CS 540: Introduction to Artificial Intelligence

Homework Assignment #2

Assigned: Tuesday, February 9
Due: Monday, February 22

Hand-In Instructions

This assignment includes written problems and programming in Java. **The written problems must be done individually but the programming problem can be done either individually or in pairs.** Hand in your work electronically to the Moodle dropbox called "`HW2 Hand-In`". Your answers to each written problem should be turned in as separate pdf files called `P1.pdf` and `P2.pdf` and put into a folder called `<wisc username>-HW2` For example, if your wisc username is `jdoe`, then put the pdf files in a folder called `jdoe-HW2` If you write out your answers by hand, you'll have to scan your answer and convert the file to pdf. Put your name at the top of the first page in each pdf file. **Both people on a team *must individually answer* and submit their answers to the written problems; no collaboration on these problems is allowed!**

For the programming problem, only one person in a pair needs to turn in their code. The person who turns it in should copy a file called `xxxPlayer.java` *and any helper classes you and your partner wrote* (but not other supplied files) into the same folder called `<wisc username>-HW2` For example, if your wisc username is `jdoe`, then you should put a file called `jdoePlayer.java` into the folder called `jdoe-HW2` Also put a file called `README.txt` in this folder that says who your partner is, including both their real name and their wisc username.

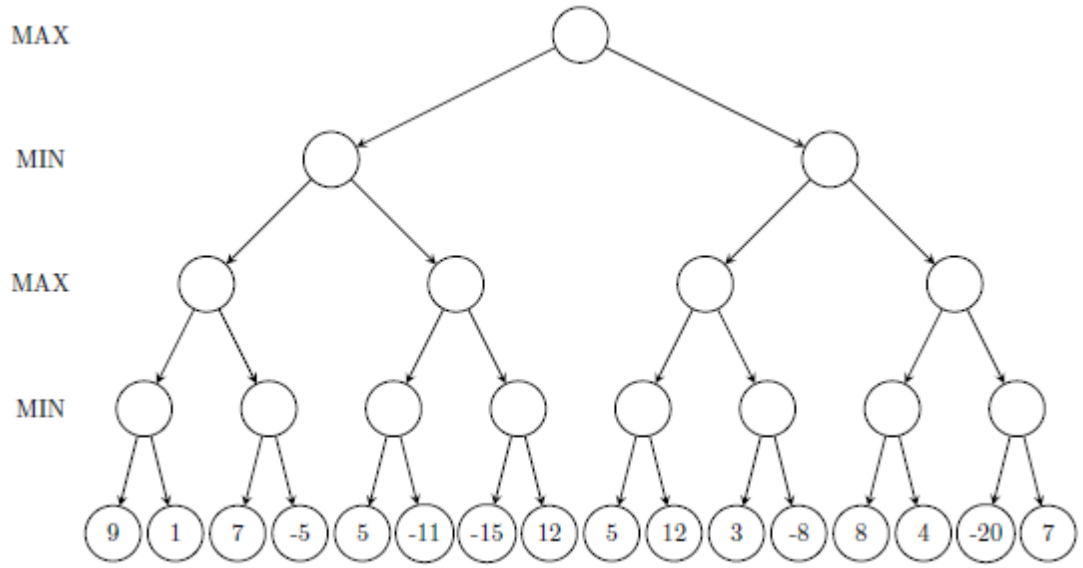
Each person on a team should compress their own folder to create `<wisc username>-HW2.zip` and copy this one file into the Moodle dropbox.

