CSC442: Intro to AI

Project 1 Report: Mancala

Name: Soubhik Ghosh, Rich Magnotti

NetId: sghosh13, rmagnott

September 15, 2019

Introduction

This project is about implementing an interactive 2 player game called Mancala and conducting various experiments to decide the robustness of our AI agent in Mancala. This project has been collaboratively executed by the following 2 members:

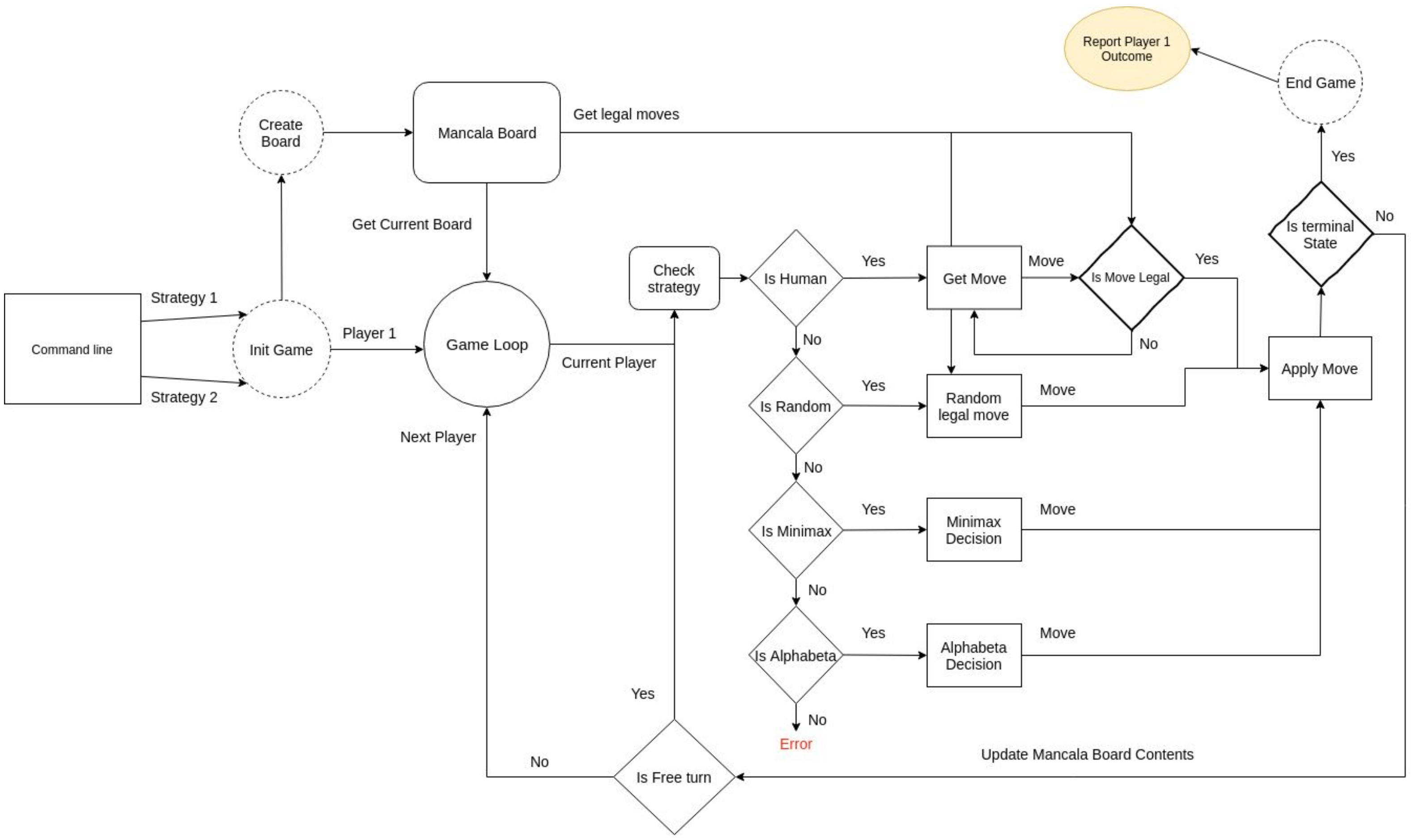
1. Soubhik Ghosh
2. Rich Magnotti

In this project we develop the US version of Mancala called Kalah and has the following rules:

- At the beginning of the game, four stones are placed in each house. This is the traditional method.
- Each player controls the six pits and their stones on the player's side of the board. The player's score is the number of stones in the store to their right.
- Players take turns to land their stones. On a turn, the player removes all stones from one of the pits under their control. Moving counter-clockwise, the player drops one stone in each pit in turn, including the player's own store but not their opponent's.
- If the last stone lands in an empty pit owned by the player, and the opposite pit contains stones, both the last stone and the opposite stones are captured and placed into the player's store.
- If the last stone lands in the player's store, the player gets an additional move. There is no limit on the number of moves a player can make in their turn.
- When one player no longer has any stones in any of their pits, the game ends. The other player moves all remaining stones to their store, and the player with the most stones in their store wins.

Program design

The following is a bird's eye view of the various components of our mancala game program



We implemented the above flow chart completely in C++ on the vim editor. We chose C++ due to its high efficiency and fine grained control on a program's performance.

The various components and interactions in the above flow chart are detailed as follows:

- The program starts by fetching the strategies from the command line arguments and validating them. Validation is successful when there are exactly 2 strategies and both of them are from the following C++ style strings: "human", "random", "minimax", "alphabeta".

- The game starts running by initializing the mancala board and spawning a game loop that runs until the game terminates

- The game starts with the Mancala board where each pit has 4 stones and each store has zero stones.

- The first strategy goes first as player 1 and the game loop runs its first iteration on player 1 as the current player and future iterations alternate between the 2 strategies as the current player.

- In a game loop iteration the current player is evaluated by running its strategy, deciding a move and applying the move on the current board state.

- Before deciding the strategy, the set of legal actions is fetched from the current Mancala board state for use in the human and random strategies.

