COMP1100 2016 Assignment 2

Name: Alexander Jones

University ID: U5956709

Tournament ID: 345

Tutor: Nathan Yong

Table of Contents

Section 1: Development of Board Evaluation Heuristics 2

Section 2: Development of Search Algorithm 5

Section 3: Time Complexity of Algorithms 11

Section 4: Problems and Challenges 13

Section 5: Comparison of Players Developed 14

Section 6: Programming Language Features Employed 15

Section 7: Further Improvements 16

Section 8: Conclusion 17

Section 1: Development of Board Evaluation Heuristics

The board evaluation function heuristic is a way of giving a value to a board. Boards that maximise my chance of winning and my opponent's chance of losing are evaluated favourably. When developing the heuristic, I went to the game of chess to get some ideas.

A simple chess heuristic is called Material, it is `value_of_my_pieces - value_of_op_pieces`. In Kalaha, a similar heuristic for Kalaha is bank difference:

```
evaluate_board = my_bank - op_bank -- heuristic 1
```

This is a simple and effective heuristic. It gives a higher value to boards that maximise the number of pebbles in my_bank and minimise the number of pebbles in op_bank. I used bank difference as the base and tried to improve upon it by considering things such as:

- How close I am to winning
- How close the opponent is to winning
- Whether it is my turn or opponent's turn
- Pond configurations

Improving upon bank difference is not an easy task. The first AB pruning minimax bot I uploaded used bank difference and it was able to get into the top 10 after 300 games.

The first challenge I faced while developing new heuristics was testing. Testing locally against Player_B takes about 4 minutes on average and although the results are consistent for the most part, they can vary. Testing on the tournament server takes much longer but the results are more deterministic.

The first heuristic I developed to improve upon 1 is called 2:

```
evaluate_board = (sum my_ponds) + my_bank - opponent_bank -- heuristic 2
```

Heuristic 2 aims to stack my ponds with pebbles, which is beneficial in the late game but not in the early game. It also increases the chance of a large capture for better opponents with more lookahead. Testing this heuristic locally produced varied results but on the tournament it was significantly worse than 1. The reason for this is that it is a situational heuristic, it wins against specific bots and losing to some of the average ones so I abandoned this heuristic.

One issue with the bank difference heuristic is that it doesn't give any value to how close me or the opponent is to winning. Therefore, I developed the next heuristic, logically called 3, which evaluates boards that allow me to win favourably and boards that allow my opponent to win unfavourably.

```
evaluate_board -- heuristic 3
  | (my_bank > 36) = my_bank - op_bank + 72
  | (op_bank > 36) = my_bank - op_bank - 72
  | otherwise      = my_bank - op_bank
```

Heuristic 3 lost to 1 locally and didn't perform well on the tournament server, losing to some of the lower ranked players. This was surprising as I thought it is a definite improvement upon 1. In spite of losing to bank difference, I didn't abandon this heuristic because I like the concept behind it.

The next heuristic, called 4, is similar to the 3 but slightly favours boards that give me a turn.

```
evaluate_board -- heuristic 4
  | (my_bank > 36)         = my_bank - op_bank + 72
  | (op_bank > 36)         = my_bank - op_bank - 72
  | (turn board == player) = my_bank - op_bank +  2
  | otherwise              = my_bank - op_bank
```

Heuristic 4 gave mixed results but overall it was worse than 1. The reason for this is that it is not always advantageous to favour boards in which it is my turn, especially when it is not strictly a repeated turn.

The next and possibly the final heuristic that I will develop is 5. It considers pond configurations as well. Certain pond configurations are more desirable than others. For example, ponds that give me another turn are desirable, especially the 6th pond because it gives me another turn without altering the other 5 ponds. Having 13 pebbles in a pond is also desirable because it guarantees gaining at least two points in my bank but it ruins my pond structure and gives pebbles to my opponent. However, there is a chance that it achieves a large capture.

