Homework Assignment #1

Assigned: Saturday, January 28th
Due: Monday, February 6th

Hand-In Instructions

This assignment includes written problems and programming in Java. Hand in all parts electronically by uploading them in *a single zipped file* to the assignment page on Canvas.

Your answers to each written problem should be turned in as separate pdf files called <wisc NetID>-HW1-P1.pdf and <wisc NetID>-HW1-P2.pdf. You can use your program of choice – pencil and paper, word processor, LaTeX – as long as you show your work fluidly and neatly. If you handwrite your solutions to written problems, make sure to scan them in. *No photographed assignments will be accepted.*

For the programming problem, put *all* files needed to run your program, including ones you wrote and ones provided, into a folder called <wisc NetID>-HW1-P3.

Once you are finished, put your programming component folder and your two PDF files into a single directory. Zip it, name it <wisc NetID>-HW1, and upload it to the assignment Canvas page.

Make sure your program compiles on CSL machines this way! Your program will be tested using several test cases of different sizes.

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.

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 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: Search Concepts [12]

a. Design a (small) search tree where **iterative deepening search** performs dramatically worse than **depth-first search** in terms of **time complexity**.* Label the start and goal states on the graph, and justify your answer using precise counts. [4]

**For example, iterative deepening search might run at $O(n^2)$, but depth-first search on that same tree might run at $O(n)$.*

b. Is h(n) = 1/h*(n), where h*(n) is the true minimum cost path from n to the goal, an admissible heuristic, consistent heuristic, or neither? [4]

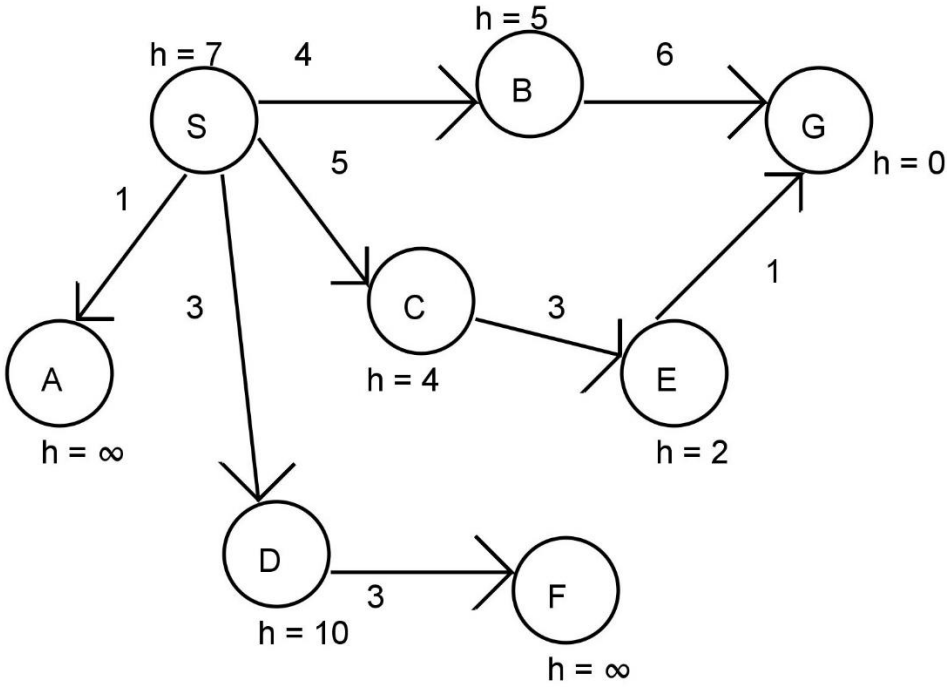
c. Draw a small, non-trivial graph (> 3 nodes) labelled with start and goal states, edge weights and heuristic function values in which **A-Search** expands more nodes to find a path from the start to the goal than **breadth-first search**. Justify your answer. [4]

Problem 2: Fun with Graphs [28]

Given this directed graph, find paths from start state S to goal state G using the search algorithms below. Add children of the same node in alphabetical order, and break ties within the frontier using alphabetical order.