- If the strategy is:
    - **Human**, then the human is prompted to enter a move. ‘`Enter a move`’. If the user enters an invalid move, the program calls it out as ‘`Illegal move`’ and re-prompts the user to enter a move.
    - **Random**, then a move is randomly chosen from the set of legal moves in a uniform manner.
    - **Minimax**, then the program runs the minimax algorithm with MAX as the current player upto a certain depth for time constraints. Hence we are actually using heuristic minimax in this project
    - **Alphabeta**, then the program runs the same minimax algorithm but with alpha-beta pruning.
- Though we get significant speedup for alphabeta than its non-pruned counterpart, we still search upto a certain depth for time constraints.

- Once a valid move is returned by one of the above strategies, the game applies the move on the current board state with respect to the current player.

- If the updated board doesn’t have any stones remaining in one of the player’s side of houses/pits, the game terminates and reports the outcome (win/loss/draw) from the player 1’s perspective.

- If the updated board does not correspond to the terminal state (no empty side), the game checks if the last stone landed in the current player's store for a free turn.

- If there is a free turn, the current player repeats the evaluation described above, in the current game iteration. Otherwise, the game loop goes to the next player and performs the evaluation.

<u>Game playing I/O</u>

The program starts by showing the following output:

```
*************** WELCOME TO MANCALA *****************


P1: "human" VS P2: "alphabeta"


P1 turn...


Current board contents:


       6    5    4     3    2    1

    --------------------------------

 00 | 04 | 04 | 04 || 04 | 04 | 04 | P2

----------------------------------------

 P1 | 04 | 04 | 04 || 04 | 04 | 04 | 00

    --------------------------------

       1    2    3     4    5    6
```

Here we show the initial board contents before actually making any move. Next depending on the strategy we prompt the user to make a move or programmatically ‘construct’ the move which is then applied to show the board contents again. If the strategy is alphabeta or minimax we show the time elapsed in milliseconds after making the move

eg. `Elapsed time: 12.6912 ms`

If the current move awards a free turn we show the following line (assuming its still P1’s turn):

```
P1 gets another turn…
```

Else we display:

```
P2 turn…

Current board contents:

      6    5    4     3    2    1
    --------------------------------
 00 | 04 | 04 | 04 || 04 | 05 | 05 | P2
----------------------------------------
 P1 | 04 | 04 | 00 || 00 | 06 | 06 | 02
    --------------------------------
      1    2    3     4    5    6
```

and the process continues till the game terminates

Code and Problem Description

Our implementation is organized into 3 C++ files.

- `mancala.h`
- `search.h`
- `play.cpp`

`mancala.h`

- This file contains all the functions and states specific to playing mancala

- The file also defines the 2 players in an enum called `Turn`

- `Turn::PLAYER1` always owns the bottom side (P1) of the board and `Turn::PLAYER2` always owns the top side (P2) of the board as shown:

```
         6    5    4     3    2    1

     --------------------------------

 00 | 04 | 04 | 04 || 04 | 04 | 04 | P2

----------------------------------------

 P1 | 04 | 04 | 04 || 04 | 04 | 04 | 00

     --------------------------------

         1    2    3     4    5    6
```

- In the file we define a `namespace` called `mancala` which contains a class definition for representing and manipulating the state for a mancala board

- We define 2 main constants inside the namespace which hard codes the number of pits in the board (`PITS`) and the initial number of stones in each pit (`STONES_PER_PIT`)

- The other constants are just to store precomputed values to speed up computations.

- The `BoardState` class represents a mancala board state in the form a 1D array of size 14. Indices 0 to 5 are pits belonging to P1 and index 6 is P1's store. Indices 7 to 12 are pits belonging to P2 and index 13 is P2's store. (Initial State: [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0])

- The array's data type is defined as a `int8_t` (1 byte long) as the maximum number of stones in any pit can be just 48

- An object of this class is a state which is created by each node in a search tree

- The actions/moves of a player are the numbers 1,2,3,4,5,6 which are nothing but the pit numbers. Internally we reduce these to 0-index for ease of programming. Hence when a player chooses the move 3, this means remove all the stones from pit 2 of the player's side of the board (See board above) and distribute them in the counter-clockwise manner.

- The `actions` method of the class returns a C++ vector of integers which are the indices of pits having non-zero stones for the current player. This method corresponds to the textbook function:
  ACTIONS (s): Returns the set of legal moves in a state

- The `result` method takes in a valid move/action (0-index based) and applies the move to the board. This function updates the current board by redistributing the stones in the pits/stores by the rules of Mancala (See introduction). This method also returns a boolean which tells whether the applied move gave the current player any free turn or not as per the rule of Mancala. This method corresponds to the textbook function:
  RESULT (s, a): The transition model, which defines the result of a move.

- The `utility` method returns the difference between the current player's store count and the opponent's store count. This assumes that the stones in the non-empty side of the board have been claimed. This method corresponds to the textbook function:
  UTILITY (s, p)

- The `terminal_test` method returns a boolean on whether one of the player's side of the board is empty or not. This method corresponds to the textbook function:
  TERMINAL-TEST (s)

- The `cutoff_test` method is used for performing depth limited search by checking when a hard coded depth is reached.

- The `eval` method calculates the various heuristic functions applied at the cutoff (More on this later).

`search.h`

- This file contains the function definitions of minimax and alphabeta algorithms and their helper functions.

- All the functions are defined statically inside a generic `struct` called `Search`

- The entry point for minimax is the public method `minimax_decision`.

- This method takes the inputs `BoardState` object, current player and a depth, and returns an integer between 0 to 5 inclusive which is an optimal move as per the minimax algorithm.

- `minimax_decision` depends on 2 other helper methods `max_value` and `min_value` which have the same signature as `minimax_decision`.

- These 2 helpers are mutually recursive and stop building the search tree when either a terminal node or cutoff is reached. Hence we are using the heuristic version of minimax.

- Usually heuristic minimax is called with a specific depth which decrements at every step. Except that in our implementation the depth remains unchanged when there is a free turn. For example if MAX generates a child node which earns it a free turn, then instead of calling `min_value(d - 1)`, it recursively calls `max_value(d)`. See source code for more details.

- Similarly the entry point for alphabeta pruning is the public method `alphabeta_decision`

- It has its own helper functions `min_value` and `max_value` which are overloaded versions of the minimax helper functions because of the 2 extra parameters alpha and beta.

