CS 540: Introduction to Artificial Intelligence

Homework Assignment #2

Assigned: Tuesday, September 24
Due: Sunday, October 6

Hand-in Instructions

This homework assignment includes two written problems and a programming problem in Java. Hand in all parts electronically to your Canvas assignments page. For *each* written question, submit a single **pdf** file containing your solution. Handwritten submissions *must* be scanned. **No jpg or other file types allowed**. For the programming question, submit a **zip** file containing *all* the Java code necessary to run your program, whether you modified provided code or not.

Submit the following **three** files (with exactly these names):

```
<wiscNetID>-HW2-P1.pdf
<wiscNetID>-HW2-P2.pdf
<wiscNetID>-HW2-P3.zip
```

For example, for someone with UW NetID crdyer@wisc.edu the first file name must be: `crdyer-HW2-P1.pdf`

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.

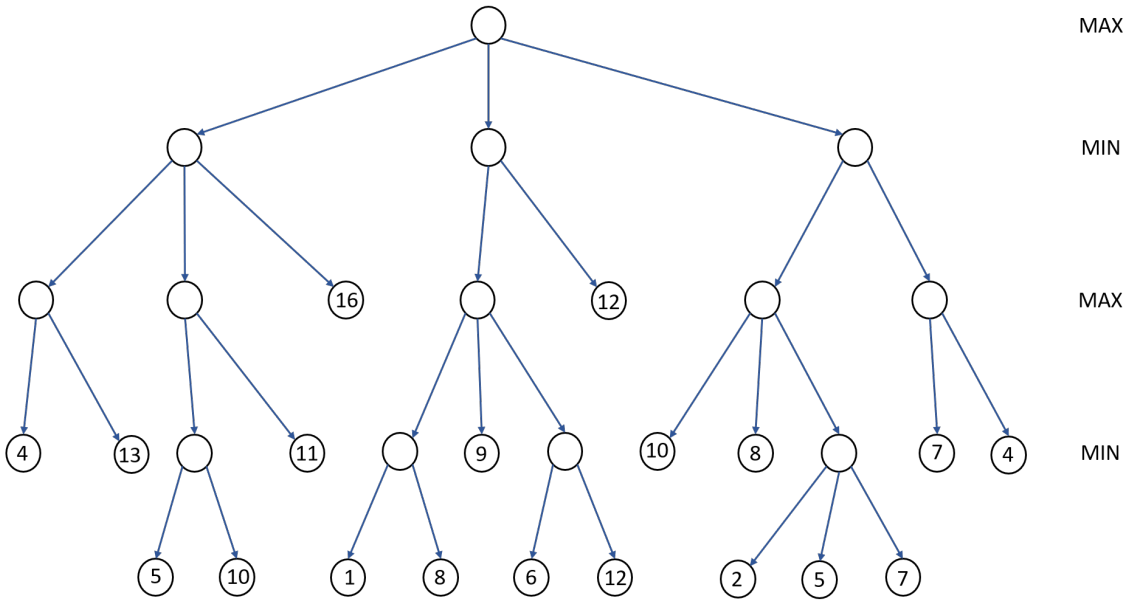
Collaboration Policy

You are to complete this assignment individually. However, you may discuss the general algorithms and ideas with classmates, TAs, peer mentors and instructor in order to help you answer the questions. But we require you to:

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

Problem 1: Minimax and Alpha-Beta [15 points]

(a) [4] 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 to prune the game tree above assuming child nodes are visited from left to right. Show all final alpha and beta values computed at the root, each internal node visited, and at the top of pruned branches. Note: Follow the algorithm in Figure 5.7 in the textbook. Also show the pruned branches.

(c) [2] For a general game tree (i.e., *not* limited to the above tree), are there *any* cases that the Alpha-Beta algorithm gives a *different value at the root node* than the Minimax algorithm? If yes, show an example; if no, just say no.

