CSC 261 **Assignment 2: Adversarial Search** **Fall 2009**

Assigned: Monday 21 September 2009

Due: Monday 28 September 2009

Topics: Minimax search, $\alpha - \beta$ pruning, heuristics

Objective: Gain greater understanding of adversarial search implementations and the performance implications of a game heuristic

Expected Time: 4-6 hours (for those who are comfortable with the mechanics of search and Scheme)

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy of source code. Your non-programming components should be typed, nicely formatted, and feature a logical organization, complete sentences, and proper grammar, spelling, and punctuation. Only one submission (both paper and digital) per group is necessary.

1 Introduction

Mancala is "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 world Mancala champion Carrie Gasparov. 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 6 "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 play, dropping the last stone into the mancala would garner another turn. Since that makes minimax search somewhat more complicated, we will ignore this rule, 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
```

You may use *DrScheme* (Choose "Pretty Big" as your language set) or *MediaScheme* for this assignment, whichever you prefer.

2.1 Game Interface

In order to promote code reuse, the search routine you will write takes a generic interface we're calling a `game` (defined in the file `game.scm`.) Encapsulated within a `game` value are

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

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

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

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

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

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

Note that only the first two of these procedures will be involved in implementing your adversarial search. You may examine the the corresponding procedures documentation for greater details.

Several example games have been provided:

`barranca.scm`, for which you can manually generate and trace an entire search tree to verify the searches, as we have done in class

`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 procedure that returns a procedure) for creating a player procedure that takes a state and produces an action, which is determined using minimax search. The `minimax-search` procedure takes a `game`, the current `state`, and a utility function (for a particular player), and 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 returned `action-value` pair.

There is a great deal of built-in display code that may be enabled by setting the value `DO-DISPLAY` at the top of the file to `#t`. 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. (It is not recommended you do this for tic-tac-toe, though.)

The file `cutoff-minimax.scm` employs the cutoff strategy suggested by the text. It is similar to the `minimax.scm` routines except that it uses a state evaluation heuristic function, rather than a utility function, and only searches to a particular number of plies.

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 alpha and beta 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 to a depth of a 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 Examples

3.1 Getting Started: Barranca and Minimax search

So where do we start? First we need to establish a game. Let's go with Barranca, first.

```
(define barranca (make-barranca-game 4 7))
```

This gives us a barranca game with $n = 4$ number possibilities and a target $k = 7$.

When we play the game, anyone using Minimax-Decision needs a Utility function that takes a state as a parameter and produces a number. The state of the game is represented the same way for *both* players. Therefore, the utility function will need to distinguish whether a state is advantageous for the player or its opponent. Thus, in creating a utility we will specify who we are advocating for (that is, *whose* definition of utility, player 1's or player 2's).

In barranca, the state contains the player getting the next turn, the list of numbers each player has, and the list of remaining numbers. Thus, when evaluating utility, the procedure needs to know which list of player-possessing numbers belongs to it. In all the provided games, player 1 is represented by `#t` and player 2 represented by `#f`. We will use `barranca-utility-fun` (provided in `barranca.scm`) to create a utility for each player. This gives 1 for wins, 0 for draws, and -1 for losses. We also need it to know what the target is (so it can determine how good the numbers being held are).

```
(define barranca-player1-utility (barranca-utility-fun #t 7))
```

Ignoring complete game play for a moment, we could directly ask our search routine to find us the best opening move (and its value) for player 1.

```
> (max-value barranca
             (game-start-state barranca)
             barranca-player1-utility)
(2 . 1)
```

This tells us that the best move is to choose 2, and the corresponding value is 1. After tracing the search tree, this shouldn't be a surprise. Since the utility is 1, this guarantees a win for player 1. Of course, usually we'll just want the optimal move, so we could ask for that more directly.

```
> (minimax-search barranca
                  (game-start-state barranca)
                  barranca-player1-utility)
2
```

Now that we're primed for back and forth game-play, we'll need to tell the game playing engine the routines that will be used to select moves.

