CS 540-3: Introduction to Artificial Intelligence
Homework Assignment #2: Search and Game Playing

Assigned: Wednesday, February 11
Due: Monday, February 23

Hand-in Instructions

This homework includes written problems and a programming problem. Hand in all parts electronically by copying them to the Moodle dropbox called "`HW2 Hand-In`", including your answers to the written problems. Your answers to each written problem should be turned in as **separate pdf** files called `P1.pdf`, `P2.pdf` and `P3.pdf`. (If you write out your answers by hand, you'll have to scan your answers and convert the file to pdfs.) Put your name at the top of the first page in each file. For the programming problem, put all `.java` files (including all modified and unmodified files given in the skeleton code) in a folder file called `P4` and then put all four of these into a zipped folder called `<wisc username>_HW2.zip`

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 anytime 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 after the due date, regardless of whether any free late days are used. 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. Free late days will be used automatically on a first-needed basis.

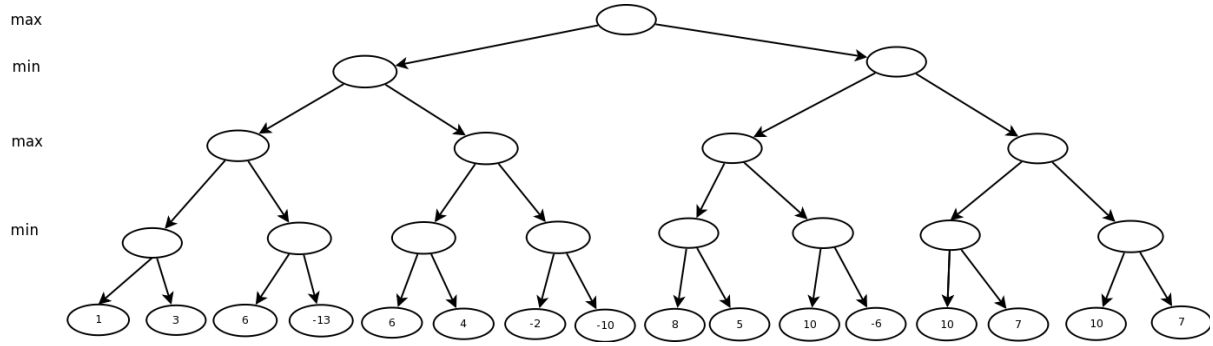
Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Collaboration Policy

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, 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 the answers
- not copy answers or code fragments from anyone or anywhere
- not allow your answers to be copied
- not get any code from the web

Problem 1: [20] Game Playing



(a) Use the Minimax algorithm to compute the minimax value at each node for the game tree above.

(b) Use the Alpha-Beta Pruning algorithm to compute the final alpha and beta values at each node in a second copy of the tree above. Also mark the pruned branches, if any. Use the Alpha-Beta algorithm given in the lecture slides, visiting children left to right.

Problem 2: [20] Hill-Climbing

If you are given a list of cities and the distances between each pair of cities, how could you find the shortest possible route that visits each city exactly once *and* then returns to the first city? This is called the travel salesperson problem (TSP). We decide to try to solve the TSP using the Hill-Climbing algorithm. Here is the set-up for the algorithm. Each state is a permutation of all the cities, called a *tour*. For example, state s = <A-B-C> means you travel from A to B to C to A. The successor operator, *Succ*(*s*), generates all neighboring states of *s* by swapping two cities. For example, if *s* = <A-B-C> is a tour, then <B-A-C>, <C-B-A> and <A-C-B> are the three neighbors generated by *Succ*(*s*). Use as the evaluation function for a state to be the total distance of the tour, including the last step of returning to the first city. Assume ties in the evaluation function are broken randomly.

(a) Is the hill-climbing algorithm guaranteed to find the minimum cost tour? If yes, explain why. If no, explain what you do find using the algorithm.

(b) How many states does *Succ*(*s*) generate assuming there are *n* cities? Explain briefly how you determined this number.

(c) Imagine some Pokemon characters are hidden in Chicago, New York, Madison, and Los Angeles. You want to visit all the cities once and return to the starting city to catch all the Pokemons. The goal is to find a tour as short as possible. The distance matrix between these cities is given in the table below.

	Madison	NYC	Chicago	LA
Madison	0	808.11	122.47	1669.74
NYC	808.11	0	711.83	2448.30
Chicago	122.47	711.83	0	1744.26
LA	1669.74	2448.30	1744.26	0

Assume the start state is: <NYC-Madison-Chicago-LA>.

i. What is the successor state of the initial state that is reached by Hill Climbing, or explain why there is no successor state.
ii. Will a global optimal solution be found using Hill Climbing from this initial state? If so, give the sequence of states. If not, explain why not.

