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.

Solution:

```
import random

class Graph():

"The class Graph that encodes an undirected graph. For this implementation, the graph
is represented as an Adjacency Matrix."
"The graph is initialised as an empty graph (A_{ij}=0 for all i, j) with numNodes nodes.
The nodeSet contains the indices of the nodes, namely [0, ..., numNodes]."
"It has methods add_edge and delete_edge for adding and deleting edges.
The __str__ method has been defined to print the Adjacency Matrix."

def __init__(self, numNodes):
self.numNodes = numNodes
self.AdjacencyMatrix = [[0 for col in range(numNodes)] for row in range(numNodes)]
self.nodeSet = set(i for i in range(numNodes))

def __str__(self):
return str(self.AdjacencyMatrix)

def add_edge(self,node1, node2):
```

```
self.AdjacencyMatrix[node1][node2] = 1
self.AdjacencyMatrix[node2][node1] = 1

def delete_edge(self,node1,node2):
self.AdjacencyMatrix[node1][node2] = 0
self.AdjacencyMatrix[node2][node1] = 0

class diGraph(Graph):

"The class digraph is a subclass of graph for directed graphs. I
t overloads the methods add_edge and delete_edge to ensure that the edges are directed."

def add_edge(self,node1, node2):
self.AdjacencyMatrix[node1][node2] = 1

def delete_edge(self,node1,node2):
self.AdjacencyMatrix[node1][node2] = 0

class wdiGraph(diGraph):

"The class wdigraph (for weighted digraph) is a subclass of digraph. It overloads
the method add_edge to also specify the length of the edge. The adjacency matrix"
"now is not a binary matrix, but a matrix with numbers corresponding to the weights.
It also adds the new method 'edge length' for finding the length of an edge,
given by its endpoints."

def add_edge(self,node1, node2, length):
self.AdjacencyMatrix[node1][node2] = length

def delete_edge(self,node1,node2):
self.AdjacencyMatrix[node1][node2] = 0

def edge_length(self, node1, node2):
return self.AdjacencyMatrix[node1][node2]


def findNextNode(graph, S, spDistances):
"Function that finds the next node to be considered by Dijsktra"
"S is the set of explored nodes, feel free to use 'explored' instead
if you find it more informative. spDistances is the list of shortest
path distances from the start_node of Dijsktra."

unexploredNodes = graph.nodeSet.difference(S)
# The set of nodes that have not been explored yet, i.e., V-S

minDist = 1e7 # Set the distances to a sufficiently large value
nextNode = random.choice(list(unexploredNodes))
# Set an arbitrary (here random) node to be the current node.
# This will be updated in the nested for loops below.

# The for loops consider every node in V-S and for that, every edge to any node in S.
# The node u for which the value dist (i.e., d'(u) in the slides/KT book) is minimized
# is chosen as the next node.
# The distance is also return because this will be spDistances[u]
# (i.e., d(u) in the slides/KT book).
```

```
for newNode in unexploredNodes:
for oldNode in S:
if graph.edge_length(oldNode,newNode)!=0:
# If there is an edge (oldNode, newNode) - note that all edges will be strictly positive,
# '0' indicates the non-existence of an edge.
dist = spDistances[oldNode] + graph.edge_length(oldNode, newNode)
if (dist < minDist):
nextNode = newNode
minDist = dist

return nextNode, minDist

def dijsktra(graph, start_node):

# Initialise the set S of explored nodes to contain the start_node.
S = set()
S.add(start_node)

# Initialise the array of shortest path distances with the distance d(start_node)
# of the start_node from itself, which is 0. For all other nodes,
# add a string 'U' for 'unexplored'.
spDistances = ["U" for i in range(graph.numNodes)]
spDistances[start_node]=0

# The main while loop of Dijkstra. Repeat until all the nodes are explored (i.e., until S = V)
while S!= graph.nodeSet:

# Select node according to the min distance criterion, using the helper function findNextNode.
nextNode, nextNodeDist = findNextNode(graph, S, spDistances)