- They share the same implementation with heuristic minimax except the additional steps for pruning and bookkeeping alpha, beta values.

`play.cpp`

- This file contains the game loop and strategy evaluation.

- When we pass command lines to the program, they are verified by the function `get_strategy_types`. The verification process simply checks whether there are 2 arguments (other than the program name) and that they are valid player strategies. Failure to meet any of these requirements terminates the program.

- These strategies are then passed to the `run_game` function which runs the game loop and returns an outcome when the game terminates. The outcome is one of the 3 enum values 1 (Win), 2 (Loss) and 3 (Draw) from the player 1(strategy 1)'s perspective. The run_game function also creates the initial board state using the constructor of the `BoardState` class

- The game loop calls the evaluation function for player 1 and player 2 in each iteration.

- The evaluation function does the following (Consult flowchart in program design):

    - Gets a vector of legal moves from the current board state and for the current player.

    - Decides and executes the strategy for that player and gets a move.

    - The current player then executes the move and updates the current board state.

    - The board is then checked for a terminal state. If it is, then the evaluation function returns true.

    - Next the board is checked for a free turn in which case the evaluation process repeats for the current player or returns to the game loop for the next player.

- This file also contains a wrapper function `run_game_wrapper` which is used to call the `run_game` function from a python file.

The following 2 files are used to build the project and conduct tests

- `Makefile`
- `Mancala_tester.py`

`Makefile`

- This file compiles and links the C++ source files to create a binary file `play` and a shared object file `libplay.so`

- The most important compilation flag we found is the -Ofast which reduced the time of minimax with depth 8 from 40 seconds to 5 seconds

`Mancala_tester.py`

- This file loads the `libplay.so` file generated above and calls `run_game_wrapper` function from the python script.

- We do this to easily write and conduct tests in python. We also used the matplotlib library to plot our observations of win/loss/draw

Member Contributions

Soubhik Ghosh:

- Responsible for creating the design and implementation of minimax and alphabeta pruning in C++
- Responsible for implementing branching factor

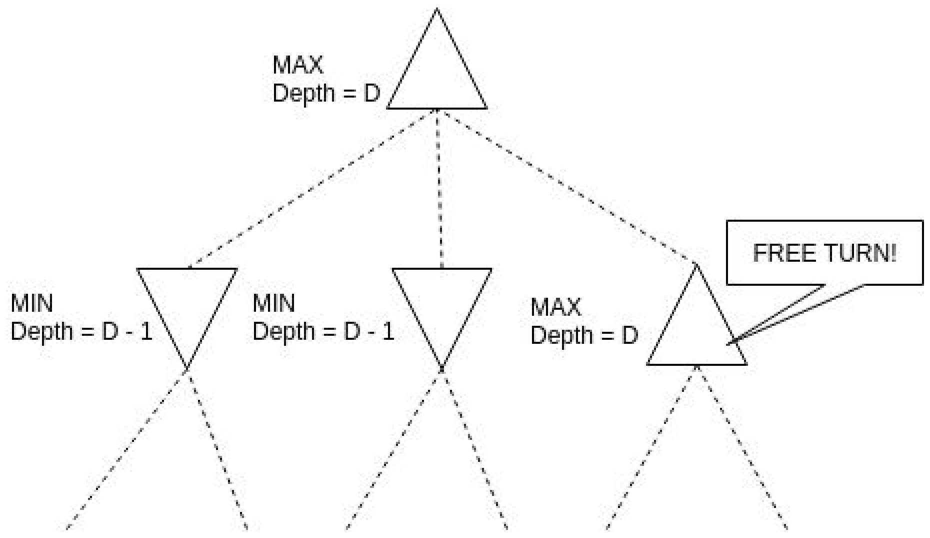
Richard Magnotti:

- Responsible for writing python test script and conducting experiments
- Responsible for finding heuristics to make minimax and alphabeta faster

Experiments

Overview

All of our experiments were done on versions of minimax and alphabeta pruning which uses a cutoff test and heuristic functions for making real-time decisions. But to still make intelligent moves and increase win rates, we tweaked the algorithms of minimax and alphabeta. In our modified versions, we decrement the cutoff depth as we go deeper into tree (towards the goal) just like heuristic minimax (and alphabeta) but skip it when there is a free turn. Free turn in the search tree can be seen if a child node of the parent MAX node remains a MAX node (due to the board state awarding a free turn) instead of a MIN node (See figure below). Same goes with the MIN node.



We did this to improve the winrate of the player getting free turns by allowing the player to search deeper into the tree. It was also more trivial than the other choice of constructing a heuristic function that considers this strategy. We still came up with some heuristic functions that were evaluated when the cutoff depth is reached as detailed below.

Based on our understanding of the game and observations we settled on 3 simple heuristics for the minimax evaluation function:

- H1: Difference between the stones in player's store and opponent's store
- H2: Difference between number of the player's and opponent's non empty pits
- H3: Number of stones on the player's side of pits where the stones towards the left side were preferred more than the stones towards the right side (near the player's store)

Our utility function returned the actual difference in scores, but our evaluation function used these heuristics.

The following experiments were done with the evaluation function: H1 + H2 (See `mancala.h`) and shallower depth values (for real time decisions) as we observed no improvement in decision making with H3. But it might be something to consider for deeper depth values where time is not a constraint and we want more accurate heuristic values.

Results, Testing, and Discussion

Our testing was done in batches of various sizes ranging from 5, 10, and 20. Empirically, 20 consecutive matches seemed a sufficient amount to make obvious any trends in the data. Furthermore, a notable constraint was the depth that each search algorithm should explore. A depth of 8 appeared to be a happy medium between *time per match* and *efficiency of the algorithm*. At this depth, the matches are able to finish in an amount of time that is not overly gratuitous (about 1-2 minutes per one match worst case when using minimax on our personal machines – and only about <30 seconds per one match on alphabeta!) while simultaneously being deep enough to achieve meaningful results. For the sake of completeness, we also tested various adversarial matchups with different cutoff depth. However the bulk of the testing we will be analyzing below was done with depth 8, and is indicated otherwise when relevant.

It is also worth noting that we performed the adversary vs human matchups in primarily only a 5-match batch per adversary for the sake of both time and not taking advantage of people's kindness in being willing to play.

