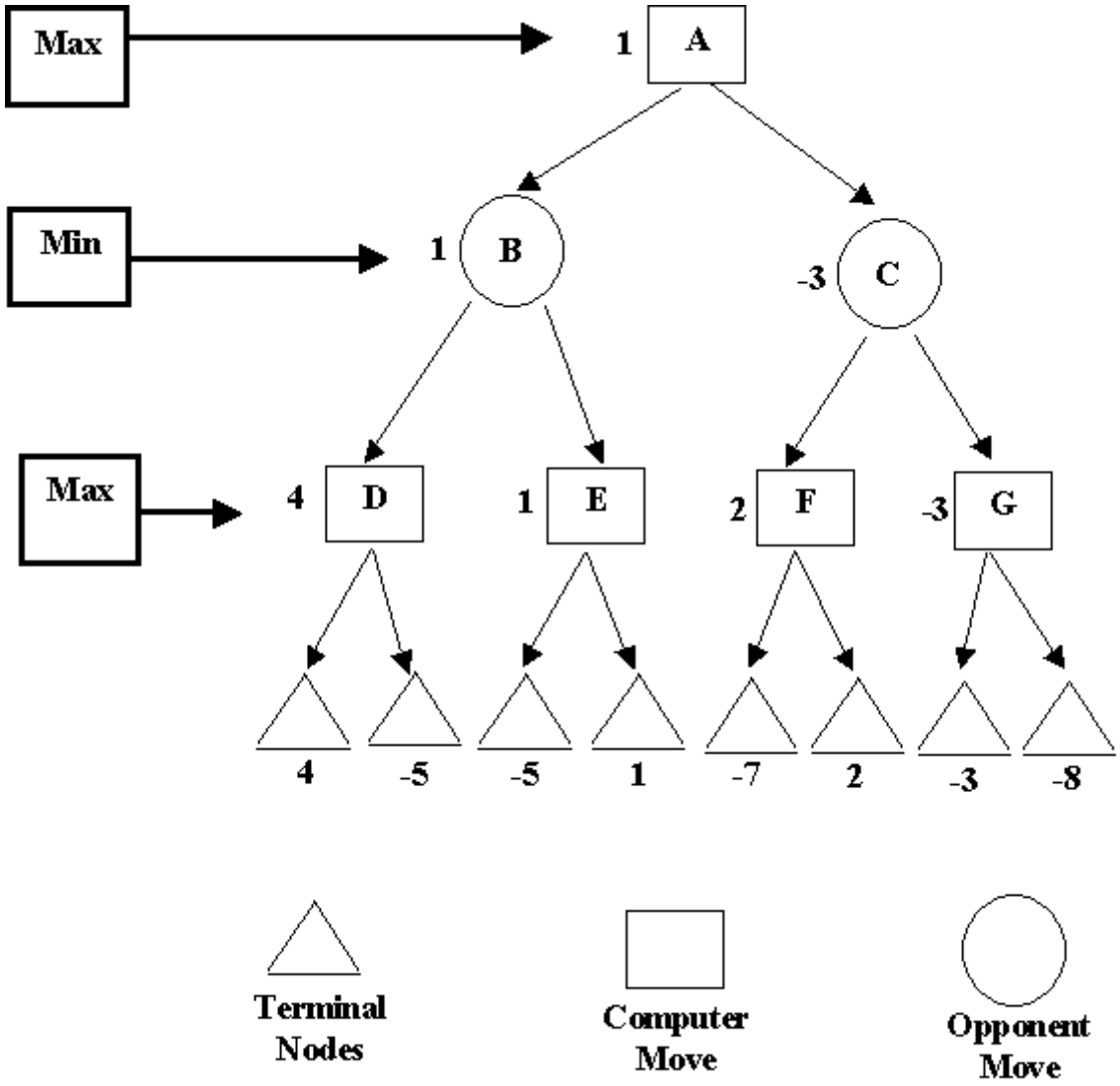
As one could imagine, there are an extremely large number of possible games. In Tic-Tac-Toe, there are 9 possible first moves (ignoring rotation optimization). There are 8 on the second turn, then 7, 6, etc. In total, there are 255,168 possible games. The goal of a game tree is to look at them all (on the first move) and try to choose a move that will, at worst, make losing impossible, and, at best, win the game.

