Assignment 3 (C)
Game Trees and Alpha-Beta Pruning

Computer Science 182 — Fall 2009

Electronic Submission due Wednesday, October 7 at 2:00 pm.

1 Introduction

Have you ever wondered why computers cannot perform many of the simplest and most natural human activities, yet excel at some of the most difficult ones? Why is it that no computer to date has been able to learn how to tell a joke, yet some computer systems are among the world's top 100 chess players, one beating former world champion Gary Kasparov in 1997 and another recently beating world champion Vladimir Kramnik in 2006 (Deep Fritz, on dual Core2 Duos)?

In fact, computers are very good at playing games because, unlike the environment in which most everyday human activities take place, gaming environments are *accessible, deterministic*, and *discrete*. Such environments sparked some of the earliest work in artificial intelligence, when a lot of early research concentrated on game playing techniques. This area is now well developed and many research results have found their way into successful commercial applications.

In this assignment, you will learn how to design and implement a relatively simple system in Lisp that can act as a formidable opponent in a game of Connect Four. To make your life easier, we have provided you with a large body of pre-written code to handle the user interface as well as the internal representation of the playing grid. Your task is to write a move generator that will find the next move in the game using state-space search with alpha-beta pruning, as explained below.

The fully functional system will be able to play a decent game of Connect Four against itself, a human opponent, or another system, always finding the best move quickly and without running out of memory.

- Tasks that you must perform for assignment 3 are bulleted like this, throughout this document.

2 Connect Four: The Game

2.1 The Rules

Connect Four is a two-player strategy game similar to tic-tac-toe. It is played using 42 tokens (usually 21 red tokens for one player and 21 black tokens for the other player), and a vertical grid that is 7 columns wide. Each column is able to hold a maximum of 6 tokens. The two players take turns. A move consists of a player dropping one of her tokens into the column of her choice. When a token is dropped into a column, it falls until it hits the bottom of the column or the top token in that column. A player wins by creating an arrangement in which at least four of her tokens are aligned in a row, column, or diagonal. For example, consider the state of the game shown in Figure 1.

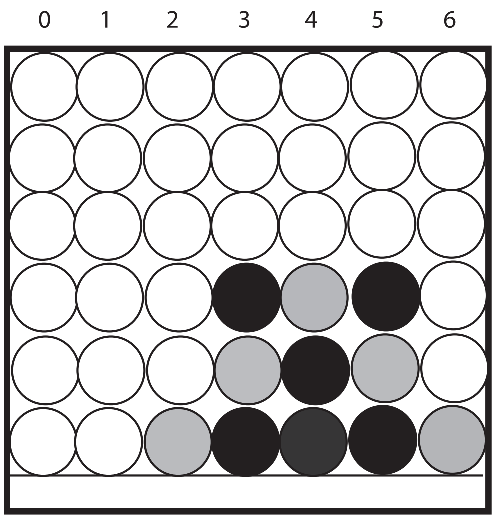

Figure 1: Sample Connect Four Board

The two kinds of tokens here are Black and Gray; it is Gray's turn to move. If Gray drops a token into column 3 or column 5, she wins.

It is quite possible for a game of Connect Four to end in a draw, i.e. in a state where all 42 tokens have been used, the grid is full, but there are not four tokens of either color aligned in any direction at any location.

2.2 The User Interface

The files required for this assignment are available in ∼`lib182/asst3`. We have provided you with a package that keeps track of the state of the Connect Four grid, and permits moves to be undone in the reverse order in which they were made. This code also provides a high-level user interface, allowing a human player to start a new game of Connect Four with the grid in some initial state (i.e., with some moves already made), make a move manually, or force the computer to generate a move automatically. All of these features are available through the code in the file `manage-board.lisp`. We recommend that you keep a printout of `manage-board.lisp` handy as you read on to use as reference.