Testing Methods:

I Minimax – 5 matches, 10 matches, 20 matches, 50 matches, 100 matches
- (a) Vs alphabeta
- (b) Vs minimax
- (c) Vs random
- (d) Vs human

II Alphabeta Pruning - 5 matches, 10 matches, 20 matches, 50 matches, 100 matches
- (a) Vs alphabeta
- (b) Vs minimax
- (c) Vs random
- (d) Vs human

III Random
- (a) Vs. alphabeta
- (b) Vs. minimax
- (c) Vs. random
- (d) Vs. human

IV Various Other Conditions
- (a) Verifying correctness
- (b) Alphabeta vs Online agent

(c) Minimax vs Online agent
V Effective Branching Factor

Section I: Minimax

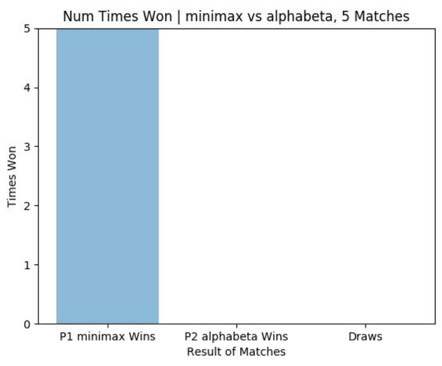
Winners			
P1-P2 ↓\Rounds →	5	10	20
Minimax v. Alphabeta	P1 Minimax	P1 Minimax	P1 Minimax
Minimax v. Minimax	P1 Minimax	P1 Minimax	P1 Minimax
Minimax v. Random	P1 Minimax	P1 Minimax	P1 Minimax
Minimax v. Human	P1 Minimax	X	X
Human v. minimax	P2 minimax	X	X

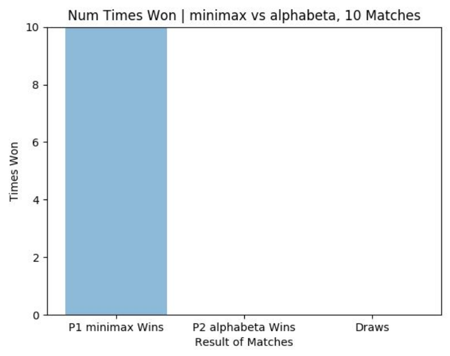
(a) Minimax vs Alphabeta
Minimax won unanimously each time against alphabeta - but why?
Simply because Minimax moved first, it was given the first move advantage. While first move advantage is generally not an issue for casual play, minimax and alphabeta are both designed to perform optimally. Therefore, minimax winning across the board is not so surprising. This can be seen for the case of 5 consecutive matches, 10, and finally 20.

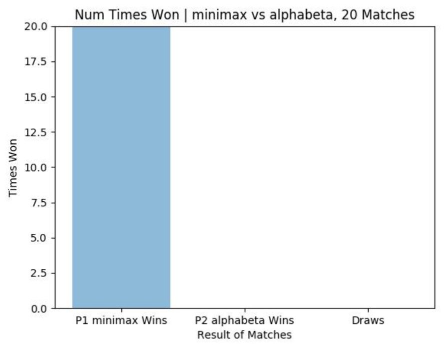
(b) Minimax vs Minimax
Minimax as player one is unsurprising for the same reason minimax vs alphabeta is unsurprising. In fact, this adds even further proof because both algorithms should behave identically - and the optimality causes the first move advantage of P1 minimax to make just enough difference in the outcome.

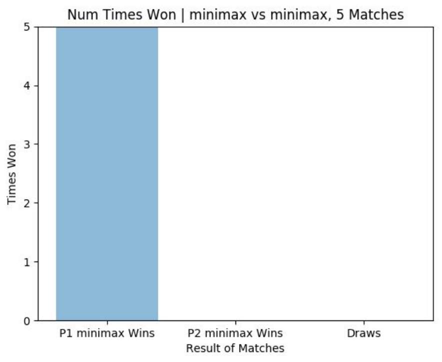
(c) Minimax vs Random
It is reasonable for a random strategy to lose against an optimal one. In theory this should be the case every time, even if random is given P1 advantage. Understandably, this is the case in our tests.

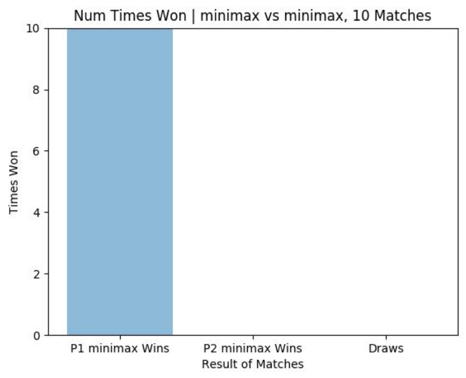
(d) Minimax vs Human
Now for the bad news (but good for the project!): the hard truth is that the optimal algorithm Minimax is better than most of us at Mancala. Our pitiful results can be seen below.

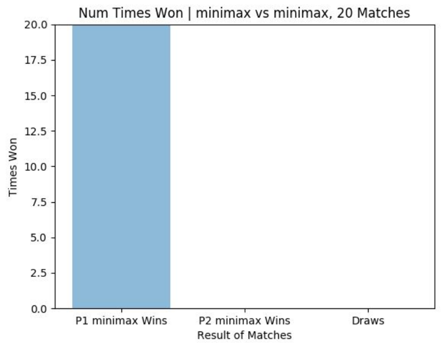

Num Times Won | minimax vs alphabeta, 5 Matches
Times Won
0
1
2
3
4
5
P1 minimax Wins
P2 alphabeta Wins
Draws
Result of Matches

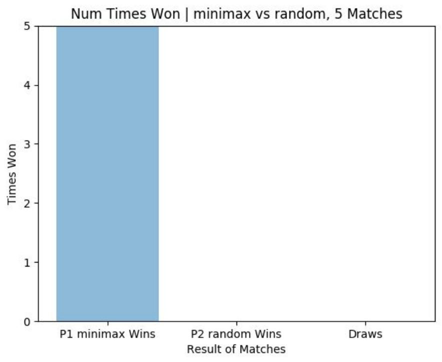

Num Times Won | minimax vs alphabeta, 10 Matches
Times Won
0
2
4
6
8
10
P1 minimax Wins
P2 alphabeta Wins
Draws
Result of Matches

