CSC 350/500 – Project 1: Subway Search

Due Date: October 2, 2023

Introduction

In this project, you will implement and compare search algorithms in real-world domains. I supply skeleton code and useful classes for Python and Java. You may use either language for this project. The last page of this document briefly summarizes these libraries and the data sets that go with this project.

File to Edit: You have been supplied with several code files, but you should only need to modify one. If using Python, you will edit search.py in the indicated places, as well as when defining new classes. If using Java, you will modify Search.java. You may also need to create other classes. No other files should be modified.

Please *do not* change the names of any of the provided functions or classes. Modifying code that you were instructed not to modify is grounds for losing credit on this assignment.

Getting Help: You are not alone! If you find yourself stuck on something, contact me for help sooner rather than later. I want these projects to be rewarding and instructional, not frustrating or demoralizing. But I do not know how or when to help unless you ask.

Also, remember, discussion of the assignment is perfectly acceptable. However, this discussion should not extend to sharing code. If in doubt about what would constitute academic dishonesty, contact me.

What to Submit: Zip all project files for your chosen language, the data files, and your answers to Q7. Submit the zip file to the assignment folder on Blackboard on or before the due date. Any submission after the due date is subject to the late policies described in the syllabus.

Assignment Questions

The London Underground (commonly called "The Tube") is a complex public transportation system which can be difficult for tourists to navigate. Boston's MBTA Subway (commonly called "The T") is less complex, though songs have been written about people trapped forever "beneath the streets of Boston." As part of this assignment, you will implement search algorithms to help these lost souls.

Question 1 (3 points)

Implement a sub-class of Problem that represents the Subway Navigation problem of finding a path between two given stations. The class should be independent of the city. For the purposes of this question, you may ignore the `h()` function. (You will need it later though!)

Some guidance for this question:

- I have provided a library which generates representations of the London Tube and Boston T systems and answers questions about these networks. This library includes functions that generate the networks in the first place. You do not need to delve into the data files yourself to build a data structure.
- Java users may wish to (but are not required to) sub-class State and Action as part of this question.
- Boston is largely for your benefit as a simpler network so that you can more easily debug. Most of my grading throughout this assignment will be based on the London network.

Question 2 (4 points)

Implement the depth-first-search (DFS) algorithm in the `depth_first_search` function (in search.py or Search.java). To make your algorithm *complete*, write the graph search version of DFS, which keeps track of the already visited states.

Modify the main method so that it creates an instance of a particular Subway Navigation Problem, where the city name, search algorithm, initial station, and destination are specified as command line arguments. For example, to run the Boston problem from Fenway to South Station using DFS, I expect to be able to enter the following (for Python and Java, respectively):

```
>python search.py boston dfs Fenway "South Station"

>java search.Search boston dfs Fenway "South Station"
```

For London, finding a path from Piccadilly Circus to Wimbledon would be run from the command line as follows:

```
>python search.py london dfs "Piccadilly Circus" Wimbledon

>java search.Search london dfs "Piccadilly Circus" Wimbledon
```

Note that any station name containing spaces must be entered on the command line with surrounding quotes.

I should be able to enter any pair of stations in Boston or London (as appropriate) as the initial state and destination. Your code should output the final path (station names will suffice), total cost, and number of search nodes visited (*visited* = extracted from the frontier).

Question 3 (4 points)

Implement the breadth-first-search (BFS) algorithm in the `breadth_first_search` function (in search.py or Search.java). Again, write a graph search algorithm that avoids expanding any already visited states. As with DFS, your output should contain the final path, total cost, and number of nodes visited.

Modify your main method so that `bfs` is now a valid command line argument that calls the BFS algorithm, for example:

```
>python search.py boston bfs Fenway "South Station"

>java search.Search boston bfs Fenway "South Station"
```

Question 4 (4 points)

Implement the A* algorithm in the `astar_search` function (in search.py or Search.java). Modify your Subway Navigation problem class from Question 1 by filling in the `h()` function using the straight-line distance (SLD) heuristic. This information can be calculated using the libraries provided for Question 1.

Modify your main method so that `astar` is now a valid command line argument that calls the A* algorithm, for example:

```
>python search.py boston astar Fenway "South Station"

>java search.Search boston astar Fenway "South Station"
```

The expected output contents are the same as DFS and BFS.

Question 5 (3 points)

Let's get a little more intelligent with our navigation. The Boston and London maps are diagrams, not accurate geographic representations. Consequently, some stations that appear far apart on the map are actually quite close in real-life. As examples, in Boston, Reservoir (Green D branch) and Cleveland Circle (Green C branch) are only about a block apart. In London, Paddington and Lancaster Gate are separated by about a ¼ mile.

