Introduction to AI Techniques

Game Search, Minimax, and Alpha Beta Pruning

June 8, 2009

Introduction

One of the biggest areas of research in modern Artificial Intelligence is in making computer players for popular games. It turns out that games that most humans can become reasonably good at after some practice, such as GO, Chess, or Checkers, are actually difficult for computers to solve.

In exploring how we could make machines play the games we play, we are forced to ask ourselves how *we* play those games. Although it seems that humans use some notion of "intelligence" in playing a game like chess, our approaches in solving such games have not progressed much farther than the sort of brute force approaches that we experimented with in the 50s. Unfortunately, present computer players usually rely on some sort of search over possible game outcomes to find the optimal move, rather than using what we would deem intelligent behavior.

In this discussion we will see some of the ideas behind these computer players, as well as future directions the field might take, and how these computer approaches can both help us learn to play the games better as well as point out some fundamental differences between human play and machine play.

As a quick time line to show how (not very) far we have come since Claude Shannon's (a famous MIT professor, the father of Information Theory, etc.) *Programming a Computer Playing Chess, 1948*:

- 1948 Claude Shannon

- 1951 Alan Turing works out a plan on paper for a chess-playing computer program.
- 1966-1967 Mac Hack 6, developed at MIT, first chess program to beat a person in tournament play
- 1997 Deep Blue beats Kasparov, the reigning world chess champion at the time, in a best out of 6 match. This was seen as a landmark in the chess program world, but really Deep Blue was just like previous chess playing machines with bigger and better computing power, and no more "intelligence" than any previous model.

Well-known Players

The most popular recent game to be solved is checkers, which had up to 200 processors running night and day from 1989 to 2007. Checkers has $5 * 10^{20}$ possible positions on its 8 by 8 board. It is now known that perfect play by each side results in a draw. You can play around with the database on the Chinook project's website: www.cs.ualberta.ca/ chinook/. The game is strongly solved, and for every move Chinook tells you whether it leads to a winning strategy, a losing strategy, or a draw.

Another famous computer player is Deep Blue, who beat chess world champion Garry Kasparov in 1997, which was capable of evaluating 200 million positions per second.

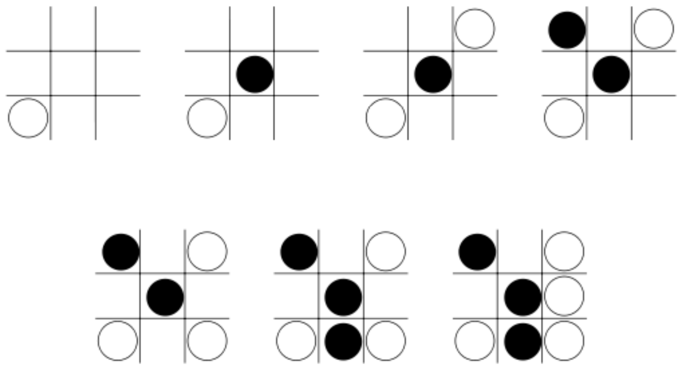
How To Solve a Game?

What if we just give the computer simple rules to follow in what is known as a **knowledge based approach**. This is how a lot of beginner and sometimes advanced human players might play certain games, and in some games it actually works (we'll take a closer look using Connect Four next time). Take the following rules for tic-tac-toe, for instance. You give it the following instructions to blindly follow in order of importance:

1. If there is a winning move, take it.
2. If your opponent has a winning move, take the move so he can't take it.

3. Take the center square over edges and corners.
4. Take corner squares over edges.
5. Take edges if they are the only thing available.

