CSC 261 **Lab 4: Adversarial Search** **Spring 2018**

Assigned: Tuesday 13 February

Due: Monday 19 February, 11:59 p.m.

Objectives:

- Understand **adversarial** search **implementations**
- Explore **performance** implications of game **heuristics**

Collaboration: This laboratory will be completed in pairs assigned by the instructor.

1 Introduction

Mancala is a "count and capture" game dating back to sixth century Africa that is still played worldwide. While the rules are fairly simple, it does require sophisticated foresight to achieve successful play. In this assignment, you are called upon to create an agent that will compete against the human world Mancala champion "Jen Kennings". How will you do it? The search tree for optimal game play is hopelessly large and deep (at least for our MathLAN machines), so something more clever is required. Fortunately, we've just learned about $\alpha - \beta$ pruning for minimax search algorithms. Now all we need is a solid implementation and a reasonable evaluation function for a game board state.

1.1 Game play

Mancala is a two player game, where each player has six "holes" that each start with four stones in them and one larger hole called the mancala. Taking turns, a player removes all of the stones from one of his or her holes (excluding the mancala) and deposits them one by one in other holes on the board in a counter-clockwise direction. Along the way, stones may be deposited in the player's own mancala, but the opponent's mancala is skipped. If the last stone is deposited in an empty hole belonging to the player, all of the stones in the opponent's adjacent hole are captured (along with the singleton in the previously empty hole) and moved to the player's mancala. The goal of the game is to end up with the most stones in your mancala (and potentially remaining in the other holes). The game ends when one player has no available moves.[1]

```
(  ) ( 4) ( 4) ( 4) ( 4) ( 4) ( 4) (  )
(  )  12   11   10    9    8    7  (  )
( 0)                                ( 0)
(  ) ( 4) ( 4) ( 4) ( 4) ( 4) ( 4) (  )
(  )   0    1    2    3    4    5  (  )
```

Photo Credit: http://www.flickr.com/photos/revgriddler/2736706261 (Used by permission.)

The figure above shows children in Cambodia playing a version of the game. On the right is a textual representation of the board including hole numbers and the two mancala bins on the left and right sides.

[1] In true mancala, dropping the last stone of a play into the mancala would garner another turn. Because that rule makes minimax search somewhat more complicated, we will ignore it, allowing strict alternation.

2 Code and environment

For this assignment, you will need to copy some starter code from the MathLAN directory:

```
~weinman/courses/CSC261/code/adversarial
```

2.1 Game interface

To promote code reuse, the search routine you will write takes a generic interface we're calling a `game` (defined in `game.scm`.) For search, you will need to use two procedures encapsulated within a `game` value:

`(game-successors-fun` *`game`*`)` produces the successor function for a game state (giving the actions available to the player whose turn it is along with the resulting states), and

`(game-terminal?` *`game`*`)` produces the terminal predicate for identifying when the game has ended.

Other encapsulated procedures include

`(game-win?` *`game`*`)` produces a "win" predicate for identifying who has won the game,

`(game-starting-state` *`game`*`)` produces a starting state for the game,

`(game-display-fun` *`game`*`)` produces a display procedure for a game state, and

`(game-play` *`game player1 player2`*`)` runs a game play engine pitting two decision procedures, *`player1`* and *`player2`*, against each other.

See each procedure's documentation for greater details. We will be using the following example games:

`barranca.scm`, for which it is reasonable to manually generate and trace an entire search tree to verify the searches, as we have done in class,[2]

`tictactoe.scm`, for which it is computationally feasible to run a complete MINIMAX search, and

`mancala.scm`, for which alpha-beta search with cutoff is required.

2.2 Adversarial search

2.2.1 Minimax search

The file `minimax.scm` contains a complete implementation of minimax search. The function `make-minimax-player` is simply a "functor" (a function that returns a function) for creating a player procedure, which takes a state and produces the action determined by a minimax search. The `minimax-search` procedure takes a *`game`*, the current *`state`*, and a *`utility`* function (for a particular player); it produces the optimal action by calling `max-value`. The procedure `max-value` takes a *`game`* and a *`state`* of the game; it determines the best action and its value, both of which are stored in the returned pair. The best action is determined by recursively calling `min-value`, which behaves similarly. When `max-value` returns to `minimax-search`, the optimal action is given by taking the `car` of the `action-value` pair produced by `max-value`.

You may enable some built-in display code by setting `DO-DISPLAY` to `#t` at the top of the `minimax.scm` file . This will allow you to see the utility of terminal states, and the actions and values for each max and min node in the tree. (I do not recommend you enable displays for tic-tac-toe, as the tree is quite large.)

