Efficient Implementation of Game Trees

Nicholas Gorski — DigiPen Institute of Technology
nicholasgorski@ngorski.com

When Deep Blue beat Chess World Champion Gary Kasparov, computer-powered combinatorial game playing established itself as a worthwhile pursuit. Since then, countless techniques and improvements have been made to computer play. In this article, we will examine the current evaluation techniques used for two-player games and demonstrate the implementation of a computer opponent for a two-player board game.

We will first describe our game of choice, the Game of the Amazons. Afterwards, we will examine games from a theoretical viewpoint to establish a model to work with, and look at methods of tree searching. From there, we will discuss both the general and game-specific aspects of efficient implementation, as well as proper game evaluation.

Game of the Amazons

Game of the Amazons[1], or just *Amazons*, is a two-player board game invented by Walter Zamkauskas in 1998 [Pegg99]. The game is governed by only four simple rules:

1. Two players alternate turns, choosing one of their Amazons to move. By convention, white will start playing first.
2. An Amazon can move a "Queen's move", like a Queen in chess, in any direction from its starting point. From there, the Amazon will fire an arrow a Queen's move away. The arrow acts as a wall, effectively removing that space from the board.
3. Neither an Amazon nor an arrow may pass through anything but empty space.
4. If a player cannot move, the game ends and the other player is the winner.

The starting position of a standard 10x10 game of Amazons is shown in Figure 1.

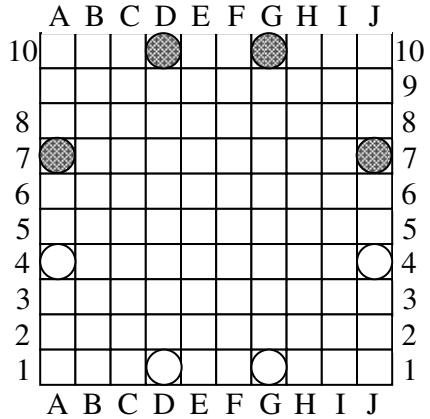

Figure 1. The starting position of Amazons.

[1] In its original Spanish: *El Juego de las Amazonas*.

Although the game description is simple, Amazons is an immensely rich game. The first player has 2,176 possible moves on his first turn alone, compared to 20 in Chess. Because Amazons is easy to understand yet so difficult to play, it makes it an attractive option for computer play.

Our goal is to win the game. Ideally, we would find *the* winning strategy and play it; is this feasible? In a simple game like Tic-Tac-Toe it is, but for more complex games it may not be. The quantity we'd like to measure is the *game-tree complexity*. This quantity measures the number of nodes we would need to evaluate to know which player has the winning strategy from any position from the start, and what that strategy it is.

To calculate it, we determine the average branching factor for the game, and the average depth, and calculate b^d. For Tic-Tac-Toe, the final result is $\approx 10^5$. The branching factor is the number of moves a player may make on a given position; in Amazons each player has about 336 possible options per move on average. Along with the average game length, 84 moves, we can determine our value: $\approx 10^{212}$ [Hensgens01]. This is more than the number of atoms in the visible universe ($\approx 10^{80}$), so we clearly can't evaluate the entire game tree. We'll have to search it.[2]

Game Theory

Before we can begin evaluating games, we need to describe what a game is and how we can represent our game in a model that lends well to our question. Unfortunately a comprehensive lesson is out of the scope of this article, but the basics will work.

A game can be defined recursively as a set of possible moves for each player, each to another game. The base case is a terminal game, where a player has lost. This recursive definition lends itself naturally to a tree representation, as shown in Figure 2.

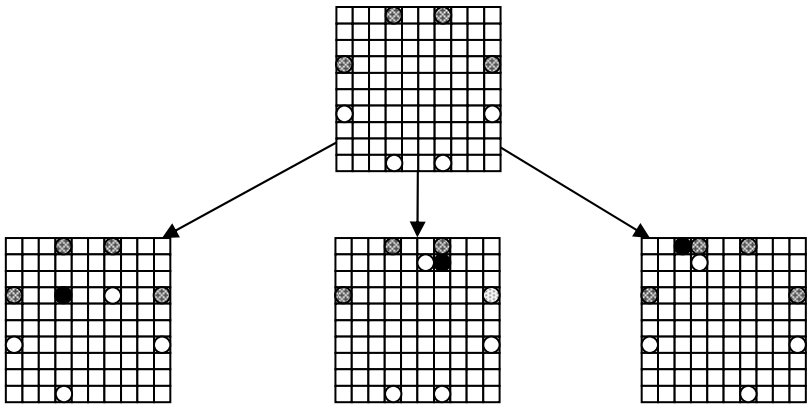

Figure 2. Some of the tree from the initial Amazons position.