This heuristic was able to beat all the above locally but it performed poorly on the tournament server, losing to average players. I played around with the numbers but that didn't make a difference so I abandoned it.

```
evaluate_board -- heuristic 5
  | (my_bank > 36)         = my_bank - op_bank + 72
  | (op_bank > 36)         = my_bank - op_bank - 72
  | (turn board == player) = my_bank - op_bank +  2
  | otherwise              = my_side - op_side
    where
      my_side = ((evaluate_ponds my_ponds) + my_bank)
      op_side = ((evaluate_ponds op_ponds) + op_bank)
      evaluate_ponds :: Ponds -> Int
      evaluate_ponds ponds = case ponds of
        [_,_,_,_,_,1]  -> 10
        [_,_,_,_,2,_]  ->  7
        [_,_,_,3,_,_]  ->  5
        [_,_,4,_,_,_]  ->  3
        [_,5,_,_,_,_]  ->  3
        [6,_,_,_,_,_]  ->  3
        [_,_,_,_,_,13] ->  3
        [_,_,_,_,13,_] ->  3
        [_,_,_,13,_,_] ->  3
        [_,_,13,_,_,_] ->  3
        [_,13,_,_,_,_] ->  3
        [13,_,_,_,_,_] ->  3
        _              ->  0
```

The following heuristic is a combination of the above but was the only one that beat player 30 twice. It attempts to maximise the number of turns I get and minimise the number of turns my opponent gets.

```
evaluate_board – heuristic 6
  | (my_bank > 36)         = my_bank - op_bank + 36
  | (op_bank > 36)         = my_bank - op_bank - 36
  | (turn board == player) = my_bank - op_bank +  2
  | (turn board /= player) = my_bank - op_bank -  2
  | otherwise              = my_bank - op_bank
```

The heuristic that I will submit for this assignment is heuristic 6, because outperformed all the heuristics above it. The only heuristic that can compete with it is 1 but 1 performed worse on the tournament server.

Another challenge is reasoning about the performance of a heuristic prior to testing is another challenge. Developing a heuristic requires experience and knowledge of the game of Kalaha, which comes from playing the game. Playing Kalaha is not relevant to Haskell programming so I decided to focus on achieving maximum search depth through polishing my search algorithm and by developing search heuristics like move ordering rather than trying to develop more board evaluation heuristics, which only marginally increase win percentage, or even decrease it as a more complex heuristic can fall into traps and lose to lower ranked players.

Section 2: Development of Search Algorithm

The final iteration of my search methodology uses the minimax algorithm with AB pruning and an implicit tree data structure.

The first thing I did once I cloned the assignment repository was importing System.Random and generating a random number between 1 and 6. However, this resulted in a compilation error and a lot of headaches until I figured out that the keyword system was banned.

Stage 1: Hard Coding

The first functioning program that I made is a very simple bot that always chooses the right-most legal move:

```
select_move board _ = head $ filter (legal_move board) [6,5,4,3,2,1]
```

The reason why I decided to use the complete list of moves instead of a list comprehension (i.e. `[6,5..1]`) is because having to generate the list is less efficient that having a readily available list. I know that these small optimisations won't necessarily make my bot better but I was hoping that they add up and give me one extra lookahead.

After this, I implemented a hard coded bot in an attempt to gain a higher ranking that player 90. The hard coded bot used about 50 guards of instructions and successfully gained a higher ranking than player 90. This does not use any lookahead. It only considers the current game state and tries to figure out a good legal move. For example, legal moves that give me a free turn, legal moves that can potentially capture opponent pieces and ponds with 13 pebbles. Although this bot is very basic, I believe that it was an important stage of development because I based my move ordering search heuristic on this bot.

Stage 2: Minimax without Alpha Beta Pruning

The initial implementation of the minimax algorithm does not attempt to reduce the size of the searchable game tree in any way. It goes through possible future boards and chooses the one with the highest value if it is at a maximiser node and the one with the lowest value if it is at a minimiser node until it runs out of the time. On average, my minimax program achieved about 4 lookaheads, which is very little compare to AB pruning which achieves an average lookahead of 15.

