CS 540-2: Introduction to Artificial Intelligence

Homework Assignment #2

Assigned: Thursday, February 15
Due: Sunday, February 25

Hand-in Instructions

This homework assignment includes two written problems and a programming problem in Java. Hand in all parts electronically to your Canvas assignment HW#2 page. For *each* written question, submit a single **pdf** file containing your solution. Handwritten submissions *must* be scanned or converted to pdf. **No photos or other file types allowed**. Each file should have your name at the top of the first page. For the programming problem, submit *just* the `studentAI.java` file and do *not* modify any of the other provided code.

You should submit the following three files (with exactly these names) for this homework:

```
problem1.pdf    problem2.pdf    studentAI.java
```

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 discussed with a TA within one week after the assignment is returned.

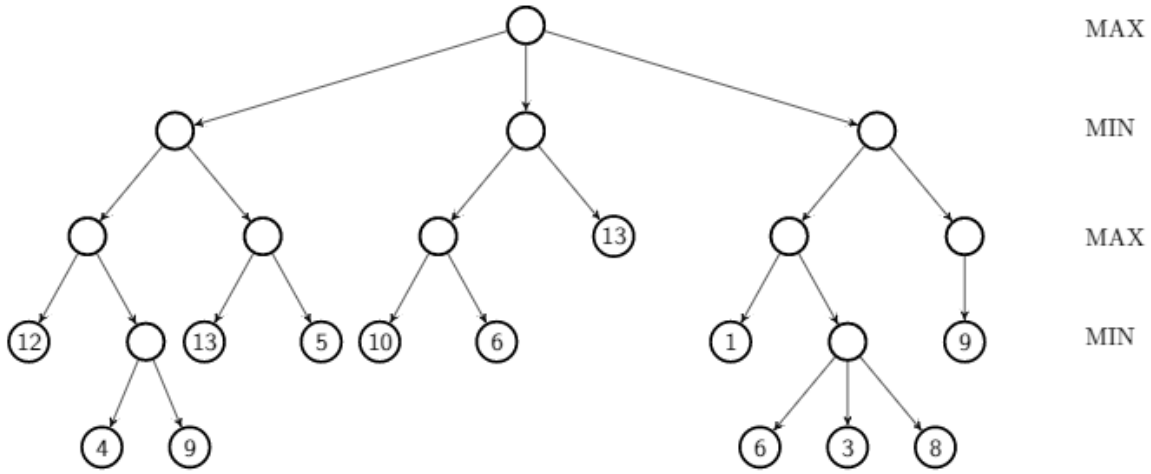
Collaboration Policy

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, peer mentors and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

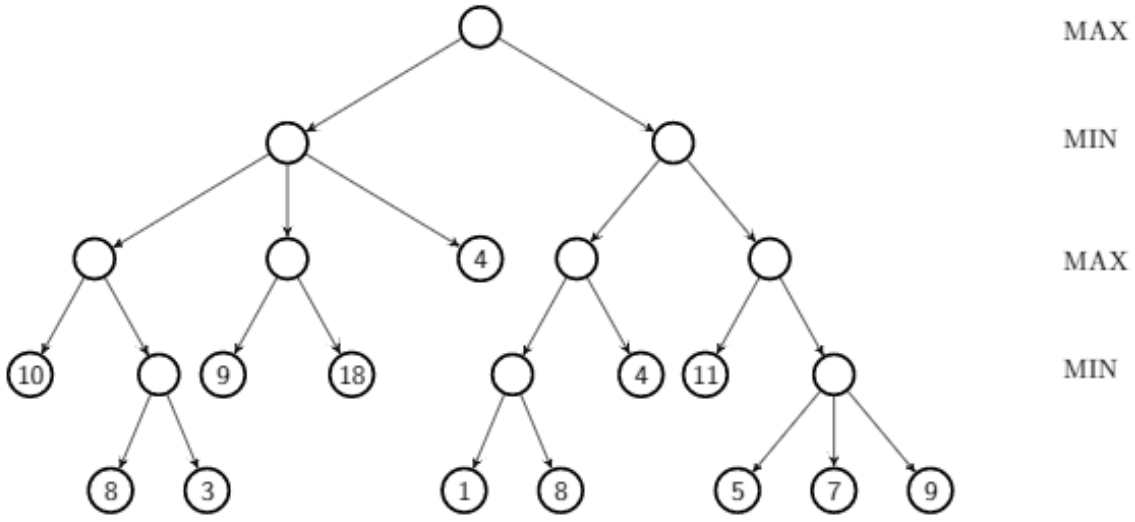
- not explicitly tell each other answers
- not copy answers or code from anyone or anywhere
- not allow your answers to be copied
- not get any code from the Web

Problem 1: [15] Minimax and Alpha-Beta

(a) [5] Use the Minimax algorithm to compute the minimax value at each node for the game tree below.