Summary of What the Computer Does

- The computer begins evaluating a move by calculating all possible moves. In Tic-Tac-Toe, it is filling any empty square on the grid, thus the number of possible moves will be equal to the number of empty squares.
- Once it has found these moves, it loops over each of the possible moves, and tries to find out whether the move will result in a win for the computer or not. It does this by running this same algorithm (recursion) over the position (obtained by performing one of the computer's original possible moves) but trying to calculate the *opponent's* best move.
- To find if a move is "good" or "bad" the game tree is extended and continues to branch out until the game ends (a "terminal node"). Terminal nodes are then assigned values based on the result of the game; the higher the value, the better the result is for the computer. Because there are only two possible results (a win or a loss), the values of "-1" and "1" can be used to represent who has won (the example below offers a greater variety of values than "-1" and "1"). Now that the terminal nodes' values have been determined, the node above them (nodes D, E, F, and G in the picture below). If the node represents the computer's choice of moves it is a "max node", if it represents the player's choice of moves it is a "min" node. The value of a max node becomes that of the highest node beneath it. The value of a min node becomes that of the lowest node beneath it. The value D in the example below is 4 and the value of E is 1. The value of B becomes 1 (the lowest of 4 and 1). By continuing to move up, each node is given a value. The top node (A) then assumes the highest value of the nodes beneath it; the node with the highest value is the move that the computer should make.

Note: If the position is that of a finished game, and the winner is the opponent, the position is regarded as a winning position for the opponent (i.e. a "bad" move for the computer), and vice-versa.

.



Components

All game trees require the same script concepts. The names given to these scripts in this article are not official, but rather represent the functions of the scripts.

- A **movement script** — This returns an Array of the positions possible to be obtained by a move made by a given player.
- A **make move script** — This selects the best of the moves returned by the above script, with the logic described in the above section.
- A **winner script** — This takes in a position and outputs if it is a completed game, and if so, who won.

Pseudocode

```
function get-best-move-for:(aPlayer) in-game:(aPosition)

if (has-won:(aPlayer) in-game:(aPosition)) {// if the player has won, return the positio
        report (aPosition.labelAsGood)
}
if (has-lost:(aPlayer) in-game:(aPosition)) {// if the player has lost, return the posit
        report (aPosition.labelAsBad)
}
if (has-tied-in-game:(aPosition)) {// if the game is tied, return the position itself, l
        report (aPosition.labelAsOk)
}
moves = get-all-moves-for:(aPlayer) in-game:(aPosition) // get all possible moves to cho
```

```
checking = 0
while: (checking < length-of:(moves)) do: { // iterate through each possible move
        currentMove = item:(checking) of:(moves) //get a move to test
        opponentMove = get-best-move-for:(opponent-to:aPlayer) in-game:(currentMove) //
        if (opponentMove.labeledAsGood) {// If the opponent will win, it's a bad move fo
                label:(item:(checking) of:(moves)) as:"bad"
        }
        if (opponentMove.labeledAsTie) {//If the opponent will win, it's ok for us
                label:(item:(checking) of:(moves)) as:"ok"
        }
        if (opponentMove.labeledAsBad) {// If the opponent will lose, it's a good move f
                label:(item:(checking) of:(moves)) as:"good"
        }
        change:(checking) by:(1) // test next possible move
}
report (best-labeled-move-among:(moves)) // move the best possible move (where "good" be
```

Optimization

The above method shows only the basic application of a game tree. In reality, the above method is not applicable to many games without some modifications. The above script should lay the ground-work for any game tree, but is not, by any means, optimal. Here are some ways to improve a game-tree's speed or performance.

Not Searching to Completion

Though searching every move is possible, it is often faster to just think forwards through the game about 3-6 moves. 6 moves are often enough to determine how good or bad a move may be, and does not sacrifice speed too much. Humans play a game similarly. We do not calculate through the entire game for each move; we always think through the consequences of a given move 3-6 moves in the future, and decide based on them. To put this into perspective, if the average chess position has 20 moves (an underestimate), and you want to search a game to completion, assuming an average of 40 moves per game (another underestimate), you'll have to search 20^40 positions (about 10^52). Even modern computers with incredibly powerful programming languages are incapable of searching anywhere near this. But computers are easily capable of searching 4 or 5 moves deep, and looking at the position to see who is winning, and this is what they do. Instead of searching 40+ moves deep, they search 2-8 moves deep and evaluate the end positions.