[2] For comparison, the average branching factor for Chess is about 35, with an average game length of 80, giving it a game-tree complexity of $\approx 10^{123}$.

We can now model any game by moving further down the tree graph, playing the game. Our question can now be rephrased: which way should we move down the tree to win?

We need a way of ranking one possible move above another. A simple way to do that would be to assign numbers to the nodes, as we already know how to order numbers. We can make game values positive if they are better for first player or negative if they are better for second player. To see how this might be done, consider the small tree below:

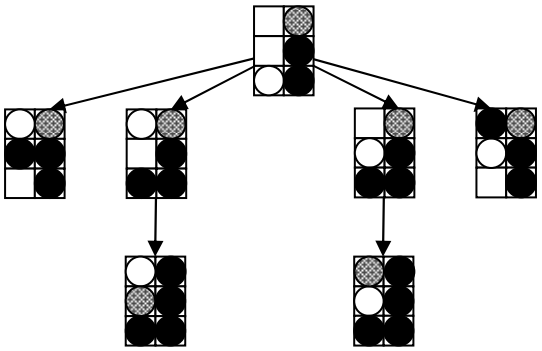

Figure 3. A complete game tree.

Here we see that white, the first player, has four choices from the start. Two of those are immediate wins, as second player has no response. However, the other two leave black with a single reply, completing the game with a loss for white. Therefore, white should prefer the former moves.

Sticking to our number scheme, a first player win should be the highest positive value, and a second player the lowest negative value. If we assign ∞ to a first player win (the highest positive "value"), and -∞ to a second player win, we can propagate the game value upwards in the tree, as in Figure 4.

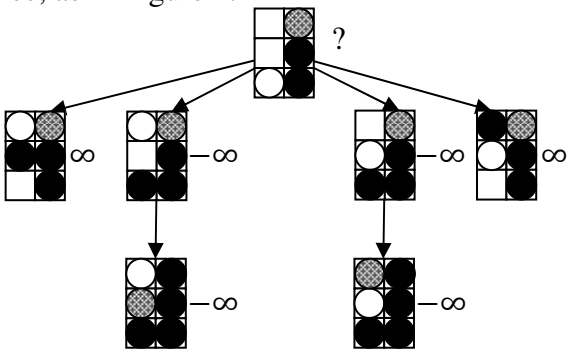

Figure 4. Game values propagated through tree. If a game has only one possible option left, its value must be the same as that option. What about multiple children nodes?

Propagation is easy when the node is the only child of the parent, as the parent can only go to that node, therefore copying its value. But now we have four choices for the value of the root node, so which value do we choose? As first player we want to move to higher values, and we indeed have a move (two, in fact) to the highest value there is. Since we're finding the optimal strategy, our move would be to that highest node. So the root node has a value of infinity for first player.

In other words, as first player we want to select the *maximum* value of the children nodes, and the second player will want to select the *minimum* value of children nodes. This method is how we will try to search game tree.

Game Tree Searching

As we described in the last section, first player wants to maximize the node values, and second player wants to minimize them (maximizing his own value); this concept is known as the Minimax decision rule. We will first cover the basics of searching starting from this principle, and make gradual improvements to the search.

Basics

The basic search idea is this: play the board in a depth-first manner, and when a terminal node is reached evaluate it and all of its siblings. To evaluate the parent node apply the Minimax decision, selecting the optimal child node for the current player. We can do this to the parent's siblings as well, and then likewise for their parents, until we reach the starting game position. Completed, we will know the value of the game and the optimal strategy.

But as noted in the beginning, we cannot do this in practice because the tree is too large; we would never be able to evaluate every terminal node. This means we need to limit the scope of our search, and the most obvious way to do that is to limit the depth to a hard value. Later we will see a better solution to this problem, but for now we can simply set an arbitrary limit, in order to examine the algorithm.

In a full search, there is no question as to what a terminal node's value is, because it's either a win for white or a win for black. But with our depth limited, we will terminate at games in which no player has yet won. What value should a node have when depth-limited? This is where game-specific knowledge comes into play: the value should be the best possible *approximation* of the true value. However, until this point we've only seen positive and negative infinity! It would be disastrous to incorrectly approximate a node as a first player win if it were actually a second player win, because the first player would work towards this node…in effect helping the second player win.

