# Dijkstra Algorithm Implementation

Maria Grazia Berni

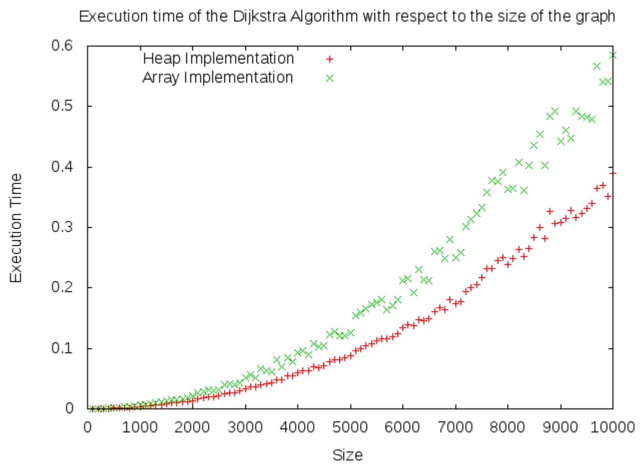
## Brief Description of the Implementation

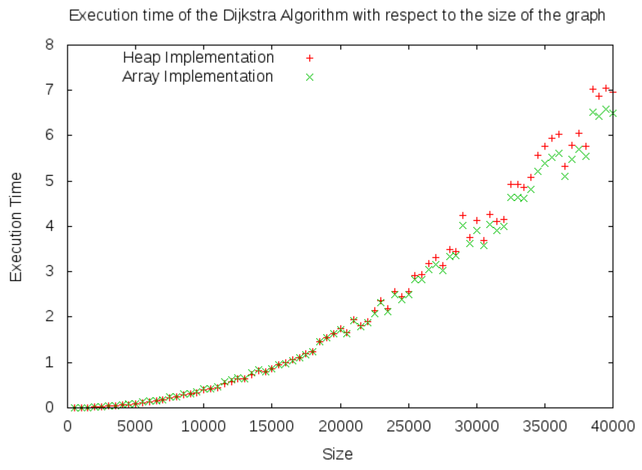
The goal of Dijkstra algorithm is to search the shortest path in a weighted graph from a source node to all the others nodes in the graph. The nodes of the graph are represented by the struct node, consisting of an integer *id*, which identify the node, an integer *pred*, wich identify the predecessor of the node in the path, and an integer *distance*, the distance from the source. The graph is represented by a struct containing the array of the nodes of the graph itself, the adjacency matrix and an integer $infinity$, set to a value greater than the sum of the weights of all the edges in the graph, so that if the distance of a node from the source is equal to this value, it means that that it is not reachable from the source. In the array based implementation of the Dijkstra algorithm, the queue is initialized by saving in it all the memory addresses of the original nodes in the graph. In the minimum search operation, the address of the minimum is saved in the first position of the queue and then the queue is pointed one position forward. Therefore, these operations do not correspond to any movement of the original nodes. Then we proceed to iterate on the nodes still present in the queue verifying the connection with the node just extracted from the queue and updating, when appropriate, the distance from the source and the predecessor. This must be repeated until the queue will be empty.

In the heap based implementation, the version of the heap used doesn't switch the elements of the heap, but uses an array of indices, so also in this case there are no displacements of the original nodes in the memory. The total order of the heap is guaranteed by comparing the distance argument of the struct node. The implementation is straightforward using the array which records only the locations of the nodes that are still in the heap after the minimum extraction. We proceed as in the previous case but the distance is set by the function $decrease\_Key$.

## Time Execution Analysis

Setting $n$ as the number of nodes in the graph and $E$ as the number of edges in the graph, the complexity of the Dijkstra algorithm using the array based implementation is $\mathcal{O}(n^2 + E)$, while the complexity using the heap implementation is $\mathcal{O}((n + E)\log n)$. In my implementation I used the adjacency matrix in both cases, rather than an adjacency list. This has no negative effect on performance when using dense matrices. Here some performance test:





As shown in the pictures, the heap version of the Dijskra algorithm performs better then the array version for small sizes of the graph, while for bigger sizes it starts performing worse. In fact, in all the tests I have used very dense matrices to represent the edges of the graph, so that the number of the edges is of the order of $n^2$, so the complexity in the case of the array version is $\mathcal{O}(n^2)$ while in the case of the heap version it becomes $\mathcal{O}((n^2)\log n)$. So in the case of dense graphs, the heap version asymptotically performs worse then the array version.