Let's see what happens when the computer plays this game (picture taken from Victor Allis's Connect Four Thesis):



This approach clearly will not always work. There are so many exceptions to rules that for a game like chess enumerating all the possible rules to follow would be completely infeasible. The next logical option to try is search. If a player could predict how the other player would respond to the next move, and how he himself would respond to that, and how the next player would respond next, etc., then clearly our player would have a huge advantage and would be able to play the best move possible. So why don't we just build our computer players to search all the possible next moves down the game tree (which we will see in more detail soon) and chooses the best move from these results? I can think of at least two of many good reasons:

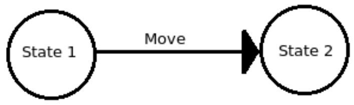
- Complexity - As we will see below, if a game offers players b different possible moves each turn, and the game takes d moves total, then the possible number of games is around b^d. That's an exponential search space, not looking good! For tic-tac-toe, there are about 255,168 possible games. Definitely reasonable. But for chess, this number is around 36^{40}, something like more than the number of particles in the universe. No good.
- It's not intelligence! Brute computational force is not exactly intell-gience. Not very exciting science here, at least not for us theoretical

> people. Maybe exciting for the hardware guys that build faster processors and smaller memory to that we have the computational power to solve these games, but other than that not very cool... It would be much more exciting to come up with a "thinking" player.

So what should we do? We can't use just simple rules, but only using search doesn't really work out either. What if we combine both? This is what is done most of the time. Part of the game tree is searched, and then an evaluation, a kind of heuristic (to be discussed more soon) is used. This approach works relatively well, and there is a good deal of intelligence needed in designing the evaluation functions of games.

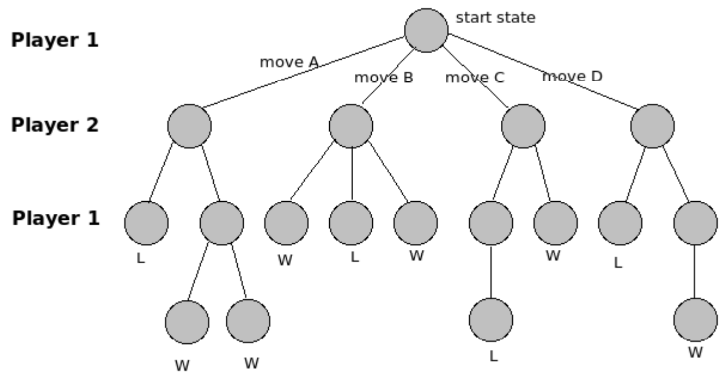
Games as Trees

For most cases the most convenient way to represent game play is on a graph. We will use graphs with nodes representing game "states" (game position, score, etc.) and edges representing a move by a player that moves the game from one state to another:



Using these conventions, we can turn the problem of solving a game into a version of graph search, although this problem differs from other types of graph search. For instance, in many cases we want to find a single state in a graph, and the path from our start state to that state, whereas in game search we are not looking for a single path, but a winning move. The path we take might change, since we cannot control what our opponent does.

Below is a small example of a game graph. The game starts in some initial state at the root of the game tree. To get to the next level, player one chooses a move, A, B, C, or D. To get to the next level, player two makes a move, etc. Each level of the tree is called a ***ply***.



So if we are player one, our goal is to find what move to take to try to ensure we reach one of the "W" states. Note that we cannot just learn a strategy and specify it beforehand, because our opponent can do whatever it wants and mess up our plan.

When we talk about game graphs some terms you might want to be familiar with are:

- **Branching factor (b)**

  The number of outgoing edges from a single node. In a game graph, this corresponds to the number of possible moves a player can make. So for instance, if we were graphing tic-tac-toe, the branching factor would be 9 (or less, since after a person moves the possible moves are limited, but you get the idea)

- **Ply**

  A level of the game tree. When a player makes a move the game tree moves to the next ply.

- **Depth (d)**

  How many plys we need to go down the game tree, or how many moves the game takes to complete. In tic-tac-toe this is probably somewhere around 6 or 7 (just made that up...). In chess this is around 40.