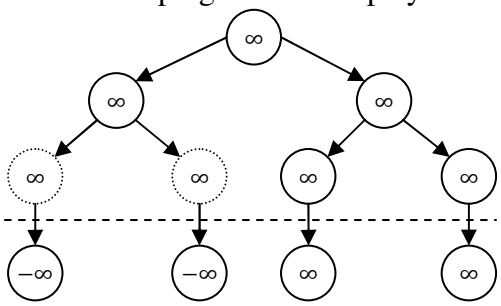

Figure 5. The dashed line is the horizon that the first player cannot search past due to the depth limit. The nodes with a semi-solid outline were incorrectly evaluated by the evaluator. If the first player moves on the left branch (neither branch is preferable), he actually moves to a losing position. Infinite values are too binary for estimation.

Clearly we need finite values: while marking wins or losses at infinity is useful and meaningful for the search, it's too black and white to make "unsure" decisions that can later be improved. Luckily, finite values are also meaningful. Consider a single segment:

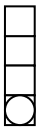

Figure 6. A single Amazons segment with a white token.

In this position white has several moves. The worst of these is to move the token to the top and shoot an arrow to the space under the token, removing any further moves. Clearly a much better is to move the token anywhere and shoot the arrow to either the very top or bottom, preserving moves. This is how finite values can be meaningful: the value of a non-terminal position can be the number of moves remaining for the player (and in the case of two players, the difference). During maximization, the worst move would have the lowest value, and other moves would be preferable.

But this is only one possible evaluation function, and choosing an evaluation function is non-trivial. Is the difference of move count really meaningful? What if tokens are separated into exclusive regions? Is the number of moves remaining the only important aspect of a position? What is important in a position? These questions are difficult to answer, but the better we answer them the better we can guess a game value. We will discuss Amazon heuristics later, so for now we will evaluate a simple mystery game with oracle evaluations.

The algorithm in pseudo-code:

```
int minimax(node n, unsigned depthRemaining) {
    if (n.terminal || depthRemaining == 0) {
        n.value = estimated_value(n);
        return n.value;
    }

    // node.player is 1 for first player, -1 for second
    int bestScore = -n.player * INFINITY;

    for each child node of n {
        int score = minimax(child, depth - 1);

        if (n.player == 1) // maximizing
            bestScore = max(bestScore, score);
        else // minimizing
            bestScore = min(bestScore, score);
    }

    n.value = bestScore;
    return n.value;
}
```

```
// initial call:
minimax(currentPosition, maximumDepth);
```

Try tracing the evaluation in Figure 7 yourself; player one to move.

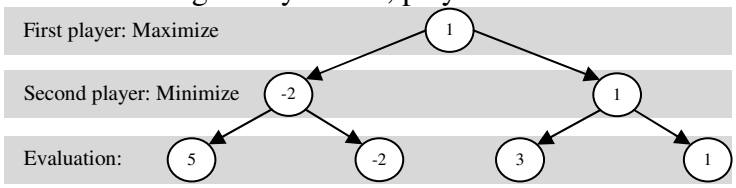

Figure 7. An evaluated game tree up to depth two.

We can now evaluate a game tree up to a fixed depth, playing the best known or best approximated move. However, we have the glaring problem that we simply assumed an arbitrary maximum depth. In order to play arbitrary amounts of time, for time control, and for better evaluation, to avoid evaluating too much or too little, we're going to need to figure out the best way to handle depth. The solution is called *iterative deepening*.

The idea is simple but radically effective: first evaluate up to depth 1, then to depth 2, etc., until a solution is found or time runs out. As we will see shortly, not only does this solve both obstacles above, but it enables an enormous opportunity for optimization that we'll discuss later. For now, though, just note the search is now only limited to the time it has to process. Later we'll show how to use previous lower-depth searches to save time.

Implementing this requires no modification to the search algorithm itself, just the way it's called:

```
// initial call:
for (unsigned depthLimit = 1;
        currentPosition.value != INFINITY &&
        currentPosition.value != -INFINITY &&
        timeLeft > 0;
            ++depthLimit)
    minimax(currentPosition, depthLimit);
```

In practice, time control is better implemented by searching within a thread, and stopping the thread when time runs out, to allow interruptions during evaluations.

