Homework

Mami Ciro Antonio

May 2021

1

1.1 Graph and Binheap

Starting from the graph itself; I've implemented a class edge, with properties source, destination and weight, and a main class graph, which contains the nodes, stored in a list, and an adjancency list. The latter is implemented as a dictionary where the keys are the sources, and the value is a list of tuple (destination,weight). Before going through the implementation of Dijkstra I performed some modification to the class binheap implemented during the lectures, indeed there were no way to access a particular value of the heap without knowing his position, and whenever we perform an operation, e.g. remove a value, the position of the values in the heap list change. In order to solve this problem I've added another list to the properties of the heap, which store the position of each value in the original list of the heap. Before being more precise I must specify that in the dijkstra algorithm I will use as values for the heap tuples $(node, distance from the source)$. Regarding the position list I've implemented it in such a way that at the index $[node - 1]$ of the position list we will find the index of the heap list which store $node$ (I'm assuming that the node in the graph are enumerated from 1 to n, while of course the index of a list goes from 0 to n-1). Moreover, to preserve the consistency of this list I modified the function "swap_keys" so that whenever we swap two values of the heap we also swap their index in the position list. Finally I've modified "decrease_key", indeed now if I want to decrease the node i, I pass to the function $i - 1$ and then retrieve its position from the position list.

1.2 Dijkstra

For what concern Dijkstra algorithm I've followed the pseudo-code, the function Dijkstra takes two inputs, the graph and the source; the first thing I do (after checking that the source is in to the graph) is to initialize two list, one which stores the distance from the source, and one which stores the predecessor, again of course due to the indexes of python the distance of node i will be in position $i - 1$. Then I initialize the Heap with a list of tuples $(node, distance)$, after, while the Heap is not empty, I extract the minimum value u from the Heap and for each edge starting from u I compute Relax. The Relax function takes as inputs the graph, the Heap, u, v which is the destination of the edge starting from u, the weight of that edge and finally the two list distance and predecessor. Inside Relax, if the condition is verified, I decrease the value of the distance of v inside the heap, and also modify its distance in the list and its predecessor.

2

a) In order to adding shortcut to the Graph I've implemented two function: A function named contract, which will take as inputs the graph, the nodes to contract and a list of nodes already contracted (it will come useful for bidirectional dijkstra, if we contract just one node it will contain only that node); and a function named contraction_dijkstra which is a modified version of dijkstra. Regarding the function contract, as first thing I find out which are the nodes which have edges that lead to the node to be contracted (only if those are not already contracted of course), and I save their value and the weight of the edge. Then for each of those nodes (let's call them sources) that lead to the value to be contracted (let's call it s for simplicity) I scan all the edges starting from s and save the distance from sources to the destination of those edges, this will be weight of the shortcut. Nevertheless we only add this shortcut if there is not another path from sources to the destination not passing from s which is shorter, this is checked with the modified version of Dijkstra; which takes the graph, the source s, the destination t, the weight of the shortcut, and the nodes already contracted. In this function we save a lot of computation w.r.t. to dijkstra thanks to mainly three things: if we extract t from the Heap we can return since we already computed the shortest path from s to t, second we do not have to check the path which start or end in one of the nodes already contracted (this is useful when we contract many nodes), and lastly if the weight of an edge we are checking is greater than the shortcut, then we can avoid computing Relax, since we are sure that no path passing from that edge will be shorter than the shortcut.

b) I've implemented the bidirectional Dijkstra in this way: the function takes as input the graph, the source s and the destination t. Firstable I perform a contraction on all the nodes of the graph, in this way we have all the shortcut. Then I initialize two new graph, a forward, which contains only the edges going from a less important node to a more important one, and a backward, with only edges which goes from more important to less, but in this case they are reversed w.r.t. to the original graph. Afterward, I initialized two list distance (from s and from t) and two list predecessor, as well as two Heap, one for the source and one for the destination. Then while nor of the Heap are empty, I extract the minimum from both the Heap (u and v). After this I scan all the edges of u and all the ones of v, computing Relax as usual, but then I update the distance as the minimum of the distance until that moment and the sum of the distance of s from the destination of the edge starting from u plus the distance from t to the destination of the edge of u (this is for the edge from u, the same but reversed happen for the edge from v). In this way we get our distance even if the forward and the backward do not arrive at the same node in the same moment.