Minimax

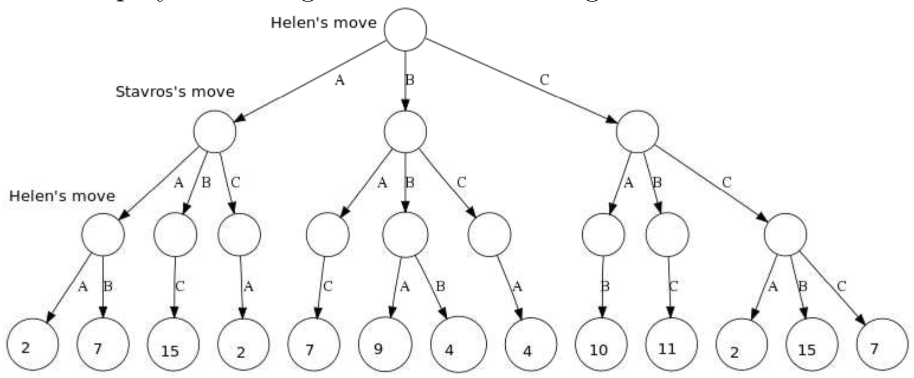
The most used game tree search is the ***minimax*** algorithm. To get a sense for how this works, consider the following:

Helen and Stavros are playing a game. The rules of this game are very mysterious, but we know that each state involves Helen having a certain number of drachmas at each state. Poor Stavros never gets any drachmas, but he doesn't want Helen to get any richer and keep bossing him around. So Helen wants to maximize her drachmas, while Stavros wants to minimize them. What should each player do? At each level Helen will choose the move leading to the greatest value, and Stavros will move to the minimum-valued state, hence the name "minimax."

Formally, the minimax algorithm is described by the following pseudocode:

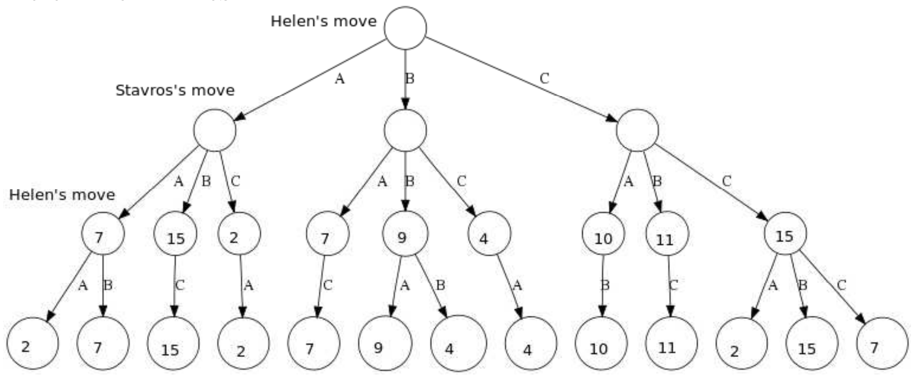
```
def max-value(state,depth):
   if (depth == 0): return value(state)
   v = -infinite
   for each s in SUCCESSORS(state):
     v = MAX(v,min-value(s,depth-1))
   return v

def min-value(state,depth):
   if (depth == 0): return value(state)
   v = infinite
   for each s in SUCCESSORS(state):
     v = MIN(v,max-value(s,depth-1))
   return v
```

We will play out this game on the following tree:

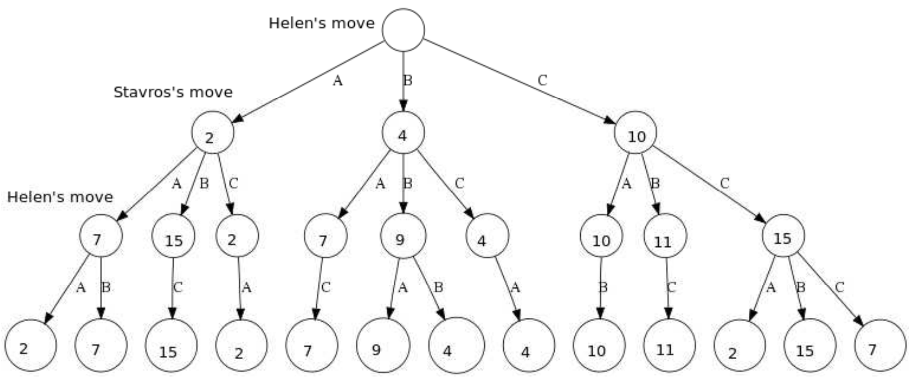


The values at the leaves are the actual values of games corresponding to the paths leading to those nodes. We will say Helen is the first player to move. So she wants to take the option (A,B,C,D) that will maximize her score. But she knows in the next ply down Stavros will try to minimize the score, etc. So we must fill in the values of the tree recursively, starting from the bottom up.

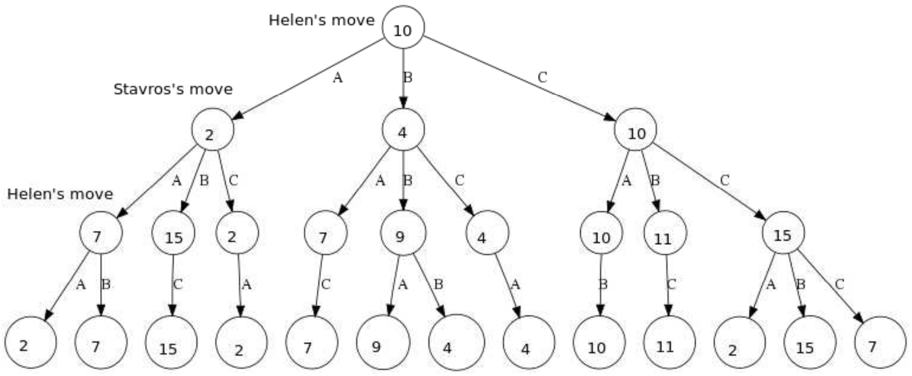
Helen maximizes:



Stavros minimizes:



Helen maximizes:



So Helen should choose option C as her first move.

This game tree assumes that each player is *rational*, or in other words they are assumed to always make the optimal moves. If Helen makes her decision based on what she thinks Stavros will do, is her strategy ruined if Stavros does something else (not the optimal move for him)? The answer is no! Helen is doing the best she can given Stavros is doing the best he can. If Stavros doesn't do the best he can, then Helen will be even better off!

Consider the following situation: Helen is smart and picks C, expecting that after she picks C that Stavros will choose A to minimize Helen's score. But then Helen will choose B and have a score of 15 compared to the best she could do, 10, if Stavros played the best he could.

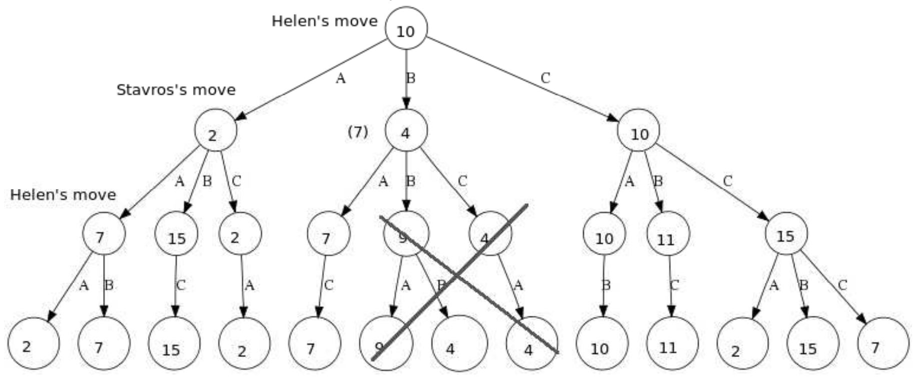
So when we go to solve a game like chess, a tree like this (except with many more nodes...) would have leaves as endgames with certain scores assigned to them by an evaluation function (discussed below), and the player to move would find the optimal strategy by applying minimax to the tree.