I uploaded this bot close to the second deadline and unfortunately it had a fatal bug as I assumed that I am always Player_A. This caused me to drop in rank to around 200 but luckily I was barely above player 90 by the end of the second deadline.

My minimax algorithm is based on a post Chris CL made on Piazza, which I am very thankful for. My minimax algorithm does not use an explicit game tree data structure. It basically performs operations on lists of possible boards at each level and the game tree is recursively implied.

The base case for the algorithm occurs when we reach depth = 0, i.e. the leaves of the game tree or when the game is finished. If we are in either of these situations, we call the board evaluation function to give the possible boards values, then we go through all the possible boards and pick the one that has the maximum value if we are at a maximiser node (player is me) or the one with the lowest value if we are at a minimiser node (player is opponent).

```
minimax :: Players -> Lookahead -> Board -> Score
minimax me depth board
| (turn board == Finished) || depth == 0 = heuristic
| player == me && depth > 0 = maximum possible_scores -- max node
| player /= me && depth > 0 = minimum possible_scores -- min node
| otherwise = error "Error"
where
possible_scores = map (evaluate_move player (depth - 1)) possible_boards
possible_boards = map (pick_n_distribute board) moves
```

Stage 3: Minimax with Alpha Beta Pruning

The minimax algorithm without alpha beta pruning considers every possible future board as it traverses the game tree. This means that it considers future board states that would never be selected because a better board has already been found. AB pruning attempts to reduce the size of the game tree. It operates by saving the best values that the parent node currently has and passes it down to its children. If a better value is found in a child node, then the current best value is updated with the new value. If a child node has a worse value than the one that the parent currently has, then that child node is cut out of the game tree, this is called the pruning condition. If the pruning condition is not reaches and there are no more children to consider, them the current best value is taken by the parent. This means that useless parts of the game tree are not searched, thus reducing the size of the game tree and potentially leading to deeper search.

The effectiveness of AB pruning depends on the order of the children nodes. If the children are visited from lowest to highest value for a maximiser node and from highest to lowest for a minimiser node, then pruning will not occur, thus producing the same results minimax without AB pruning. In general, we want to visit the node with the best value for the parent first, this way pruning occurs earlier.

The most important resource that I used to learn AB pruning is the following YouTube video: https://www.youtube.com/watch?v=J1GoI5WHBto

I also used the following 3 websites:

https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm

http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html

https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

In AB pruning, alpha represents the current best score that the maximising node has. It is initiated to the worst possible score that the maximising player can receive, which is -infinity. Best represents the current best score the minimising node has. It is initiated to the worst possible score that the minimising player can receive, which is +infinity.

There are basically 4 possible cases to consider in an AB pruning function:

1. There are no more children nodes to consider
2. Pruning Condition
    a. For maximiser parent node, it occurs when new_alpha >= current_beta
    b. For minimiser parent node, it occurs when new_beta <= current_alpha
3. A better value than the current value is found
    a. For maximiser parent node, new_alpha > current_alpha
    b. For minimiser parent node, new_beta < current_beta
4. None of the above cases occur.

What the algorithm needs to do in each case:

Case 1: Base Case

a. If we are at a maximiser parent node and there are no more boards to consider, then we just take the current best value, which will be current_alpha.
b. If we are at a minimiser parent node and there are no more boards to consider, then we just take the current best value, which will be current_beta.

Case 2: Pruning Condition

a. If new_alpha >= current_beta, then we need to break the search and not traverse down that child node because the parent node would never consider choosing that child node. In this case we just return the new_alpha value.
b. If new_beta <= current_alpha, then we need to break the search and not traverse down that child node because the parent node would never consider choosing that child node. In this case we just return the new_beta value.

Case 3: A better value than the current value is found

a. If new_alpha > current_alpha for a maximiser parent node, it means a better value for alpha is found. If this happens, then we take this new value of alpha and pass it up to the parent node and move on to the next child node. We recursively do this until either the base case or pruning condition is met.
b. If new_beta < current_beta for a minimiser parent node, it means a better value for beta is found. If this happens, then we take this new value of beta and pass it up to the parent node and move on to the next child node. We recursively do this until either the base case or pruning condition is met.