We now have the baseline functionality necessary to evaluate any game position in standard time control. However, it still does too much unnecessary work to be used for a strong computer player. Luckily, there are a variety of ways to improve the search.

Improvements

Most improvements fit into two categories: simplification and pruning. Simplifications are modifications to the algorithm that take advantage of the properties of the game values and search results in order to perform fewer operations. Pruning algorithms

evaluate the nodes in a manner that avoids nodes it knows will have no effect on the final value. We will first look at a simplification.

NegaMax

The first simplification comes from observing we're evaluating a zero-sum game; that is, a game where one players gain is exactly the other player's loss. In that case, a player's best move is exactly equal to the other player's worse move. Observe how the tree from Figure 7 changes in NegaMax form:

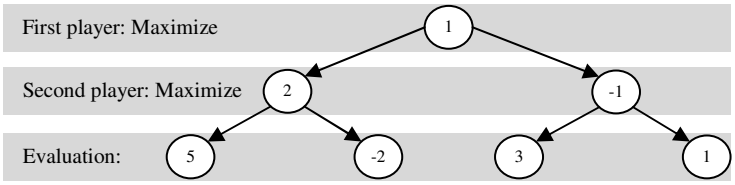

Figure 7b. In NegaMax form, both players are choosing the largest value from the negation of the child evaluation. They are maximizing the next player's lowest number.

Note that the evaluation function should also change. In Minimax, it always evaluated from the first player's perspective, with positive meaning first player advantage and negative meaning second player advantage. However, in NegaMax the evaluation function needs to be adjusted to match the currently evaluating player; otherwise the first player will minimize his own gains. The algorithm is now simplified:

```
int negamax(node n, unsigned depthRemaining) {
    if (n.terminal || depthRemaining == 0) {
        n.value = estimated_value(node);
        return n.value;
    }

    int bestScore = -INFINITY;

    for each child node of n {
        int score = -negamax(child, depth - 1);
        bestScore = max(bestScore, score);
    }

    n.value = bestScore;
    return n.value;
}
```

This simplification helps in multiple ways: from the programming point of view, it's easier for the programmer to reason about, and is easier to maintain. To the compiler the function is also easier to reason about, aiding in optimization, and contains less branching. In both cases, any optimizations made can be made in a single location.

Alpha-beta Pruning

Having looked at a simplification improvement, we'll now look at the most important pruning improvement: alpha-beta pruning. The idea is simple, but requires a bit of practice to become fully intuitive.

Before we look at any trees, we'll look at it from a simple mathematical viewpoint. Note that we can answer the following question: What is the value of $max(3, min(1, \mathrm{X}))$? The answer is 3, regardless of the value of X! No matter what value X has, $min(1, \mathrm{X})$ will be *at most* 1; and $max(3, 1)$ is 3.

In other words: if we are minimizing a node and know that the parent node has a pending maximum that's higher than the lowest evaluation at the minimizing node, no other nodes in the minimizing node will make a difference: the pending maximum will be chosen over the lowest minimum. Figure 8 shows the alpha-beta evaluation of a Minimax tree:

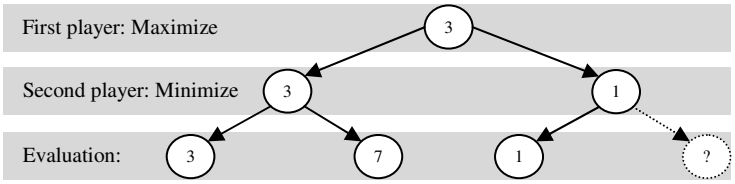

Figure 8. After evaluating the left branch, the algorithm knows one argument to the maximizer: three. When the right branch evaluates its terminal node and determines its value will at most be one, the evaluation can end because the first player already knows that branch won't be taken, as three is higher than one.