Alpha-Beta Pruning

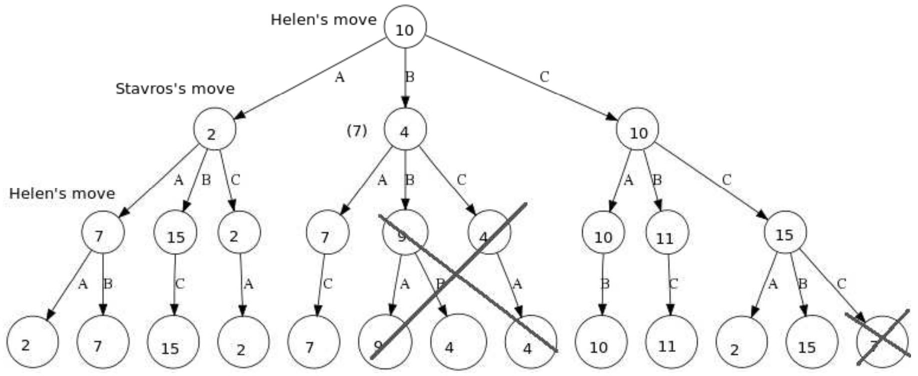
While the minimax algorithm works very well, it ends up doing some extra work. This is not so bad for Helen and Stavros, but when we are dealing with trees of size 36^{40} we want to do as little work as possible (my favorite motto of computer scientists... we try to be as lazy as possible!).

In the example above, Helen really only cares about the value of the node at the top, and which outgoing edge she should use. She doesn't *really* care about anything else in the tree. Is there a way for her to avoid having to look at the entire thing?

To evaluate the top node, Helen needs values for the three nodes below. So first she gets the value of the one on the left. (we will move from left to right as convention). Since this is the first node she's evaluating, there aren't really any short cuts. She has to look at all the nodes on the left branch. So she finds a value of 7 and moves on to the middle branch. After looking at the first subbranch of her B option, Helen finds a value of 7. But what happens the next level up? Stavros will try to minimize the value that Helen maximized. The left node is already 7, so we know Stavros will not pick anything *greater* than 7. But we also know Helen will not pick anything in the middle branch *less* than 7. So there is no point in evaluating the rest of the middle branch. We will just leave it at 7:



Helen then moves on to the rightmost branch. She has to look at the 10 and the 11. She also has to look at the 2 and 15. But once she finds the 15, she knows that she will make the next node up at least 15, and Stavros is going to choose the minimum, so he will definitely choose the 10. So there is no need to evaluate the 7.



So we saved evaluating 6 out of 26 nodes. Not bad, and often alpha-beta does a lot better than that.

Formally, the alpha-beta pruning optimization to the minimax algorithm is as follows:

```
a = best score for max-player (helen)
b = best score for min-player (stavros)
initially, we call max-value(initial, -infinite, infinite, max-depth)

def max-value(state, a, b, depth):
    if (depth == 0): return value(state)
    for s in SUCCESSORS(state):
       a = max(a, min-value(s,a,b,depth-1))
       if a >= b: return a \\ this ia a cutoff point
    return a

def min-value(state, a, b, depth):
    if (depth == 0): return value(state)
    for s in SUCCESSORS(state):
       b = min(b,max-value(s,a,b,depth-1))
       if b <= a: return b \\ this is a cutoff point
    return b
```

There are a couple things we should point out about alpha-beta compared to minimax:

- Are we guaranteed a correct solution?

  Yes! Alpha-beta does not actually change the minimax algorithm, except for allowing us to skip some steps of it sometimes. We will always get the same solution from alpha-beta and minimax.

- Are we guaranteed to get to a solution faster?