```
(define barranca-player1 (make-minimax-player barranca barranca-player1-utility))
```

Thus, `barranca-player1` is a procedure that takes a state (given to it by the game playing engine) and returns the move it wishes to make. The game playing engine can then make this move and give the resulting state to the opposing player for consideration. This process would repeat until the game terminates.

What's actually going on inside minimax-search? If we set `DO-DISPLAY` in `minimax.scm` to `#t`, we will end up with complete trace of the search tree. Here is the portion corresponding to the two left-most nodes of the search tree

```
> (minimax-search barranca
                  (game-start-state barranca)
                  barranca-player1-utility)
MAX state: (#t () () (1 2 3 4))
MIN state: (#f (1) () (2 3 4))
MAX state: (#t (1) (2) (3 4))
MIN state: (#f (3 1) (2) (4))
MAX state: (#t (3 1) (4 2) ())
Player 1: 3
Player 2: 8
MAX Utility -1
MIN ((4 . -1))
MIN state: (#f (4 1) (2) (3))
MAX state: (#t (4 1) (3 2) ())
Player 1: 4
Player 2: 6
MAX Utility -1
MIN ((3 . -1))
MAX ((3 . -1) (4 . -1))
[Remaining output elided]
```

We start by following the alternating max and min nodes down the left side of the tree, as 1 is chosen by player 1, then 2 is chosen by player 2, 3 is then chosen by player 1, and finally 4 is chosen by player 2. Here the two products are 3 and 8. Since the move belongs to max (player 1) and there are no remaining possibilities, the utility is reported as -1 (3 is much farther from the goal of 7 than player 2's value of 8).

The min node above this then reports that its only option is to choose 4, and the utility is -1. The process is repeated for the case when the max node has chosen 4 as its second value, instead of three. This leaves player 2 (min) with only one possibility, and the final max node with no remaining moves. The utility is again calculated, turning out again to be in favor of min (player 2). The min node reports that its only action is to choose 3, with a utility of -1. Just above this in the tree, the max node now has a choice between 3 and 4, but both of these have a utility of -1. You are encouraged to trace the remaining output through the tree.

3.2 Moving' on up: Tic-Tac-Toe

Let's create a tic-tac-toe game. There are no free parameters in this game, so the call is straightforward.

```
(define ttt (make-tictactoe-game))
```

Just like barranca, a tic-tac-toe decision making procedure has to know *who* it is playing for. The procedure `tictactoe-utility-fun` simply takes an argument indicating whether it needs to produce a procedure for player 1 (`#t`) or player 2 (`#f`). We can put these together and create two optimal players (using minimax search) for tic-tac-toe.

```
(define ttt-smart-player1 (make-minimax-player ttt (tictactoe-utility-fun #t)))
(define ttt-smart-player2 (make-minimax-player ttt (tictactoe-utility-fun #f)))
```

Of course, optimal play in tic-tac-toe should always lead to a draw. How can we pit these fearless competitors against each other? We simply use the game-play engine, giving it the game (so it knows the rules) and the two decision-making procedures.

```
> (game-play ttt ttt-smart-player1 ttt-smart-player2)
- - -
- - -
- - -
Player 1 chooses 0
X - -
- - -
- - -
Player 2 chooses 4
X - -
- O -
- - -
[Several moves elided]
X O X
X O O
O X -
Player 1 chooses 8
X O X
X O O
O X X
Draw!
```

It's comforting that things to in fact lead to a draw. Of course, the actions being reported may not be meaningful to you, but they correspond to the rules of the game, as internally represented. (It's not so hard though: the slots are numbered from top-to-bottom, left-to-right, starting at 0).

What if we don't have an optimal player? Let's create a remarkably simple agent decision function that always chooses the first possible action.

```
(define ttt-lazy-player
   (lambda (state)
      (caar ((game-successors-fun ttt) state))))
```

The successor function gives us a list of pairs. Since the first item in the list is a pair `(first-action . resulting-state)`, taking the `car` of the first item gives us the actual action choice. (Combining the two leads to the `caar` shown above.) The astute reader will notice that we haven't checked for the case when there are no successors. We need not, since the game play engine will not ask us for a play when there is none to make. What happens when our lazy agent squares off against the optimal player?

