Minimax with Konane:

An evaluation of the Minimax and the Minimax with α-β Pruning algorithms on the board game Konane

Derek Reimanis

October 11, 2012

**Abstract**

*In this paper, we look at the construction process of the Konane board game and the construction of two different agents that will play on this board. The first agent presented is a Minimax agent, and the second agent presented is the same Minimax agent with the addition of the α-β pruning algorithm. We compare results from several thousand iterations of the game to determine how the type of algorithm coupled with variable game look-ahead depth affect win percentage.*

**1 Introduction**

Artificial Intelligence, that is, an ‘intelligent’ problem solver, generally revolves around the idea of a model in which an agent and an environment exist. The agent can perceive the environment using its percepts. An agent can also take actions that alter the environment using its actuators. In combination, an agent perceives the environment and returns an action that alters the environment. In what is referred to as a reflex-based model, an agent will apply the current state of the environment to a set of internal rules, thus modeling a virtual environment. This is done without taking any actions upon the actual environment, allowing the agent to effectively ‘think’ about future environment states. An agent can now plan ahead, using iterations of a virtual environment state to eventually reach a goal state. This is referred to as a goal-based agent; that is, an agent that applies the rules of the environment to a virtual copy of the environment to eventually reach a desired state (Stuart Russell, 2010).

The inclusion of a goal state than an agent is often not enough for an agent to generate high quality behavior (Stuart Russell, 2010). Because there are many ways of reaching a goal, questions of optimization occur in which we must reach the goal state in the most efficient manner. A typical utility-based model involves a utility function that returns a value of how desirable an inputted state is. As this agent nears the goal state of the environment, the utility value of the environment would generally increase.

Naturally, as we begin to evaluate the efficiency in which we reach a goal state, we need to evaluate the manner in which we do it. Often times an agent will exhaust multitudes of resources attempting to find a goal state. If an optimal function can instead be applied to reach a desirable state, not necessarily the goal state, then certainly it should be used. We can also assume that we get closer to the goal state as the desirability of neighboring states increases. This idea gives rise to applying search techniques to our model, in the form of a tree. If we represented our starting state as the root of a tree, then we can represent known future states as children in the tree (David Poole). The edges of this tree would be the action that was applied to get from parent to child. The problem then becomes finding the shortest path to get from the root to some desirable node, or solution. Since our agent has realized a desirable node is much easier to find than the solution we must apply a heuristic to that node. A heuristic is a reasonable estimation of how near the absolute optimal solution we are. A heuristic may even give us the absolute optimal solution. It is akin to understanding the saying, “Close, but good enough”. Goal-based agents can thus use a heuristic to alter the environment with close approximation to their goal

One strong application of goal-based models is the world of games. For the purpose of this paper, we will focus on games that involve two players, both of which alternate moves to reach a win/loss state. In these games, there is one clear winner and one clear loser. This form of game is often referred to as zero-sum. Zero-sum games have the same numerical payoff for every instance of the game (Stuart Russell, 2010). This means one player loses (0 points) and one loses (1 point), or they draw (1/2 point each). When playing a game like this with the goal of winning, a player must play the best they can. If a player were to play randomly, surely they would not do as well as if they applied a strategy.

It is as this point that we notice our definition for a utility-based agent model matches a zero-sum game. We describe the environment as the game’s board, and a set of rules we can apply to the game board as the possible moves of either player depending on the environment. We define a rule that results in a losing state as a negative utility function, and rules that give winning states as positive utilities. We can apply our heuristic as well to pick the ‘best’ path to get from an initial state to (ideally) a winning state. We can, theoretically, outplay any human of the game. We could even play the game in the ‘best’ way possible, from any move. The difficulty here, of course, is how we actually do that.

In section 2 of this paper, we will explore the definition of games and how we can build algorithms that utilize key features of games to return good move choices. In section 3, we discuss experiments performed to validate the algorithms and game definitions presented in section 2. In section 4, results from select experiments are shown and described. Finally, in section 5, we discuss the importance of the results, and hint on possible future studies.

**2 Game Search**

The first challenge of playing a game on a computer is, of course, building the game. We know that at an abstract level, games can be defined as having the following (Stuart Russell, 2010):

* *S*0: The initial state, which specifies how the game is set up at the start.
* Player(*s*): A definition of which player has the move in a state.
* Action(*s*): The set of legal moves in a state.
* Result(*s, a*): A definition of the results of a move.
* Terminal-Test*(s)*: A test that is true when the game is over.
* Utility(*s, p*): A function that defines the payoff for player p for a given state, *s*.