Case 4: If none of the above cases occur, then we need to move on to the next child node (next possible board) for the current parent node without updating the values for alpha and beta, i.e. passing up current_alpha and current_beta to the parent node.

Putting this all together results in the following two functions:

```
alpha_beta_max current_alpha current_beta moves' = case moves' of
  [] -> current_alpha
  (m : ms)
    | new_alpha >= current_beta -> new_alpha
    | max new_alpha current_alpha == new_alpha -> alpha_beta_max new_alpha current_beta ms
    | otherwise -> alpha_beta_max current_alpha current_beta ms
    where
      new_alpha = minimax player (depth - 1) (pick_n_distribute board m) current_alpha
current_beta

alpha_beta_min current_alpha current_beta moves' = case moves' of
  [] -> current_beta
  (m : ms)
    | new_beta <= current_alpha -> new_beta
    | min new_beta current_beta == new_beta -> alpha_beta_min current_alpha new_beta ms
    | otherwise -> alpha_beta_min current_alpha current_beta ms
    where
      new_beta = evaluate_move player (depth - 1) (pick_n_distribute board m) current_alpha
current_beta
```

Stage 4: Beyond Alpha Beta Pruning

I made post on Piazza asking what I can do beyond alpha beta pruning. Dr. Uwe Zimmer suggested 3 things:

- Visit https://chessprogramming.wikispaces.com/
- Implement search heuristics like move ordering
- Not to waste time with fiddling around with board evaluation heuristics.

Stage 4a: Move Ordering

Pessimal ordering of nodes (worst to best) removes the benefit of AB pruning, resulting in a time complexity of $O(b^d)$, Optimal move ordering (best to worst) imrpved the time complexity to $O(b^{d/2})$. Achieving optimal move ordering in Kalaha is a very difficult task. The time complexity for my move ordering is $\sim O(b^{3d/4})$ or slightly better.

Initially, when I passed the moves to my AB functions I simply used:

```
filter (legal_move board) [6,5,4,3,2,1]
```

This is not the worst case move ordering, in fact, it is very good and can achieve a very high ranking in the tournament.

I then moved on to implement a better way to order the list of available moves and came up with the following:

```
moves
  | (no_of_pebbles board 6 == 1)  = [6]
  | (no_of_pebbles board 5 == 2)  = 5 : filter (legal_move board) [6,4,3,2,1]
  | (no_of_pebbles board 4 == 3)  = 4 : filter (legal_move board) [6,5,3,2,1]
  | (no_of_pebbles board 3 == 4)  = 3 : filter (legal_move board) [6,5,4,2,1]
  | (no_of_pebbles board 2 == 5)  = 2 : filter (legal_move board) [6,5,4,3,1]
  | (no_of_pebbles board 1 == 6)  = 1 : filter (legal_move board) [6,5,4,3,2]
  | otherwise = filter (legal_move board) [6,5,4,3,2,1]
```

This resulted in a significant improvement over the initial implementation. It also resulted in unexpected and significant improvements on the tournament servers, allowing my bot 345 to be the second highest ranked student bot after 294 and in the top 10 overall.

In my move ordering, I decided to always choose 6 if it has 1 pebble. I believe that there is no situation in which this is not an advantageous move because it does not alter the remaining 5 ponds so if there was a good move in the other 5 ponds, like a huge capture, I can always execute that move in my second turn.

I tried many things to improve upon this move ordering, for example, positioning ponds with 13 pebbles at the front of the list, i.e. (no_of_pebbles board 1 == 13) = 1 : filter (legal_move board) [6,5,4,3,2]. I also considered using mod for ponds that give me a free turn, i.e. (no_of_pebbles board 6 % 13 == 1) = 6 : filter (legal_move board) [5,4,3,2,1]. Finally, I considered move that possible allow a capture, i.e. (no_of_pebbles board 6 == 0) && (no_of_pebbles board 5 == 1) = 5 : filter (legal_move board) [4,3,2,1]. However, all of these made my bot slower and lost to the initial method I had above.

