Homework Batch III: Routing Algorithms

Algorithmic Design - A.A. 2020-2021

Valentina Blasone

Due date: 18/05/2021

Instructions

1. Implement the binary heap-based version of the Dijkstra's algorithm.
2. Consider the contraction hierarchies presented during the course. Assume to deal with graphs that can be fully represented in the memory of your computer. Implement:
   (a) an algorithm to add the shortcuts to a graph;
   (b) a bidirectional version of Dijkstra algorithm that can operate on the graphs decorated by the algorithm at Point 2a.

Solution

As a general comment, in all implementations the graph is to be considered implemented as an *adjacency list*. In python, the adjacency list is implemented using a list of lists. Moreover, we consider to deal with the internal representation of the graph, hence the nodes are to be assumed labelled as $0, ..., n \in \mathbf{N}$ ordered and with no missing numbers.

A final general consideration is on the definition of the value used to initialize the vector of distances in Dijkstra's algorithm. In the implementation, *inf* provided by the *math* module in python is used. A possible alternative to this solution, could have been to compute the sum of all the weights in the graph, and finally add 1 to the obtained value. The function is included in the python file `hw03_vb.py` as `compute_inf`.

Exercise 1

The idea is to use a binary heap as the data structure to handle the queue in the Dijkstra's algorithm. The binary heap is implemented in the python file `binheap.py`, in in the `binheap` class. The data structure is implemented similarly to what we did during lectures. The main difference is in the introduction of an additional list to store the structure of the heap, which is implemented as a member variable, `self._V`. To be compliant throughout the implementation, the methods `_swap_keys`, `insert`, `remove_minimum` and `decrease_key` were modified accordingly.

Then, the auxiliary functions necessary for the Dijkstra algorithm were implemented, i.e., `init_sssp` and `relax`. The implementation reflects the pseudo-code in the slides, therefore no more comment are added.

Finally, the Dijkstra algorithm is implemented, again taking as reference the pseudo-code included in the slides. A new total order is defined, based on the weight values. The algorithm takes as input a graph, `G`, and a source node, `n` and returns in output two lists (of length equal to the number of nodes in the graph):

- `dist`: which stores the length of the shortest path, from `s` to each of the nodes in `G`;
- `pred`: which stores the predecessor of each destination node in the shortest path with `s` as source.

The algorithm is implemented in the `dijkstra` function in `hw03_vb.py`.

Exercise 2 (a)

Some additional auxiliary functions were implemented: `find_predecessors`, `remove_node`, `find_weight`. A brief explanation can be found in the documentation of the code.

It is assumed that the original graph should not be modified, therefore two auxiliary graphs are considered:

- `G_overlay`: this graph, for a fixed node `n`, represents the corresponding overlay graph of `n`, i.e., the graph in which all the nodes with importance lower than n and n itself have been removed and the corresponding shortcuts have been added.
- `G_updated`: this is the graph the will be returned as output of the algorithm. It is the original graph (not removing any node) enriched with the shortcuts. In the assumption that the input graph `G` can be modified in place, `G_updated` is not necessary and we can directly work with `G`, without other changes in the code.

The algorithm loops over all the nodes in the graph. Given a node `n`, first its predecessors in `G_overlay` are derived. Then, a list containing its adjacency nodes in `G_overlay` is also created. At this point, the node `n` is removed from `G_overlay`, by calling the function `remove_node`. Then, we loop on the predecessors of `n` (derived above). We call Dijkstra in order to derive the shortest distances from the predecessor `p`. In a nested for loop we loop over the adjacent nodes of `n` (derived above). The idea is to sum the weight of the edge from `p` to `n` and the weight of the edge from `n` to its adjacent node, `a`. If the distance obtained is strictly smaller than the distance from `p` to `a` find with Dijkstra (which, we should remember, is computed in the overlay graph of `n`, where `n` has been removed), then it means that removing `n` modified the sssp from `p` to `a` and we need to add the shortcut. The shortcut is added to both `G_overlay` and to `G_updated`. Note that we allow for a single edge between two nodes, therefore if a node from `p` to `a` with a larger weight was already present, this edge is removed when the shortcut is added.

Finally `G_updated` is returned. The code is implemented in the `shortcuts` function in `hw03_vb.py`.

Exercise 2 (b)

The algorithm receives in input a graph decorated with all its shortcuts (as the output from the code in Exercise 2 (a)), a source node `s` and a destination node `d`. It returns in output the shortest distance between the two nodes.