The top level initialization function for the Connect Four system is called `start-new-game`. This function takes two optional keyword arguments: `grid-list` and `level`. `grid-list` is a list-of-lists representation of the initial state of the playing grid (in case we want to continue an existing game, for example). `level` is a non-zero integer indicating the computer's "skill level": the greater this number, the longer the move generator will take to find the best move. Under the present implementation, `level` is simply the depth cut-off value used by the move generator, so a `level` value of 6 would result in a 6-ply search of the game tree. Both `grid-list` and `level` are optional arguments; `grid-list` defaults to the empty grid, and `level` defaults to 4.

To begin a new game with a clear board, a user will type: `(start-new-game :level n)`, where n is some positive integer. To start a game with a particular layout of tokens, the user will type something such as:

```
-> (start-new-game :grid-list  '((_ _ _ _ _ _ _)
                                 (_ _ _ _ _ _ _)
                                 (_ _ _ _ _ _ _)
                                 (_ _ _ _ _ _ _)
                                 (X _ _ O _ X O)
                                 (X _ O X O X O))
                   :level 3)
```

This results in the following output:

```
The initial state of the game is:


 _ _ _ _ _ _ _
 _ _ _ _ _ _ _
 _ _ _ _ _ _ _
 _ _ _ _ _ _ _
 X _ _ O _ X O
 X _ O X O X O

It is now X's turn.

NIL
USER(3):
```

Note that the user enters the initial grid as a list of rows, each row being a list of slots. An empty slot is denoted by an underscore. For convenience, an entirely empty row may be notated as the empty list.

To make a specific move, the user would call the function `play`. `play` takes a keyword argument `:move` which, if supplied, specifies the column into which the current player's next token is to be dropped. For example, `(play :move 4)` would result in the current player's next token being dropped into column 4. To have the computer generate a move, the user would call play with no arguments. To make a move, and have the computer generate a countermove, the user could call the `play-round` function, which takes a keyword argument in the same way as `play` does. Finally, to have the computer play against itself until the game ends, the user would call the `finish-game` function with no arguments.

2.3 The Internal Representation

In addition to these high-level functions, `manage-board.lisp` contains many low-level functions used to access the data structures underlying the internal representation of the game state that you will need to use, e.g. `list-legal-moves` and `game-next-turn`.

The function `print-board` is the print function for the struct `board`. At any point in your code, you may include `print` or `format` calls in order to print out the current board stored in the global structure `*current-game*` for debugging reasons. These calls will implicitly invoke `print-board`.

3 Writing a Move Generator

Having understood how the high-level functions in `manage-board.lisp` work, your next task is to implement a *move generator*. The `play` functions in `manage-board.lisp` call the function `generate-move` (also in `manage-board.lisp`) which, in turn, calls the function `compute-move` to find a move; You will write the `compute-move` function, along with any helper functions, in the file `select-move.lisp`. Your move generator should be capable of using the minimax algorithm as well as minimax with alpha-beta pruning; however, we leave decisions about implementation details to you. You will use the `*alpha-beta-p*` flag to toggle alpha-beta pruning, where `T` invokes minimax with alpha-beta and `nil` invokes minimax. As usual, you should strive for a concise, easy to understand functional decomposition.

3.1 Implementing the basic minimax algorithm

Read and understand Russell and Norvig's treatment of the minimax algorithm in the text (Secs. 6.1-6.2; pp. 161-167), and consult your lecture notes as necessary.

- Implement full minimax by filling in the function `compute-move`.
- If ties result in evaluating states in the game tree, you should break them to the left.

The main idea is simple: take the current state, generate successor states by trying all possible moves, evaluate them, and pick the move that leads to the most attractive successor state. Of course, the tricky bit is evaluating the successor states. One does this by putting oneself in the opponent's position, and assuming that the opponent will always pick the move that leads to a successor state that is most favorable to her.

Take the following hypothetical endgame situation as an example:

```
0 1 2 3 4 5 6
_ _ _ X O X _
O X X O O X O
O O X X O O X
O X O O X X O
X O X X X O X
X O X O X O X
```

> If the current player is X, and the opponent is O, X might think along the following lines: "If I move to column 2, O will only have three possible moves: columns 0, 1 and 6. If she moves to column 0, she wins. If she moves to column 1, I can move to column 0, and the game will end in a draw. If she moves to column 6, I can move to column 1, and win. So she will definitely move to column 0 because that way she'll win, hence I must not move to column 2 in the first place...,"and so on.