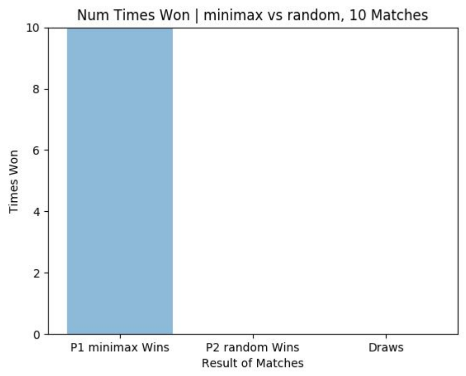

Num Times Won | minimax vs alphabeta, 20 Matches
Times Won
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
P1 minimax Wins
P2 alphabeta Wins
Draws
Result of Matches

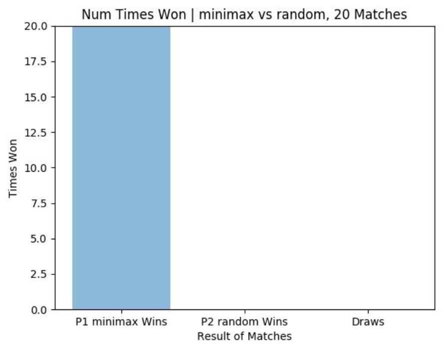

Num Times Won | minimax vs minimax, 5 Matches
Times Won
0
1
2
3
4
5
P1 minimax Wins
P2 minimax Wins
Draws
Result of Matches

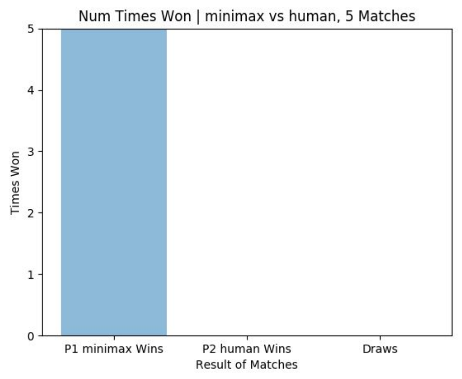
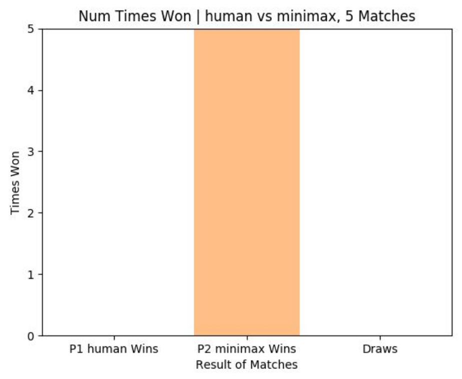

Num Times Won | minimax vs minimax, 10 Matches
Times Won
0
2
4
6
8
10
P1 minimax Wins
P2 minimax Wins
Draws
Result of Matches


Num Times Won | minimax vs minimax, 20 Matches
Times Won
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
P1 minimax Wins
P2 minimax Wins
Draws
Result of Matches


Num Times Won | minimax vs random, 5 Matches
Times Won
0
1
2
3
4
5
P1 minimax Wins
P2 random Wins
Draws
Result of Matches


Num Times Won | minimax vs random, 10 Matches
Times Won
0
2
4
6
8
10
P1 minimax Wins
P2 random Wins
Draws
Result of Matches


Num Times Won | minimax vs random, 20 Matches
Times Won
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
P1 minimax Wins
P2 random Wins
Draws
Result of Matches

Section II: Alphabeta

Winner			
P1-P2 ↓/Rounds →	5	10	20
Alphabeta v. alphabeta	P1 alphabeta	P1 alphabeta	P1 alphabeta
Alphabeta v. minimax	P1 alphabeta	P1 alphabeta	P1 alphabeta
Alphabeta v. random	P1 alphabeta	P1 alphabeta	P1 alphabeta
Alphabeta v. human	P1 alphabeta	X	X

Human v. alphabeta	P2 alphabeta	X	X

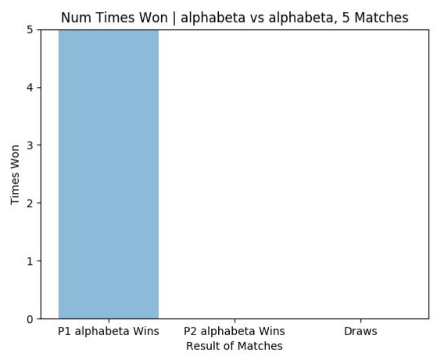
(a) Alphabeta vs Alphabeta
Because of the first turn advantage, P1 alphabeta dominated P2 alphabeta across the board for all batch sizes. Because alphabeta uses the same search algorithm as Minimax but more efficiently (via pruning), we expected to see the same level of optimization with quicker run time. This was the exact result we found (see below for run-time analysis of minimax and alphabeta).

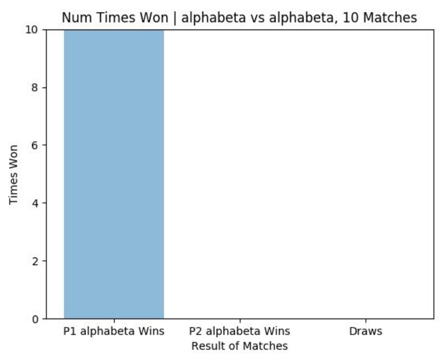
(b) Alphabeta vs Minimax
Also an expected result because of P1 advantage (you're probably starting to see a theme here). P1 alphabeta dominates P2 minimax for all batch sizes.

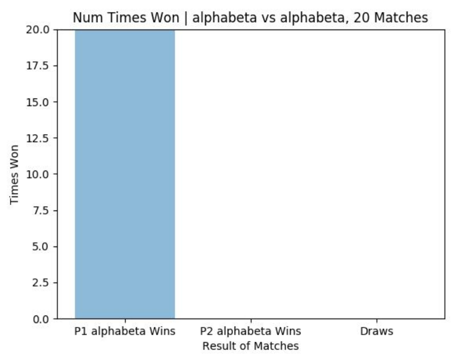
(c) Alphabeta vs Random
It almost goes without saying that alphabeta would win all matches vs random for the same reasons minimax did in the previous section discussion.