We now have an abstract definition for a game. In this paper, we instantiate the above definition to construct the board game, Konane. Also known as Hawaiian Checkers, Konane is a two-player game played on a square board. Players initialize the game board by placing pieces on squares of their color, as shown in Figure 1 below. Player 1 controls the black pieces, and player 2 controls the white pieces. Player 1 then removes a black piece from either the corners of the board, or the middle squares. Player 2 follows suit by removing a white piece directly adjacent to the square player 1 removed a piece from. After the initial two moves, the protocol is to use the pieces you control to jump over your opponent’s pieces. A piece that is jumped over is removed, and the piece that does the jumping now lies in a new square. Jumps may only be done horizontally or vertically, and the square the jumping piece lands in must be vacant. Multiple jumping is allowed, provided that it stays horizontal or vertical throughout the entire move. The game ends when one player has no moves remaining. The winning player is the player that still has moves remaining.

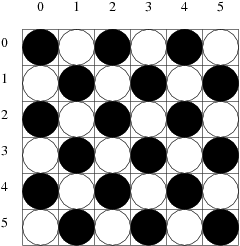


Figure -- The initial Konane game board

Implementing this game involves constructing the functions listed above; that is, functions that define which player has the move, a set of actions in a state, the results of actions in a state, a terminality test to check if the game has ended, and a value for a state respective to a player. For Konane, we define the terminal-test to be true when a state has no possible actions. Now, we construct a utility-based agent to interact with this model.

**2.1 Traditional Minimax Search**

The first agent we look at is an agent that utilizes the minimax search algorithm. The minimax search algorithm is a recursive algorithm that evaluates all the possible states of the game from the actions available from the current state. Minimax recursively finds new actions for each new state and continually proceeds until a terminal state is reached. At this point, the utility value of the terminal state is found and returned.

Minimax has a unique property that makes it optimal in two player games, but relatively difficult to apply in multi-agent games. As minimax recursively descends, it effectively builds a tree of all possible states from all possible moves. This rather agnostic approach does not get us anywhere, because we need to make a move, and we want that move to be of favorable utility down the road. Minimax accounts for this by labeling the nodes of every other ply, or tree depth, as minimum nodes or maximum nodes. Maximum nodes are the possible states that you will be in, and minimum nodes are those of your opponents. Minimax assumes that the opponent will make a decision most befitting to them, so it applies a minimum to any opponent’s ply. A minimum state is a maximum state in the opponent’s point of view. So Minimax assumes the opponent will make the best decision they can, and evaluates its own moves based on that fact. We are left with a recursive ‘flip-flopping’ of the best and worst states. The minimax algorithm is shown below in Figure 2.1 (Stuart Russell, 2010).

Figure 2.1 – The minimax algorithm

As you can see from the algorithm, we descend until a terminal state is found. Then, we return a numerical association based on our heuristic of a state’s utility. It may also be appropriate to apply a maximum depth to our tree, to ensure it does not exceed stack size boundaries when executing. This is done by effectively treating nodes that are at the depth cap as terminal nodes. In the experimental section of this paper, you will see this depth cap used as a variable in an attempt to see its effect on the algorithm’s performance.

**2.2 Minimax with Alpha-Beta Pruning**

The traditional minimax algorithm, while performing quite admirably, does have an issue. Because of its recursion until a terminal state is reached, the tree it builds expands until all terminal states are reached. For the observant reader, you may have noticed that this extension is not necessarily required to still achieve the meaning of minimax.

Take, for example, the following Figure 2.2:

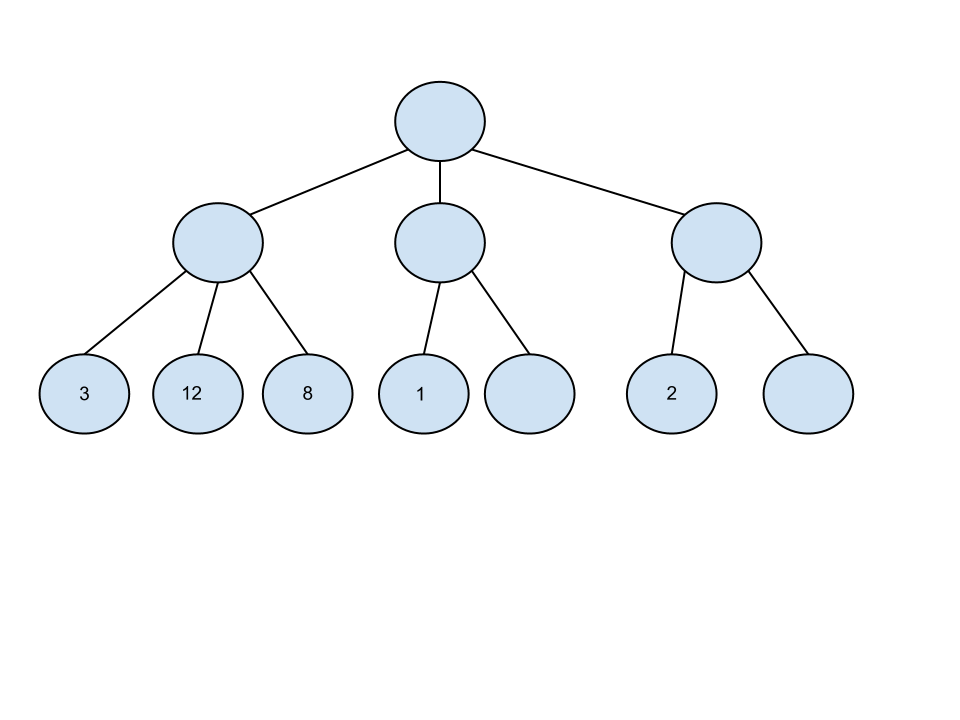


Figure .2 -- A snapshot of a minimax tree in the middle of its iterations

Notice that as the terminal states with values 3, 12, and 8 are evaluated, only the minimum will be used by the opponent. This means that the move associated with game state value 3 will be selected. As the opponent continues to choose minima, we notice something. The opponent will chose 1 as a minimum next. Now, the unknown utility value of the next cell does not need to be calculated. We know that the opponent will pick the minimum, and has already chosen a minimum from the left branch of the tree. But, the values for the next unknown state can only be smaller than 1 for it to propagate up. In either case, it doesn’t matter because since we are going to choose the maximum of 3 and 1, we will choose 3. That node does not even need to be evaluated. Continuing on, we see the exact same is true with the right branch. A value less than 3 is propagated up, which means we do not need to continue evaluating. This is the minimax with α-β pruning algorithm, which is formally shown below in Figure 2.3:

Figure 2.3 – Minimax with alpha-beta pruning

The implementation of this algorithm is actually very similar to that of traditional minimax. The only difference being two new utility variables, alpha and beta, tracking the smallest and largest utilities found at their levels.

**3 Experimental Methods**

Using the game of Konane, the traditional minimax algorithm and the minimax with alpha-beta pruning algorithm were both tested. A human player was also incorporated, to compare how the two algorithms performed against an opponent that would not always do the move with the highest utility. A board size of 8x8 tiles was used; the purpose of this size is to create a game with numerous possible moves. As mentioned previously, a tree depth cap was utilized. The purpose of the depth cap is to compare if the ability to look ahead further than the opponent affects the outcome of the game. The depth cap was either 1 or 20. A depth cap of 1 corresponds to a player that only evaluates the other player’s move. A depth cap of 20 is enough to at least capture any possible move in the 8x8 grid. These variables were swapped interchangeably with traditional minimax and minimax with alpha-beta pruning.

The utility function used gave numeric output for an inputted state. For every piece under your control in the corners of the board, ten points were awarded. For controlled squares directly adjacent to the corners, seven points were awarded. For controlled squared directly adjacent to the squares worth seven points, five points were awarded. Five points were also awarded to all controlled squares on the edges of the board. Finally, three points were awarded to all controlled squares adjacent to the squares worth five points. Another set of points was awarded based on remainder of moves remaining from a state. For each remaining move in a state, a numeric value of three times the number of moves remaining was applied.

The purpose of this utility function is to force the pieces to make their ways to the edge of the board, while still maintaining large move sets. Pieces on the edge of the board have less likelihood of being jumped because they border the board and pieces cannot jump off the board.

It is worth mentioning that the two algorithms randomize their first two moves. The underlying purpose of this is to avoid replicate games that may occur when the same sequence of introductory events happen.