# Add the new node to the set of explored nodes S and record its shortest path distance.
S.add(nextNode)
spDistances[nextNode] = nextNodeDist

return spDistances



# Empty graph with 6 nodes, for the KT Figure 5.7. example the mapping will be
# s->0, u->1, v->2, x->3, y->4, z->5
myGraph = wdiGraph(6)

# Adding the edges with weights as in KT Figure 5.7.
myGraph.add_edge(0,1,1)
myGraph.add_edge(0,2,2)
myGraph.add_edge(0,3,4)
myGraph.add_edge(1,4,3)
myGraph.add_edge(1,3,1)
myGraph.add_edge(2,5,3)
myGraph.add_edge(2,3,2)
myGraph.add_edge(3,5,2)
myGraph.add_edge(3,4,1)


# Execute Dijsktra to find the shortest path distances
shortestDistances=dijsktra(myGraph, 0)
```

```
# Print the shortest path distances to the screen
print(shortestDistances)
```

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?

Solution:

```
import random

def greedyEFT(intervals):

"The 'Earliest Finishing Time' Greedy Algorithm, which considers the intervals
in terms of non-decreasing finishing time."
```

```
# This will be the output schedule of the algorithm, represented as a list with
# the order being the order in which the intervals where added to the output schedule.
schedule = []

# This sorts the intervals in terms of earliest finishing time.
sortedIntervals = dict(sorted(intervals.items(), key=lambda e: e[1][1]))

# The schedule is initialised to contain the first interval.
firstInterval = list(sortedIntervals.keys())[0]
schedule.append(firstInterval)

# The for loop in which the algorithm schedules the remaining intervals.
# They are considered in terms of non-decreasing finishing time.
for currInterval in sortedIntervals:
# This is the starting time of the interval we are considering for
# addition to the output schedule.
currIntervalStartTime = sortedIntervals[currInterval][0]
# If the starting time is before the finishing time of the last interval we
# scheduled, then we don't add it. Note that here the last interval to be added
# to the schedule is the one with the latest finishing time, so its is sufficient
# to only check for overlap with that.
lastIntervalFinishTime = sortedIntervals[schedule[-1]][1]

# If there is an overlap, do not add the interval to the schedule.
if currIntervalStartTime < lastIntervalFinishTime:
continue
else:
schedule.append(currInterval)                                    # Otherwise, add it.



return schedule


def greedyEST(intervals):

"The 'Earliest Starting Time' Greedy Algorithm, which considers the intervals
in terms of non-decreasing starting time."

# This will be the output schedule of the algorithm, represented as a list with
# the order being the order in which the intervals where added to the output schedule.
schedule = []

# This sorts the intervals in terms of earliest starting time.
sortedIntervals = dict(sorted(intervals.items(), key=lambda e: e[1]))

# The schedule is initialised to contain the first interval.
firstInterval = list(sortedIntervals.keys())[0]
schedule.append(firstInterval)
lastIntervalFinishTime = sortedIntervals[firstInterval][1]

# The for loop in which the algorithm schedules the remaining intervals.
# They are considered in terms of non-decreasing starting time.
for currInterval in sortedIntervals:
# This is the starting time of the interval we are considering for addition to the output schedule.
currIntervalStartTime = sortedIntervals[currInterval][0]
```

```
# If there is an overlap, do not add the interval to the schedule.
if currIntervalStartTime < lastIntervalFinishTime:
continue
else:
schedule.append(currInterval)                                    # Otherwise, add it.

# Update the latest finishing time of any interval in the output schedule to
# be the one of the newly added interval. Note that it is not possible for the
# newly added interval to finish before any of the intervals already scheduled,
# as then it would not be compatible with that inteval.
lastIntervalFinishTime = sortedIntervals[currInterval][1]



return schedule

def greedySmallest(intervals):

"The 'Smallest Interval' Greedy Algorithm, which considers the intervals
in terms of non-decreasing size."