Problem 3: [20] Simulated Annealing

(a) How could you change the Simulated Annealing algorithm so that it performs the same as Hill Climbing?

(b) With regards to Simulated Annealing, what is the probability of accepting each of the following moves? Assume the problem is trying to maximize the objective function. Use the Simulated Annealing algorithm given in the lecture slides.

Current State's Evaluation	Neighbor State's Evaluation	Current Temperature, T
16	15	20
25	13	25
76	75	276
1256	1378	100

Problem 4: [40] Take-Stones Game

In this problem you are to implement the Minimax algorithm to play a two-player game called Take-Stones. The game starts with *n* stones numbered 1, 2, 3, ..., *n*. Players take turns removing one of the remaining numbered stones. At a given turn there are some restrictions on which numbers (i.e., stones) are legal candidates to be taken. The restrictions are:

- At the first move the first player *must* choose an odd-numbered stone that is less than *n*/2. For example, if *n* = 7, the legal numbers for the first move are 1 and 3. If *n* = 6, the only legal number for the first move is 1. This number is saved as the value of `lastMove`.
- At subsequent moves each player alternates turns. The number that a player can take must be a multiple or factor of the `lastMove` (note: 1 is a factor of all other numbers). Also, this number may *not* be one of those that has already been taken. After taking a stone, the number is saved as the new `lastMove`. If a player *cannot* take a stone, she loses the game.

An example game is given below.

Sample Input
n=7

Sample Output
```
Player 1: 3
Player 2: 6
Player 1: 2
Player 2: 4
Player 1: 1
Player 2: 7
Winner: Player 2
```

Methods to be Implemented
You must implement the following five methods for the class `PlayerImpl` given in the supplied skeleton code:

```
public class PlayerImpl implements Player {
   public ArrayList<Integer> generateSuccessors(int lastMove, int []
   takenList);
   public double max_value(GameState s, int depthLimit);
   public double min_value(GameState s, int depthLimit);
   public int move(int lastMove, int [] takenList, int depthLimit);
   public double stateEvaluator(GameState s);
}
```

A description of these five methods is given below.

1. **`public ArrayList<Integer> generateSuccessors(int lastMove, int [] takenList):`**

`lastMove` is the number that was taken at the previous move (by your opponent). If it is the first move, this value will be -1. `takenList` is an integer array that lists all the numbers that have already been taken at earlier turns. If `takenList[i]`$\neq$ `0` then the number `i` has already been

taken. Thus the length of `takenList` will be (*n* + 1), with `takenList[0]` never used. You can access the number *n* from the member variable *n* of the class `PlayerImpl`. Given these two inputs, this method returns an `ArrayList` of integers that are valid numbers (i.e., stones) that can be taken.

`2. public double max_value(GameState s, int depthLimit):`
This method takes a `GameState s` and an integer `depthLimit` as its input. It returns the Minimax value with respect to the MAX player and updates the `GameState s`. More details on the input and output of this method are given next.

`GameState s`:
If you check `GameState.java`, you will find that the class has four variables. Before calling the `max_value` method, you should assign values to two of these variables, `takenList` and `lastMove`. `takenList` is the same variable described in `generateSuccessors`. The constructor of `GameState` uses a copy of `takenList` so you don't have to worry about it when you manipulate `takenList` within a `GameState` object. `lastMove` is the number taken at the previous move (by your opponent) that lead to the current `GameState s`.

Before the method returns, you should set the values of the two other variables, `leaf` and `bestMove`. The variable `leaf` identifies if the given state is a terminal state (i.e., the game is over because there are no legal moves, or equivalently, there are no successors of this node) or not. `bestMove` is the number that when taken will yield the best minimax value for the player. It should be noted that multiple numbers might yield the same best minimax value. In that case, choose the maximum of those numbers. One special case of this is if all possible moves have the same minimax value of -1, meaning they are all loses for MAX. Even in that case, assign the maximum of those numbers as the `bestMove`. Update these two variables before the method returns. If there is no possible move or the `bestMove` is unknown, set `bestMove` to -1. Hint: Pay attention to possible underflow issues since you are comparing two doubles.

`int depthLimit`:
`depthLimit` is the maximum depth that you should search. More specifically,

1. `depthLimit` = -1 means there is no limit on the search depth, so do an exhaustive minimax search until reaching terminal states. (Do this only for small values of *n*.)
2. `depthLimit` = 0 means terminate the search at the current state even if it is not a terminal state. If current `GameState` is a terminal state, return -1 as the minimax value (because MAX just lost). Otherwise, call the `stateEvaluator` method to get the estimated static game value.
3. `depthLimit` = some positive integer means keep searching below the current `GameState s`. Initialize `GameState s` and `int depthLimit` for each child (remember to decrement `depthLimit`) before calling `min_value`.