For a small example tree the benefits are hard to see, but in practice this technique can cut enormous chunks of the tree from evaluation. In code, we add this improvement by including a *window* to the search. We call the lower bound *alpha* and the upper bound *beta*: alpha is the lowest value the parent node will pick, and beta the highest. We start with the window at $[-\infty, \infty]$ and as the child nodes are evaluated the window is updated. Namely, the first player will maximize alpha, because he intends to pick nothing lower, and second player will minimize beta. If the window is ever broken, we can stop. In code:

```
int alphabeta(node n, unsigned depthRemaining,
                int alpha, int beta) {
    if (depthRemaining == 0) {
        n.value = estimated_value(node);
        return n.value;
    }

    for each child node of n {
        int score = alphabeta(child, depth - 1,
                              alpha, beta);

        if (n.player == 1) // maximizing
            alpha = max(alpha, score);
            if (alpha >= beta) break;
```

```
        else // minimizing
            beta = min(beta, score);
            if (beta <= alpha) break;
    }

    n.value = (n.player == 1) ? alpha : beta;
    return n.value;
}

// initial call:
alphabeta(currentPosition, currentDepth,
            -INFINITY, INFINITY);
```

In NegaMax, the window for a child node is the negative reverse of the parent window and only alpha is updated, since both players are maximizing. This observation allows us to keep our simplified code, with minor additions.

Move Ordering

We now have a simple and efficient method for searching a tree. However, there is one more powerful optimization to be made, and was alluded to when iterative deepening was first mentioned. This optimization aids the alpha-beta pruning improvement by ordering moves, from best to worst. The idea is simple: the better the evaluation, the higher alpha is and the closer to cut-off we get. Alpha is raised the most when the node we evaluate gives the highest value. So by ordering from highest to lowest, we minimize the size of the window as quickly as possible.

This is a fine idea, but how do we know which way to order the moves when we first come upon them? After all, the point of the search is to figure out the values of the nodes: if we could order the moves from the start we'd have no need to search. The answer lies in iterative deepening. Recall that iterative deepening will first open up the children nodes at depth one and evaluate them. It then starts the search over but stops at depth two, and so on. The key is that when starting a new search at depth D, the nodes at depth D - 1 contain values. Namely, they contain the estimated values of the nodes, from the evaluator, which will be refined by further searching. We can use these estimations to give a rough order to the moves; and the better the evaluator, the better the ordering.

Because the number of moves at depth D is exponentially less on average than the moves at depth D + 1, taking the time to sort the existing nodes is well worth the effort, because it helps make significant alpha-beta cuts:

```
int negamax(node n, unsigned depthRemaining,
                int alpha, int beta) {
    if (n.terminal || depthRemaining == 0) {
        n.value = estimated_value(node);
        return n.value;
    }
```

```
    for each child node of n, sorted by value {
        int score = -negamax(child, depth - 1,
                                -beta, -alpha);
        alpha = max(alpha, score);
        if (alpha >= beta)
           break;
    }

    n.value = alpha;
    return n.value;
}
```

We now have the standard practical starting point for a game tree search.

Implementation

Searching game trees is resource-intensive. If done with disregard, memory can be quickly exhausted and searches will be wastefully slow.

Board Storage

One of the primary things we'll be doing is looking at board positions, for navigating the tree. In many search algorithm implementations the child node loop looks similar to:

```
// implementing: for each child node of n, sorted by value
sort(n.moves)
for each move from n {
    make(move);

    // search

    unmake(move);
}
```

The board is modified temporarily for searches, but in the end it remains identical. This is extremely space efficient, as no boards are copied, but it has a deficiency: it cannot be made concurrent. The board can only have one active move at a time, and if we try to optimize our search with threads we'll run into problems. While we won't cover concurrent searches, we can plan accordingly. We need to optimize memory usage.

The first thing to observe is the simplicity of a board square. In Amazons, a space has four possible values: empty, blocked, a player one token, or a player two token. We can represent a board space with just two bits; this is excellent, because our integral data types happen to have bit-sizes that are powers of two. We can pack four spaces into eight bits, a huge space saver compared to any other data type we might use. We can store a

single 10x10 board in just 25 bytes, which not only saves memory for the game tree, but is efficient to copy during processing.

A second improvement is to note that leaf nodes are expected to be cut from the search quite often. In fact, if the ordering is good enough most leaf nodes will *never* be evaluated because they'll always be cut off by a previous move. It would be a waste of memory for these leaf nodes to have a copy of the board. This introduces the concept of a *tentative move*. The idea is that when a parent node expands to add its children to the tree, they're all actually identical to the parent board except for the move from the parent node. Therefore, they can all share the same board representation, and only store their move. Only when a node with a tentative move is expanded does it commit its move, gaining its own representation of the board in memory; this allows its children to make tentative moves. At any point of the search, only stem nodes contain a copy of the board.[3]

