Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 5: Dynamic Programming

In this lab we will be implementing *dynamic programming algorithms* that were covered in the lectures.

Task 1: Implementation of the Bellman-Ford Algorithm

We will first implement the Bellman-Ford algorithm. In the previous lab sheet we implemented (weighted) graphs using the adjacency matrix representation. In particular, for those graphs we had $A_{ij} = \ell_{ij}$ if and only if there is a directed edge (i, j) of length ℓ_{ij} in G. Contrary to the previous lab sheet though, we will assume that for all $(i, j) \in E$, we have $\ell_{ij} \in \mathbb{R}$. We will use $A_{ij} = L$ to denote that $(i, j) \neq E$, where L will be defined to be some very large global variable (e.g., L=1000).

Your task will be to implement the Bellman-Ford algorithm using the pseudocode presented in Lecture 20. In particular, you may define the $2D$-array M to be a list of lists: for example the element $M[3, 5]$ will be the 5th element of the third list (indexed from 1). The algorithm should designate a node v (given as input as an integer), and compute the costs of the cost-minimizing paths from every node to v. In particular, define a function

```
bellman_ford(graph, target_node)
```

The graph argument will be of the class `graph` that you defined in Lab Sheet 4. The `target_node` will be an integer corresponding to a node of the graph. The function should return the $2D$-array M (as a list of lists).

Run your function on the graph of Figure 6.23a in Kleinberg-Tardos (the same example that was used in Lecture 20) with node t as input. Compare the outcome of your function with the table presented in Figure 6.23b.

Task 2: Algorithms for the Weighted Interval Scheduling Problem

In Lab Sheet 4, you implemented three different algorithms for the interval scheduling problem, namely

- the algorithm that chooses the compatible interval with the earliest finishing time, `greedyEFT`,

- the algorithm that chooses the compatible interval with the earliest starting time, `greedyEST`,

- the algorithm that chooses the smallest compatible interval, `greedySmallest`,

and performed some experiments with them on random inputs.

Exercise 1:
In this task you will consider the weighted interval scheduling problem and you will implement two more algorithms:

- the algorithm that chooses the compatible interval with the largest weight,

- the dynamic programming algorithm that finds an optimal schedule that we saw in the lectures.

Each algorithm will be a function that inputs a set of intervals with starting and finishing times as before, but now it will also input the weights of the intervals. For that, we can enhance our dictionary representation from Lab Sheet 4 to now have a list of three elements as the value, namely the starting time, the finishing time, and the weight, respectively. The output will be a set of compatible intervals.

For example, a possible input could be

```
{1: [0.3, 0.5, 4], 2:[0.4,0.6, 3], 3:[0.1, 0.8,7],
4:[0.7,0.8], 5:[0.2,0.3,3]}
```

The output 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 to sort the dictionary in terms of the third element of their value (which is a list), which is the weight of the interval. You can do that very similarly to the way you sorted the dictionaries in Lab Sheet 4.

For the dynamic programming approach, you can use a list for the $1D$-array M. The list will store at $M[i]$ the weight of the optimal solution using the intervals $1, \ldots, i$. You may refer to the pseudocode as given in the lectures.

Exercise 2:
Now that you have the implementations of all five algorithms, you will again test them on randomly generated instances to test and compare their performance. 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. For each interval, also draw a weight in $[0, 20]$ uniformly at random. Calculate the total weight of intervals that are included in the output

of each algorithm. Repeat this K times (e.g., for $K = 100$) and calculate the average weight 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.
In addition to the performance of the algorithms in terms of the optimisation objective, you can also test their running time. Using the `timeit` function, measure the average time that each algorithm requires over the sequence of K runs. Print the measured times on the screen.