A few students were interested in using my move ordering method and told me that they will reference me in their reports. I wouldn't claim that I invented this method as it is very simple but I also don't want to get into any trouble in case they don't reference me.

Stage 4b: Futility Pruning

Reference: https://chessprogramming.wikispaces.com/Futility+pruning

I visited the chess programming web site and was interested in futility pruning because the idea seemed reasonable to me and implementing it only required adding a few lines of code to my existing program.

```
| (2 * (36 - op_bank)) < alpha = alpha
| (2 * (op_bank - 36)) > beta  = beta
```

Futility pruning discards the moves that have no potential of raising alpha or no potential of decreasing beta, thus potentially achieving deeper search into the game tree.

This was a failure as it lost to my simple alpha beta program with no move ordering. I am unsure if my implementation is wrong or if futility pruning is not as effective as alpha beta pruning alone.

Beyond move ordering and futility pruning, I looked into negamax and negascout but after doing some research I found out that these two algorithms perform poorly against alpha beta pruning so I abandoned them and instead I focused on polishing and testing what I already had.

Section 3: Time Complexity of Algorithms

Time complexity is an abstract measure of how long it takes to execute the program of an algorithm.

Time complexity for minimax algorithm without AB pruning:

Reference: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Let branching factor be represented by b and depth by d.

The branching factor is the number of children at each node. For the game of Kalaha, the maximum branching factor is 6, which is the number of possible ponds we can choose from to generate future possible boards.

The game tree for Kalaha has one node at its zeroth level, b nodes at its first level, b^2 nodes at its second level, b^3 nodes at its third level and so on. At max depth it has b^d nodes.

For branching factor = 6, we have the following tree.

Each * represents a node.

```
                                6^0 * level 0

                            + 6^1 * * * * * * level 1

                    + 6^2 ****** ****** ****** ****** ****** ****** level 2

                                        .

                                        .

                                        .

                + 6^d ****** ****** ****** ... ****** ****** ****** level d
```

Therefore, the number of nodes is equal to

$$\sum_{n=0}^{d} b^n$$

To find the time complexity, we need to find out how long it takes the algorithm to generate each level of the tree. For Kalaha, we generate new nodes using pick_n_distribute. Assume that it takes constant time t to generate a node using pick_n_distribute. At level 0 we have 1 node, so it takes time t, at level 1 we have 6 nodes so it takes time 6t and so on. At level d it takes time 6t^d.

$$O\left(t\sum_{n=0}^{d} b^n\right) = O\left(tb^d\right) = O\left(b^d\right) = O(6^d)$$

However, this is only an estimate as the branching factor for Kalaha is not constant, it deceases overtime as more ponds become empty. Perhaps a more accurate estimates is:

$$O(4^d)$$

Time complexity for minimax algorithm with AB pruning:

Reference: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

The maximum number of leaf node positions evaluated is (as shown above):

$$O(b^d)$$

This occurs when nodes are ordered from worst to best. In the worst case, no pruning occurs, therefore the full tree is examined (or until the program runs out of time allowed to make a move).

If the move ordering for the nodes is from best to worst, then pruning can happen earlier at each parent node. In the best case, each node will examine $2(6) - 1$ child nodes until pruning occurs. This means that in the best case minimax with alpha beta pruning evaluates the following number of nodes:

$$O\left(b^{\frac{d}{2}}\right)$$

However, the nodes for my Kalaha program are not ordered in an optimal way. I use move ordering but for the most part the nodes are ordered randomly.

Therefore, the time complexity is for my program is:

$$O\left(b^{\frac{3d}{4}}\right) = O\left(6^{\frac{3d}{4}}\right) \approx O\left(4^{\frac{3d}{4}}\right)$$

I am assuming that the branching factor is around 4 on average.