Problem 2: Hill-Climbing [20 points]

Given a set of locations and distances between them, the goal of the Traveling Salesperson Problem (TSP) is to find the shortest tour that visits each location exactly once. Assume that you do *not* return to the start location after visiting the last location in a tour. We would like to solve the TSP problem using a greedy hill-climbing algorithm. Each state corresponds to a permutation of all the locations (called a *tour*). The operator *neighbors*(*s*) generates all neighboring states of state *s* by swapping two locations. 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 *neighbors*(*s*). Use as the evaluation function for a state the length of the tour, where each pairwise distance is given in a distance matrix. For example, if *s* = <*A*-*B*-*C*> is a tour, then its total length is *d*(*A*, *B*) + *d*(*B*, *C*) where *d*(*A*, *B*) is the distance between location *A* and *B*.

(a) [3] If there are *n* locations, how many neighboring states does the *neighbors*(*s*) function produce?

(b) [3] What is the *total size* of the entire search space, i.e., the total number of states, when there are *n* locations?

(c) [14] Imagine that a student wants to hand out fliers about an upcoming programming contest. The student wants to visit the Memorial Union (M), Wisconsin Institute of Discovery (W), Computer Sciences Building (S), and Engineering Hall (E) to deliver the fliers. The goal is to find a tour as short as possible. The distance matrix between these locations is given as follows:

	M	W	E	S
M	0	1.1	1.4	0.9
W	1.1	0	0.6	0.7
E	1.4	0.6	0	0.5
S	0.9	0.7	0.5	0

The student starts applying the hill-climbing algorithm from the initial state: <*W*-*M*-*E*-*S*>

(i) [2] What is the length of the tour associated with the initial state?

(ii) [4] What are the possible neighboring states of <*W*-*M*-*E*-*S*> and what are each of their tour lengths?

(iii) [2] What is the *next* state reached by hill-climbing? Or explain why there is no next state, if there isn't one.

(iv) [6] If (iii) had a next state, continue the algorithm until it terminates, and list the sequence of states found, from initial state to final state. What is the tour length associated with the final state?

Problem 3: Take-Stones Game Playing [65 points]

Implement the Alpha-Beta pruning algorithm to play a two-player game called Take-Stones. Follow the algorithm pseudocode described in Figure 5.7 of the textbook. Different versions of the Alpha-Beta algorithm may lead to different values of alpha and beta.

Game Rules

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 strictly less than *n*/2. For example, if *n* = 7 (n/2 = 3.5), the legal numbers for the first move are 1 and 3. If *n* = 6 (n/2 = 3), the only legal number for the first move is 1.
- At subsequent moves, players alternate turns. The stone number that a player can take must be a ***multiple or factor*** of the last move (note: 1 is a factor of all other numbers). Also, this number may ***not*** be one of those that has already been taken. After a stone is taken, the number is saved as the new last move. If a player ***cannot*** take a stone, he/she loses the game.

An example game is given below for *n* = 7:

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

Program Specifications

There are 2 players: player 1 (called Max) and player 2 (called Min). For a new game (i.e., no stones have been taken yet), the Max player always plays first. Given a specific game board state, your program is to ***compute the best move for the current player*** using the given heuristic static board evaluation function. That is, only a *single* move is computed.

Input

A sequence of positive integers given as command line arguments separated by spaces:

```
java Player <#stones> <#taken_stones> <list_of_taken_stones> <depth>
```

- `#stones`: the total number of stones in the game
- `#taken_stones`: the number of stones that have already been taken in previous moves. If this number is 0, this is the first move in a game, which will be played by Max. (Note: If this number is even, then the current move is Max's; if odd, the current move is Min's)
- `List_of_taken_stones`: a sequence of integers indicating the indexes of the already-taken stones, ordered from first to last stone taken. Hence, the last stone in the list was the stone taken

in the last move. If `#taken_stones` is 0, this list will be empty. You can safely assume that these stones were taken according to the game rules.
- `depth`: the search depth. If `depth` is 0, search to end game states (i.e., states where a winner is determined).

For example, with input: `java Player 7 2 3 6 0`, you have 7 stones while 2 stones, numbered 3 and 6, have already been taken, so it is the Max player's turn (because 2 stones have been taken), and you should search to end game states.