You are required to write down the following for each algorithm:

- At each step until the goal is tested, you must list state positions (+ priority if applicable) in the frontier.
- The solution path found by the algorithm.
- The total # of nodes expanded.



a. Iterative Deepening Search [8]
b. Dijkstra's Search aka Uniform Cost Search [8]
c. A-Search. Also, is this heuristic admissible? Prove why or why not. [10]

Problem 3: Informed Search in Mazes [60]

Write a Java program that finds a path through a maze from a given start position to a given goal position. Your task is to write a program that reads in a maze and finds a solution by executing:

FindPath maze search-method

The first argument, maze, is a text file containing the input maze as described below. The second argument, search-method, can be either "dfs" or "astar" indicating whether the search method to be used is depth-first search (DFS) or A* search, respectively.

The Maze

A maze will be given in a text file as a matrix in which the start position is indicated by "S", the goal position is indicated by "G", walls are indicated by "%", and empty positions are where the navigator can move. The outer border of the maze, i.e., the entire first row, last row, first column and last column will *always* contain "%" characters. The navigator is allowed to move only horizontally or vertically, not diagonally.

The Algorithms

For DFS, push available neighboring states to the stack in counterclockwise order starting from the left. More specifically, move-Left(L), move-Down(D), move-Right (R), and move-Up (U) are pushed on to the stack that implements the *Frontier* set for this search method in this order. This means that U will be the first neighbor to be popped from the stack and expanded.

Assume all moves have cost 1. Repeated state checking should be done by maintaining both *Frontier* and *Explored* sets. If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*; otherwise, throw away node *n*.

For A* search, use the heuristic function, *h*, defined as the Euclidean distance from the current position to the goal position. That is, if the current position is (u, v) and the goal position is (p, q), the Euclidean distance is $\sqrt{(u-p)^2 + (v-q)^2}$. When expanding a node, add neighboring states into the priority queue clockwise from the top -- U, R, D, L. This means that, in the invent of a tie, U will removed from the priority queue first. Assume all moves have cost 1. For A* search, repeated state checking should be done by maintaining both *Frontier* and *Explored* sets as described in the Graph-Search algorithm in Figure 3.14 in the textbook. That is,

- If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*.

- If a newly generated node, *n*, has the *same* state as another node, *m*, that is already in *Frontier*, you must compare the *g* values of *n* and *m*:
  - If $g(n) \geq g(m)$, then throw node *n* away (i.e., do *not* put it on *Frontier*).
  - If $g(n) < g(m)$, then remove *m* from *Frontier* and insert *n* in *Frontier*.
- If new node, *n*, has the *same* state as previous node, *m*, that is in *Explored*, then, because our heuristic function, *h*, is consistent (aka monotonic), we know that the optimal path to the state is guaranteed to have already been found; therefore, node *n* can be thrown away. So, in the provided code, *Explored* is implemented as a Boolean array indicating whether or not each square has been expanded or not, and the *g* values for expanded nodes are not stored.

Output
After a solution is found, print out on separate lines:
1. the maze with a "." in each square that is part of the solution path
2. the length of the solution path
3. the number of nodes expanded
4. the maximum depth searched
5. the maximum size of the *Frontier* at any point during the search.

If the goal position is *not* reachable from the start position, the standard output should contain the line "No Solution" and nothing else.

Code
You must use the code skeleton provided. You are to complete the code by implementing search() methods in the AStarSearcher and DepthFirstSearcher classes and the getSuccessors() method of the State class. **You are permitted to add or modify the helper classes, but we require you to keep the IO class as is for automatic grading.** The FindPath class contains the main function.

Compile and run your code using an IDE such as Eclipse. To use Eclipse, first create a new, empty Java project and then do File → Import → File System to import all of the supplied java files into your project.

You can also compile and run with the following commands in a terminal window:
javac *.java
java FindPath input.txt dfs

Testing
Test both of your search algorithms on the sample test input files and check the results against the sample output files, which will be posted to Canvas. Make sure the results are correct on CSL machines. It shouldn't be necessary to import any third-party libraries.

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 <wisc NetID>-HW1-P3. Move this into the <wisc NetID>-HW1 directory with your written problems, then compress this folder to create <wisc NetID>-HW1.zip and upload it the Canvas assignment page.