Analysis of Dijkstra's Algorithm

IT251 Data Structures and Algorithms II

Srinivasa R
Department of Information Technology
National Institute of Technology Karnataka, Surathkal

***Abstract*—This is an analysis of Dijstra's Algorithm by comparing the efficiency of Fibonacci Heap and Minimum Priority Queue ADTs for the queue used by the algorithm.**

I. Introduction

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph. It is a greedy algorithm that chooses the shortest possible path at each step.

II. Algorithm

The pseudocode for the general Dijkstra's Algorithm is as follows:

Algorithm 1: General Dijkstra's Algorithm

Data: Graph G, Min-Priority Queue Q, Source node $s \in G.V$
Result: $v.d = \delta(s, v)\ \forall\ v \in V$ where $\delta(u, v)$ is the shortest distance between nodes u and v
Function `Initialize`(G, s, Q)**:**
  foreach $v \in G.V$ **do**
    Q.`Insert`(v, ∞)
    $v.d \leftarrow \infty$
    $v.\pi \leftarrow$ NIL
  end
  $s.d \leftarrow 0$
end
Function `Relax`(u, v, W, Q)**:**
  if $v.d > u.d + W(u, v)$ **then**
    Q.`DecreaseKey`$(v, W(u, v))$
    $v.d \leftarrow u.d + W(u, v)$
    $v.\pi \leftarrow u$
  end
end
`Initialize`(G, s, Q)
$S \leftarrow \phi$
while $Q \neq \phi$ **do**
  $u \leftarrow Q$.`ExtractMin`$()$
  $S \leftarrow S \cup \{u\}$
  foreach $v \in G.Adj[u]$ **do**
    `Relax`$(u, v, G.W, Q)$
  end
end

III. Theoretical Analysis

Let n be the number of vertices in the graph. Doing a time complexity analysis, from the above algorithm, it can be seen that the function 'Relax' is called some $a(\propto n)$ times in every in every iteration of the loop which runs some $b(\propto n)$ times. Within the 'Relax' function, the 'DecreaseKey' function is called once. Let the time complexity of the 'DecreaseKey' function be $\mathcal{O}(f(n))$. Then, the time complexity of Dijkstra's algorithm (implemented with an adjacency matrix) is:

$$\begin{aligned}\mathcal{O}(n) &\approx a \times b \times \mathcal{O}(f(n)) \\ &\propto n \times n \times \mathcal{O}(f(n)) \\ \implies \mathcal{O}(n) &\approx n^2\mathcal{O}(f(n))\end{aligned}$$

Thus it can be seen that if a Minimum Priority Queue is used, the time complexity of Dijkstra's algorithm would be $\mathcal{O}(n^2 \log n)$ since the 'DecreaseKey' function has a time complexity of $\mathcal{O}(\log n)$ for a Minimum Priority Queue.
Continuing this analysis, it can be seen that using a Fibonacci Heap instead of a Minimum Priority Queue can decrease the time complexity of Dijkstra's Algorithm since the 'DecreaseKey' function has an amortized time complexity of $\mathcal{O}(1)$ in a Fibonacci Heap. Thus, using a Fibonacci Heap, the time complexity of Dijkstra's algorithm can be brought down to $\mathcal{O}(n^2)$ vastly improving the performance of the algorithm for larger graphs. The reduction in time complexity is similar for an implementation of Dijkstra's Algorithm with an adjacency list.

TABLE I
Comparison of Minimum Priority Queue and Fibonacci Heap

ADT	Insertion	FindMin	ExtractMin	DecreaseKey
Min Priority Queue	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Fibonacci Heap	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

IV. Implementation and Testing

The code to implement the Fibonacci Heap data structure and the Minimum Priority Queue data structure is within the '`fibheap.py`' and '`priorityqueue.py`' files respectively. The program can be run with '`python main.py`'. Upon running the program and setting the size of the graph to perform the tests on and the number of iterations (number of different graphs), we can observe that for larger graphs, using the Fibonacci Heap results in much lower time and much greater performance.

```
Enter the number of nodes in the graph to simulate Dijkstra's algorithm: 3000
Enter the number of iterations to compute and take average of: 10

Average time taken (Fibonacci Heap): 0.06599848999976529 s
Average time taken (Minimum Priority Queue): 0.753813970000192 s
```

Fig. 1. Average performance on 10 graphs each with 3000 nodes

In the above example, running Dijkstra's algorithm with the Fibonacci Heap as the ADT is more than 10 times faster than running it with the Minimum Priority Queue as the ADT thus supporting the theoretical analysis.