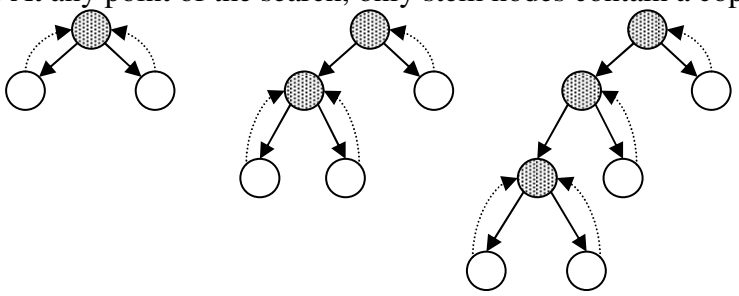

Figure 9. As the tree is searched, the leaf nodes share board memory with their parent, waiting to commit their move until they are examined. (Grey nodes contain board data.)

Note that querying a board space becomes a costly operation with tentative moves, because every query is no longer a simple check to the board representation. It has to additionally check if the position being queried was involved in the tentative move, and adjust the result if so. Thus a new problem is introduced: evaluations, which query constantly, slow down tremendously. Observe, though, that most of our memory use is from the tree being stored in memory, ergo most of our memory savings come from boards in the tree. If the board's existence is temporary, tentative moves make little sense, as the memory will soon be released. Therefore, we can solve our problem by making a temporary non-tentative board during evaluation, which has only quick space queries, at the cost of a single board and copy. The board's size in memory is small enough, due to the first optimization, that the time to copy a tentative board is minimal and well worth the effort.

Game Tree Searching

Game tree searching is straightforward, and nearly matches the ideas expressed in the pseudo-code in previous sections. There are, however, a few worthwhile optimizations to be made.

The first and most important is to reuse previous search data during play. While a single search will use the existing information optimally, by ordering moves and pruning, consecutive searches will start the entire search from scratch. This is a waste: rather than

[3] A stem node is a node with leaf nodes for children.

start the tree over, the tree can simply trim away the nodes that are no longer accessible and keep the previously evaluated nodes. We do this by letting the tree *observe* a move: it finds the child node that corresponds to that move and takes its data, making it the root.

The second optimization is another pruning improvement, called NegaScout[4]. The idea behind NegaScout is along the same lines as that of alpha-beta: we will trim nodes that won't affect the result of our search. The difference is that NegaScout tries to do so early, by forcing children to use a narrower window than they should; namely, by using a *null window*, where the size is one. In effect, it says, "If the child node value doesn't beat the current best value *with* a handicap, then it also wouldn't beat it *without* the handicap." By adding a handicap, we shrink the alpha-beta window and prune more nodes, increasing search speed greatly. However, if it turns out there is a better move, even with the handicap, we have to remove the handicap and search the tree with a full window, to get the actual highest value. This trade-off is justified with good move ordering, where re-searches are minimized due to correctly predicted cut-offs. We then have:

```
int negamax(node n, unsigned depthRemaining,
                int alpha, int beta) {
    if (n.terminal || depthRemaining == 0) {
            n.value = estimated_value(node);
        return n.value;
    }

    int b = beta;
    for each child node of n, sorted by value {
        int score = -negamax(child, depth - 1,
                             -b, -alpha);
        if (child is not the first &&
                score > alpha && score < beta) {
            // found a better move in the
            // handicap window, full search
            score = -negamax(child, depth - 1,
                             -beta, -alpha);
        }

        alpha = max(alpha, score);
        if (alpha >= beta)
           break;

        b = alpha + 1; // reset null window
    }

    n.value = alpha;
    return n.value;
}
```

[4] This search optimization is also called the Principal Variation Search.

Other Improvements

Like any application, there are numerous smaller improvements that can be tested with a profiler. Some test results from this case study were: do use an improved memory allocator; do allocate the board statically, instead of dynamically; do not sort the moves concurrently; do not scan the board concurrently, do not search the tree concurrently.

In this case "concurrently" means "naively concurrently": efficient concurrency for game tree searching is not something to attempt without good knowledge of concurrency.

Evaluating Amazons

As a relatively young game, the evaluation of Amazons positions is still quite uncharted. There is no sure-fire way to generate a heuristic, and most simply come from experience. However, there are three primary evaluation methods we can examine for good results.

Mobility

The first method measures the mobility of the pieces. Calculation for a player's mobility is simple: for each piece of the player, count how many squares it can reach in one move. The evaluator returns the mobility of player one subtracted by the mobility of player two.

