Analysis of Dijkstra’s Algorithm[*]

Sachin Prasanna

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

***Abstract* – This paper gives an analysis of the Dijkstra’s Algorithm, which solves for the single source shortest path on a positive weight edged directed acyclic graph.**

Key Terms* – *Dijkstra’s Algorithm, Vector, Priority Queue, Set, Fibonacci Heap

I. Introduction

Dijkstra’s Algorithm is a greedy algorithm which solves the shortest path problem by visiting each node of the graph, and trying to relax all edges connected to that node. This process of relaxation gives the shortest path from the source node to all nodes of the graph. The algorithm may not work if some edges are negative weighted.

II. GENERAL PSEUDO CODE

```
DIJKSTRA (G, w, s)
1    INITIALIZE-SINGLE-SOURCE (G, s)
2    S = Φ
3    Q = G.V
4    while Q ≠ Φ
5        u = EXTRACT-MIN (Q)
6        S = S ∪ {u}
7        for each vertex v ∈ G.Adj[u]
8            RELAX (u, v, w)
```

III. TIME COMPLEXITY ANALYSIS

Conducting a line-by-line time analysis, we have

1. This step takes O(V), as all vertices are initialized with their distances from source.
2. This set contains all visited nodes, and takes O(1) time.
3. This takes O(V), a data structure is created storing the nodes.
5. This is the **extract minimum** of the data structure, which extracts the minimum element of the chosen data structure. This varies according to the data structure chosen.
7. Runs for (V-1) vertices in the worst case, so the complexity of this line translates to O(V).
8. Relaxation basically translates to the **decrease key** operation. This operation is used to update the data structure used when the distance to a vertex is reduced. This operation involves removing the vertex from the data structure, updating its distance, and inserting it back into the data structure with its new priority.

IV. PICKING THE OPTIMAL DATA STRUCTURE

Clearly, the crux of the algorithm depends on the data structure depends on the data structure chosen to hold the node and its corresponding shortest distance during the run time of the algorithm. Four data structures (vector, set, priority queue, Fibonacci heap) were tested for the results of this paper. The time complexities of the data structure dependent lines of code are as follows:

Data Structure	**Extract Minimum**	**Decrease Key**
Vector	O(V)	O(1)
Set	O(logV)	O(logV)
Priority Queue	O(logV)	O(logV)
Fibonacci Heap	O(logV)	O(1) (amortized)

After testing all four data structures, each of them had comparable run times and got the better of the other on different occasions. This basically means that it depends on the structure of the inputted graph.
For example, for a graph with fewer nodes and not so densely connected, vector is expected to have the best run time.

Conversely, for a densely connected graph with a relatively higher number of nodes, Set (Binary Search Tree), Priority Queue (Min Heap) and the Fibonacci Heap have better runtimes.

V. FINAL TIME COMPLEXITY

The main load of the algorithm lies in the 2 loops, (line 4 and line 7). In the worst case scenario (a completely connected graph – where each node is one edge away from all the other nodes), line 4 and line 7 combine to give $O(V^2)$ and the relaxation further adds a O(logV). Finally, the time complexity boils down to $\mathbf{O(V^2 logV)}$.

But by thorough observation, the number of times relaxtion is done equals the number of edges in the graph. So, the final time complexity is **O(ElogV).**

[*] This work is based on the algorithm given by Edsger Wybe Dijkstra.