(b) [7] Use the Alpha-Beta pruning algorithm to prune the game tree below, assuming children are visited left to right. Show the final alpha and beta values computed at each internal node, and at the top of pruned branches. ***Note*: Follow the algorithm in Figure 5.7 in the textbook or, equivalently, the pseudocode on page 11 of the game playing lecture notes**. Different versions of the algorithm may lead to different values of alpha and beta.



(c) [3] For a general game tree (i.e. not limited to the above two trees), are there any cases that the Alpha-Beta algorithm makes a *different* move than the Minimax algorithm? If yes, show an example; if no, explain briefly why not.

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 T or F 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 assignment of values associated with the state. This algorithm is usually called GSAT. Assume that ties in the evaluation function are broken randomly.

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

(b) [3] What is the total size of the search space, i.e., the total number of states in the sample space? Assume again that there are *n* Boolean variables.

(c) [10] Consider the following problem containing 6 clauses (in parentheses) and 5 variables:

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

From the starting state (A=F, B=F, C=F, D=F, E=F), what is the *final state* reached by hill-climbing? At each step show all the successors and their corresponding evaluation scores. Will a global optimal solution be found by hill-climbing from this initial state?

(d) [5] Consider the following problem containing 4 clauses and 3 variables:

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

Find an initial state assignment that is a non-goal state (i.e., a non-satisfying assignment) that is on a *plateau* in our hill-climbing space. Also prove that this initial state is at a plateau by showing the evaluations for its successor states.

Problem 3: [65] Awari

Implement the Alpha-Beta pruning algorithm to play a two-player game called Awari. Follow the algorithm pseudocode in the lecture notes and in the textbook in Figure 5.7.

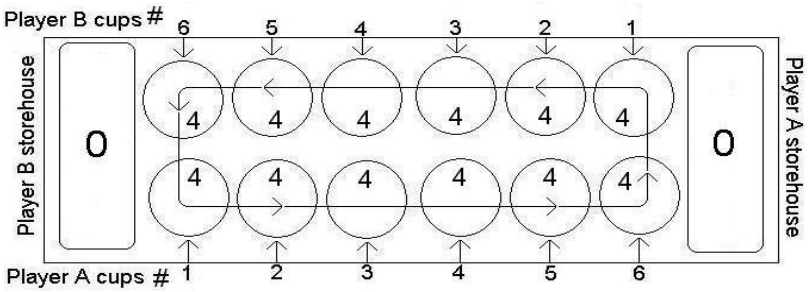
RULES

Awari is an Ashanti abstract strategy game in the Mancala family of board games (pit and pebble games) played worldwide. Here we will use slightly modified Abapa rules set out by Awari International for tournaments. They are as follows:

The objective is to win stones by placing your last stone being moved into a pit of your opponent that already contains one or two stones, making the final count two or three, and then capturing and placing these stones into your storehouse.

Board and initial setup

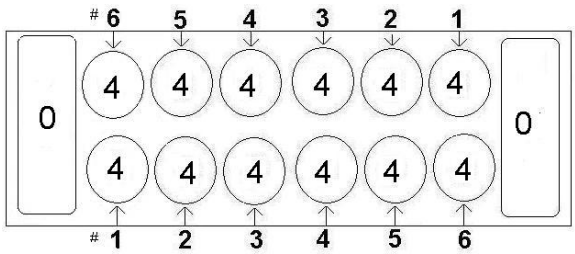
The board is comprised of twelve pits. Each side belongs to one player. The numbers in each pit represents the number of stones in that pit.



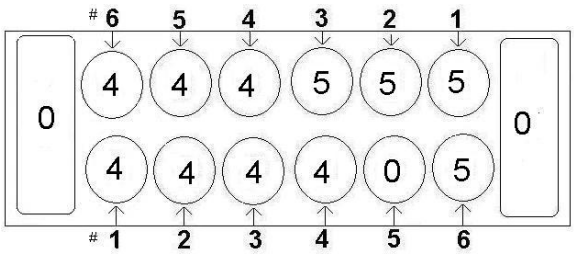
The direction of play is counterclockwise, moving left to right around the board in a circle. **Each player has six pits** on one side of the board and a storehouse to their right, which is for storing stones won during the game. **To start the game**, place a constant number (usually 4) stones into each player's pits, and none in the storehouse.

