CS 540: Introduction to Artificial Intelligence

Homework Assignment #1

Assigned: Thursday, January 28
Due: Monday, February 8

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 `P1.pdf` and `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. 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 username>-HW1` For example, if your wisc username is `jdoe`, then create a folder called `jdoe-HW1` 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. Finally, also put into the folder your files `P1.pdf` and `P2.pdf`. Then compress this folder to create `<wisc username>-HW1.zip` and copy this one file into 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.

Collaboration Policy

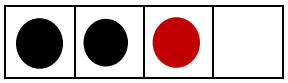
You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate 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. [15] Black and Red Marbles

Consider the following 1x4 board containing 2 black and 1 red marbles:



Let's call this the *initial* state S. A *goal* state is defined as one in which there is no black marble anywhere to the left of a red marble. You are allowed the use of two operators to reach a goal state:

1. A marble can "slide" to its left or right one position into a blank square. **Cost: 1**
2. A marble can "jump" over another (one) marble to its left or right provided there is a blank square at the destination. **Cost: 2**

Your task for this problem is to use **Uniform-Cost Search (by hand)** to reach a goal state with the lowest possible cost. Use the representation **[b, b, r, o]** to represent the start state S above. Use throughout: **b for black, r for red** and **o for open**. Make use of an **Explored** set and a **Frontier** set in order to discard repeated states when generating successors (see the algorithm in Figure 3.14 in the textbook).

(a) [10] Use the algorithm to find the first lowest-cost goal state. Break ties randomly. Show expanded nodes and the current priority queue at each step as you search, using the representation defined above.

(b) [5] Show *all* lowest-cost paths, including the cost, using the representation above. Give all states along each path.

Problem 2. [25] Stimulus Plan

The Government has *N* (a positive integer) dollars for economic recovery that it wants to spend as quickly as possible. It needs your help to come up with a spending plan. Here are the rules:

- In year 1, it must spend exactly 1 dollar.
- Say in year *t*-1 the government spent *X* dollars. In year *t*, the government is allowed to spend *X*, *X*+1, or *X*-1 dollars. Note the spending can only be non-negative. In addition, the pool of money can only be non-negative too.
- In the last year, it must spend exactly 1 dollar and use up the money.

The goal is to **minimize the number of years needed to use up the money.**

(a) [8] Use formal notation to formulate your states, state space, your cost function, and how to generate successor states. An example formulation is shown for the Water Jugs Problem in the lecture slides.

(b) [4] Is "the total amount of money remaining in the pool at time *t"* an admissible heuristic? Explain why or why not.

(c) [8] State and justify a different non-trivial, *admissible* heuristic.

(d) [5] For any *N*, is it guaranteed there will be a plan for the government to use up the money? Briefly explain.

Problem 3. [60] Maze Search

Write a Java program that finds a path through a maze from a given start position to a given goal position. 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. A robot is allowed to move only horizontally or vertically, not diagonally.

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 is a text file containing the input maze as described above. The second argument can be either "`dfs`" or "`astar`" indicating whether the search method to be used is depth-first search (DFS) or A* search, respectively. For DFS, first push move-Left, second push move-Down, third push move-Right, and lastly push move-Up onto the stack that implements the *Frontier* set for this search method. In this way the Up move will be popped and visited *first*. For A* search use as the heuristic function, *h*, the City-Block 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 City-Block distance is $|u-p| + |v-q|$. In case of ties, use the priority order: U, R, D, L. That is, first pop U, then R, etc. Assume all moves have cost 1.

For *both* DFS and 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.

Test both of your two search algorithms on three mazes: `smallMaze.txt`, `mediumMaze.txt` and `bigMaze.txt`

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, and (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. An example solution to `smallMaze.txt` can be found in `smallMazeSolution.txt`

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 classes, but we require you to **keep the IO class as is** for automatic grading. The `FindPath` class contains the main function. To compile and run your code you use an IDE such as Eclipse. To use the supplied code in 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.