Comp 251: Assignment 3

Instructor: Jérôme Waldispühl

Due on March 26th at 11h59

- Your solution must be returned electronically on MyCourse.
- Written answers and programming questions must be returned in separate submission folders on MyCourse.
- The only format accepted for written answers are PDF or text files (e.g. `.txt` or `.rtf`). PDF files must open on SOCS computers. Any additional files (e.g. images) must be included in the PDF.
- Do not submit a compressed files (e.g. zip files). Upload instead each PDF or text file individually.
- The solution of programming questions must be written in java. Submit the Java source file only (i.e. `.java`). Your program should compile and execute of SOCS computers in a terminal. Java files that do not compile or execute properly on SOCS computer will not be graded.
- To some extent, collaborations are allowed. These collaborations should not go as far as sharing code or giving away the answer. You must indicate on your assignments the names of the persons with who you collaborated or discussed your assignments (including members of the course staff). If you did not collaborate with anyone, you write "No collaborators" at the beginning of your document. If asked, you should be able to orally explain your solution to a member of the course staff.
- Unless specified, all answers must be justified.
- When applicable, your pseudo-code should be commented and indented.
- The clarity and presentation of your answers is part of the grading. Be neat!
- Violation of all rules above may result in penalties or even absence of grading (Please, refer to the course webpage for a full description of the policy).
- Partial answers will receive credits.
- The course staff will answer questions about the assignment during office hours or in the online forum at `https://osqa.cs.mcgill.ca/`. We urge you to ask your questions as early as possible. We cannot guarantee that questions asked less than 24h before the submission deadline will be answered in time.

1. (40 points) We will implement the Ford-Fulkerson algorithm to calculate the Maximum Flow of a directed weighted graph. Here, you will use the files WGraph.java and FordFulkerson.java, which are available on the course website. Your role will be to complete two methods in the template FordFulkerson.java.
   The file WGraph.java is the similar to the file that you used in your previous assignment to build graphs. The only differences are the addition of setters and getters methods for the Edges and the addition of the parameters "source" and "destination". There is also an additional constructor that will allow the creation of a graph cloning a WGraph object. Graphs are also encoded using a similar format than the one used in the previous assignment. The only difference is that now the first line corresponds to two integers, separated by one space, that represent the "source" and the "destination" nodes. An example of such file can be found on the course website with the file `ff2.txt`. These files will be used as an input in the program `FordFulkerson.java` to initialize the graphs. This graph corresponds to the same graph depicted in [CLRS2009] page 727.
   Your task will be to complete the two static methods `fordfulkerson(Integer source, Integer destination, WGraph graph, String filePath)` and `pathDFS( Integer source, Integer destination, WGraph graph)`. The second method `pathDFS` finds a path through a Depth First Search (DFS) between the nodes "source" and "destination" in the "graph". You must return an ArrayList of Integers with the list of unique nodes belonging to the path found by the DFS. The first element in the list must correspond to the "source" node, the second element in the list must be the second node in the path, and so on until the last element (i.e., the "destination" node) is stored. The method `fordfulkerson` must compute an integer corresponding to the max flow of the "graph" and the graph itself. The method `fordfulkerson` has a variable called `myMcGillID`, which must be initialized with your McGill ID number.
   Once completed, compile all the java files and run the command line `java FordFulkerson ff2.txt`. Your program must use the function writeAnswer to save your output in a file. An example of the expected output file is available in the file `ff226000000.txt`. This output keeps the same format than the file used to build the graph; the only difference is that the first line represents now the maximum flow (instead of the "source" and "destination" nodes). The other lines represent the same graph with the weights updated with the values that represent the maximum flow. The file `ff226000000.txt` represent the answer of the example showed in [CLRS2009] Pag 727. You are invited to run other examples of your own to verify that your program is correct.

2. (40 points) We want to implement the Bellman-Ford algorithm for finding the shortest path in a graph where edge can have negative weights. This question extends the previous question on the implementation of the Dijkstra's algorithm done in the assignment 2. You will need to execute this program to use the same auxiliary class `Wgraph` used in question 1. Your task is to fill the method `BellmanFord(WGraph g, int source)` and `shortestPath(int destination)` in the file `BellmanFord.java`.
   The method `BellmanFord` takes a object `WGraph` named `g` as an input (See Assignment 2) and an integer that indicates the source of the paths. If the input graph `g` contains a negative cycle, then the method should through an exception. Otherwise, it will return an ob-

ject `BellmanFord` that contains the shortest path estimates (the private array of integers `distances`), and for each node its predecessor in the shortest path from the source (the private array of integers `predecessors`).
The method `shortestPath` will return the list of nodes as an array of integers along the shortest path from the source to the node destination. If this path does not exists, the method should throw an exception.
Input graphs are available on the course webpage to test your program. Nonetheless, we invite you to also make your own graphs to test your program.

3. (10 points) Give an $\mathcal{O}(n \lg n)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (Hint: Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i-1$. Maintain candidate subsequences by linking them through the input sequence.)
We do not ask you to prove your algorithm, but a *complete exact* proof will receive a bonus.

4. (10 points) Apply the Neddleman-Wunch algorithm to find the optimal sequence alignment between the sequences $\omega_1 = AACT$ and $\omega_1 = GAT$. We set all the substitution, deletion and insertion costs at 1, and the match score to -1. Here, your goal will be to compute the alignment with the minimum score. Show the complete dynamic programming table filled by the algorithm, and show the path(s) corresponding to the optimal sequence alignment. Write down the alignment and its cost. (Note: Alternatively, you can choose to use the edit scores used in class and maximize the score.)