DSSC - Algorithmic Design

Homework 6

Patrick Indri

May 14, 2019

1 Introduction

The aim of this assignment is to implement both an array-based and a heap-based version of the Dijkstra's algorithm. The `/code` folder contains both implementations.

Compile and run

The `makefile` provided can be used to compile the code. Using `make` or `make all` the code gets compiled in a verbose version `main.x`, useful to test the program and see its output. Running `make benchmark` results in the compilation of `benchmark.x` which benchmarks the code and provides a cleaner output, ready to be plotted. The script `run.sh` can be used to perform a benchmark of the code.

2 Implementation

The implementation exploits the `Graph` structure, which stores a `IntMatrix` containing the matrix representation of the graph, an array of vertices (implemented by means of the `Vertex` structure) and the size of the graph.

A graph must be initialised using the `InitialiseGraph` function, passing the matrix of weights. The `RandomlyFillSparseIntMatrix` can be used to allocate such matrix (a parameter regulates how sparse the matrix will be). Graphs can be reset using the `ResetGraph` function, which resets all the distance information stored in the vertices,

preparing the graph to be freshly passed to another algorithm. `DeallocateGraph` takes care of deallocating both the array of vertices and the matrix of weights. It should be noted that, as presented, the Dijkstra's algorithms assume the `Graph` they investigate to be correctly initialised. Once the graph has been processed by one of the algorithms (which require the choice of the starting node), the `ShortestPath` function gives access to the distance of each of the nodes and prints the best path.

The Dijkstra's algorithm requires to treat infinite distances. The `C` programming language does not handle infinity when dealing with integers. For this reason, infinity has been encoded as the maximum integer available (`INT_MAX` from `limits.h`). This does not avoid overflows: the user must be cautious in dealing with extremely large weight matrices or extremely large weights.

2.1 Array-based

The array-based implementation uses the `QArray` structure to implement an array-based queue. The array contains (similarly to the `Graph`) an array of vertices and an integer array. The integer array encodes the extraction process: once a position of the `Vertex` array has been extracted, the same position of the integer array is turned from `0` to `1`, and the same position will be no longer considered when looking for minimums. The relaxing phase of the algorithm updates both the distances in the queue and in the graph.

2.2 Heap-based

To implement a heap-based queue, the heap implementation proposed in *Homework 3* and *Homework 4* has been taken as a starting point. The functions were adapted to deal with `QHeap` structure, which holds the heap nodes implemented as `HeapElem`. Each node of the heap queue contains the index of the element it contains and a pointer to its distance (the node distance acts as the ordering key for the heap). This choice allows distances to be allocated on a separate array: as long as an elements remains in the heap, the distance corresponding to its index has a fixed location in memory and can be directly accessed. This simplifies the relaxation procedure, since it avoids traversing the queue to update the distance of a specific index [1]. Once distances have been update, running `Heapify` on the `QHeap` restores the heap property.

3 Benchmark

`benchmark.x` tests both implementations with graphs of `100` to `5000` vertices, printing the execution time.

Fig. 3.1

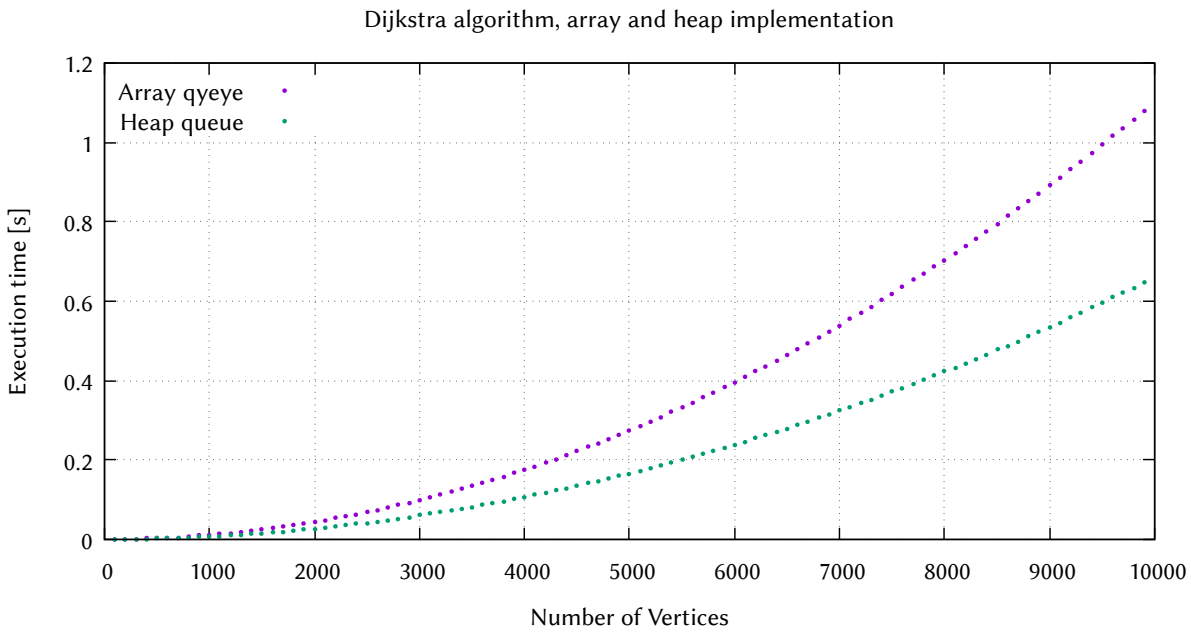

Fig. 3.1 Shows the results of the benchmark: the heap-based implementation provides better results in terms of execution time and, as expected, seems to scale better with the number of vertices.

References

[1] Mo Chen et al. *Priority queues and dijkstra's algorithm*. Computer Science Department, University of Texas at Austin, 2007.