# This will be the output schedule of the algorithm, represented as a list with the order
# being the order in which the intervals where added to the output schedule.
schedule = []

# Create a new dictionary that has the interval ids as keys and their lengths as values.
intervalDifferences = {}
for interval in intervals:
intervalDifferences[interval] = intervals[interval][1]-intervals[interval][0]

# Sort the intervals in terms of non-decreasing length
sortedIntervals = dict(sorted(intervalDifferences.items(), key=lambda e: e[1]))

# The schedule is initialised to contain the first interval.
firstInterval = list(sortedIntervals.keys())[0]
schedule.append(firstInterval)

# The for loop in which the algorithm schedules the remaining intervals. They are considered
# in terms of non-decreasing starting time.
for currInterval in sortedIntervals:

# This is the starting time of the interval we are considering for addition to the output schedule.
currIntervalStartTime = intervals[currInterval][0]

# This is the finishing time of the interval we are considering for addition to the output schedule.
currIntervalFinishTime = intervals[currInterval][1]

overlap = False
for otherInterval in schedule:

# Use variables for ease of reference when checking for overlap
otherIntervalStartTime = intervals[otherInterval][0]
otherIntervalFinishTime = intervals[otherInterval][1]

# If there is an overlap, do not add the interval to the schedule.
```

```
# PLEASE MAKE SURE TO EDIT THIS TO BE IN ONE LINE IF YOU C/P
if otherIntervalStartTime < currIntervalFinishTime and
               currIntervalStartTime < otherIntervalFinishTime:

overlap = True
break

if overlap == False:
schedule.append(currInterval)  # Otherwise add the interval.

return schedule


def randomIntervals(numIntervals):

"A function that generates a number of intervals (given as input) uniformly
at random from the [0,1] interval."

intervals = {}

for interval in range(1,numIntervals):

a1 = random.uniform(0,1)
a2 = random.uniform(0,1)

# Set the smallest of the two drawn values as the starting time
# and the largest as the finishing time.
intervals[interval] = [min(a1,a2),max(a1,a2)]

return intervals


def experiment(K):

"The main function which runs the experiment. K is the number of repeats
for each value of the number of intervals. The function calculates"
"the number of intervals scheduled by each of greedyEFT, greedyEST,
and greedySmallest for uniform randomly generated intervals."

for numIntervals in [5,10,20,50,100]:

# The lists where the number of scheduled intervals for each run will be stored.
numIntervalsEST = []
numIntervalsEFT = []
numIntervalsSmallest = []

for _ in range(1,K): # Repeat the experiment K times.

intervals = randomIntervals(numIntervals) # Generate random intervals

# Run the algorithms and compute the number of intervals they schedule.
scheduledIntervalsEFT = greedyEFT(intervals)
numIntervalsEFT.append(len(scheduledIntervalsEFT))

scheduledIntervalsEST = greedyEST(intervals)
numIntervalsEST.append(len(scheduledIntervalsEST))
```

```
scheduledIntervalsSmallest = greedySmallest(intervals)
numIntervalsSmallest.append(len(scheduledIntervalsSmallest))

# Compute the averages.
averageIntervalsEST = sum(numIntervalsEST)/K
averageIntervalsEFT = sum(numIntervalsEFT)/K
averageIntervalsSmallest = sum(numIntervalsSmallest)/K

# Print the results on the screen
# PLEASE MAKE SURE TO EDIT THESE TO BE IN ONE LINE EACH IF YOU C/P
print("When we have ", numIntervals, " intervals, GreedyEFT schedules on average,"
,averageIntervalsEFT, " intervals.")
print("When we have ", numIntervals, " intervals, GreedyEST schedules on average,"
,averageIntervalsEST, " intervals.")
print("When we have ", numIntervals, " intervals, GreedySmallest schedules on average,"
 ,averageIntervalsSmallest, " intervals.")

return

experiment(100)
```