CS 540-2: Introduction to Artificial Intelligence

Homework Assignment #1

Assigned: Wednesday, January 25
Due: Friday, February 3

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 "`HW1 Hand-In`". 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` (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* 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` To test your program, we will `cd` into your directory, and compile it using: `javac FindPath.java` Make sure your program compiles on CSL machines this way! Your program will be tested using several test cases of different sizes. Compress this folder to create `<wisc NetID>-HW1-P3.zip` and copy this file to the Moodle dropbox. Make sure all three files, `<wisc NetID>-HW1-P1.pdf`, `<wisc NetID>-HW1-P2.pdf` and `<wisc NetID>-HW1-P3.zip` 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 or a TA within one week after the assignment is returned.

Collaboration Policy

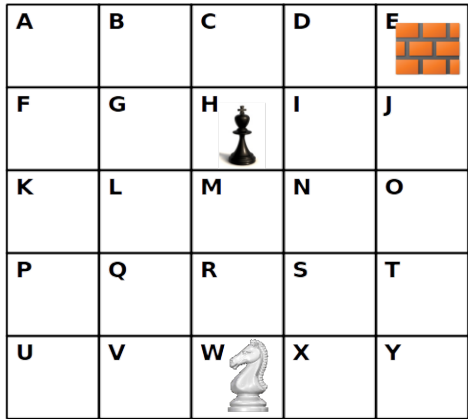
You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate 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: [21] Search Algorithms

In the following figure is a modified chess board on a 5×5 grid. The task is to capture the black king (*fixed* in square **H**) by moving the white knight (starting in square **W**) using only "knight moves" defined in Chess. Assume the successor function ***expands*** legal moves in a clockwise order: (1 right, 2 up); (2 right, 1 up); (2 right, 1 down); (1 right, 2 down); (1 left, 2 down); (2 left, 1 down); (2 left, 1 up); (1 left, 2 up). Note that not all of these moves may be legal from a given square. It is *not* legal for a knight to move to a square with a wall on it (square **E**). Submit your answers to the following questions in a single pdf file called <wisc NetID>-HW1-P1.pdf



(a) [6] **Depth-First Search**
List the squares in the order they are expanded, including the goal node if it is found. State **W** is expanded first. Using a stack for the *Frontier* means you should add states to the stack in the *reverse order* from that given above. So, state **N** will be popped and expanded second. Use the Graph-Search algorithm (Figure 3.7), which uses both a *Frontier* set and an *Explored* set so that each square will be expanded at most once. Write down the list of states you expanded in the order they are expanded, including the goal state **H**. Write down the solution path found (if any), or explain why no solution is found.

(b) [6] **Iterative-Deepening Search**
Draw the trees built at each depth until a solution is reached. Use the same order of expanding nodes as in (a), and also use the same method of repeated state checking as in (a).

(c) [3] **Heuristic Function**
Let each "move" of the knight have cost 1. Consider the heuristic function $h(n) = |u - p| + |v - q|$, where the grid square associated with node n is at coordinates (u, v) on the board, and the goal node **H** is at coordinates (p, q). That is, $h(n)$ is the "city-block" distance between n and **H**. Is $h(n)$ admissible? Why or why not?

(d) [6] **Algorithm A Search**
Apply **Algorithm A** with the heuristic function $h(n)$ defined in (c). In the case of ties, expand states in alphabetical order. Use repeated state checking by keeping track of both the *Frontier* and *Expanded* sets. If a newly generated node, n, does not have the same state as any node already in *Frontier* or *Expanded*, then add n to *Frontier*. If n's state already exists in either

Frontier or *Expanded*, then check the *g* values of both *n* and the existing node, *m*, as follows:

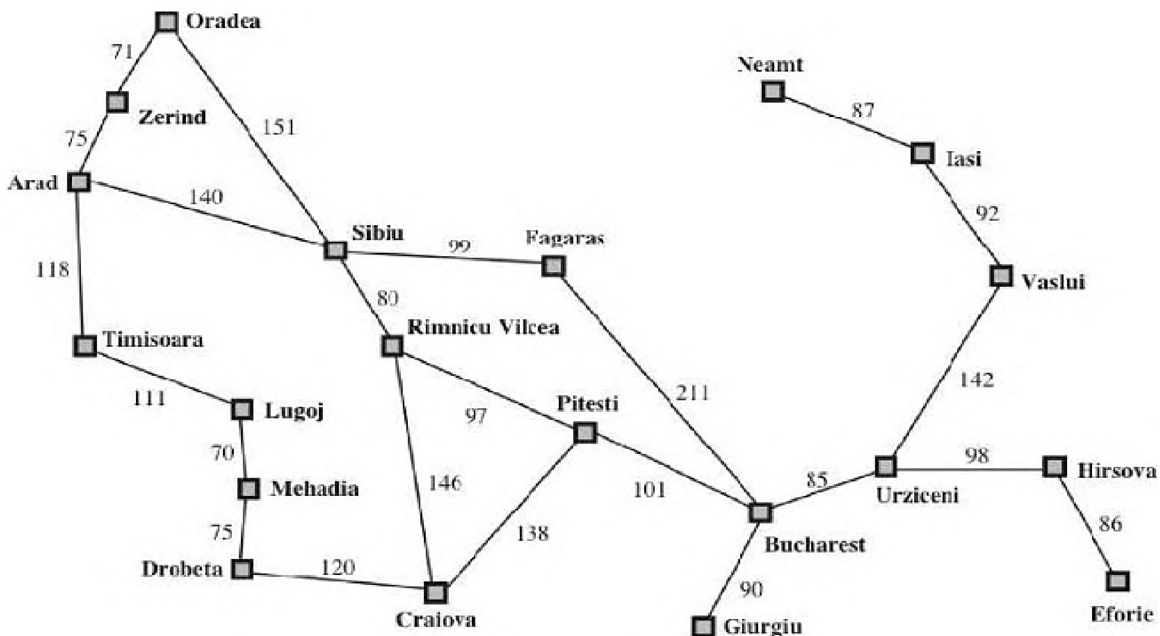
- If $g(n) \geq g(m)$, then just throw away the new node, *n*.
- If $g(n) < g(m)$, then remove *m* from either *Frontier* or *Expanded*, and insert *n* into *Frontier*.

Fill in a table of the form shown below that shows at each iteration of the algorithm the contents of the *Frontier* and *Expanded* sets. For each node in *Frontier*, include in parentheses its *f* value and its "parent" square in the search tree. The leftmost node in *Frontier* will be the one expanded in the next step. The number of steps (i.e., rows in the table) will correspond to the number of nodes expanded, including the goal node **H** if it is found. Finally, list the solution path found, if any, or explain why no solution is found.

Step	*Frontier*	*Expanded*
1	**W**(*f* value, parent square in search tree)	---
2		

Problem 2: [14] Meet Your Friend

Two friends live in different cities on the following map. On every turn, we must simultaneously move each friend to one of its neighboring cities on the map. That is, each person cannot stay in the same city for two consecutive turns, though she can move to a neighboring city and then on the next turn move back to the previous city. The amount of time needed to move from city *i* to neighbor city *j* is equal to the road distance *d*(*i*, *j*) between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on her cell phone) before the next turn begins. The goal is to have the two friends meet in the same city as quickly as possible. Submit your answers to all parts of this problem in a single pdf file called `<wisc NetID>-HW1-P2.pdf`



(a) [8] Write a formal description including following details for this search problem. (An example formulation is shown for the Water Jugs Problem given in the lecture slides)
(i) State space
(ii) Successor function
(iii) Goal test
(iv) Step cost function (i.e., the definition of the cost of a move from a state to a successor state)

(b) [3] Let *D(i, j)* be the straight-line distance between cities *i* and *j*. Which of the following heuristic functions are admissible?
(i) 1
(ii) $2 \times D(i, j)$
(iii) $D(i, j)/2$

(c) [3] Are there completely connected maps for which no solution exists? Briefly explain why or why not.

Problem 3: [65] Maze Search

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 "`bfs`" or "`astar`" indicating whether the search method to be used is breadth-first search (BFS) 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 robot 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. A robot is allowed to move only horizontally or vertically, not diagonally.

The Algorithms
For BFS, add move-Left (L), move-Down (D), move-Right (R), and move-Up (U) in that order to the queue that implements the *Frontier* set for this search method. In this way, moves will be visited in the same order as insertion, i.e., L, D, R, U. 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 $\sqrt{(u-p)^2+(v-q)^2}$. Add moves in the order L, D, R, U to the priority queue that implements the *Frontier* set for A* search. 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 `BreadthFirstSearcher` classes and the `getSuccessors()` method of the `State` class. **You are permitted to add or modify the 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 bfs
```

Testing

Test both of your search algorithms on the sample test input file: `input.txt` and compare results with the two output files: `output_astar.txt` and `output_bfs.txt`. Make sure the results are correct on CSL machines.

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` Compress this folder to create `<wisc NetID>-HW1-P3.zip` and copy this file to the Moodle dropbox.