This is the simplest and quickest evaluation method, but lacks the ability to notice important game features. For example, in maximizing mobility the pieces will tend to stay in the center of the board, so much so that the other player can easily capture the surrounding area for himself.

Territory

Rather than measure mobility, we can try measuring territory. Territory is a measure of how many squares a player "owns". We say a player owns a square when he can reach that square in less moves than the other player. Squares that can be reached at the same time count towards neither player.

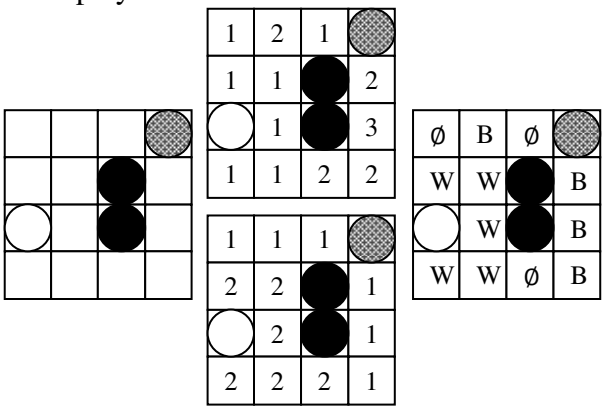

Figure 10. The leftmost board is the game position. The top center board is the move counter for first player and the bottom center board for the second player. The rightmost board indicates the owner of a space. The value of the entire board is the number of squares white owns minus those of black. In this case, the value is 1.

This is a large improvement, because pieces are now aware of their influence over things nearby. However, this heuristic isn't without its problems as well: pieces will tend to sacrifice themselves for territory that seems good, but is actually only good because it's unattended. Once one or two opponent pieces enter, it may end up being quite bad. This heuristic doesn't account for multiple contenders.

Territory-Mobility

Luckily, we can acquire a strong heuristic by combining the previous two values. We do a territory evaluation, but also keep track of the "mob" number of a square: the number of pieces that can reach it in one move. Whenever a player owns a square, he adds not just one, but additionally the mob number of the square. This is like measuring the assuredness of the square.

In practice, the importance of these two measurements is not equal; territory is still more important. Hashimoto suggests a weight of four is best for the territory portion, a result gained after testing several values [Hashimoto01]. Therefore, for each owned square the player gets four points, along with the mob number.

Other Heuristics

There are other heuristics for Amazons that have not been implemented in the demo. Some of these are discussed by Qian Liang's thesis paper [Liang03], with promising results. Another approach is to apply learning algorithms, to adjust the specific weights of each evaluation.

Future Directions

With the above implementation, the computer can play at an intermediate level. However, various improvements can be made to strengthen the program.

The most important improvements are that of speed: the deeper the program can search, the better it can evaluate current nodes. Luckily, there is much room for improvement. There is a class of optimizations know as *forward pruning* that attempt to eliminate uninteresting nodes from being expanded. Some keywords to search for are: killer move heuristic, history heuristic, and refutation tables. Of these, the killer move heuristic is simplest.

The raw speed of evaluation can be increased as well. The board can be transformed into an undirected graph representation, where squares are bidirectionally connected to other squares. Not only does this simplify navigation, but many different looking boards end up having the same representation. Lastly, we can try to apply concurrency to the evaluation, but great care must be taken to scale properly and avoid shared data.

Conclusion

In this article, we discussed game trees and tree searches, along with the various optimizations that can be made. We discussed memory saving techniques and alluded to more advanced pruning techniques. Finally, we applied the theory to the Game of the Amazons, resulting in a performant intermediate level artificial intelligence, ready to be modified and improved.

References

[Pegg99] Pegg Jr., Ed, "Amazons," available online at http://www.chessvariants.org/other.dir/amazons.html, March 10, 1999.

[Hensgens01] P.P.L.M. Hensgens, P.P.L.M., *A Knowledge-based Approach of The Game of Amazons*, Universiteit Maastricht, 2001.

[Hashimoto01] T. Hashimoto, Y. Kajihara, N. Sasaki, H. lida, J. Yoshimura. *An evaluation function for amazons*, in: H.J. van den Herds, B. Monien (Eds.), Advances in Computer Games, Vol. 9. Universiteii Maastricht, Maastricht, 2001,pp. 191-203.

[Liang03] Liang, Qian, *The evolution of Mulan: Some studies in game-tree pruning and evaluation functions in the game of Amazons*, University of New Mexico, 2003.