Computer Science 2XC3: Final Lab Part I

This project will include a final report and your code. Your final report will have the following. You will be submitting **.py (NOT *.ipynb)** files for this final project.

- Title page
- Table of Content
- Table of Figures
- An executive summary highlighting some of the main takeaways of your experiments/analysis
- An appendix explaining to the TA how to navigate your code.

For each experiment, include a clear section in your lab report which pertains to that experiment. The report should look professional and readable.

PLEASE NOTE: This project has 6 parts. Part 1-3 are released this week (Final Lab Part I), Parts 4 – 6 will be released next week (Final Lab Part II).

Part 1 : Single source shortest path algorithms

Part 1.1: In this part you will implement variation of Dijkstra's algorithm. It is a popular shortest path algorithm where the current known shortest path to each node is updated once new path is identified. This updating is called *relaxing* and in a graph with n nodes it can occur at most $n-1$ times. In this part implement a function dijkstra (graph, source, k) which takes the graph and source as an input and where each node can be relaxed on only k times where, $0 < k < N - 1$. This function returns a distance and path dictionary which maps a node (which is an integer) to the distance and the path (sequence of nodes).

Part 1.2: Consider the same restriction as previous and implement a variation of Bellman Ford's algorithm. This means implement a function bellman_ford(graph, source, k) which take the graph and source as an input and finds the path where each node can be relaxed only k times, where, $0 < k < N - 1$. This function also returns a distance and path dictionary which maps a node (which is an integer) to the distance and the path (sequence of nodes).

Part 1.3: Design an experiment to analyze the performance of functions written in Part 1.1 and 1.2. You should consider factors like graph size, graph. density and value of k, that impact the algorithm performance in terms of its accuracy, time and space complexity.

Part 2: All-pair shortest path algorithm

Dijkstra's and Bellman Ford's are single source shortest path algorithms. However, many times we are faced with problems that require us to solve shortest path between all pairs. This means that the algorithm needs to find the shortest path from every possible source to every possible destination. For every pair of vertices u and v, we want to compute shortest path $distance(u, v)$ and the second-to-last vertex on the shortest path $previous(u, v)$. How would you design an all-pair shortest path algorithm for both positive edge weights and negative edge weights? Implement a function that can address this.

Part 3: A* algorithm

In this part, you will analyze and experiment with a modification of Dijkstra's algorithm called the A* (we will cover this algorithm in next lecture, but you are free to do your own research if you want to get started on it). The algorithm essentially,

Part 3.1: Write a function A_Star(graph, source, destination, heuristic) which takes in a directed weighted graph, a sources node, a destination node , and a heuristic "function". 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 source to destination.

Part 3.2: In your report explain the following:

- 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?

This is partial release.

…. Part 4-6 to be released next week.