CS 480: Assignment 07 Report
Zachary Ferguson

# Mancala

## Design and Implementation:

My implementation for Min-Max with alpha-beta pruning and computer make-move follow those of the slides. I did however condense alpha-beta down to maximize the readability of the code. The main feature of my state space search is the heuristic to evaluate the board.

I based my heuristic off one describe here, http://blog.hackerrank.com/mancala/. The Heuristic can be summarized in the following formula:

$$h(s) = (\text{Number of Stones on Max's side} + \text{Number of stones in Max's Mancala}) - (\text{Number of Stones on Min's side} + \text{Number of stones in Min's Mancala})$$

$$h(s) \in [-\text{Number of Stones}, \text{Number of Stones}]$$

This is a very simple and easy to compute heuristic (see **Performance** for a further analysis). The basic idea of this heuristic is that a player can at max gain the number of stones on their side plus the stones they already control. This in turn encourages the computer to pick moves that will keep the most number of stones on their side while also encouraging them to put stones in the mancala.

As indicated above this heuristic is in the range of the number of stones available to claim. But the board evaluation function should return a value in the range [min-wins, max-wins]. To convert to this range I implement a simple mapping function. This function works by normalizing the input range and scaling to the output range. In fact this function is modeled after the map() function in the standard Arduino library, https://www.arduino.cc/en/Reference/Map.

## Performance:

In order to maximize the performance, I use a couple methods. First, I simply declaim the optimize to be most important. Second, I define the helper functions to be in-line. Lastly, I use a simple heuristic that takes minimal time to compute.

By using the optimize setting in Lisp, I attempt to maximize the performance of my implementation. In practice, I saw little speedup, however. This probably implies SBCL was already optimizing the performance, so I get little speedup.

For both the sum-player-owned-pits and map-range-to-range functions I declare them as in-line. I do this to reduce the stack space overhead and hopefully avoid any unnecessary computation for these simple tasks.

In terms of performance, the most important thing I do is to use a simple, but effective, heuristic. As outlined above, the heuristic I use is a simple summation and difference of four values. This allows for quick evaluation of a given state, which implies the Min-Max search is able to search more in depth.

A possible improvement on my code would be to memoize the states of the board. Because the states repeat themselves, memoizing would eliminate the need the need search certain board states. I chose not to implement this however because of time constraints and a general lack of knowledge for Lisp hash-set data structures.

**Evaluation:**

To properly evaluate the performance of my state space search I ran the following experiments. For each of the experiments, I run a tournament with two of my players. I time this tournament with Lisps time function. The only parameter tweaked is the max-depth for each player.

To start I test with a max depth of 10 for both players. The tournament takes a total of 11.71 seconds to run and the final sum score is zero. If I repeat this experiment, I get the same results. In general the system is stable with two of my players because they will always choose the same moves depending on the depth they are allowed to search.

To test a simple example I set max-depth to five for both players. This tournament only takes 0.219 seconds. As a stress test, I set the max-depth to 16 for both players. This takes a longer amount of time due to the exponential search space increase. In sum, the tournament takes 522.85 seconds to run. The searches take a decreasing amount of time as the game plays out because the search find end game states prior to reach a max depth.

To check my search is functioning properly, I pit two of my players against each other with varying max depths. I first give the player one a depth of five and player two a max depth of 10. Because of these depths player two always wins. Similarly, if I swap max depths, player 1 always wins. In general, the player with the larger max depth always wins. This verifies that my search works correctly.