Output

You are required to print out the following information to the console, after your alpha-beta search has completed:
- The best move (i.e., the stone number that is to be taken) for the current player (as computed by your alpha-beta algorithm)
- The value associated with the move (as computed by your alpha-beta algorithm)
- The total number of nodes visited
- The number of nodes evaluated (either an end game state or the specified depth is reached)
- The maximum depth reached (the root is at depth 0)
- The average effective branching factor (i.e., the average number of successors that are *not* pruned)

For example, here is sample input and output when it is Min's turn to move (because 3 stones have previously been taken), there are 4 stones remaining (3 5 6 7), and the Alpha-Beta algorithm should generate a search tree to maximum depth 3. Since the last move was 2 before starting the search for the best move here, only one child is generated corresponding to removing stone 6 (since it is the only multiplier of 2). That child node will itself have only one child corresponding to removing stone 3 (since it is the only factor of 6 among the remaining stones). So, the search tree generated will have the root node followed by taking 6, followed by taking 3, which leads to a terminal state (Max wins). So, returning from these nodes, from leaf to root, we get the output below.

Input:
```
$java TakeStones 7 3 1 4 2 3
```

Output:
```
Move: 6
Value: 1.0
Number of Nodes Visited: 3
Number of Nodes Evaluated: 1
Max Depth Reached: 2
Avg Effective Branching Factor: 1.0
```

All doubles/floats should be printed with 1 decimal place.

Static Board Evaluation

The static board evaluation function should return values as follows:
- At an end game state where Player 1 (MAX) wins: 1.0
- At an end game state where Player 2 (MIN) wins: -1.0

- Otherwise,
  - if it is Player 1 (MAX)'s turn:
    - If stone 1 is not taken yet, return a value of 0 (because the current state is a relatively neutral one for both players)
    - If the last move was 1, count the number of the possible successors (i.e., legal moves). If the count is odd, return 0.5; otherwise, return -0.5.
    - If last move is a prime, count the multiples of that prime in all possible successors. If the count is odd, return 0.7; otherwise, return -0.7.
    - If the last move is a composite number (i.e., not prime), find the largest prime that can divide last move, count the multiples 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; otherwise, return -0.6.
  - If it is Player 2 (MIN)'s turn, perform the same checks as above, but return the opposite (negation) of the values specified above.

Other Important Information

- Set the initial value of alpha to be `Double.NEGATIVE_INFINITY` and beta to be `Double.POSITIVE_INFINITY`
- To break ties (if any) between child nodes that have equal values, pick the smaller numbered stone. For example, if stones 3 and 7 have the same value, pick stone 3.
- Your program should run within 10 seconds for each test case.

The Skeleton Code

You can use the provided skeleton code or feel free to ***implement your program entirely on your own.*** However, you should ensure that your program:

- Has a main class named `TakeStones` (we will call `TakeStones` to test)
- Receives input from the command line as specified
- Outputs exactly match our given format

If you use the skeleton, you should implement the two methods in class `GameState`, two methods in class `Helper`, and three methods in class `AlphaBetaPruning`. You may also need to add some additional class fields or helper methods.

`Class GameState`: defines the state of a game

- `size`: the number of stones
- `stones`: a Boolean array of stones, a false value indicates that that stone has been taken. Notice that 0 is not a stone, so `stones[0] = false`, and `stones` has a size of `size` + 1
- `lastMove`: index of the last taken stone

`Class TakeStones`: the main function that parses the command line inputs and runs the search algorithm

`Class AlphaBetaPurning`: where the Alpha-Beta pruning algorithm should be implemented

`Class Helpers`: helper functions

More details can be found in the comments in the supplied code.

There are three more test cases in the `testcases` folder to help verify your implementation. The `testcase.txt` file contains the command line arguments for the three cases, and the correct outputs are in files `output[1-3].txt`

Deliverables

Put *all* `.java` files needed to run your program, including ones you wrote, modified or were given and are unchanged, into a folder called `<wiscNetID>-HW2-P3` Compress this folder to create `<wiscNetID>-HW2-P3.zip` and upload it to Canvas. For example, for someone with UW NetID crdyer@wisc.edu the file name must be: `crdyer-HW2-P3.zip` You should *not* include any package path or external libraries in your program.