Computer Science 2XC3: Final Lab/Project

Dr. Vincent Maccio

Please read the following carefully. There are no completion/participation grades in labs. However, the TAs are there to guide and aid you in your tasks.

Purpose

This lab focuses on implementing, analysing, and applying shortest path algorithms. This lab will span the remainder of the lab time of this course. Think of it as a final project. Each week I will be releasing a new part of the project for you to consider/work on.

Part 1

- Empirically analyze and deduce runtimes of traditional/mystery shortest path algorithms
- Gain a general comprehension of when and why to use one shortest path algorithm over another
- Implement shortest path algorithm variations

Part 2

- Do independent research on another shortest path algorithm
- Implement the A* shortest path algorithm
- Give a discussion on how this algorithm would compare against Dijkstra's algorithm; when would you prefer one over the other (if ever)

Part 1

Throughout this lab you will be creating and ultimately submitting a project report. Your project report should look professional and complete. It should include the following:

- Title page
- Table of Content
- Table of Figures
- An executive summary highlighting some of the main takeaways of your experiments/analysis (this can be presented in bullet form if you wish)
- An appendix explaining to the TA how to navigate your code (for example, which .py file to find which implementation/experiment in)

For each experiment, include a clear section in your lab report which pertains to that experiment.

Shortest Path Approximations

In lecture we saw "single source" versions of Dijkstra's and Bellman-Ford's algorithms (we'll see Bellman-Ford this week). Both algorithms take the approach of keeping track of the currently known shortest path to each node. When a node's new shortest path is updated (for example, when Dijkstra's algorithm decreases the key of a node in the min-heap), this is called "relaxing" that node/vertex. In general, in both algorithms, a vertex can be relaxed up to V – 1 times.

Implement versions of Dijkstra's and Bellman-Ford's algorithms which relax each node at most k times. Name these functions:

- dijkstra_approx(G, source, k)
- bellman_ford_approx(G, source, k)

Have both of these functions returns a distance dictionary which maps a node (integer) to the distance of the shortest path known to the node (based off the approximation).

Experiment Suite 1

Use what you have learned thus far about empirical experiments and approximations to design and conduct some interesting experiments. Use your judgement on how you should measure certain things (hint see the total_distance function in the given code). However, some general things which you should be considering are how things are dependent on the graph size, the graph density, and the value of k. Play around with the numbers and identify some interesting observations.

- An explicit outline of the experiments you ran. Number of nodes, edges, relaxations, etc. In addition to the outline, give an explanation to why you choose this specific set of parameters, what were you trying to elicit/observe? You should have at least 3 separate experiments.
- At least three different clearly labeled and professional graphs corresponding to the experiments you chose to run.
- A discussion and conclusion regarding the results. There should be a paragraph or so for each experiment you ran.

Mystery Algorithm

Dijkstra's and the Bellman-Ford algorithm are both single-source shortest path algorithms. You give them a node *v* (a single source) and they return the shortest path from *v* to every other node. An all-pairs shortest path algorithm takes a graph as input and returns the shortest path from every node to every other node. This is usually returned as a matrix of values.

Imagine if you were tasked with devising two different all-pairs-shortest path algorithm; one for positive edge weights and one for potentially negative edge weights. If correctness was your only concern, what would you do? It should not take too much convincing that running a single source algorithm for each potential "source" solves the all-pairs-shortest path problem.

From 2C03 (and Wikipedia) you know Dijkstra has complexity $\Theta(E + V\log V)$, or $\Theta(V^2)$ if the graph is dense. Moreover, Bellman-Ford has complexity $\Theta(VE)$, or $\Theta(V^3)$ if the graph is dense. Knowing this, what would you conclude the complexity of your two algorithms to be for dense graphs? Explain your conclusion in your report. You do not need to verify this empirically.

In the code posted with this document, you will find a mystery() function. It takes a graph as input. Do some reverse engineering. Try to figure out what exactly this function is accomplishing. You should explore the possibility of testing it on graphs with negative edge weights (create some small graphs manually for this). Determine the complexity of this function by running some experiments as well as inspecting the code. Hint: to determine the degree of the polynomial function consider graphing it on a log/log plot – you likely saw something similar to this in your high school sciences classes. Given what this code does, is the complexity surprising? Why or why not?

Part 2

There are many shortest path algorithms; it is one of the more practical graphing problems there is. We have seen some classic/traditional ones in the previous part of this project. For this next part, you are going to do your own research on a variation/modification to Dijkstra's algorithm called the A* algorithm (pronounced "A star", you know, like a regular expression).