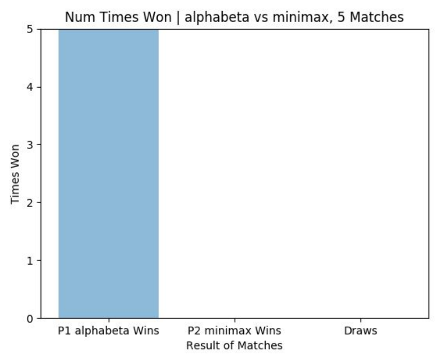
(d) Alphabeta vs Human
Alphabeta had no problem achieving unanimous victories against a human opponent - even when some strategies and heuristics were implemented by the human.

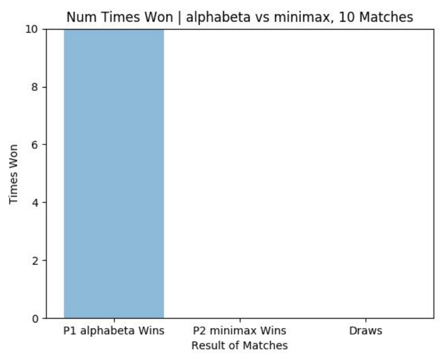
(e) Unfortunately, even with P1 advantage, the poor human opponent stood no chance against mighty alphabeta.

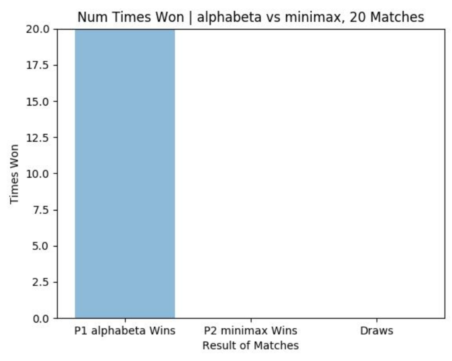


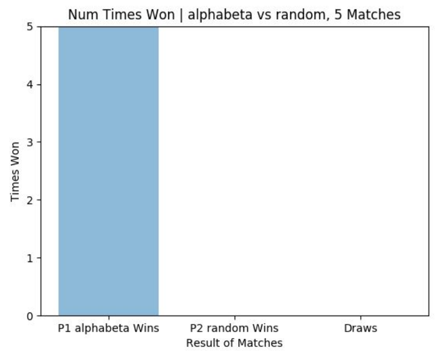
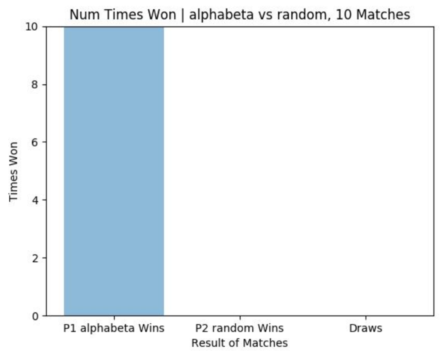
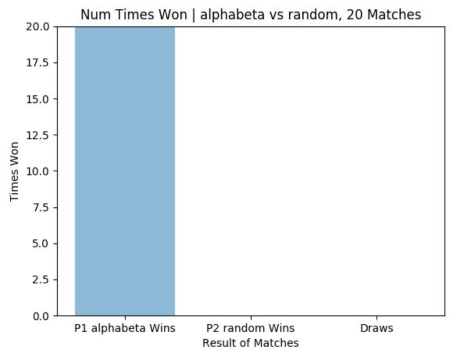
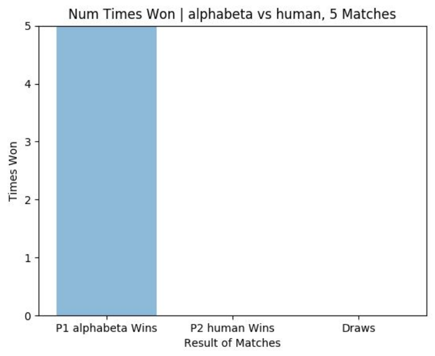
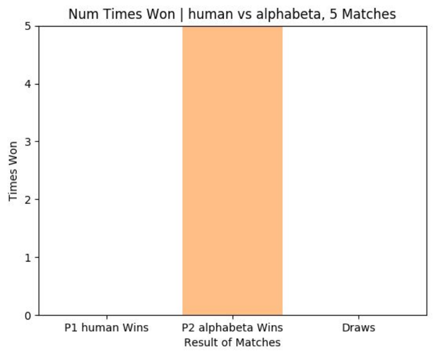










Section III: Random

Winner			
P1-P2 ↓/Rounds →	5	10	20
Random v. alphabeta	P2 alphabeta	P2 alphabeta	P2 alphabeta

Random v. minimax	P2 minimax	P2 minimax	P2 minimax
Random v. random	P1 random	P1 random	P1 random
Random v. human	P2 human	X	X
Human v. random	P1 human	X	X

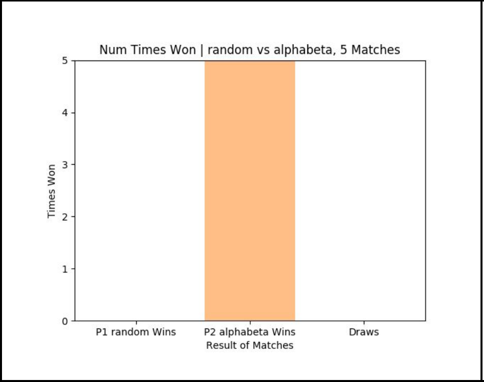
(a) Random vs Alphabeta
It seems even with P1 advantage, the random move algorithm was no match against an optimization algorithm - this was entirely expected and is a good sign that our search algorithms are implemented properly.