Keeping with the Boston example, suppose I am a rider clueless about the geography and relying only on the map. If I board at Riverside station (the end of the D branch) and intend to reach Cleveland Circle, the map suggests I would have to go all the way into Kenmore and transfer to a Green C headed back to Cleveland Circle. If I knew better, I would be willing to save a lot of time and walk a block from Reservoir to reach Cleveland Circle.

Create a new Problem subclass that remains the Subway Navigation problem, but considers as a goal state any station within some distance *d* of the station (expressed in kilometers) that I name as my goal.

For the problem above, my command would be:

```
>python search.py boston astar Riverside "Cleveland Circle" 0.25

>java search.Search boston astar Riverside "Cleveland Circle" 0.25
```

That is, I want to go from Riverside to Cleveland Circle, or any station within 0.25 km (SLD) of Cleveland Circle.

Make sure to modify the Search class to allow for the additional command line argument and new subclass of Problem. This problem should work with both cities and all 3 search algorithms. The expected output remains the same.

Question 6 (4 points)

You may have noticed that the search algorithms that you developed in Questions 2-4 are quite separate from the problem definitions that you wrote in Question 1 and Question 5. When written correctly, the algorithms will work on any problem that has the same structure. In other words, the search algorithms are independent of the problems themselves. For A* search, we define the heuristic function as part of the problem, rather than as part of the search algorithm, because developing an admissible heuristic requires knowledge of the specific problem.

In order to verify that our search algorithms work on any well-structured problem, let's create a new Problem in a domain that is completely different from subway navigation.

Implement a sub-class of Problem that represents the 8-puzzle problem. Your `successor()` function should be written so that it gives the result of moving the blank up, left, down, and right in that order. (Thus actions represent the movement of the blank space.) Use Manhattan distance as your heuristic.

The initial state will be given as part of the command line arguments. For example:

```
>python search.py eight bfs 142305678

>java search.Search eight bfs 142305678
```

Note that 0 represents the blank space. Your goal state should be defined such that the blank space ends in the upper left corner. Be sure to modify your main method to read in the eight puzzle and so that `eight` is a valid command-line argument to specify the problem.

If you have written your code generically enough, each of your search algorithms should work immediately.

Question 7 (3 points)

In a separate document (TXT, DOCX, PDF), answer the following questions:

a. Identify an admissible heuristic for the Subway Navigation problem other than straight-line distance. (You can assume access to any information you would need to create one.)

b. The Boston and London data each have some situations where the distance between two linked stations is less than the calculated straight-line distance between those stations. In some cases, the difference is over 200 meters or 1/8 mile. (This is a legitimate concern when compiling datasets from different sources, as I did for this project.) What is the consequence of this fact? How significant is this consequence?

c. When given the example input shown in Question 6, BFS and A* search find solutions quickly, while DFS takes a very long time. Would modifying DFS to use iterative deepening help the running time? Why or why not?

Included Code

Search Problem Classes (search.py, or named Java class)

- Search – Main class, includes search functions. In Python, these functions are not part of a class.
- Problem – Abstract class that defines a search problem. You will have to subclass this class in order to define the domains that we will use for this assignment.
- Node – Defines a search node that may get visited during the search process.
- Tuple – Defines a state/action pair. (Not included in Python code. Uses built-in Python tuple instead.)
- State – Defines a state. You may want to subclass this class for specific problem domains. (Not necessary in Python code, but you are free to add it if you wish.)
- Action – Defines an action. You may want to subclass this class for specific problem domains. (Not necessary in Python code, but feel free to add it.)

Subway Problem Classes (subway.py, or named Java class)

- SubwayMap – A class defining a Subway Map (effectively an adjacency-list implementation of a graph). Use the public buildBostonMap() and buildLondonMap() functions to automatically create each city's network.
- Station – A subway station and its location (effectively a graph vertex with geographic information included).
- Link – A connection between two subway stations (effectively a graph edge, with a distance and a route).

About the Datasets

The Boston data includes all stations and connections on this map on the Red, Blue, Orange, and Green lines. The Silver line and commuter rail are not included. If you want to know more about the data, check the readme file.

The London data includes all stations and connections on this map that are part of the 11 Underground lines, shown as solid colored lines in the diagram. The 6 routes shown as hollow lines (DLR, Elizabeth Line, London Overground, London Trams, IFS Cloud Cable Car, and Thameslink) are not included. If you want to know more about the data, check the readme file.

For both networks, I have removed punctuation from station names in the dataset, except ampersands, slashes, and hyphens. Also, the London network contains stations with duplicate names (Edgware Road, Hammersmith, and Paddington). Short abbreviations are added to duplicate names that indicate the lines that stop there. If

you’re not sure how a station is listed, open boston_stations.csv or london_stations.csv from the data folder and use search commands to figure it out.