To put this in more formal terms, let us assign a value of -1 to a state that leads to O's victory, +1 to a state that ensures X's triumph, and 0 to a state that leads to a draw. In the example just cited, X is trying to evaluate the state resulting from his moving to column 2. The value of this state is going to be the smallest value (i.e. the *minimum*) of the values of its successor states, because X is assuming that O will pick the move leading to a state that is least favorable for X. The possibilities are -1 (if O moves to column 0), 0 (if O moves to column 1), and +1 (if she moves to column 6). X assumes that O will pick the minimum of these values, so the state resulting from X's move to column 2 will get a value of -1, and X will definitely keep looking for a better solution. After trying moves to columns 0, 1 and 6 as well, and evaluating the resulting states, X will end up picking the move that leads to the state with the highest (i.e. *maximum*) value.

Minimax is an inherently recursive algorithm, and you should implement it as such. The most straightforward way to do this is to write a recursive helper function that uses minimax to find the value of the current state, using `do-move` and `undo-move` as appropriate and have `compute-move` call this helper function. Note, however, that if you were to implement Minimax as just described, you wouldn't want to test your move generator starting with an empty grid since, due to the size of the game tree, you might have to wait hundreds of thousands of years before getting an answer! Something like the above example would constitute a more realistic test.

Please Note: In this assignment you will ultimately be implementing a depth-limited Minimax algorithm capable of using alpha-beta pruning. In order to do this you will have to extensively modify the generic minimax implementation. If you would like to incorporate these changes (sections 3.2 and 3.4) from the start, you may certainly do so.

3.2 The Static Evaluator

Naive minimax always works in principle, but fails miserably on large game trees in practice. To address this problem, perform the following:

- Modify your move generator to keep track of the search depth, and have it stop generating new successor states after the depth limit (i.e. the value of the `depth-limit` field in the `search-info` structure of the current game) has been reached. Then, have the move generator call a *static evaluator* helper function to estimate the value of the current state, and return this value.

The speed with which your program plays will depend largely on how efficient the static evaluator is. For some games, writing an evaluator is trivial; e. g. for tic-tac-toe, checking whether the middle square on the 3x3 board is occupied by X, O, or neither, and assigning the state a value of +0.5, -0.5, and 0, respectively, is perfectly sufficient. For other games such as chess or Go, however, good evaluators are few and far between.

It turns out that the following simple evaluation function for Connect Four (based on R. L. Rivest, *Game Tree Searching by Min/Max Approximation*, AI 34 [1988], pp. 77-96) performs surprisingly well:

```
a win by X has a value of +512,
a win by O has avalue of -512,
a draw has a value of 0,
```

otherwise, take all possible straight segments on the grid (defined as a set of four slots in a line–horizontal, vertical, or diagonal; see Figure 2), evaluate each of them according to the rules below, and return the sum of the values over all segments, plus a *move bonus* depending on whose turn it is to play (+16 for X, -16 for O), as depicted in Figure 2.

The rules for evaluating segments are as follows:

```
-50 for three Os, no Xs,
-10 for two Os, no Xs,
- 1 for one O, no Xs,
  0 for no tokens, or mixed Xs and Os,
  1 for one X, no Os,
 10 for two Xs, no Os,
 50 for three Xs, no Os.
```