The values kept track of during the runs were: number of wins per player, which was simply a running counter of wins per player at the end of a game, average number of moves per game, and various time measurements. The time measurements include the move that took the shortest time, the move that took the longest time, and the average time of all moves. Time measurements were recorded by a timer that started immediately before a call to an algorithm to make the next move, and ended immediately after returning. Both player 1 and player 2 had the exact same timer.

**4 Results**

In total, 160,000 tests were run. The first set of tests run compared two traditional minimax players with varying search depths of 1 or 20. Four tests of 10,000 game iterations each were performed. Their results are shown on Table 1 in Appendix A.

In the second set of tests, two minimax with alpha-beta pruning players were compared. As in a similar manner as the first test, varying search depths of 1 or 20 were applied to both players 1 and 2. Four tests of 10,000 game iterations each were performed. Their results are shown on Table 2 in Appendix A.

In the third set of tests, player 1 was a minimax with alpha-beta pruning player. Player 2 was a traditional minimax player. Both used varying combinations of 1 or 20 search depths. Four tests of 10,000 game iterations each were performed. Their results are shown in Table 3 in Appendix A.

In the fourth and final set of tests, player 1 was a traditional minimax player, while player 2 was a minimax with alpha-beta pruning player. Again, combinations of 1 and 20 search depths were applied to four tests of 10,000 game iterations. The results may be seen in Table 4 of Appendix A.

Figure 4 -- Average wins per 10,000 games

Figure – Total Wins for both players

Figure 6-- Average Wins Per Max Tree Depth

Figure 5-- Average Time Per Move

These tests cater to the experiments checking for how tree depth affects play ability, the differences of using alpha-beta pruning methods as opposed to traditional.

**5 Discussion**

Looking at the results, we notice that an interesting anomaly has occurred. In Figure 3, player 1’s wins clearly outweigh player 2’s wins. The first and second moves were both set up as random, yet the player with the first move appears to have an advantage. A puzzling situation. Next, we look at Figure 4 and how being a different player may or may not affect the ability of an algorithm to perform well. We can see that when the traditional minimax algorithm was applied to player 1, the highest average of win percentage was garnered. This is interesting, as the alpha-beta algorithm is but an extension of traditional minimax, implying that perhaps alpha-beta wins should be higher. This is reflected in the player 2 choice, as alpha-beta more average wins than traditional. Perhaps a likely cause is that randomizing the first and second moves is affecting the results in more ways than thought.

Figure 5’s results are to be expected; that is, alpha-beta times were on average nearly twice as fast as the traditional minimax’s. This is because alpha-beta cuts out many of the calculations traditional minimax would normally do. Finally, looking at how the depth of a tree may affect win likelihood, we can see that larger tree depth has a lesser effect on win likelihood than do shallow trees. This is an interesting fact, and might point the implication that the utility function is perhaps inverted. Instead of pushing all nodes out to the board edges, if a utility function gave more points to controlled pieces near the middle of the board, more jumps would exist.

A defensive position is assumed by moving controlled pieces to the edge of the board. It is a strategy of waiting for the opponent to run out of moves. However, the results suggest that maybe an offensive approach is more suited to this game. Where, by taking control of the center of the board, you force your opponent to hug the edges thus removing the likelihood that piece stays involved in the active gameplay.

**6 Conclusion**

In this paper we looked at how two different search algorithms play a game of Konane. Both of the algorithms apply the same utility function to play the game. This utility function must be a good heuristic or it runs the risk of turning a winning situation into a losing one.

For future work, changing the definition of the utility function to allow a more ‘offensive’ approach to the game may be an interesting addition. Then run the same tests as presented in this paper to see if any difference occurs. Also, a test suite without a randomization of the first two moves may be prudent, just to make sure the randomization has no effect on data.

References

David Poole, A. M. (n.d.). *Computational Intelligence A Logical Approach.* Retrieved 10 12, 2012, from University Of British Columbia: http://www.cs.ubc.ca/~poole/ci/slides/ch4/module04.pdf

Sheppard, John (2012). CSCI 446 Artificial Intelligence.

Stuart Russell, P. N. (2010). *Artificial Intelligence: A Modern Approach 3rd Edition.* Upper Saddle River, New Jersey: Pearson Education Inc.

Appendix A

Table -- Traditional minimax



Table -- alpha-beta pruning



Table -- Traditional minimax as Player 2



Table -- Traditional minimax as Player 1