Late Policy

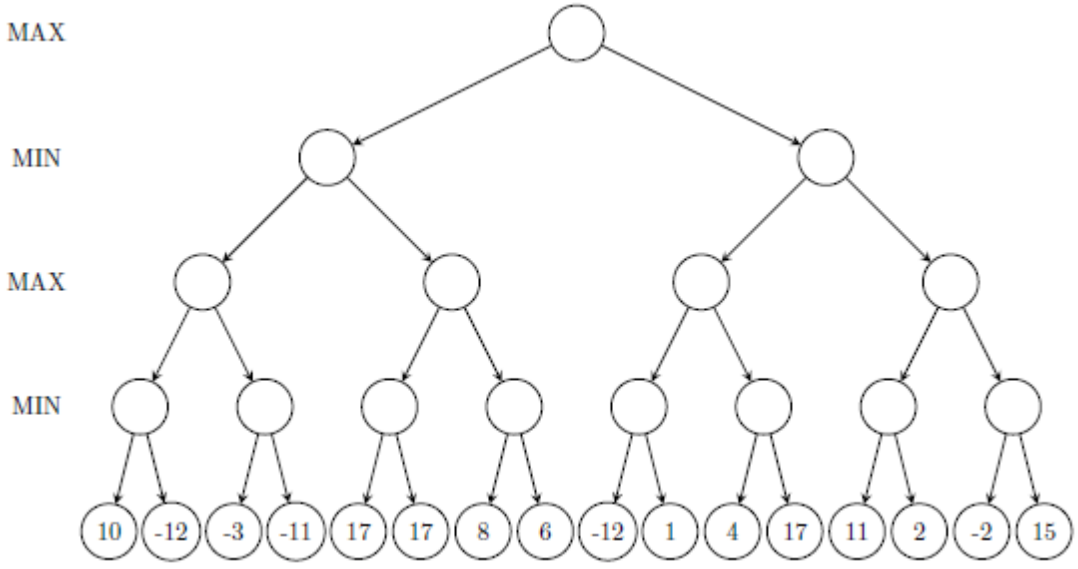
All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be raised with the instructor or a TA within one week after the assignment is returned.

Problem 1. [15] Minimax and Alpha-Beta

(a) [6] Use the Minimax algorithm to compute the minimax value at each node for the game tree below.



(b) [9] Use the Alpha-Beta Pruning algorithm in the textbook to compute the minimax value at each node for the game tree below, assuming children are visited left to right. Also show the final alpha, beta and *v* values computed at each node. Show which branches are pruned.



Problem 2. [20] Hill-Climbing

We would like to solve the Boolean Satisfiability problem (SAT) using a greedy hill-climbing algorithm. Each state corresponds to a complete assignment of *true* or *false* to each Boolean variable. The successor operator, *Successors*(*s*), generates all neighboring states of *s*, which we define as all total assignments that differ by exactly one variable's truth value. So, for example, given the state with assignments {A = *true*, B = *false*} , the neighboring states would be {A = *false*, B = *false*} and {A = *true*, B = *true*}. Define the evaluation of a state to be the number of clauses that are satisfied given the assignments at the state. A clause is defined as a disjunction (ORs) of variables. Assume that ties in the evaluation function are broken randomly and we do not perform any repeated state checking for successor generation.

(a) [2] Given *n* Boolean variables, how many neighboring states does the *Successors*(*s*) function produce?

(b) [2] What is the total size of the search space, i.e., how many possible states are there in total? Assume again that there are *n* Boolean variables.

(c) [8] Consider the following problem containing 5 clauses (in parentheses) and 4 variables, where A is shorthand for A = *true* and ¬A is shorthand for A = *false*:

$$(A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (B \vee C) \wedge (B \vee \neg C) \wedge (\neg B \vee D)$$

From the starting state (A = false, B = true, C = false, D = false), what is the next state reached by hill-climbing, or explain why there is no successor state. Will a global optimal solution be found by hill-climbing from this initial state? Briefly explain why or why not.

(d) [6] Consider the following problem containing 4 clauses and 3 variables:

$$(A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee B \vee \neg C)$$

Come up with a non-goal state (i.e. a non-satisfying assignment) that is a local optimum in the hill-climbing space.

(e) [2] Consider the case when neighboring states of *s* are defined as all total assignments that differ by up to *two* variables' truth values. How many neighboring states would the *Successors*(*s*) function produce assuming there are $n \geq 2$ Boolean variables? Briefly explain your answer.

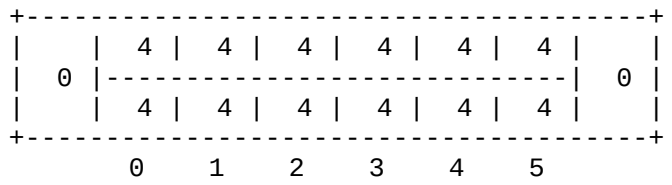
Problem 3. [65] Game Playing

This problem may be done either individually or in a team of two. Find a teammate on your own or via Piazza. Write a Java program that plays the game Mancala (also called Kalaha or Wari), which is an ancient African and Middle Eastern combinatorial game. The game is two-player and turn-based, and uses a board with 12 small *bins* (6 for each player) and two *mancalas* (the large "scoring" bins, one for each player), and a set of small stones. The 6 small bins in front of you and the mancala on your right are yours.