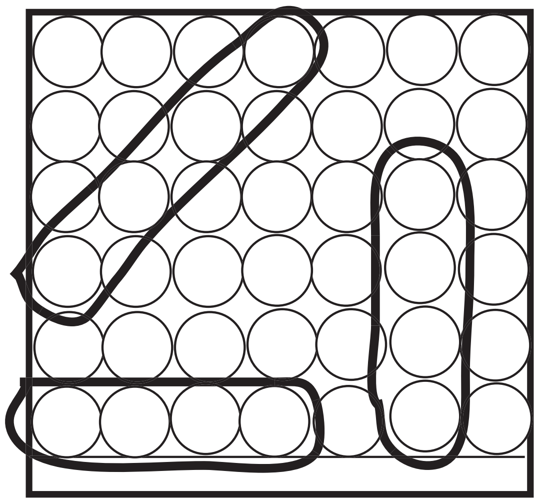

Figure 2: Sample Segments: Evaluation must be over all such segments

We have included a sample call to the evaluator that specifies the expected value in `select-move.lisp` to help you debug your evaluator function.

While this evaluation function is guaranteed to work, it is not necessarily the best or the easiest to implement efficiently. Notably, it doesn't favor quicker wins. That is, if X can win in both 23 and 17 moves, there is no incentive for him to win faster.

After having made the appropriate additions and modifications to your `compute-move` function, you should be able to have the computer generate reasonable moves in any game situation.

3.3 Bookkeeping

By now you will no doubt have noticed that every time the computer displays the state of the grid following a move, it also outputs information pertaining to search statistics. In particular, it is supposed to give the number of moves considered, the average branching factor and the maximum search depth reached. You have to make this information available to the system by updating the `search-info` structure as follows: whenever you look at a new search node, increment `moves-considered`, check whether it is at a greater depth than the current value of `depth-reached`, and if so, set `depth-reached` to this value. The node of the current game state (i.e. the root node) is defined to have a depth of 0. Any child node resulting from a move will then be at a depth of 1, and the children of these nodes would be at a depth of 2. Also, each time you expand a node, count the number of valid successor nodes, and push the result onto the list of branch-counts in the `branching-factors` field.

Note that if you are doing everything correctly, the branching factor should always be equal to the number of columns (at least in the early stages of the game, when no columns are full), while the maximum search depth should always be equal to the depth cutoff value (except at the end, when there are fewer moves than this value).

3.4 Alpha-Beta Pruning

The final improvement you will make to the move generator is the addition of *alpha-beta pruning.* To implement this, you will need to make significant changes to your minimax function, since in order to perform alpha-beta pruning, you'll have to keep track of the best moves found by both players at every level in the search tree. As you expand nodes at any given level, you will have to keep updating the best value found by the current player so far (say, `alpha`). Once this value reaches the best value found by the opponent at the *previous* level (in this case, `beta`), you may abort the search at the current level, and return the current value, because you know that the opponent won't try this course of action anyway.

- Modify your function to include alpha-beta-pruning. Russell and Norvig describe this process on pages 167-171. You will also find the algorithm on page 170 useful.
- One again, if ties result in evaluating states in the game tree, you should break them to the left.

In addition to the bookkeeping you are already doing, every time you stop expanding nodes at a given level because of alpha-beta pruning, increment the `search-info` field `moves-pruned` by the number of nodes left unexpanded at this level. You might be surprised at how large this number gets in comparison to the total number of hypothetical moves made! Herein lies the power of alpha-beta pruning...

Hint - If you run into problems debugging your alpha-beta pruning procedure, try running your code on a small tree with a small depth-limit, and have it print out the nodes and alpha and beta values as it goes.

4 Analysis

Now that your code is working, fill in the table in `analysis.txt`. You will play the computer against itself using various search depths and toggling alpha-beta on and off. Recall you can toggle the depth search by writing `(start-new-game :level 5)`, setting the search depth to 5 in this case. You can set the `*suppress-output-p*` flag to true to display only game summary information and suppress individual move output. To calculate time, use the builtin `time` function, e.g. `(time (finish-game))`. Feel free to use `analysis.in` to run these tests; however, be aware that you may need to comment out higher level runs if they do not finish in time.

5 Finishing Up

When you are done, use `(run-file-to-file "connect-four.in")` to test your move generator; output should appear in the file `connect-four.out`. Submit your solutions using make submit, as usual. You must submit *electronically* by Wednesday, October 7 at 2:00 pm.