CS 540-2: Introduction to Artificial Intelligence

Homework Assignment #2

Assigned: Monday, February 6

Due: Saturday, February 18

Hand-In Instructions

This assignment includes written problems and programming in Java. Hand in all parts electronically by copying them to the Moodle dropbox called "HW2 Hand-In". Your answers to each written problem should be turned in as separate pdf files called <wisc NetID>-HW2-P1.pdf and <wisc NetID>-HW2-P2.pdf (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. Copy these pdf files to the Moodle dropbox. For the programming problem, put all the java files needed to run your program, including ones you wrote, modified or were given and are unchanged, into a folder called <wisc NetID>-HW2 Compress this folder to create <wisc NetID>-HW2-P3.zip and copy this file to the Moodle dropbox. Make sure your program compiles and runs on CSL machines. Your program will be tested using several test cases of different sizes. Make sure all three files are submitted to the Moodle dropbox.

Late Policy

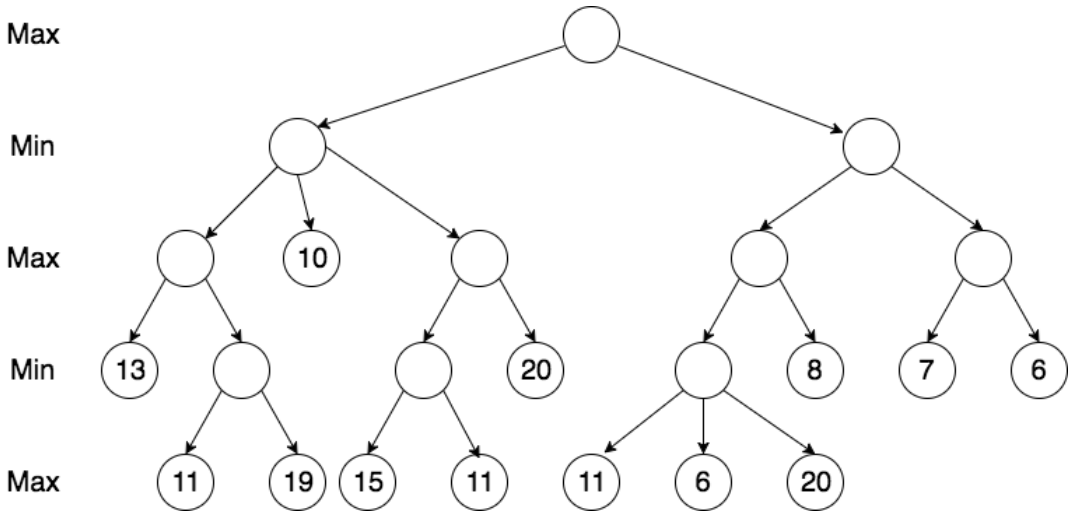
All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between 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 within one week after the assignment is returned.

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 to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code on the Web

In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Problem 1. [15] Minimax and Alpha-Beta



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

(b) [7] Use the Alpha-Beta pruning algorithm to prune the game tree above, 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 pseudocode in the lecture notes and textbook. Different versions of the algorithm may lead to different values of alpha and beta.

(c) [3] For a fixed search tree, are there any cases that the Alpha-Beta algorithm makes a *different* move from the Minimax algorithm? If yes, show an example; if no, explain briefly why not.

Problem 2. [15] Hill Climbing

Given a set of locations and distances between them, a tour is a path that visits each location exactly once and ends at the start location. For each tour, we fix the start location (and hence the end location as well). We would like to find the shortest tour starting from a given location using a greedy Hill-Climbing algorithm. Below is the specification of the search problem:

- Each state corresponds to a permutation of all the locations except the starting and ending locations, e.g. <*A*-*B*-*C*-*A*> is a tour from *A*. For this problem, symmetric paths aren't considered different, e.g. <*A*-*C*-*B*-*D*-*A*> and <*A*-*D*-*B*-*C*-*A*> are considered to be the same state.

- The operator '*neighbors*(*s*)' generates all neighboring states of state *s* by swapping the order of any two locations except the first and last location.

- We can set the evaluation function for a state to be the total distance of the tour, where each pairwise distance is given in an *n* x *n* distance matrix (where *n* stands for the *n* locations, and edges are undirected, i.e., *A*-*B* and *B*-*A* have the same distance). Assume that ties in the evaluation function are broken by choosing the first one in alphabetical order; for example, if states <*A*-*B*-*C*> and <*A*-*C*-*B*> have the same cost, choose <*A*-*B*-*C*>.