Mancala Resources
Odds are, some of you are unfamiliar with how to play Mancala. The rules are detailed below, but here are several versions that you can play online to familiarize yourself with the game dynamics and strategies (note that the rules for these versions might vary slightly from ours, though not by much):

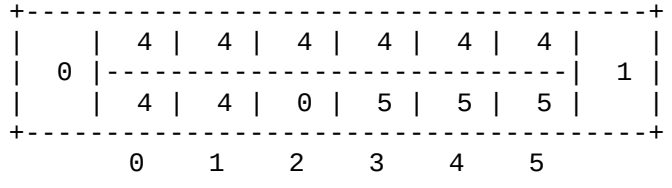
- Mancala Snails!
- Mancala Web v2.0

Rules of Play
There are many different versions of rules for this game, but the ones we will use are as follows. The game starts with 4 stones (or sometimes 3 or 5) in each small bin. The object of the game is to collect the most stones in your mancala.

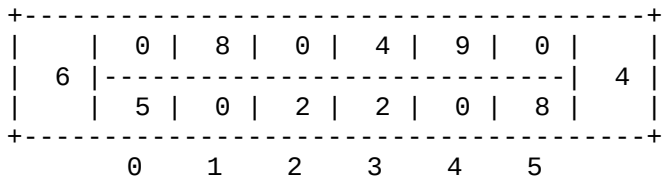


On your turn, you select one of your own bins. The bin must have one or more stones in it. You pick up all the stones in that bin and place one stone in each bin to the right (i.e., going counter clockwise). If you still have stones when you reach your mancala, then you put a stone in your mancala, and begin to place stones in each of your opponent's bins going back around. If you still have stones when you reach your opponent's mancala, skip it and proceed through your bins again, and your mancala, etc., until you run out of stones.

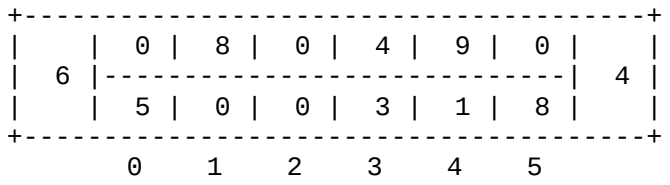
For example, if you have the first move, you could pick up the stones in bin 2, which results in this board:



Note that your last stone ended up in your mancala. When this happens, you get to take another turn. Now suppose that later on in the game, the board looks like this:



This time when you choose bin 2, you get the following board:



Note that the last stone ended up in an *empty* bin *that belongs to you*. When this happens, you get to take that stone, plus *all* of the stones in the opponent's bin on the opposite side, and put them in your mancala:

And then it is your opponent's turn. The game is over when all the bins on one side of the board are empty. All the remaining stones go into the opponent's mancala (e.g. if Min has no more stones on her side of the board, then Max gets to put all of the remaining stones in his mancala, and the game is over). The player with the most stones at the end of the game wins.

Writing a Mancala-Playing Agent

We have provided a skeleton program and a Mancala game framework, which handles all of the game board data structures, rule aspects of the game, and provides an GUI interface for playing against your agent. In this framework there is an *abstract* class defined in the file `Player.java`, which defines all the methods necessary for some agent to interface with the game framework.

- Download the skeleton code
- There is no makefile, so to compile everything, use this command:
  `% javac *.java`

Your Programming Task

Your task is to write an `xxxPlayer.java` file, where `xxx` is your wisc username (e.g., jdoe@wisc.edu's code would be written in `jdoePlayer.java` The skeleton of `xxxPlayer.java` is provided to you as `studPlayer.java` It is important that your login name appears verbatim, and "Player" is capitalized, with no dashes or underscores or anything else).