Rule 1: MOVING On their turn, the player chooses any pit on their side, removes all the stones in that pit, and, starting at the next pit to the right moving counterclockwise, puts one stone into each pit that comes next without skipping any pits, except the pit moved from and the storehouses. For example:

Before

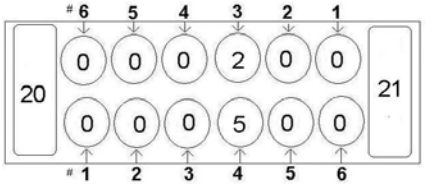


After

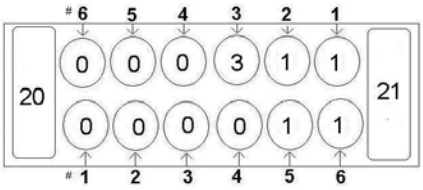


Rule 2: WINNING STONES IN ONE PIT If a player places their *last* stone into a pit on the opponent's side that makes the new count in that pit either 2 or 3 stones, the player wins these stones and places them in their storehouse. For example:

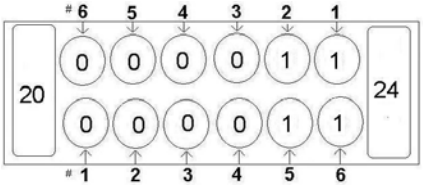
Before move



After move, before capture



After capture

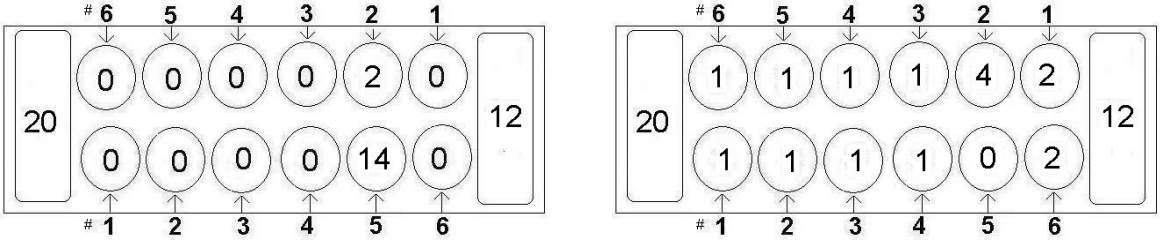


Rule 3: WINNING STONES IN MULTIPLE PITS If the last stone does capture the pit's stones, then the previous pit is checked. If that pit, too, has a count of 2 or 3 stones, then those stones are also placed into the current player's storehouse, and the pit previous to it is checked next. This capturing continues until a checked pit cannot be captured, or it resides outside your opponent's set of pits.

Important Note: No capture is possible if it would leave an opponent with *no* stones in any of their pits. **That is, you cannot entirely wipe-out an opponent in one move** (known as a grand-slam). The move is allowed, but no capture will occur.

Rule 4: SKIPPING PITS If a player chooses a pit containing 12 or more stones, the player will skip the pit they started from and continue to the next pit. For example:

Before move **After move**



Rule 5: MANDATORY MOVE If any player has six empty pits, the next player *must* move at least 1 stone into the pits of the empty player if they have any pits with enough stones to reach an empty pit. If not, the game ends and the remainder of the stones are forfeit. The player with the largest number of stones in their storehouse is the winner.

The game also ends when, at any time, a player has captured a majority of the stones, which in the standard game is 25 of the 48 total stones.

Writing an Awari-Playing Agent

We have provided a skeleton program and an Awari game framework, which handles all of the game board data structures, rule aspects of the game, and provides a GUI interface for practicing. In this framework there is an abstract class defined in the file

`Player.java`, which defines all the methods necessary for an agent to interface with the game framework. Use this command to run the game:

```
% javac *.java
% java Awari <Player1Class> <Player2Class> <board.txt> [maxDepth]
```

The first two arguments are the playing agents' class names, the third argument is a text file for board setup, and the last argument is the maximum search depth for the AI (it is not required when both players are human). We have provided a human player class, `HumanPlayer`, which takes input from the GUI interface. You can use this class to help you get familiar with the game mechanism. For example, you can practice with your AI using:

```
% java Awari HumanPlayer studentAI setup.txt 15
```

The first line in the setup file defines the current scores for players 1 and 2 respectively, and the next two lines define the number of stones in each of the pits. For example, the setup file for the default initial board state is:

```
0 0
4 4 4 4 4 4
4 4 4 4 4 4
```

Note we may use different board states in testing. When the game is run with no parameters, it is a two-human-player practice game with the default initial board setup.