Answer these questions:

(a) [3] If you have a start location and *n* other locations, how many neighboring states does the *neighbors*(*s*) function produce (include same states)? Show your computation.

(b) [3] What is the total size of the search space, i.e., how many possible distinct states are there in total? Assume again that there are *n* other locations. Show your computation.

(c) [9] Imagine that a student wants to hand out fliers about an upcoming programming contest. The student stands at Memorial Union (M), and want to visit the Wisconsin Institute for Discovery (W), Computer Science Building (S), Capitol Building (C), and Engineering Hall (E) to deliver the fliers, and then go back to the Memorial Union. The goal is to find the shortest possible tour. The distance matrix between individual locations is as follows:

	M	**C**	**W**	**E**	**S**
M	0	0.9	0.6	0.8	0.7
C	0.9	0	1.3	1.5	1.3
W	0.6	1.3	0	0.2	0.3
E	0.8	1.5	0.2	0	0.2
S	0.7	1.3	0.3	0.2	0

***Table 1**. Distance matrix*

The student starts applying the greedy Hill-Climbing algorithm from the initial state: <M-E-C-S-W-M>. Identify 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? Explain why or why not. If an optimal tour can be found, list the sequence of states in the tour.

Problem 3. [70] Game Playing: Take-Stones

Implement the Alpha-Beta pruning algorithm to play a two-player game called Take-Stones. Follow the algorithm pseudocode in the lecture notes and 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 taking a stone, the number is saved as the new last move. If a player ***cannot*** take a stone, 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 next turn of the next player***. 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 is number is 0, it means 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, there will not arguments for this list.
- depth: the search depth. If depth is 0, search to end game states

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, the next turn is the Max player (because 2 stones have been taken), and you should search to end game states.

Output

You are required to print out to the console:

- A trace of the alpha-beta search tree that is generated. That is, *at each node* of the search tree generated, you are to print the following **right before returning from the node** during your recursive depth-first search of the tree:
  - The number of the stone taken to reach the current node from its parent
  - The list of currently available stones at the current node, output in increasing order and separated by a space
  - The final Alpha and Beta values computed at the current node, separated by a tab

  Note: The above information is printed for you in the provided code.
- At the completion of your alpha-beta search, print the following three lines:
  - "NEXT MOVE"
  - The next move (i.e., stone number) taken by the current player (as computed by your alpha-beta algorithm)
  - A list of the stones remaining, in increasing order separated by spaces, after the selected move is performed

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. So, returning from these nodes, from leaf to root, we get the output below.

Input:

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

Output:

```
3
5 7
alpha: -1000    beta: 1000
6
3 5 7
alpha: 100      beta: 1000
2
3 5 6 7
alpha: -1000    beta: 100
NEXT MOVE
6
3 5 7
```

Static Board Evaluation

The static board evaluation function should return values as follows:

- At an end game state where Player 1 (MAX) wins: 100
- At an end game state where Player 2 (MIN) wins: -100
- Otherwise,
    - 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 lastMove is 1, count the number of the possible successors (i.e., legal moves). If the count is odd, return 5; otherwise, return -5.
    - If lastMove is a prime, count the multiples of that prime in all possible successors. If the count is odd, return 7; otherwise, return -7.
    - If lastMove is a composite number (i.e., not prime), 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 6; otherwise, return -6.

Other Important Information

- Set the initial value of alpha to be -1000 and beta to be 1000
- To break ties (if any) between multiple 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 roughly 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 public class named `TakeStones` (we will call `TakeStones` to test)
- Receives input from the command line
- Outputs exactly match our given format

If you use the skeleton, you should implement four methods in class `TakeStones` and two methods in class `Helper`. I/O methods have been implemented for you.

The following five methods for the class `Player` are given in the supplied skeleton code:

`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`:

- `public int alphabeta(GameState state, int depth, int alpha, int beta, boolean maxPlayer)`
- `public int next_move(GameState state, int depth, boolean maxPlayer)`

- `public ArrayList<Integer> generate_successors(GameState state)`
- `public int evaluate_state(GameState state);`

`Class Helpers: helper functions`

Details of parameters can be found in the comments in the supplied code.

Submission

Submit ***all*** your java files**.** You should not include any package path or external libraries in your program. Copy all source files into a folder named `<Wisc NetID>-HW2`, then compress this folder into a zip file, called `<wisc NetID>-HW2-P3.zip`