  No! Even using alpha-beta, we might still have to explore all b^d nodes. A LOT of the success of alpha-beta depends on the ordering in which we explore different nodes. Pessimal ordering might causes us to do no better than Manama's, but an optimal ordering of always exploring the best options first can get us to only the square root of that. That means we can go twice as far down the tree using no more resources than before. In fact, the majority of the computational power when trying to solve games goes into cleverly ordering which nodes are explored when, and the rest is used on performing the actual alpha-beta algorithm.

Interesting Side Note - Konig's Lemma

I will use this opportunity to introduce an interesting theorem from graph theory that applies to our game graphs, called Konig's Lemma:

Theorem: Any graph with a finite branching factor and an infinite number of nodes must have an infinite path.
Proof: Assume we have a graph with each node having finitely many branches but infinitely many nodes. Start at the root. At least one of its branches must have an infinite number of nodes below it. Choose this node to start our infinite path. Now treat this new node as the root. Repeat. We have found an infinite path.

How does this apply to our game trees? This tells us that for every game, either:

1. It is possible for the game to never end.
2. There is a finite maximum number of moves the game will take to terminate.

Note that we are assuming a finite branching factor, or in other words, each player has only finitely many options open to them when it is his or her turn.

Implementation

As we have said over and over again, actually implementing these huge game trees is often a huge if not impossible challenge. Clearly we cannot search all the way to the *bottom* of a search tree. But if we don't go to the bottom, how will we ever know the value of the game?

The answer is we don't. Well, we guess. Most searches will involve searching to some preset depth of the tree, and then using a ***static evaluation function*** to guess the value of game positions at that depth.

Using an evaluation function is an example of a ***heuristic*** approach to solving the problem. To get an idea of what we mean by heuristic, consider the following problem: Robby the robot wants to get from MIT to Walden Pond, but doesn't know which roads to take. So he will use the search algorithm he wrote to explore every possible combination of roads he could take leading out of Cambridge and take the route to Walden Pond with the shortest distance.

This will work... eventually. But if Robby searches every possible path, some other paths will end up leading him to Quincy, some to Providence, some to New Hampshire, all of which are nowhere near where he actually wants to go. So what if Robby refines his search. He will assign a heuristic value, the airplane (straight line) distance to each node (road intersection), and direct his search so as to choose nodes with the minimum heuristic value and help direct his search toward the goal. The heuristic acts as an estimate that helps guide Robby.

Similarly, in game search, we will assign a heuristic value to each game state node using an evaluation function specific to the game. When we get as far as we said we would down the search tree, we will just treat the nodes at that depth as leaves evaluating to their heuristic value.

Evaluation Functions

Evaluation functions, besides the problem above of finding the optimal ordering of game states to explore, is perhaps the part of game search/play that involves the most actual thought (as opposed to brute force search). These functions, given a state of the game, will compute a value based only on the current state, and cares nothing about future or past states.

As an example evaluation, consider one of the type that Shannon used in his original work on solving chess. His function (from White's perspective) calculates the value for white as:

- +1 for each pawn
- +3 for each knight or bishop
- +5 for each rook
- +9 for each queen
- + some more points based on pawn structure, board space, threats, etc.

it then calculates the value for black in a similar manner, and the value of the game state is equal to White's value minus Black's value. Therefore the higher the value of the game, the better for white.

For many games the evaluation of certain game positions have been stored in a huge database that is used to try to "solve" the game. A couple examples are:

- OHex - partial solutions to Hex games
- Chinook - database of checkers positions

As you can see, these functions can get quite complicated. Right now, evaluation functions require tedious refinements by humans and are tested rigorously through trial and error before good ones are found. There was some work done (cs.cmu.edu/ jab/pubs/propo/propo.html) on ways for machines to "learn" evaluation functions based on machine learning techniques. If machines are able to learn heuristics, the possibilities for computer game playing will be greatly broadened beyond our current pure search strategies.

Later we'll see a different way of evaluating games, using a class of numbers called the surreal numbers, developed by John Conway.