Of course, the computer's decision will not be perfect if it does not search to the completion of the game. Thus, depth is often the control behind difficulty levels. A more difficult computer player will search to a higher depth, but will have the disadvantage of being slower.

More Complex Evaluation Scripts

In chess, a more suitable way to see who is "winning" is to count out the points each piece is worth. For instance, the "value" of a board could be calculate by adding up all the "good pieces' points" and subtracting all the "bad pieces' points" (each chess piece is generally assigned a value, with a pawn being valued a 1-point, and a queen at 9 points). The more accurate your evaluation script, the better your moves will be, but the longer (in general) they'll take to run. It is important to find a balance between accuracy and speed. More speed lets you search deeper, while greater accuracy makes the depth you search more effective.

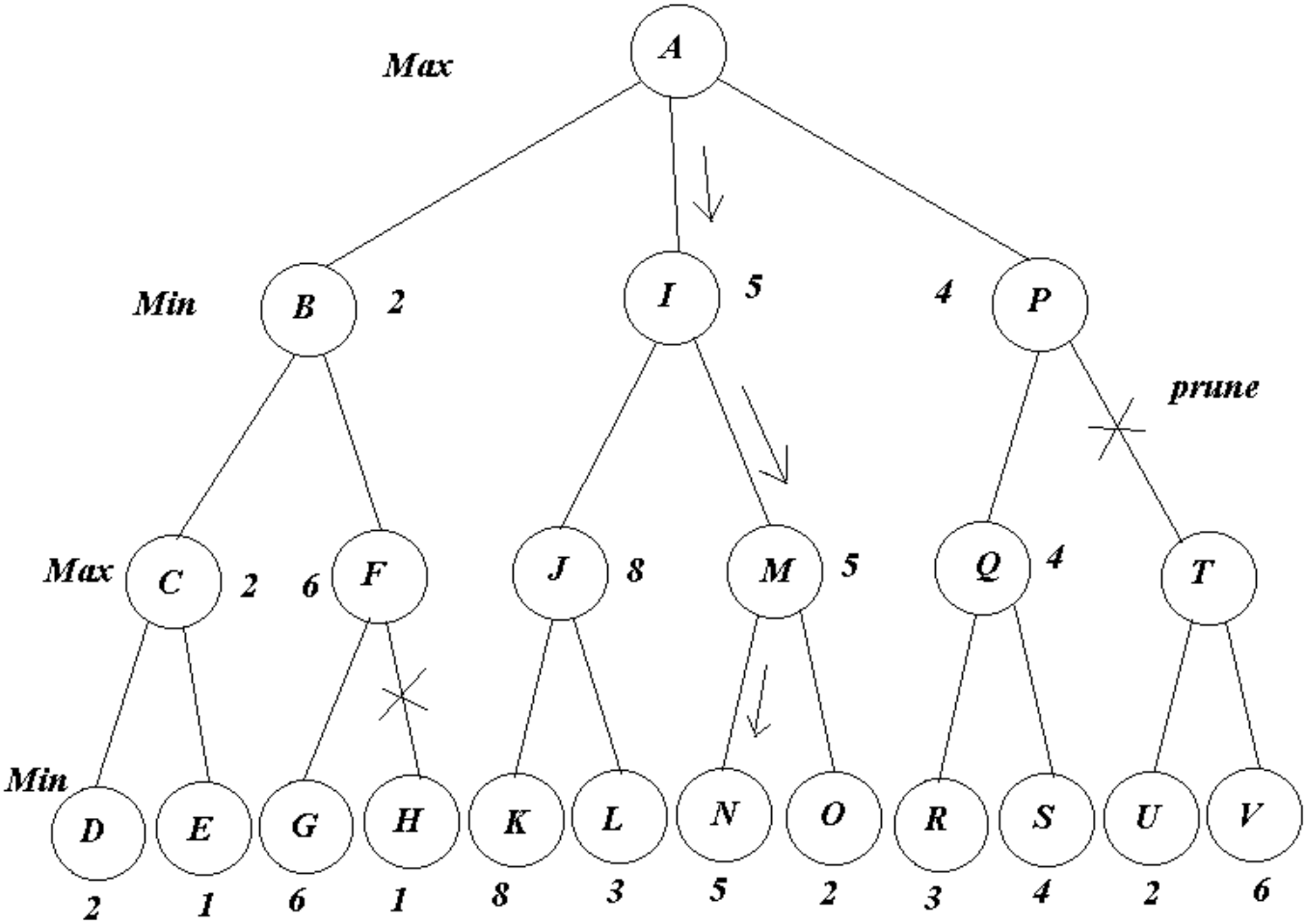
Other Ways to Improve Your Script

- Pruning (trying to skip positions that do not seem to be important to make your searching faster)
  - Alpha-Beta Pruning is one of the most effective ways to eliminate searches and does not sacrifice accuracy (that is, the same decision will be reached by a computer searching to the same depth using the same evaluation function, regardless of whether or not Alpha-Beta pruning is implemented). In the picture below, imagine the computer is searching from left to right. It searches all the way down to D, and gives it a value of 2. Then it gives E a value of 1. The result is giving C (a max node) the value of 2. The computer than looks at G and, seeing that it is higher than the 2 (it is a 6), it stops looking at H (or any other moves that F might result in). The reasoning behind this is this:

    F has to be at least 6 (it is a max node). Therefore, F has to be greater than or equal to C (which is 2). Since F has to be greater or equal to C, the opponent (the min node) will never pick F rather than C; instead the

min node will have to pick C, the lower of the two values.

- While the elimination of one move seems insignificant, in larger branching factors, more moves can be eliminated. Continuing on to branch P, one can see the reverse happening. Q's value has been determined to be 4, which is less than I (which is 5). Since the max node will only choose the best move (and P can not be greater than 4), continuing to search branch P (to terminal nodes U and V) is unnecessary.
