Homework III

Liberatori Benedetta

May 2021

1 Introduction

The file `graphs` contains the class `graph` which implements the data structure in Python as dictionaries of dictionaries. The outer dictionary has one key for each node in the graph. For each key then the associated value is a dictionary in which the keys are the neighbouring nodes and the values the weights of the edges. Additional functions are meant to insert edges, print the graph and retrieve the number of nodes in the graph.

The file `binheap_dijkstra` contains the class `binheap` written during the lessons, conveniently modified in order to efficiently implement the priority queue. In the following functions, heaps will contain pairs of node name and distance from a selected source, totally ordered w.r.t. such distance (`pair_order`). In order to efficiently implement the `decrease_key` function, which needs to reach the key to be updated in the heap, the class has been modified adding a member `indexes`: a list storing in the (i-1)-th position the index of the pair associated to the i-th node in the heap. This complies with the fact that heaps will contain pair corresponding to graphs' nodes, which are (or can conveniently labelled as) integers ranging from 1 to the number of nodes.

2 Exercise 1

The function `Dijkstra` implements the heap-based version of Dijkstra's algorithm. It returns two lists: one storing in the (i-1)-th position the shortest distance of the i-th node from the selected source, while the other one the predecessor in the shortest path. At the beginning these lists are initialized: the former with an unbounded upper value in each position (except for the one corresponding to the source which is set to 0), which can be represented in Python as float('inf'), and the latter with None. Then the priority queue is filled as already explained. The minimum node is extracted and the function `relax` is used to update the distance of each adjacent node if needed. As mentioned above the update of the binary heap exploit the additional member and this ensures that the cost is logarithmic in the number of nodes and not linear.

3 Exercise 2

The function `shortcut` implements a contraction of a selected node v in a graph. It takes as an additional input a set of nodes already contracted, in order for the procedure to be iterative. For each couple of not-yet-contracted predecessor-successor $u-w$ of v adds a shortcut edge from u to w iff the shortest path between these contains v. This is checked via the function `witness_search` which takes as input the two nodes u, w, the set of already contracted nodes and the distance of the path from u to w through v. It performs a forward search from u to w in a Dijkstra-like fashion, in which only the not-yet-contracted nodes are considered and which returns as soon as w is removed from the priority queue and thus finalized. In addition, the distance of the path from u to w through v is used as an upper bound and searches passing through edges with weights greater than this distance are interrupted. Indeed, a precondition for this algorithm is that the weights are positive numbers and so, if an edge through x has weight greater than this upper bound, than a possible path from u to w shorter than the upper bound does not include x. This avoids unnecessary operations, reducing the space-search for a path shorter than the possible shortcut. The function `preprocessing` wraps up this procedure decorating a given graph with all the needed shortcuts, contracting nodes w.r.t growing importance (from node 1 to node cardinality of the nodes in the graph). The result is the union of the so called overlay graphs in the Contraction Hierarchies algorithm.

The function `Bid_Dijkstra` implements a bidirectional version of Dijkstra's algorithm that can operate after the step by the `preprocessing` function. Taking a graph $G = (V, E)$, a source and a destination nodes, returns the shortest distance from the source to the destination. It builds two graphs U and D s.t.: $U = (V, \{(v, w) \in E \quad | \quad v < w\})$ and $D = (V, \{(w, v) \in E \quad | \quad v > w\})$. Edges in D are transposed in order to perform a backward search as a forward search. Then the function performs two forward searches: one in U starting from the source node and one in D starting from the destination node. The distance to be returned, initialized as float('inf'), is updated whenever a node is reached in both graphs and is is smaller the the current value computed. In order to check if a node has been already visited in one of the two searches the list storing the partial results for the shortest distances are used. If the value stored in position i-1 in the list containing the distances from the source is different from float('inf'), then the i-th node has been already encountered in the forward search and the same holds for the backward search.