An additional auxiliary function was implemented: `make_forward_backward_graphs`. The function takes in input a graph `G` and produces in output two graphs:

- `G_f`: which contains the edges (u, v) of `G` s.t. $u < v$
- `G_b`: which contains the edges (u, v) of `G` s.t. $v < u$, reversed

Assume we want to compute the distance between a source node, `s` and a destination node, `d`. The idea is to perform a bidirectional version of Dijkstra, by alternating a forward pass on `G_f`, with source `s` and a backward pass on `G_f` with source `d`. The algorithm begins by initializing the vectors which will contain the candidate distances for the forward and backward pass, `dist_f` and `dist_b`, using `initi_sssp`. Then, the relative queues are implemented, using the bin-heap as data structure. Two additional lists are initialized, `discovered_f` and `discovered_b`, which will store the nodes finalized in the forward and backward pass.

In each forward (backward) pass, a node is extracted from the corresponding queue and is added to the correct list of discovered nodes, `discovered_f` (`discovered_b`). All the adjacent nodes in `G_f` (`G_b`) are relaxed, using the `relax` function.

The stopping criterion for the algorithm consists in two considerations. The first consideration is that if in the forward (backward) pass a node already discovered in the previous backward (forward) passes is discovered, that means that the two "frontiers" of the forward and backward search have intersected. However, this condition is not enough to terminate the algorithm, because this shortest path in terms of edges number may not be the shortest path in terms of distance (weights sum). Therefore we need an additional condition, to return the correct shortest distance. The idea is to compare the smallest distance from `s` to `d` among the candidate distances with the minimum distances from s considering the nodes still in the queue.

The code is implemented in the `dijkstra_bidir` function in `hw03_vb.py`.

Testing

Herein the two graphs used for testing the algorithms are graphically shown. The graphs are called `G0` and `G1` as in the python code provided with this report.

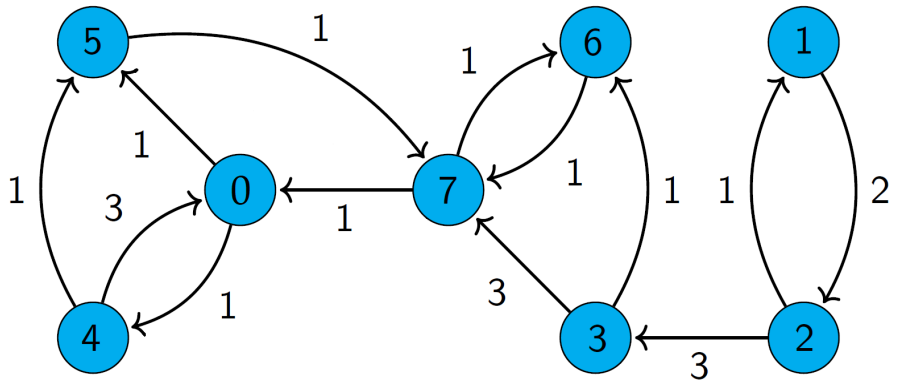

Figure 1: Graph `G0`

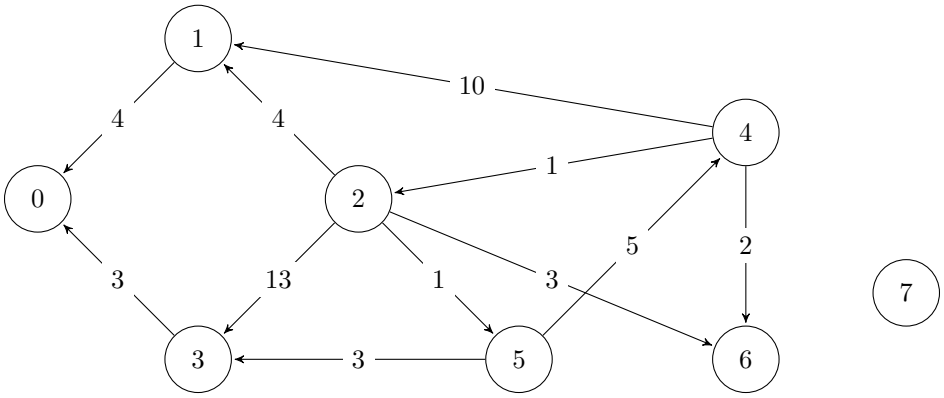

Figure 2: Graph `G1`