- If one can predict which moves are likely to be the best first, Alpha-Beta pruning becomes much more effective. For instance, if placing an "X" in the center of the Tic-Tac-Toe grid is, as a rule of thumb, more likely to be a better move than on the sides, checking the "put X in the center" move first can result in having to search less moves.



- Searching to a "stable position" can help reduce the horizon effect. Basically, if a node seems unstable (the computer just took a piece for instance), looking deeper in just that branch can often reveal a decisive move that can help the computer make a better move. While searching a layer deeper for every script is no different than simply extending your entire search deeper, trying to find positions that could lead to a big loss or gain and only searching those can be far more efficient than searching all of the possible moves, particularly at the last layer.
- Many computers search all of the moves to a given depth, and then search only the best move to an additional depth. If the best move proves to be poor (with the added input of the extra depth), the computer searches the 2nd move to an additional depth, continuing for a set number of moves, or until a move does not prove to be "defective" upon further searching.
- Many game-tree-implementing programs use additional optimizations, such as filtering rotations/reflections in games with many, commonly occurring symmetric positions (e.g. Tic-Tac-Toe), storing computations of positions in memory to be accessed later, etc.

Problems With these Optimizations

- The horizon effect is when the computer makes a poor decision because of a move that was too deep for it to search (if it is searching 5 ply and the bad consequence of a move is 6 ply deep, the computer will not see it). This can be problematic when the computer takes a piece in chess (and does not see a recapture) or when the

computer is threatened and the computer keeps trying to delay the inevitable, often resulting in poor decisions by the computer

- Computers have serious problems playing any game with a large number of moves per turn (called the "branching factor" of a game tree). The above picture has a branching factor of two. Chess has a branching factor of about 30 (an average of 30 moves per turn). Other games (Go, for instance) have a very large number of possible moves, and, as a result, computers are unable to search every move to enough depth to make a reasonable move. Novices regularly beat computers at Go, because computers are unable to cope with the large number of moves, as the computer has to search a number of positions equal to the following expression: (branching_factor)^depth_searched. In chess, it is about 30^depth, in Go, it is even larger.