`return value:`

1. If current `GameState s` is a terminal state, return -1 (because MAX just lost).
2. If current `GameState s` is not a terminal state and we terminated the search because `depthLimit` = 0, return the value of the `stateEvaluator` method. Note: The `stateEvaluator` should always return a real value greater than -1.0 and less than 1.0.
3. If current `GameState s` is not a terminal state and we can keep searching, return the maximum value of all its children.

3. public double min_value(GameState s, int depthLimit):
Same as `max_value` except this method returns the minimax value with respect to the MIN player. It should also update the object `s` as described for `max_value`. Moreover,

1. If current `GameState s` is a terminal state, then return 1 (because MIN just lost).
2. If current `GameState s` is not a terminal state and we terminated the search because `depthLimit = 0`, return the *negation* of the value returned by the `stateEvaluator` method.

4. public int move(int lastMove, int [] takenList, int depthLimit):
This is the top-level method that is used to determine the actual next move of a player. Given the three inputs, this method returns the stone number that gives the maximum game value, i.e., the minimax value for the node. This method should use the `max_value` method to determine this (because the root node of the search tree is always the MAX player). `Int depthLimit` is the maximum depth you should search in the `max_value` method. You can just pass `depthLimit` to the `max_value` method as a parameter. Note that when `depthLimit = -1`, you should do an *exhaustive* minimax search *without* a depth limit. The other player then automatically becomes the MIN player and the `min_value` method gets called from within the `max_value` method. If no numbers can be taken (i.e., no move is possible), then the `move` method should return -1.

The `move` method is called whenever a player has to make a move. The game tree generated by this method is not saved in memory. During each call of the method, the game tree is generated again. But each game tree is rooted at a different game state during different method invocations.

5. public double stateEvaluator(GameState s):
You are to implement the following evaluation function, which should return a real value that is greater than -1.0 and less than 1.0. Positive values mean the state is good for MAX and bad for MIN, and negative values mean the state is good for MIN and bad for MAX. For extra credit you can also develop a second function that is improvement on this simple one.

1. If stone 1 is not taken yet, just return 0 because the current state is a relatively neutral one for both players.
2. If `lastMove` is 1, count the number of the possible successors (i.e., legal moves). If the count is odd, return 0.5; otherwise, return -0.5.
3. If `lastMove` is a prime, count the multipliers of that prime in all possible successors. If the count is odd, return 0.7. Return -0.7 otherwise.
4. If `lastMove` is a composite number, find the largest prime that can divide `lastMove`. Count the multipliers of that prime, including the prime number itself if it hasn't already been taken, in all the possible successors. If the count is odd, return 0.6. Return -0.6 otherwise.

The I/O parts have already been written for you. So, you are NOT responsible for any console input or output. You are only responsible for implementing these five methods.

Testing Your Code
We will test your program on multiple test cases where we will vary the inputs `n` and `depthLimit`. The format for testing is:

```
java HW2 <n> <modeFlag> <depthLimit>
```

where n is the value of *n* in the Take-Stones game, `modeFlag` is an integer ranging from 1 to 3 controlling which types of players will play the game, and `depthLimit` is the maximum depth you can search in one turn. Your program should output the moves of the players and the winner.

When `modeFlag = 1`, both players are computers and you will see a game played between them. When `modeFlag = 2`, the first player is a person and the second player is the computer. When `modeFlag = 3`, the first player is the computer and the second player is a person. We will only test for `modeFlag = 1`. The other two modes are so that you can play against your own code and see if you can win!

When `depthLimit = -1`, do exhaustive minimax search *without* a depth limit. Otherwise, it should be a positive integer that is the maximum depth of the search in one turn. So, an example command for testing the code is:

```
java HW2 7 1 -1
```

meaning there are 7 stones, computer plays computer, and all the complete search tree is generated to leaves that are terminal states where the game is over because one player has won. As part of our testing process, we will unzip the file you submit, remove any class files, call `javac *.java` to compile your code, and then call the main method called `HW2` with various parameter settings.

Deliverables

Hand in all the `.java` files you wrote or modified as well as any in the skeleton code that you did not modify. The TA will just compile all the code you submit and run our test cases.

Extra Credit

In the implementation of the `max_value` function, must you always do an exhaustive search of the *entire* game tree (i.e., there is no depth limit) of all the possible moves, with all leaves being terminal states, in order find the best move at the root (i.e., the best minimax value at the root)? If so, explain why. If not, explain under what condition(s) you can stop searching early.