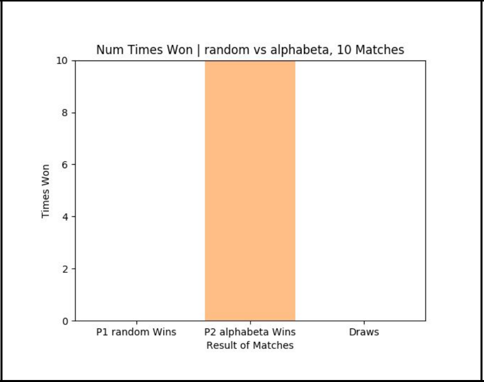
(b) Random vs minimax
Again, P1 random was swept by our optimal move search algorithm - expectedly.

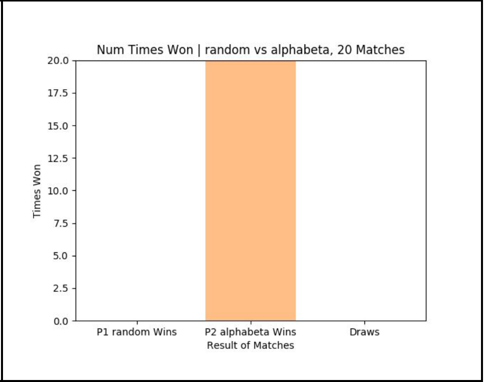
(c) Random vs Random
The results here are a little more interesting and warrant some discussion. By virtue of the stochastic nature of the random move algorithm, it makes sense here that even with P1 random first turn advantage, P1 still lost or received a draw some number of times, unlike our alphabeta and minimax. To clarify this idea, because there is no strategy in random's algorithm, there is only probability guiding the performance of its move decisions. This is exactly what we see for all batch sizes, the P1 advantage affords P1 random to win a slightly higher number of times but still losing a good amount.

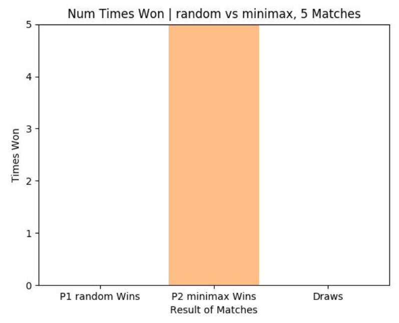
(d) Random vs Human
Even with random's P1 advantage, the human opponent defeated it a majority of the time. This is because of the aforementioned lack of strategy on random's behalf. *ANY* strategy should beat *NO* strategy.

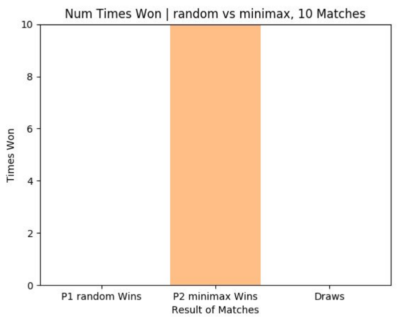
(e) Human vs Random
P1 avantage still allows P1 human to win a majority of the time against P2 random (though strangely slightly less wins than when human was P2 in part(d). But we'll chalk that up to random got lucky or perhaps the human opponent was tired).

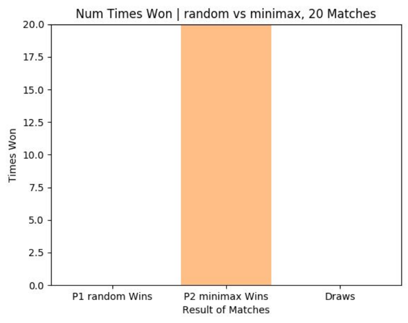


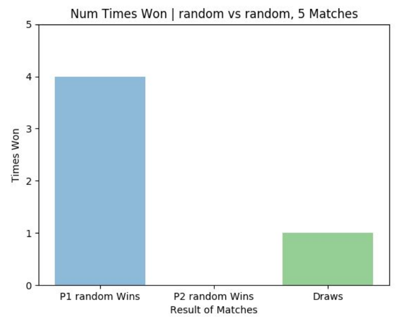


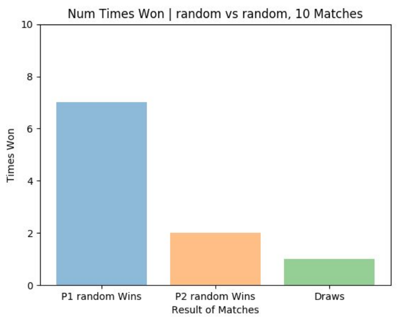


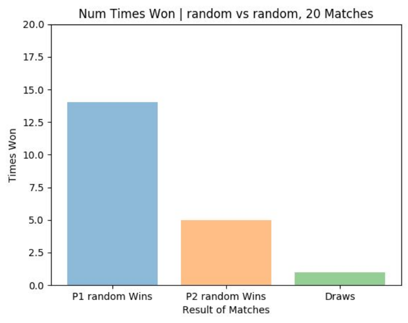


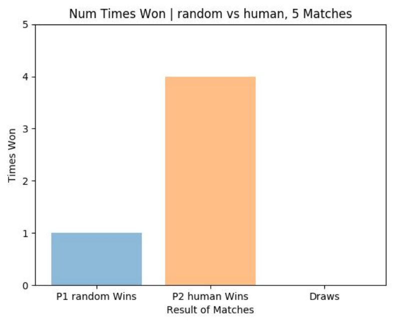


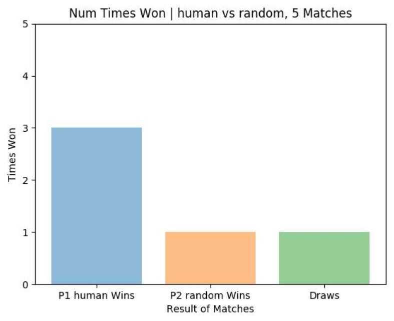












Section IV: Various other Conditions

(a) Verifying correctness

For Cutoff Depth = 8, we get the same final board contents as shown below for minimax vs minimax, alphabeta vs alphabeta, minimax vs alphabeta and alphabeta vs minimax.

```
         6    5    4     3    2    1

      --------------------------------

   09 | 01 | 00 | 01 || 12 | 00 | 00 | P2

  ----------------------------------------

   P1 | 00 | 00 | 00 || 00 | 00 | 00 | 25

      --------------------------------

         1    2    3     4    5    6
```

Similarly for Cutoff Depth = 9, we get the same final board contents for all the 4 combinations

```
         6    5    4     3    2    1

      --------------------------------

   07 | 14 | 00 | 00 || 00 | 00 | 00 | P2

  ----------------------------------------

   P1 | 00 | 00 | 00 || 00 | 00 | 00 | 27

      --------------------------------

         1    2    3     4    5    6
```

This ensured that our alphabeta gave the exact same result as minimax with the same cutoff depth verifying correctness of both the algorithms.