The file `cutoff-minimax.scm` employs the cutoff strategy suggested by the text (cf., AIMA Section 5.4.2). This implementation is similar to the routines in `minimax.scm` except that they use a state evaluation function, rather than a utility function, and only search to a given number of plies.

[2] I am indebted to John Stone for introducing me to this game, which is described in the following article.
Guy, R. K. (1990). A guessing game of Bill Sands, and Bernardo Recamán's Barranca. *American Mathematical Monthly 97*(4), 314–315. `http://www.jstor.org/stable/2324514`.

2.2.2 Alpha-Beta search

The file `alphabeta.scm` contains skeletons and specifications for the two procedures you are to write. These are similar in nature to the examples in `minimax.scm`, and `cutoff-minimax.scm` with a few key differences. Your recursive routines will require the α and β values, as well as keeping track of the search depth, so that an evaluation function can be applied.

Just like `make-cutoff-minimax-player`, the provided procedure `make-alpha-beta-player` creates a player that uses a particular evaluation function and searches down to a depth of the given number of plies. The `alpha-beta-search` procedure begins the search for a maximizing action.

You will complete the definitions of `alpha-beta-max-value` and `alpha-beta-min-value`. Of course, these are so similar that implementing one practically gives you the other.

We will have more to say about how `evaluation-fun` examines a state and calculates a score in a moment.

2.3 Mancala game

The file `mancala.scm` provides an implementation of the rules given above that meets the `game` interface specification. The only details you will need to worry about (eventually) are the displayed board format shown on the first page (if you care to see how the game is being played) and the actual state representation in Scheme for writing an evaluation procedure.

3 Assignment

Problem 1 - Creating an evaluation function [40 points]

Part A

Our in-class lab looks at a board evaluation function that is too simple; it ignores many important aspects of the end game until it is too late. Your first task is to write a better evaluation functor `best-mancala-eval`

```
(define best-mancala-eval
   (lambda (player)
      (lambda (state)
         ...
```

Place this function (and any helpers) in the starter file called `evaluation.scm`. Things that you may want to consider include

- number of stones in your player's mancala,
- the total number of stones on your player's side of the board,
- the number of empty holes on your player's side of the board,
- board configurations that can lead to large gains (or losses), or
- anything else you might think of.

Experiment with several different heuristics, systematically testing them to determine which seems to work best. You can test these heuristics with the `cutoff-minimax-search` provided as well as the alpha-beta pruning implementation you will complete.

Part B

Write a short essay about how you chose your evaluation function. What things did you try? What worked well and what didn't? Explain *how* you determined what works well.

The audience for your essay is your class peers. That being the case, you still ought to *briefly* introduce the problem and context.

Problem 2 - Implementing alpha-beta pruning [40 points]

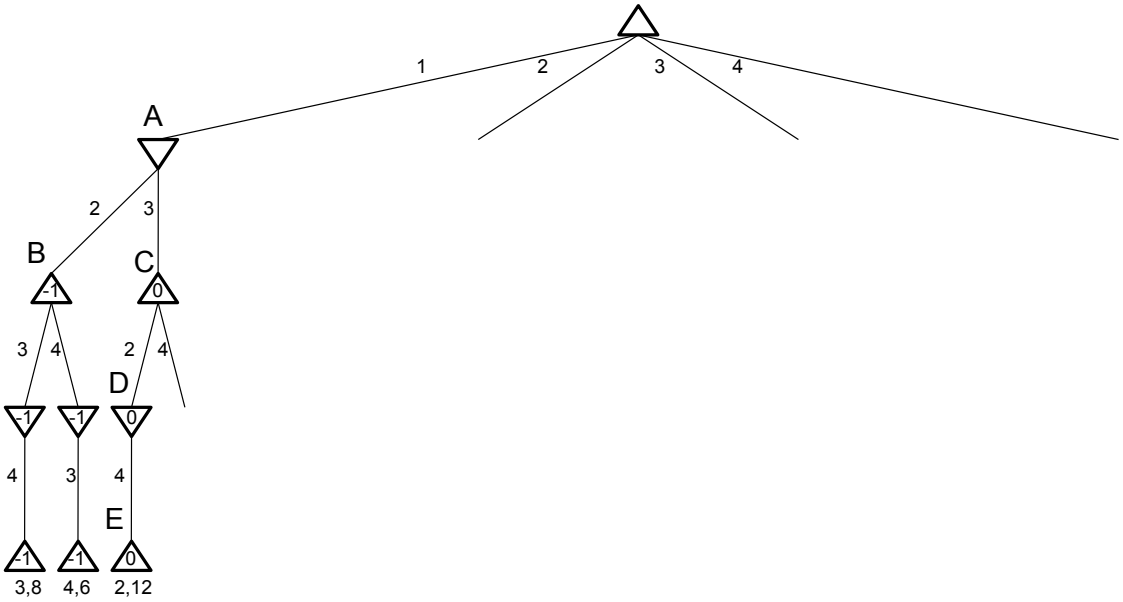
Using the procedures in `minimax.scm` and `cutoff-minimax.scm` as guides, implement `alpha-beta-max-value` and `alpha-beta-min-value` for alpha-beta search as specified in `alphabeta.scm` and described in AIMA Figure 5.7 Like `max-value` in `minimax.scm`, and `cutoff-max-value` in `cutoff-minimax.scm` `alpha-beta-max-value` returns both the optimal estimated action and its value in a pair.