Section 4: Problems and Challenges

By far the most challenging part about this assignment was developing board evaluations techniques. Developing a Kalaha heuristic in not conceptually challenging or challenging in terms of programming, it is just time consuming and in many ways useless. The problems:

1. Testing
    a. Testing locally takes a very long time per game and what is worse is that results are not deterministic because lookahead values for alpha beta pruning bots fluctuate based on CPU load. This means that in order to get reliable conclusions when testing different heuristics, multiple games need to be run. One way to deal with this is to write a script that automatically runs Kalaha matches for players with different search algorithms and different board evaluation heuristic. There is still a problem with this. I will be competing players that I developed against each other so I have no idea how the winning player will perform in the tournament.
    b. Testing on the tournament server was fine in the first few weeks but became useless in the last week as I stopped getting matches after playing 10 games.
2. Developing a heuristic does not really improve my Haskell programming skill and knowledge. It simply requires playing Kalaha and fiddling around with numbers.
3. Reasoning about the performance of a board evaluation heuristic prior to testing was never accurate. For example, I thought heuristic 3 was surely going to outperform heuristic 1 because it gives higher values to winning states. However, it didn't.
4. There is very little information on Kalaha heuristics online, especially when compared to the game of chess, where there are countless sources that explain a vast number of board evaluation heuristics in detail and provide thorough testing.

The next challenge is that AB pruning is conceptually difficult to grasp and difficult to implement in a functional programming language. Most information available on AB pruning online uses imperative pseudocode with variables and for-loops with loop breaks, which makes updating the values of alpha and beta very simple and whenever a pruning condition is reaches, the a loop break can be used. A common issue that I noticed many students were struggling with is not being able to update alpha and beta properly, which is vital to implementing AB pruning.

The next challenge is that I had difficulty figuring out how to extract the move with the highest score at select_move. Eventually, I found a way to do it using list comprehension to generate a list of tuples of score-move pairs then extracting the move with the highest score.

```
snd $ maximum $ [((evaluate_move (turn board) depth (pick_n_distribute board x)), x) | x <- moves]
```

Later I moved away from list comprehension and implemented a recursive function:

```
score_move_pairs alpha beta moves' = case moves' of
  []       -> []
  (m : ms) -> (evaluate_move (turn board) depth (pick_n_distribute board m) alpha beta, m) : score_move_pairs alpha beta ms (*)
```

This function was developed in an attempt to update the values of alpha and beta recursively when moving to the next node in the line labelled (*) as suggested by Nathan Yong. After many failures at achieving that, I gave up.

Section 5: Comparison of Players Developed

All the players and heuristics I developed are in a folder called Development_Stages.

Testing of players was mainly performed locally but before any decisions were made, players were tested on the tournament server.

Overall, alpha_beta_best performed the best out of all the players. The only other player that could compete with it is alpha_beta_better, which is the only both that beat player 30 twice, but alpha_beta_better performed poorly on the tournament and was not reliable enough to be submitted.

I decided to go with alpha_beta_best as it is a safe bet. It loses to alpha_beta_better when going second but performs very well on the tournament server.

I thought about writing a script that tests players against each other automatically but this will give me little indication of how well the winning player will perform on the tournament.

Section 6: Programming Language Features Employed

The main feature employed throughout my program is recursion, both primitive and non-primitive recursion.

I believe that implementing AB pruning in Haskell using recursive function calls is just as simple as implementing it in an imperative language, it just requires a different way of thinking. To illustrate this, take a look at the following two code snippets:

```
// Imperative AB pruning pseudocode for maximiser node from wikipedia
function alphabeta(node, depth, α, β, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic value of node
  if maximizingPlayer
    v := -∞
    for each child of node
      v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
      α := max(α, v)
      if β ≤ α
        break (* β cut-off *)
    return v

-- Haskell alpha beta function for a maximiser node
evaluate_move :: Players -> Lookahead -> Board -> Alpha -> Beta -> Score
evaluate_move player depth board alpha beta
  | (depth == 0) || (turn board == Finished) = evaluate_board
  | (turn board == player) && (depth > 0)    = alpha_beta_max alpha beta (moves board)
  where
    alpha_beta_max :: Alpha -> Beta -> [Pond_Ix] -> Alpha
    alpha_beta_max current_alpha current_beta moves' = case moves' of
      [] -> current_alpha
      (m : ms)
        | new_alpha >= current_beta -> new_alpha
        | new_alpha > current_alpha -> alpha_beta_max new_alpha current_beta ms
        | otherwise -> alpha_beta_max current_alpha current_beta ms
        where
          new_alpha = evaluate_move player (depth - 1) (pick_n_distribute board m)
current_alpha current_beta
```