```
> (game-play ttt ttt-smart-player1 ttt-lazy-player)
[Many silly moves elided]
X X X
O X -
O O -
Player 1 Wins!
```

No surprise there. But how important is this utility function anyhow? What if we gave player 1 the utility function used by player 2? Even if our latter player is extremely lazy, player 1 will in fact be choosing moves that benefit player 2.

```
> (game-play ttt ttt-smart-player2 ttt-lazy-player)
[Many silly moves elided]
O - -
O X X
O - X
Player 2 Wins!
```

Don't be misled by the variable names. Player 2 is actually `ttt-lazy-player`.

3.3 Mancala

In mancala, the state is represented as a list, whose `car` indicates which player's has the next turn and whose `cdr` is the board. The board is represented as a list of 14 holes, as follows. Indices 0-5 are the holes for player 1 with index 6 as player 1's mancala. Indices 7-12 are player 2's holes and index 13 is player 2's mancala. The figure on page one illustrates the indices along with the contents of each slot. Creating a mancala game is straightforward.

```
(define mancala (make-mancala-game))
```

Since mancala has a moderately large branching factor, we may not always be able to search a game tree all the way to terminal states. As mentioned above, we can instruct the minimax search to cutoff at a particular depth (number of plies) and then apply an evaluation function for *estimating* the utility of a given game board. The file `mancala-player.scm` has an extremely simple evaluation function that simply counts the stones in our player's mancala.

```
(define mancala-player1-eval (simple-mancala-eval #t))
```

We can sanity check this by asking it to evaluate the initial state of the game.

```
> (mancala-player1-eval (game-start-state mancala))
0
```

Sure enough, we start out with zero stones in our mancala.

We can then use this evaluation function to create a player that only searches, say, just 3 plies deep in the tree

```
(define mancala-player1 (make-cutoff-minimax-player mancala 3 mancala-player1-eval))
```

and then ask the player what opening move it chooses.

```
> (mancala-player1 (game-start-state mancala))
2
```

We could also create an even simpler (lazier) player that simply chooses the first action.

```
(define lazy-mancala-player
   (lambda (state)
      (caar ((game-successors-fun mancala) state))))
```

What happens as these two players square off?

```
> (game-play mancala mancala-player1 lazy-mancala-player)
[Many iterations elided]
(  ) ( 7) ( 0) ( 0) ( 0) ( 0) (  )
(  )  12   11   10    9    8    7  (  )
(13)                               (28)
(  ) ( 0) ( 0) ( 0) ( 0) ( 0) (  )
(  )   0    1    2    3    4    5  (  )
Player 1 Wins!
```

No surprise there. What if we use the same evaluation function for player 2, but allow it to look ahead a little bit further?

```
(define mancala-player2 (make-cutoff-minimax-player mancala 4 (simple-mancala-eval #f)
```

What happens when these play each other? (Aside from the fact that player 2 will take longer).

```
> (game-play mancala mancala-player1 mancala-player2)
[Many iterations elided]
(  ) ( 0) ( 0) ( 2) ( 1) ( 0) ( 0) (  )
(  )  12   11   10    9    8    7  (  )
(25)                               (20)
(  ) ( 0) ( 0) ( 0) ( 0) ( 0) ( 0) (  )
(  )   0    1    2    3    4    5  (  )
Player 2 Wins!
```

You should note that going any deeper will make the search painfully slow. Hence, we will need to use alpha-beta pruning to ignore irrelevant sub-trees.

4 Assignment

Note: 10 points will be credited based on an evaluation of the cleanliness, readability, and organization of your Scheme code.

Problem 1 - Creating an Evaluation Function [45 points]

Part A [30 points]

As mentioned above, the given board heuristic 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)
         ...
```

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 may test these heuristics with the cutoff-minimax provided, or you may wait until you've completed an alpha-beta pruning implementation.

Part B [15 points]

Include a writeup of a few paragraphs 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.