Remember that MINIMAX search expands the *entire* search tree. Because we may not always be able to search a game tree all the way to terminal states, you will need to impose a cutoff test limiting the number of plies that are searched. Just as in `cutoff-minimax.scm`, every transition from a MIN node to a MAX node increases the depth (number of plies) of the search.

Testing your code

To test your alpha-beta search, you will need to use a simpler game than mancala. Fortunately, you also have barranca and tic-tac-toe. Using either of these games, carefully test your algorithm by calculating the utility values of various states, making sure your search is returning the correct move and pruning the search tree appropriately.

Examine at least one example that demonstrates your search routines correctly prune the search tree. For example, consider the example barranca game ($n = 4$, $k = 7$) whose search tree you will trace in the in class lab; it is illustrated (in part) below.



Player 1 (max) has started by choosing 1. At the MAX node labeled B, player 2 has found the result of choosing the number 2. After this, the MIN node labeled A should have updated $\beta = -1$. It then proceeds to calculate the max value for C, with $\beta = -1$. Once the MIN node labeled D returns the value 0, node C knows that the value it returns will be at least 0, but in fact $0 > \beta = -1$. Because the result is necessarily higher than the other option that the parent MIN node A already has, there is no need to examine the other option. Thus, we may prune the remaining successor of C, which corresponds to choosing 4.

To verify this, one might add `display`/`printf` statements to the code that produce such an informative output, such as:

```
...
MAX state (#t (1) (3) (2 4))    Entrance for state C
MIN state (#f (2 1) (3) (4))    Entrance for state D
MAX state (#t (2 1) (4 3) ())   Entrance for state E
MAX value  0                    Value of state E
MIN value  0                    Value of state D
beta=-1.0: prune                Discontinuation in state C
MAX value  0                    Value of state C
...
```

Using a different example (from any game you like), explain to yourselves what is happening, as in the annotated example above.

In your submitted code, leave any display lines in place but commented out.

Mancala tournament

Though no part of your grade will be dependent on it, we will conduct a tournament among your mancala players. To compete, create a separate file called `tournament.scm` containing defininitions of *username*`-mancala-best-player1` and *username*`-mancala-best-player2` using your evaluation procedure `best-mancala-eval` and whatever search routine you choose. (Use only the username of the group member submitting to PWeb.) The search routines (i.e., all of your `alphabeta.scm` file) and your evaluation procedure must also be repeated (copied into) this `tournament.scm` file. You might start by concatenating your files with the following terminal command

```
$ cat alphabeta.scm evaluation.scm > tournament.scm
```

Delete any existing `module`, `load` or `require` directives from `tournament.scm`. After you have added your player definitions, add this header to the top of your `tournament.scm` file

```
(module tournament lang/plt-pretty-big
        (provide username-mancala-best-player1
                 username-mancala-best-player2)
        (require "game.scm" "mancala.scm")
```

and add the final closing paren (which matches the `module` directive) at the *very* end of your file.

To test your work, add the statement `(require "mancala.scm" "tournament.scm")` to a file saved in the same directory. Run the file and try `(`*username*`-mancala-best-player1 mancala-start-state)`

Each call to your players must take no longer than 10 seconds nor search deeper than 5 plies. Any player procedure that violates these constraints will be disqualified. The winner will gain bragging rights and (subject to availability) some sort of prize.

What to turn in

In addition to `references.txt`, your submission should include the following

- Your completed `alphabeta.scm` file
- Your `evaluation.scm` file defining your mancala heuristic `best-mancala-eval`
- A short `driver.scm` program that demonstrates your procedures are functional and correct
- A transcript `output.txt` of your test driver program's output
- A single PDF `evaluation.pdf` containing your evaluation function essay
- [Optional] Your completed `tournament.scm` file