Marco Munizaga
UFID: 91990399

Final Project Report

1) The game that was implemented was a game where different colored circles were placed on a board. Picking a circle would result in picking the cluster of neighboring circles (either up down left or right) . If the cluster was bigger than two circles, the cluster would be removed and upper circles in the original board will fall down and take their place. If there are no more rows in a column, that column is removed so the pieces shift left.

2) The game was implemented through a lot of functional composition. Which allowed a lot of the pieces to be really modular, and easy to work with. The whole game is played (both human and ai) through immutable state, which greatly reduced the amount of moving parts of the underlying system. The board is represented by a hashmap with keys corresponding to columns. The columns were also hashmaps. Because the map was only really generated once, this provided an efficient means to read and manipulate the board in O(1) time.

3) The most important functions where find-cluster, find-all-clusters for the game itself (no ai). Find cluster would be given a graph (aka the board) and a position, and it would return a vector of all the positions of items in that clusters (e.g. [ [0 0] [ 0 1]  [1 1] [1 0] ] for a square ) The find-all-clusters would iterate over every point in the graph (without repeating a point if there is a cluster for it already) and return all of the clusters as well as their pertinent graphs and their score values. This was (for the non ai part) important to recognize when the user has finished playing the game. The lesser important functions where just to manipulate the graph, and to check for valid input. The functions included

**Function Overview**

execute-moves : given a vector of moves, and a graph execute the moves on the graph and print the intermediary graphs (just to see what the ai was doing really)

filter-moves: get rid of moves that have no effect in a vector of moves, returns a vector of moves where every move changes the graph

bounded? : checks to make sure the position is within the range of the graph

choose-move : given a graph and a cluster, remove the cluster from the graph, and return the graph

goto-neighbor : given a position and a direction (i.e. :up :down :left :right) go to that neighbor node, or return nil if it is out of bounds

read-neighbor : same as goto-neight but returns a value

shift-graph-changes : function to move nodes down the board

shift-graph-cols : function to remove empty cols

AI functions...

**Genetic Algorithm**

initialize : create a strand (vector of moves) from random points on the board, or a specified length

genereate strands : returns a population of strands, specified population count, and strand length

fitness : calculates the score a strand would get playing the game

roulette-wheel-selection : picks a strand at random with a weighting towards fitter strands (aka fitness proportionate selection http://en.wikipedia.org/wiki/Fitness_proportionate_selection)

crossover : given two strands randomly mix them to generate a new strand

simple-mutate : x % of the time it will randomly change an item in a strand

breed-strands : given some strands produce new strands through crossover and mutations

circle-of-life : combines the above functions.  creates an initial population of strands and test each strand's fitness. picks the fittest to generate the next population.

**Greedy Algorithm**

greedy-depth-search :  picks the highest valued cluster from find-all-clusters, every time.

Exhaustive
exhaustive-search : recursively finds every value in the tree structure for all possible states (bad at complex graphs, take too long)

**Visual window**
Used quil library to generate window
build-visual-aid : accepts an atom (piece of state) to bind to print.

Flowchart and calling hierarchy provided on a separate page.

I**mplemented intelligence:**

Genetic Algorithm : Really cool stuff! Just picking random points and evolving them to reach a superior point value is really cool. It has no idea about the structure of the graph, just a simple this works, this doesn't, and it's fast!

Greedy Depth First : Simple nothing really new here
Exhaustive Search : Basic search (back when I thought if I made it really fast this might stand a chance, nope says the 25x25 board)

**Future design decisions**

This was the third time this program was rewritten from scratch. As of now I'm pretty happy with it. Most function are modular and can easily be replaced. The overall design is very compositional which I'm very happy about. There is no state involved, and everything is immutable (double win!). The only real change would be to better document the functions inline (clojure provides that functionality).

The problem with the first approach was that it stored the board as a vector inside a vector (array inside an array) which is fine, unless some functions reads a vector and returns a list, now that O(1) lookup time you were enjoying becomes O(n) lookup time. So you had to be really careful with functions that would return list or heaven forbid lazy sequences. And I like lazy sequences so I knew I had to rewrite it.

The second approach was just my imperative side showing. I used a ton of state. Every node on the board was a piece of state with up down left right values that referenced other nodes. The biggest problem with that is that managing all those states can get really really confusing. To remove a node you have to update the neighbor nodes, then move the upper node down, which requires updating that node's neighbors. If you want to get rid of an empty column you have to relink all of the adjacent columns' nodes together. It was big bulky and troublesome.