Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 4: Greedy Algorithms

In this lab we will be implementing *greedy algorithms* that were covered in the lectures, and testing them on certain examples.

Task 1: Implementation of Dijkstra's Algorithm

The first task will be to implement Dijkstra's shortest path algorithm. Before the implementation of the algorithms, we start from the implementation of graphs as a class.

Task 1.1: Representing Graphs

Recall that graphs can be represented as *adjacency matrices* or *adjacency lists*. In this task we will focus on the adjacency matrix representation. We will be dealing with directed *weighted* graphs, so, for a graph $G = (V, E)$, the adjacency matrix representation will take three different forms:

- For (undirected, unweighted) graphs, $A_{ij} = A_{ji} = 1$ if and only if there is an edge $(i, j) \in E$ in G.
- For (unweighted) directed graphs, $A_{ij} = 1$ if and only if there is a directed edge $(i, j) \in E$ in G.
- For weighted directed graphs, $A_{ij} = \ell_{ij}$ if and only if there is a directed edge (i, j) of length ℓ_{ij} in G. We will assume that for all $(i, j) \in E$, we have $\ell_{ij} > 0$ and we will use $A_{ij} = 0$ to denote that $(i, j) \neq E$.

Exercise 1:
Define a glass `Graph` which will correspond to undirected, unweighted graphs. The class should be initialised with the number of nodes n (e.g., `numNodes`) and should create an empty graph with n nodes (i.e., an adjacency matrix A with $A_{ij} = 0$ for all $i, j \in [n]$). The adjacency matrix should be the an attribute of the class set by the initialiser. The initialiser should also set an attribute for the set of nodes $\{1, 2, \ldots, n\}$.

For the `Graph` define the following methods:

- A method `add_edge` which inputs two nodes i, j and adds an edge (i, j) to E. The addition of the edge will be by appropriately modifying the adjacency matrix.

- A method `delete_edge` which inputs two nodes i, j and deletes the edge (i, j) from E. The deleting of the edge will be by appropriately modifying the adjacency matrix.

You should also add a method for printing the graph, i.e., the adjacency matrix that represents it. Ideally, you may redefine the `__str__` method for this.

Next, define a class `diGraph` as a subclass of `Graph`. Overload the `add_edge` and `delete_edge` methods to add and delete directed rather than undirected edges.

Finally, define a class `wdiGraph` as a subclass of `diGraph`. Overload the `add_edge` method to also input the length ℓ_{ij} of an edge (i, j) to be added. The addition of the edge will be by appropriately modifying the adjacency matrix. Additionally, define a new method called `edge_length` which inputs te endpoints of an edge (i, j) and returns the length ℓ_{ij} of the edge.

Task 1.2: Dijkstra, $O(mn)$ running time implementation

Now we are ready to implement Dijsktra's algorithm. For this lab sheet, we will be content with the "inefficient" implementation, namely the one that runs in time $O(mn)$. This is described in the first paragraph of "Implementation and Running Time" in Chapter 5.4 of the KT book, page 183. The idea is that we iterate over all nodes in $v \in V - S$ and for each such node, we consider all of its neighbours u in S. For each such pair, we compute the value ofthe distance $d(u) + \ell_{u,v}$ and we keep track of the smallest, as well as the node v that results in teh smallest distance.

Exercise 2:
Implement Dijkstra's Algorithm following the pseudocode of the lectures (also Page 180 of the KT book). In particular, define a function

```
dijkstra(graph, start_node)
```

which inputs a graph and a starting node s and finds for each node $v \in V$ of the graph, the minimum path distance from s. The function should return the list of shortest path distances (e.g., `spDistances`).

Some useful notes:

- You will need to keep track of the nodes in S and those in $V - S$ during the execution of the algorithm. You may use the Python `set` data type for this. A `list` can also be used, but the `set` is closer to the pseudocode for the algorithm, as S and $V - S$ are sets.

- It might be useful to use a helper function (e.g., `findNextNode`) which finds the next node to be added to S in the execution of the algorithm. This function will input the graph, the set S and the list of shortest path distances and will return the next node v to be considered and its shortest path distance $d(v)$.

Exercise 3:
Using the class `Graph` and its methods developed in Task 1.1. above, create the

graph of Figure 5.7. in Kleinberg Tardos (may be 4.7. in some versions). For consistency, let s, u, v, x, y, z correspond to nodes $0, 1, 2, 3, 4, 5$ respectively. Recall that the execution of Dijkstra on this graph yielded the result $[0, 1, 2, 2, 3, 4]$. Run your implementation of Dijkstra's algorithm on this graph as input and verify that you obtain the same result.

Task 2: Greedy Algorithms for the Interval Scheduling Problem

The goal of this task will be to implement three different algorithms for the interval scheduling problem, namely

- the algorithm that chooses the compatible interval with the earliest finishing time (optimal), `greedyEFT`,
- the algorithm that chooses the compatible interval with the earliest starting time, `greedyEST`,
- the algorithm that chooses the smallest compatible interval, `greedySmallest`,

and perform some experiements with them on random inputs.

Exercise 4:
Implement the three aformentioned algorithms in Python. Each algorithm will be a function that inputs a set of intervals with starting and finishing times, and outputs a set of compatible intervals. We can assume that the intervals are given to us by means of a dictionary with keys the id of the interval and values list of two elements, the starting times and finishing times. For example, a possible input could be

`{1: [0.3, 0.5], 2:[0.4,0.6], 3:[0.1, 0.8], 4:[0.7,0.8], 5:[0.2,0.3]}`.

The ouput of the function will be of a `set` data type, or of a `list` data type, containing the ids of the intervals (which are the same as the keys in the input dictionary). For example, a feasible schedule for the input above could be $\{1, 4\}$. An infeasible schedule would be $\{1, 2\}$ as intervals 1 and 2 overlap.

Remark 1: You will need a way to sort dictionaries in terms of certain elements of the lists that constitute their values. For example, for `greedyEFT` you will need a way to sort by the second element in the lists, which is the finishing time. One way to achieve this is via using

```
sortedIntervals=dict(sorted(intervals.items(), key=lambda e: e[1][1]))
```

For `greedyEST` this can be done similarly. For `greedySmallest`, you will need to sort based on the difference between the finishing and starting times, which is a bit more challenging.

Exercise 5:
Test the performance of these three algorithms in randomly generated instances. For $n = 5, 10, 20, 50, 100$, generate n intervals with their starting and finishing time drawn uniformly at random from $[0, 1]$, ensuring that the finishing time is

after the starting time. The easiest way to achieve this is to draw two values for each interval, and set the smallest one to the starting time and the largest one to the finishing time. Calculate the number of intervals that are included in the output of each algorithm. Repeat this K times (e.g., for $K = 100$) and calculate the average number of intervals that each algorithm includes in the output schedule (you may use a list to store intermediate calculations). Report your observations on the comparisons between those different algorithms. How close are the other two algorithms compared to `greedyEFT`, which is optimal?