Your Programming Task

Your task is to write a `studentAI.java` file. The skeleton code is already provided. Write your solution assuming that you are player 1. This file will extend the abstract `Player` class. Here you will write code for the abstract methods defined in `Player.java`. There are three things required for your implementation:

1. Minimax search with alpha-beta pruning
2. Cut-off search at a fixed depth limit
3. A static board evaluation (SBE) function

Specifically, you have to implement five functions in the `studentAI` class:

```
1. public void move (BoardState state);
2. public int alphabetaSearch(BoardState state, int maxDepth);
3. public int maxValue(BoardState state, int maxDepth, int
   currentDepth, int alpha, int beta);
4. public int minValue(BoardState state, int maxDepth, int
   currentDepth, int alpha, int beta);
5. private int sbe(BoardState state);
```

More details on these five functions are given below.

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

This is a wrapper function for alpha-beta search. It should use alpha-beta search to update the data member `move` (which will be returned by the `getMove()` method to the `Match` class that is controlling the game environment). Since the whole search space for Awari is extremely large, you need to cut off the search at some fixed depth limit, which is specified by the `maxDepth` class member. There is a **10-second time limit** to calculate each move. You can assume `maxDepth` will not be greater than 15.

2. `public int alphabetaSearch(BoardState state, int maxDepth);`

This function will start the alpha-beta search (see Figure 5.7 in the textbook for reference). The detailed descriptions of input and output are given below:
*@param **state*** The board state for the current player (the MAX player). You can assume the pits for the current player are always in the lower row, and that the lower row is player 1.
*@param **maxDepth*** The maximum search depth allowed.
@return Return the minimax value associated with the best move for the current player; returns the move with the *smallest* index in the case of ties. The value of the move should be in the range [0, 5], with 0 representing the leftmost pit.

3. `public int maxValue(BoardState state, int maxDepth, int currentDepth, int alpha, int beta);`

This function will search for the minimax value associated with the best move for the MAX player. The search should be cut off when the current depth equals to the maximum allowed depth. 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. The only condition for determining a leaf node (besides having reached maximin depth) is that there are no legal moves for the player to make, effectively ending the game at that state. The detailed descriptions of input and output are:
*@param **state*** The game state that the MAX player is currently searching from
*@param **maxDepth*** The maximum search depth allowed
*@param **currentDepth*** The current depth in the search tree
*@param **alpha*** The α value
*@param **beta*** The β value
@return The minimax value corresponding to the best move for the MAX player

4. `public int minValue(BoardState state, int maxDepth, int currentDepth, int alpha, int beta);`

This function is similar to `maxValue` except this function returns the best value for the MIN player.

5. `private int sbe(BoardState state);`

This function takes a board state as input and returns its SBE value. Use the following method: Return the number of stones in the storehouse of the current player minus the number of stones in the opponent's storehouse. Always assume the current player is player 1.

BoardState class

In your implementation you will be working with the `boardState` class. The following public members and methods will be of interest to you, and we highly recommend that you use them:

1. `int[]` **`score`**

This is an array with two indices. `score[0]` contains the number of stones in player 1's storehouse, and `score[2]` contains the number of stones in player 2's storehouse.

2. `boolean` **`isLegalMove`**`(int player, int move)`

This method *returns* a Boolean that indicates whether or not a proposed move is legal. The `move` parameter is the proposed pit from which to move stones. It must be an integer between 0 and 5 inclusive, with each index representing one of the six pits you can start a move from, from left to right on the bottom row. The `player` parameter indicates the player who is proposing to make that move. It must be either 1 or 2.

3. `BoardState` **`applyMove`**`(int player, int move)`

This method similar to `isLegalMove` except that it *returns* a new `boardState` with a move applied. It does NOT check if the proposed move is legal before performing it.

4. `String` **`toString`**`()`

This method will print out the current board, with sides labeled. This method is for you to test your code.

Grading

The only thing you should submit is `studentAI.java` so do **not** modify any other files in the framework. Your AI will go through the following three tests:

1. Your `alphabetaSearch`, `maxValue`, and `minValue` functions will be unit tested, which means we will call these functions explicitly with specific parameters and examine the outputs. Be sure to follow the exact specifications defined above. These tests will be worth 30% points of the possible points for this problem.

2. Your AI will be matched against a dummy AI, which will always perform the move that has the smallest index among all possible legal moves. If your AI wins, you will earn 30% of the possible points.

3. Your AI will be matched against an advanced AI in some endgame situations, but these situations are unwinnable for the opponent if you have played every move correctly. If your AI passes these tests, you will earn 40% of the possible points.