The general idea is in addition to the edge weights you provide the algorithm with a heuristic function which takes in an edge and returns a numerical value to help guide a traditional Dijkstra's algorithm to not go down potentially bad paths. You use this heuristic in addition to the edge weight as the key in the… hey you know what, this is an experiential course. you figure out how A* works. To get you started this video is relatively easy to follow:

https://www.youtube.com/watch?v=ySN5Wnu88nE

As a side note, that channel, along with Numberphile, are generally great. Although I must admit I have not watched much of them in recent years.

Do some research into how this algorithm works, and in your report give an explanation of the algorithm as well as discussing the following points:

- What issues with Dijkstra's algorithm is A* trying to address?
- How would you empirically test Dijkstra's vs A*?
- If you generated an arbitrary heuristic function (similar to randomly generating weights), how would Dijkstra's algorithm compare to A*?
- What applications would you use A* instead of Dijkstra's?

Implementation

Write an a_star(G, s, d, h) function which takes in a directed weighted graph G, a sources node s, a destination node d, and a heuristic "function" h. Assume h is a dictionary which takes in a node (an integer), and returns a float. Your method should return a 2-tuple where the first element is a predecessor dictionary, and the second element is the shortest path the algorithm determines from s to d.

Part 3

Note: This part contains arguably more work than the previous two parts (or the part to be posted next weekend).

This will have you compare the performance of Dijkstra to A*. In general, this is tricky to do. Generating random graphs is not really a meaningful option since we need a heuristic function for A*. Sure, we could randomly generate that as well but that seems like an unwise choice since the “rule of thumb” the heuristic is meant to encode is totally arbitrary, and could hurt the runtime overall. Therefore, it would be nice if we had some real-world data to use.

Take a look at two files posted alongside this assignment: London_connections.csv and london_stations.csv. Full credit to Dr. Sebastien Mosser for making me aware of these files found at: https://github.com/nicola/tubemaps.

These files contain complete data on the London subway system. *Oi, you mean the tube, guvna*. That’s right my limey friend. You will find that this subway network has many stations/stops, over 300 in fact. Find a map of the network below.

Figure 1: London "Tube" Map

Treat each station as a node in a graph. Furthermore, you add an edge between two stations if they are connected (see london_connections.csv). Put an edge between node n and m if and only if station n and m are connected. But how do you determine the weights of these edges, and perhaps even more difficult, how would you build a heuristic function for your A* algorithm?

You will note that in london_stations.csv there is latitude and longitude information on each station. This will allow you to get very good approximations for the distance the train must travel between stations which are connected (the majority are essentially straight lines). Feel free to hard code some estimations on distance based on the map for stations which are not connected by straight lines, however, there is no requirement to do this. Perhaps even more useful though, from one source station you can create a function/dictionary which takes in another station and returns the physical distance as-the-crow-flies from that station to the source station. Use this function/dictionary as your heuristic function in your A* algorithm.

Experiment Suite 2

Use the two .txt files to build a weighted graph and heuristic function. For all of your experiments in this suite, use this graph and heuristic function.

The goal of this suite is to determine if and when A* is better than Dijkstra (and perhaps vice versa). The graph itself is not that large. Therefore it may be reasonable to check all pairs shortest paths. That is, check the time each algorithm takes for all combination of stations. Give well presented results and discussion of your analysis. Some points to considers:

- When does A* out perform Dijkstra; when are they comparable in runtime; when does Dijkstra out perform A*?
- What could explain the results above?
- Is there anything you note about certain combinations of stations? For example, what do you note about stations which are:
    - On the same line?
    - On adjacent lines?
    - On lines which require several transfers?
- For the point above, there is “line” information in the files as well. Can you incorporate how many lines the shortest path uses in your results/presentation/discussion?

Part 4

The final part of this project you must refactor and organize your code into something reasonable. See the UML class diagram below:

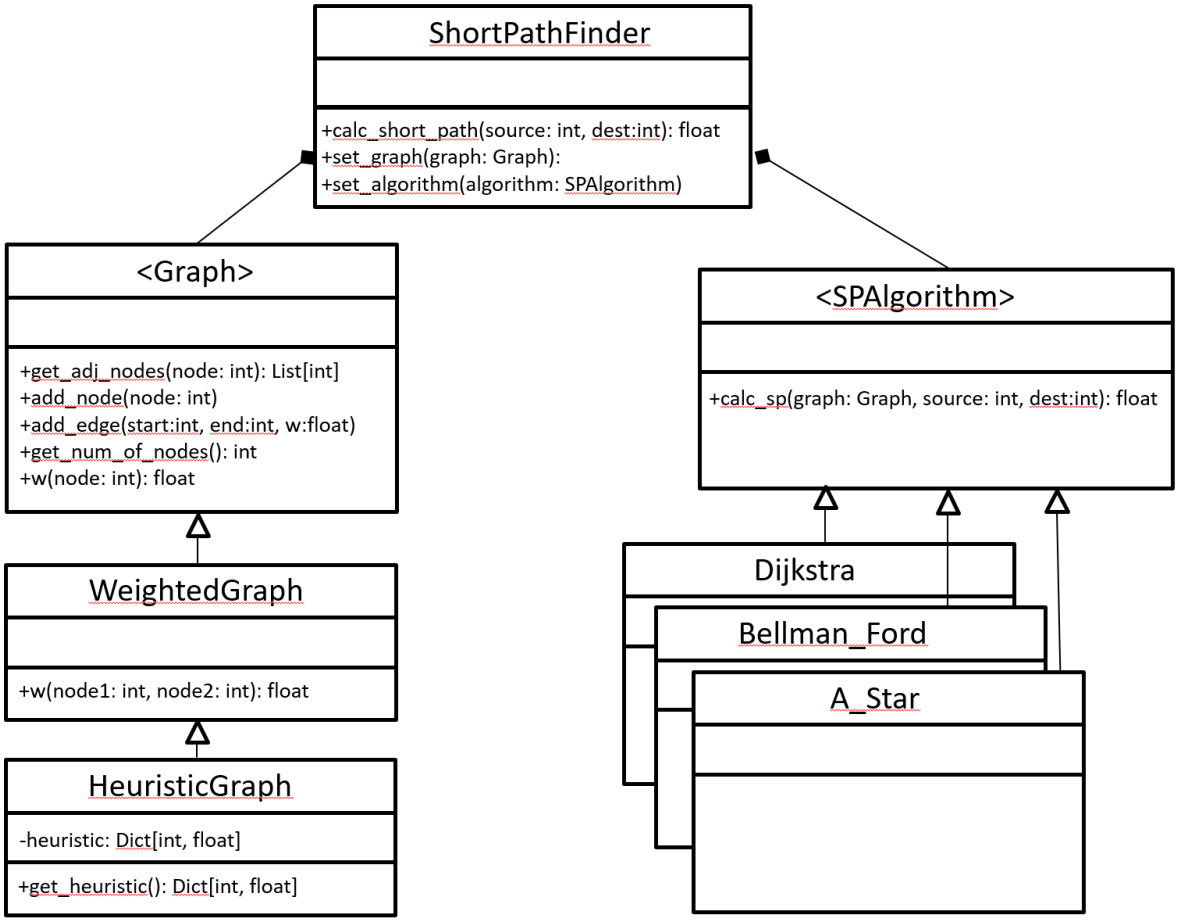

Figure 2: UML class diagram

Refactor you code to adhere to this diagram. Furthermore, consider the points listed below and discuss these points in a section labelled Part 4 in your report (where appropriate). Python is a different from Java when it comes to things like interfaces, public/private, etc. Do some research regarding how these things are done (if at all) in Python.

- The A_Star class Figure 2 does not exactly line up with the one I asked you to implement in Part 2. Do no rewrite you're A* algorithm! Instead, treat the A_Star class from Figure 2 as an "adapter".
- Discuss what design principles and patterns are being used in Figure 2.
- Figure 2 is a step in the right direction, but there is still a lot of work that can be done. For example, how graph Nodes are represented is being exposed to the rest of the application. That is, the rest of the application assume nodes are represented by integers. What if this needs to be changed in the future? For example, what if later on we need to switch nodes to be

represented at Strings? Or what if nodes should carry more information than their names? Explain how you would change the design in Figure 2 to be robust to these potential changes.

- Discuss what other types of graphs we could have implement “Graph”. What other implementations exist?

Grading and Submission

Your group will submit the following documents to Avenue:

- report.docx (or .pdf, or whatever – as long as a reasonable person can open it)
- code.zip (all your source code pertaining to the lab – including experiment code)

In addition to the grade allocations below, your report may lose up to 20% of the final grade for not looking professional, having formatting/style issues, graphs presented in a messy manner, etc. Moreover, you may lose grades for not including elements explicitly mentioned in the Part 1 section of this document. Find a rough grade breakdown below:

Part 1

Dijkstra and Bellman-Ford Approximations	10%
Experiment Suite 1	15%
Mystery Algorithm	5%

Part 2

A* Implementation	10%
A* Explanation/Discussion	5%

Part 3

Experiment Suite 2 – Results	30%
Experiment Suite 2 -- Discussion and insight	10%

Part 4

Code Refactor	15%