The only difference between these two functions is that the Haskell function doesn't have a for loop, variables or loop breaks. However, the alternative to all of these features is recursive function calls.

Another feature of Haskell that I used throughout my code is the higher order function filter, to get a list of legal moves. This allowed me to completely avoid list comprehensions in my code. I don't know for sure if this makes a difference in terms of performance or not but I am hoping that avoiding list comprehension throughout my code adds up and make a difference in performance enough to gain 1 extra lookahead above the competition.

Another feature of Haskell that I use is where clauses, which reduce the number of input parameters to functions and make my code more readable.

The final feature I want to talk about is type declarations to make function definitions more readable.

Section 7: Further Improvements

I believe that my bot is far from perfect and that there is a vast number of improvements that can be implemented to make it a more powerful Kalaha AI.

Creating a better search heuristic through achieving close to optimal move ordering would results in the most significant improvement as it would allow alpha beta pruning to be much more effective. Coming up with a general function that produces the optimum order of children at each node is an incredibly difficult task but if implemented successfully this would produce and a very powerful Kalaha AI. One idea I had was to use the board evaluation function to evaluate each move then sort the moves from best to worst:

```
-- moves ordered from best to worst based on the score given by the heuristic

moves_max board player = snd $ unzip $ (sortBy (flip compare)) $ [(evaluate_board
(pick_n_distribute board x) player, x) | x <- filter (legal_move board) [6,5,4,3,2,1]]

moves_min board player = snd $ unzip $ sort $ [(evaluate_board (pick_n_distribute board x)
player, x) | x <- filter (legal_move board) [6,5,4,3,2,1]]
```

I had very high hopes for this, but of course it failed miserable as it achieved a maximum lookahead of 8, compared to 33 by my original program. The reason for this is that it is computationally expensive to generate the list of moves. For example, unzip has a time complexity of $O(n)$.

I thought about and tried to make a lookup table for move ordering but the possibilities are incomprehensibly large and would probably take more memory than the tournament allows.

Another thing I can improve upon in my program is updating alpha and beta at top level so that when I move to the next node I have the current best values for alpha and beta, and not -infinity and +infinity.

Another things is fiddling around with the numbers in the board evaluation heuristic until the magic combination is found, however, that is not very exciting.

Beyond move alpha beta pruning, move ordering and board evaluation heuristics there is little that can be successfully done within the constraints of the assignment.

Section 8: Conclusion

Developing a powerful Kalaha player beyond alpha beta pruning isn't about coming up with more complex functions, it is about striking the perfect balance between speed and getting the best board values and move orders. This is the reason why a simple heuristic like bank difference performed so well.

There are many questions left unanswered:

- Is it worth the time to develop a more complex heuristic? Will bank difference beat it?
- Is there such a thing as the best heuristic?
- Is there a benefit in knowing the opponent's heuristic?
- Is futility pruning effective if implemented properly?
- Is there a way to come up with a none computationally intensive function that produces close to optimum move ordering?

I have been interested in the field of artificial intelligence long before I knew what it was all about, I just instinctively knew it is something I would be interested in. This assignment was my first attempt at making an AI. It was very satisfying to see my bot play Kalaha 100x better than I ever will, simply by generating possible future boards and choosing ones that maximise bank difference. This assignment further confirmed my interest in the field of AI.

This assignment was an enjoyable way to improve my Haskell programming skills. I improved my understanding of tree data structures and the use of higher order functions over the course of this assignment while also improving my understanding of recursion and lists.