This file will implement the `xxxPlayer` class, which extends the *abstract* `Player` class. Here you will actually write code for the *abstract* methods defined in `Player.java`. There are four things required for your implementation:

1. Minimax search
2. Alpha-beta pruning
3. Time management with iterative-deepening search (IDS)
4. A static board evaluation (SBE) function

Specifically, you have to implement six functions in `xxxPlayer` class. They are:

```
1. public void move (GameState state);
2. public int maxAction(GameState state, int maxDepth);
3. public int minAction(GameState state, int maxDepth);
4. public int maxAction(GameState state, int currentDepth, int
   maxDepth, int alpha, int beta);
5. public int minAction(GameState state, int currentDepth, int
   maxDepth, int alpha, int beta);
6. private int sbe(GameState state);
```

More details on these six functions are given below.

1. `public void move (GameState state);`

This method will implement the IDS algorithm to update the protected data member `move` after each iteration of IDS. (The `getMove()` method then returns that data member to the class that is controlling the game environment.) The `maxDepth` of the IDS algorithm starts from 1 and increments by 1 at each iteration until interrupted due to the time limit. Inside each iteration, you need to do a Minimax search with `maxDepth`.

2. `public int maxAction(GameState state, int maxDepth);`

This is a wrapper function for Minimax search for ease of use. The caller of this function should be the Max Player. The detailed descriptions of input and output are given below:

- `@GameState state`: The game state for the current player
- `@int maxDepth`: The maximum depth you can search in the search tree
- `@return`: Return the best step that leads to the maximum SBE value

3. `public int minAction(GameState state, int maxDepth);`

This function is similar to `public int maxAction(GameState state, int maxDepth)` except this function returns the best step for the Min Player.

4. `public int maxAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);`

This function will actually do the Minimax search with Alpha-Beta pruning. The detailed descriptions of input and output are given below:

- `@GameState state`: The game state we are currently searching
- `@int currentDepth`: The current depth of the game state we are searching
- `@int maxDepth`: The maximum depth we can search. When current depth equals `maxDepth`, we should stop searching and call the SBE function to evaluate the game state
- `@int alpha`: This variable is for alpha-beta pruning, which should be self-explanatory
- `@int beta`: This variable is similar to alpha
- `@return`: Return the best step that leads to the child that gives the maximum SBE value; return the step with the *smallest index in the case of ties*

It is important to note that we will also call the SBE function to evaluate the game state when the game is over, i.e., when someone has won the game.

5. **`public int minAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);`**

This function is similar to `public int maxAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta)` except this function returns the best step for the Min Player.

6. **`private int sbe(GameState state);`**

This function takes a game state as input and returns its SBE value. You may implement a simple SBE method as follows: Return the number of stones in the mancala of the current player minus the number in the mancala of the opponent. You may want to design a better SBE if you want to gain more points for the homework.

Running the Program

In the provided framework is the `Mancala` class, which has the main method and allows you to just play the game, play against your agent, or pit two agents against each other by using the command:

```
% java Mancala (xxxPlayer) (xxxPlayer)
```

The arguments are the agent class names. With no parameters, it is a two human player game. With one argument, you play against the provided agent, and with 2 arguments, the agents will play each other. The match gives the agents a 10-second time limit by default in which to decide on moves, so alpha-beta pruning and IDS are going to be essential in order to search for good moves. (The search may be interrupted by the game environment when the time limit up, which is why you store the best move so far in the data member `move`.) To be fair, though, a human player gets 10 seconds to move as well (or an arbitrary move is made for you).

TIPS:

- Start early!
- You will probably want to refer to the pseudocode on page 170 in the textbook but you will need to add a depth limit (as in the example from class) for IDS purposes.

- The `GameState` class has a copy constructor and an `applyMove()` method. You'll want to use these.
- For development purposes, you may alter the 10-second time limit and the 4 initial stones in each bin by editing `TIME_LIMIT` and `NUM_STONES` data members in `Mancala.java` We will be testing and grading using various time limits and numbers of stones.
- We recommend that you do *not* modify the files we've provided. Your agent *must* be compatible with the framework for grading.
- All you need to submit are `xxxPlayer.java` and any additional helper classes you may have written.

Grading

Your agent will play against several of our own agents and your grade will be based on the match results. More specifically:

- If your agent can beat our random agent, which makes a random move at each step, you can get half the total points.
- If your agent can beat our simple agent, you can get 100% of the points.
- If your agent can beat an advanced agent, you can get 100% of the points plus a bonus.

Have fun!