CSC442: Intro to AI
Project 1: Game Playing

September 15, 2019

This project is about designing, implementing, and evaluating an AI program that plays a game against human or computer opponents. You may be able to build a program that beats you, which is an interesting experience. You may find that that careful programming can mean the difference between programs that take seconds, minutes, or hours to respond. The project has three components:

1. Design and development of your game playing program.
2. Experimentation and evaluation of your game playing program.
3. A project writeup.

Mancala

For this project, you will be implementing a program to play the game Mancala. Mancala is an ancient board game with many variants. We will be foucsed on the popular (in the US) Kalah version. For details, check out: . Briefly, this variant of Mancala is a two-player board game where each player has a row of M pits where each *pit* is initially filled with K *stones*. For this project, we will use the standard setup of six pits each initally containing four stones. *(You may hardcode this, but you might be well-served to represent these numbers as global constants.)* To each player's right on the board is a *store*. The objective for each player is to accumulate as many stones as possible into their own store.

Each turn proceeds by the active player choosing a pit, then removing all the stones, then *sowing* the stones into pits counter-clockwise, including the store and opponents pits if enough stones are in the original pit. If the last stone is sowed into the store, then the player gets to take another turn. If the last stone is sowed into one of the player's own empty pits, then the player *caputures* any stones in their opponents pit directly across the board. Play continues until either player runs out of stones.

Program Design and Implementation

You must design your program so that we may test it according to the following behavior: Your program must accept at least two commandline arguments, which will represent the first and second players, respectively. For example, to play a game against a random opponent, where you want to be the second player, invoke the program as "`./play random human`". The general syntax is "`./play player1 player2`" where `player1` and `player2` are any two of the following strategies: *random*, *minimax*, *alphabeta*, *human.*

Before play, the game state is initialized to the new game state of mancala. Each turn proceeds as follows:

1. the current player is prompted for their move by calling the appropriate function, and passing it the current game state: `evaluate(playerFn, state)`.

2. The chosen move is then printed to standard out. For player one, the move and a newline are printed. Player two's moves should be preceeded by a space.

   For example, if player one chooses to sow the stones from pit 4 as their first move, then player two sows stones from the first pit the output would be: "4\n 1\n".

3. After printing the move, the board state is printed to standard error. If it is a human players move, then the human player function must prompt the player to enter a move by printing a prompt to standard error.

If at any point an illegal move is selected by a human, the message "Illegal move" should be printed to standard error, and either the human prompted to enter another move, or the game terminated. We will not enter illegal moves during testing, but you should try them while debugging.

When the game is over, one of the following three messages should be printed to standard out:

- `Player 1 wins!`
- `Player 2 wins!`
- `Draw.`

The program should terminate after printing this final message.

Display

Note that displaying the game state is meant to make your game fun to play (or at least possible). You are welcome to use any design you prefer; however, as a default we suggest five lines of text, with the top line being player 1's board and the bottom line being player 2's board. You are welcome (and encouraged!) to use different visual indications of state! You may use more or fewer lines, and other kinds of ascii designs. The goal is to develop an easy to use visual representation of a mancala board, in text.

```
    --------------------------------
 00 | 04 | 04 | 04 || 04 | 04 | 04 |
----------------------------------------
    | 04 | 04 | 04 || 04 | 04 | 04 | 00
    --------------------------------
```

You may also want to remind the player of the move numberings:

```
       6    5    4     3    2    1
    --------------------------------
 00 | 04 | 04 | 04 || 04 | 04 | 04 |
----------------------------------------
    | 04 | 04 | 04 || 04 | 04 | 04 | 00
    --------------------------------
       1    2    3     4    5    6
```

Or you could also include an indication of the players:

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

or, depending on if there is a human player, and if they are player one or two:

```
       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
```

.

State-Space Representation

You should carefully design your state-space representation. You will need to consider each of the following:

- State: number of pits, number of stones in each pit and store, current player
- Actions: choose a non-empty pit (most likely you should represent using an integer or character)
- Transition model: Takes a state, distributes stones, and toggles which player is next to play.
- Initial state: four stones distriubuted into each of six pits
- Terminal states: either player is out of stones

Once you have the representation, you *must* use the state-space paradigm to select moves for the computer player. I strongly recommend that you begin by implementing the random and human players, and make sure that you can play a legal game of mancala with your system. For the real core of this project, you will be implementing the *minimax* search algorithm. Additionally, you will extend it by incorporating *alpha-beta pruning.*

We may discuss program design considerations in class. Regardless, if you are careful then your program will have a "pluggable" search strategy, and you can easily swap other techniques and experiement with minimal engineering overhead. I recommend that you consider writing functions which accept a game state as input and return a move – even for the human strategy – then simply loop by calling the appropriate player's function until the terminal test is passed, at which point a winner is announced.

Experiments and Evaluation

You are also asked to evaluate the performance of your implementation through a series of three experiments.

- Investigate skill of minimax. You should do this by pitting your minimax implementation against random strategies and establishing win/loss/draw rates. You are encouraged to play against your random choice and minimax algorithms as well and determine your win/loss/draw rates. You should also pit minimax against alpha-beta and determine if one strategy seems to play at a higher skill level than the other, or if they are similar.

- Investigate computational effectiveness of alpha-beta pruning. You will do this by playing minimax vs random games and alphabeta vs random games and timing them. Your objective is to establish an approximate speedup for alpha-beta pruning over minimax.

- Choose either of two options:

  - Investigate first-choice bias and the pie rule. This requires to you to do additional experiements (random vs random, minimax vs minimax) to establish whether or not the first-player seems to have an unfair advantage over the second player in Mancala.

  - Instrument your code to numerically explore states expanded per second. You should calculate the total number of states expanded by minimax, the total expansions by alpha-beta, and use these numbers to calculate an effective branching factor for alpha-beta vs minimax.

Writeup

Lastly, you must produce a short PDF writeup with at least the following four sections:

1. Program design (describe the overall architecture of your program, describe state/action representations, turn taking, handling equivalent moves (random vs fixed))
2. Member contributions (describe each team member's contributions to the project)
3. Experiments
4. Discussion (what you learned, did it work, was anything surprising, most interesting challenge, etc.)

Your complete writeup should probably be in the range of 3-6 pages total.

Submission Instructions

Upload a ZIP archive with everything (source code and writeup) to BlackBoard before the deadline.

Place all your source code into a directory called `src`. You must also include a plain-text `README` file in your src directory which describes how to build and run your program, and includes the names and UR NetID's for each contributing team member. Zip your `src` directory, `README`, and PDF writup into a single file and upload to blackboard.

Submissions received by 11:59PM Sunday, September 29th will receive a flat 5% bonus credit. Submissions will be accepted until 11:59PM, Wednesday, October 2nd.

Grading

Amount	Category	Subcat
10	Overall Simulation	
	10	Random Player
	10	Human Player
40	AI Functions	
	20	Minimax Player
	20	Alpha-Beta Player
26	Writeup	
	10	Design
	10	Member Contributions
	6	Experiment Organization
24	Experiments	
	8	evaluation of minimax skill
	8	evaluation of alphabeta speed
	8	your choice