Finally we also pitted our minimax, alphabeta implementations against online players.

(b) Alphabeta vs Online Agent

In order to pit our algorithm against the online game's algorithm, we initiated a match on the following websites and another separate match on our own machines simultaneously - performing each game's respective moves on one another.

The AI agents from the online games are:

- http://arcadespot.com/game/mancala/ - Case 1
- https://www.coolmathgames.com/0-mancala - Case 2

As these should be at least as good as mediocre humans like us.

For this test we directly started with a higher value for cutoff depths as alphabeta is much faster than minimax for the same cutoff depth and to ensure alphabeta is able to defeat the agents. We also made alphabeta go first as the online games only allowed the user to start first.

For cutoff depth = 11 we pitted alphabeta against case 1 and the agent gets easily defeated with the final board contents as

```
           6    5    4     3    2    1

        --------------------------------

     09 | 00 | 01 | 01 || 06 | 00 | 01 | P2

    ----------------------------------------

     P1 | 00 | 00 | 00 || 01 | 01 | 06 | 31

        --------------------------------

           1    2    3     4    5    6
```

P1: Alphabeta

P2: Online Agent

For cutoff depth = 14 we pitted alphabeta against case 2 and this agent also got defeated with the final board contents as

```
           6     5     4      3     2     1

       --------------------------------

   09 | 00 | 00 | 00 || 00 | 00 | 00 | P2

   ----------------------------------------

   P1 | 00 | 00 | 01 || 13 | 01 | 01 | 23

       --------------------------------

           1     2     3      4     5     6
```

P1: Alphabeta

P2: Online Agent

(c) Minimax vs Online agent

For this we again used the agents from the online games:

- http://arcadespot.com/game/mancala/ - Case 1
- https://www.coolmathgames.com/0-mancala - Case 2

For cutoff depth = 8 we pitted minimax against case 1 and the agent gets easily defeated with the final board contents as

```
          6     5     4      3     2     1

       --------------------------------

   08 | 00 | 00 | 00 || 00 | 00 | 00 | P2

   ----------------------------------------

   P1 | 02 | 01 | 01 || 04 | 05 | 00 | 27

       --------------------------------

           1     2     3      4     5     6
```

P1: Minimax

P2: Online Agent

For cutoff depth = 9 we pitted minimax against case 2 and this agent also gets easily defeated with the final board contents as

```
           6    5    4     3    2    1

       --------------------------------

    12 | 00 | 00 | 00 || 00 | 00 | 00 | P2

   ----------------------------------------

    P1 | 00 | 00 | 02 || 04 | 02 | 04 | 24

       --------------------------------

           1    2    3     4    5    6
```

P1: Minimax

P2: Online agent

We finally pit minimax with cutoff depth = 2 against case 2 and in that case the agent is able to defeat our minimax with the final board contents as

```
           6    5    4     3    2    1

       --------------------------------

    19 | 00 | 02 | 02 || 01 | 05 | 01 | P2

   ----------------------------------------

    P1 | 00 | 00 | 00 || 00 | 00 | 00 | 18

       --------------------------------

           1    2    3     4    5    6
```

P1: Minimax

P2: Online agent

This was an interesting experiment we made on our free time

Section V: Effective Branching Factor (EBF)

To calculate the EBF we made use of the equation: $N + 1 = 1 + b + b^2 + \cdots + b^d$, where the variables are defined as:

N: Number of nodes

d: Solution Depth

b: Effective branching factor of the search tree with a uniform depth of d

The following entries correspond to the first move taken by alphabeta/minimax

d (Depth of the solution)	Total expansions by Minimax	Total expansions by Alphabeta	Time (in ms) taken by minimax	Time (in ms) taken by alphabeta	EBF for minimax	EBF for alphabeta
2	142	61	~0.05	~0.04	11.4	7.3
4	12215	1597	2.7	0.5	10.2	6.0
6	1396792	31733	92.5	2.3	10.3	5.4
8	103995764	618193	4224.21	38.0766	9.9	5.1
10	7048979569	8978869	281327	513.674	-	4.8

This suggests that alphabeta pruning is far efficient than minimax in terms of computational cost and performance.

General Discussion and Conclusion

This project was time consuming yet rewarding. By manually programming techniques described in AIMA, we were able to gain a robust understanding of adversarial search through our implementation of the Minimax algorithm and Alphabeta pruning. Additionally, through batch-testing we were able to see first-hand the power of intelligent search algorithms as well as the importance of heuristics and optimization techniques.

Although our implementation works, it didn't always work quite as expected. One challenge we encountered was regarding odd and even depth cutoff values. Strangely, when testing, we discovered odd depth cutoff values for minimax and alphabeta were losing almost unanimously against random. However, with *even* cutoff values, minimax and alphabeta behaved as expected. The issue was in our minimax implementation (and alphabeta) utility function. Ideally if player 1 started as MAX, the utility function should be evaluated from player 1's perspective which is player1's store count minus player 2's store count. But we instead evaluated the utility from the perspective of whatever player was executing in that search tree node. This went unnoticed for 4 days until we plotted bar graphs of win/loss/draw rates during which we immediately fixed that.

The most interesting observation is how our search algorithms defeated other implementations of Mancala. For example, when we put our alphabeta algorithm up against online Mancala implementations, ours won across the board. This could be due to several reasons: (a) our alphabeta's cutoff depth was higher than the online implementation's one, and as a consequence ours was able to find more optimized move possibilities, (b) the online implementation's search algorithm was inferior to the minimax search outlined in AIMA, or finally (c) perhaps combination of (a) and (b). Therefore, it is clear that our implementation of alphabeta (and minimax) worked as good if not better than expected.