Problem 2 - Writing the Alpha-Beta Search Algorithm [45 points]

Part A [30 points]

Using the procedures in `minimax.scm` and/or `cutoff-minimax.scm` as potential 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 6.7 Like `max-value` in `minimax.scm`, `alpha-beta-max-value` returns both the optimal action and its value.

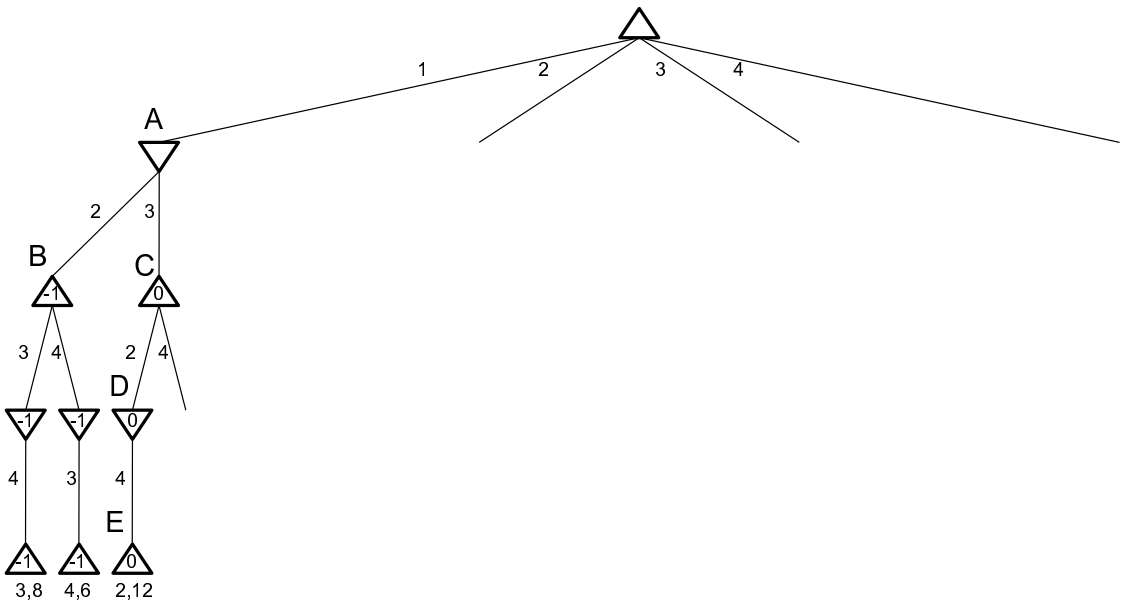
Remember that MINIMAX search expands the entire search tree. Because every single successor action needs to be evaluated, `map` can be used to apply `min-value` to all the successor states. In alpha-beta search, you will need to explicitly iterate over every successor state so that you may stop iterating and return a state-action pair when no further recursive expansion needs to be done. Thus, in addition to keeping track of the maximum value in `alpha-beta-max-value`, you will also need to keep track of the *action* that yields the maximum value because both are returned in a single cons cell.

Since we may not always be able to search a game tree all the way to terminal states, we 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.

Part B [15 points]

You will need to test your alpha-beta search on something simpler 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.

Include a writeup with at least one example that illustrates your search routines correctly prune the search tree. For example, consider the example barranca game ($n = 4$, $k = 7$) with the search tree traced in the example above and is illustrated 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 $0 > \beta = -1$, so this is higher than the other option that the parent min node A already has, so there is no need to examine the other option. Thus, we may prune the remaining successor of C, which is 4.

To write this up, one might add a `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
...
```

You should also explain what is happening, as in the example above and in the output annotations.

You must choose a different example for demonstrating the correctness of your code, though you may use any game you like. Significant credit for will be attributed to the quality and clarity of your explanation. Be sure you understand this example so you are able to produce your own.

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

Mancala Cage Match

Though no part of your grade will be dependent on it, a tournament using your own mancala players will be conducted. Somewhere in your submitted files you should define *username*`-mancala-best-player1` and *username*`-mancala-best-player2` using your evaluation procedure `best-mancala-eval` and whatever search routine you choose. However, each call to your player 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.