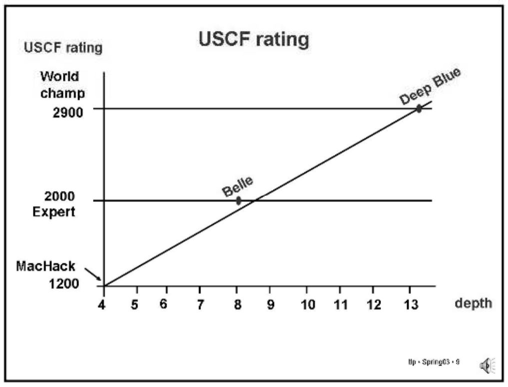
Solving a Game

We often talk about the notion of "solving" a game. There are three basic types of "solutions" to games:

1. ***Ultra-weak*** The result of perfect play by each side is known, but the strategy is not known specifically.
2. ***Weak*** The result of perfect play and strategy from the start of the game are both known.
3. ***Strong*** The result and strategy are computed for all possible positions.

How far do we need to Search?

How far do we need to search down the tree for our computer player to be successful? Consider the following graph (the vertical axis is a chess ability score):



(taken from 6.034 coursenotes).
Deep Blue used 32 processors, searched 50-10 billion moves in 3 minutes, and looked at 13-30 plys per search.

Clearly, to approach the chess -playing level of world champion humans, with current techniques searching deeper is the key. Also obvious, is that real players couldn't possibly be searching 13 moves deep, so there must be some other factor involved in being good at chess.

Is game-play all about how many moves we see ahead?

If searching deep into the game tree is so hard, how are humans able to play games like Chess and GO so well? Do we play by mentally drawing out a game board and performing minimax? It seems that instead humans use superior heuristic evaluations, and base their moves on experience from previous game play or some sort of intuition. Good players do look ahead, but only a couple of plys. The question still remains as to how humans can do so well compared to machines. Why is it hardest for a machine to do what is easiest for a human?

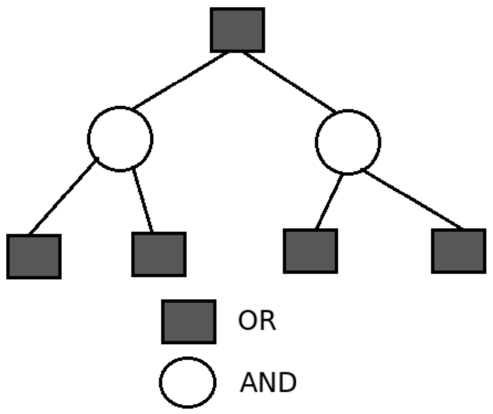
Alternative Search Methods

There are countless tweaks and alternatives to the maximin and alpha-beta pruning search algorithms. We will go over one, the proof-number search, here, and leave another variation, conspiracy number search, for our discussion next week on connect four.

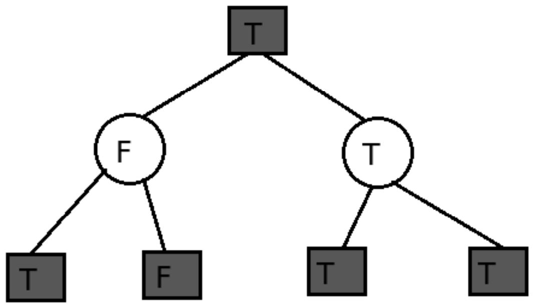
PN-search

While alpha-beta search deals with assigning nodes of the game tree continuous values, proof number search decides whether a given node is a win or a loss. Informally, pn-search can be described as looking for the shortest solution to tell whether or not a given game state is a win or a loss for our player of interest.

Before we talk about proof number search, we introduce AND-OR trees. These are two level alternating trees, where the first level is an OR node, the second level consists of AND nodes, etc. The tree below is an example:



If we assign all the leaves to values (T)rue or (F)alse, we can move up the tree, evaluating each node as either the AND of its leaves or the OR of its leaves. Eventually we will get the value of the root node, which is what we are looking for.



For any given node, we have the following definitions:

- **PN-number**: The proof number of a node is the minimum number of children nodes required to be expanded to prove the goal.
  - AND: pn = Σ(pn of all the children nodes)
  - OR: pn = (argmin(pn of children nodes))
- **DN-number**: The disproof number is the minimum number of children nodes required to disprove the goal.
  - AND: dn = argmin(dn of children nodes)
  - OR: dn = Σ(dn of children nodes)

When we get to a leaf, we will have (pn,dn) either $(0, \infty)$, $(\infty, 0)$, or (1,1), since the game is either a sure win or sure loss at that point, with itself as its proof set. The tree is considered solved when either pn = 0 (the answer is true) or dn = 0 (the answer is false) for the root node.

When we take a step back and think about it, an AND/OR tree is very much a minimax tree. The root starts out as an OR node. So the first player has his choice of options, and he will pick one that allows his root node to evaluate to true. The second player must play at an AND node: unless he can make his node T no matter what, (so F for player 1), then player one will just take one of the favorable options left for him. So an AND/OR tree is just a min/max tree in a sense, with ORs replacing the MAX levels and AND replacing the MIN levels.

PN search is carried out using the following rough outline:

1. Expand nodes, update pn and dn numbers.
2. Take the node with the lowest pn or dn, propagate the values back up until you reach the root node *.
3. Repeat until the root node has pn=0 or dn = 0.

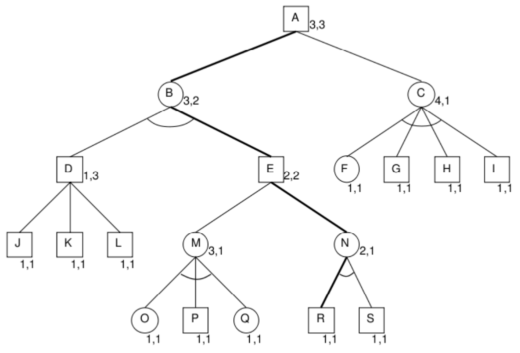
The slightly tricky part of this is the second step. What we really want to find is the ***Most Proving Node***. Formally, this is defined as the frontier node of an AND/OR tree, which by obtaining a value of True reduces the tree's pn value by 1, and obtaining a value of False reduces the dn by 1. So evaluating this node is guaranteed to make progress in either proving or disproving the tree.

An important observation is that the smallest proof set to disprove a node and the smallest proof set to prove a node will always have some nodes in common. That is, their intersection will not be empty. Why is this? In a brief sketch of a proof by contradiction, assume for the contrary that they had completely disjoint sets of nodes. Then we could theoretically have a complete proof set *and* a complete disproof set at the same time. But we cannot both prove and disprove a node! So the sets must share some nodes in common.

What we get out of all of this is that we don't really have to decide whether we will work on proving or disproving the root node, as we can make progress doing both. So now we can be certain of what we will do at each step of the algorithm. The revised step 2 from above is:

At an OR level, choose the node with the smallest pn to expand. At an AND level, choose the node with the smallest dn to expand.

The tree below is an example of pn search, taken from Victor Allis's "Searching for Solutions, " in which "R" is the "most-proving node."



Others

I will just briefly mention a couple of other variations on game search that have been tried, many with great success.

- Alpha-beta, choosing a constrained range for alpha and beta beforehand.
- Monte Carlo with PN search: randomly choose some nodes to expand, and then perform pn-search on this tree
- Machine learning of heuristic values
- Group edges of the search tree into "macros" (we will see this)
- A gazillion more.

Next Time: Connect Four and Conspiracy Number Search!

References

- Victor Allis, Searching for Solutions, http://fragrieu.free.fr/SearchingForSolutions.pdf
- Intelligent Search Techniques: Proof Number Search, MICC/IKAT Universiteit Maastricht
- Various years of 6.034 lecture notes