Dijkstra's Algorithm

Md. Irtiaz Kabir
Student ID: 2005070

Sadatul Islam Sadi
Student ID: 2005077

Wahid Al Azad Navid
Student ID: 2005089

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

March 9, 2024

Contents

List of Figures

List of Tables

1 Introduction

This report briefly describes Dijkstra's algorithm for solving the single source shortest path problem. From a single source, this algorithm determines the shortest path to every node.

1.1 Problem Definition

Given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s. Under single source shortest path problem we have to find the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves. The pure brute force solution to this problem is not feasible. One of the earliest algorithms to solve the SSSP problem in polynomial time was Dijkstra's.

1.2 Definition

Dijkstra's algorithm is the iterative algorithmic process to provide us with the shortest path from one specific starting node to all other nodes of a graph. It applies the greedy algorithm as the basis principle. This algorithm takes a graph and a source as an input and outputs the shortest path to all other nodes in linear space complexity.

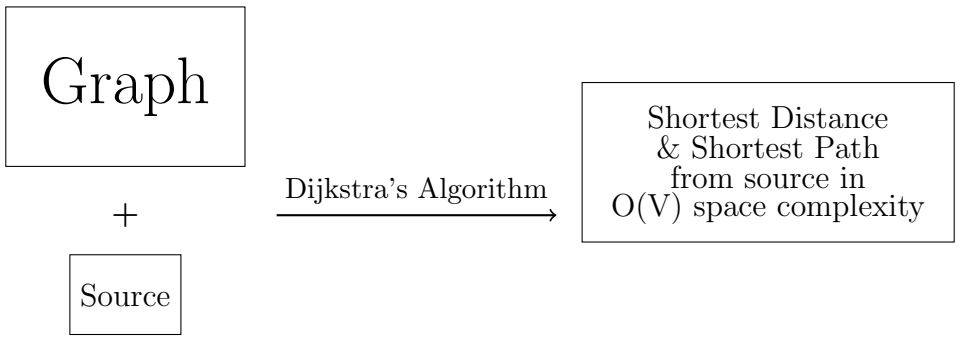

Figure 1: Basic input and output of Dijkstra's Algorithm

2 Dijkstra's Algorithm

2.1 Algorithm

Step 1: Create a priority queue[1], Q. Also define two arrays *dist* and *prev*

dist: This array contains shortest distance so far to each node.

prev: This array contains the predecessor of the node. $prev[u]$ refers to the node that has set the shortest distance of u.

Step 2: Initially $dist[s] = 0$, and for all other vertices this length equals ∞. For all nodes $v \in G.V$, $prev[v] = None$.

Step 3: The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unvisited vertex which has the least value. We use extracMin of the priority queue for this purpose.

Step 4: Next, from vertex v relaxations [2] are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value $dist[\text{to}]$. If the length of the current edge equals len, the code for relaxation is:

$$dist[\text{to}] = min(dist[\text{to}], dist[v] + len)$$

To get the actual path we use an array of predecessaors. Building this array of predecessors is very simple: for each successful relaxation, i.e. when for some selected vertex v, there is an improvement in the distance to some vertex to, we update the predecessor vertex for to with vertex v:

$$prev[\text{to}] = v$$

Also we need to perform a decrease key for vertex *to* whenever we perform a relaxation. After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates.

Step 5: Now, dist[u] shows the shortest distance from source to u. From the predecessor array of prev we can easily get the shortest path: starting with v, repeatedly take the predecessor of the current vertex until we reach the starting vertex s to get the required shortest path with vertices listed in reverse order. So, the shortest path P to the vertex v is equal to:

$$P = (s, ..., p[p[p[v]]], p[p[v]], p[v], v)$$

[1] A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

[2] The Edge Relaxation property is defined as the operation of relaxing an edge u → v by checking whether the best-known way from S(source) to v is to go from S → v or by going through the edge u → v. If it is the latter case we update the path to this minimum cost.

2.2 Pseudo Code

```
1         def Dijkstra(Graph, source):
2
3         Q ← PrioriyQueue()
4         dist ← ϕ
5         prev ← ϕ
6
7         for each vertex v in Graph.Vertices:
8             dist[v] ← ∞ if v is not source else 0
9             prev[v] ← None
10            Q.enqueue(v, dist[v])
11
12        while Q is not empty:
13            u ← Q.extract_min()
14
15            for each neighbor v of u:
16                alt ← dist[u] + Graph.Edges(u, v)
17
18                if alt < dist[v]:
19                    dist[v] ← alt
20                    prev[v] ← u
21                    Q.decrease_priority(v, alt)
22
23        return dist, prev
```

3 Analysis

3.1 Proof of Correctness

Let's call our source vertex s. Let $d(v)$ be the distance assigned by the algorithm and let $\delta(v)$ be the shortest path distance from s-to-v. We want to show that at the end of the algorithm, $\forall v,\ d(v) = \delta(v)$. This suffices to prove that the algorithm correctly computes the distances. We prove this by induction on $|Q|$ via the following lemma:

Lemma: For each $x \in R, d(x) = \delta(x)$.

Proof by Induction: *Base case:* Since Q only grows in size, the only time $|Q| = 1$ is when $Q = s$ and $d(s) = 0 = \delta(s)$, which is correct.

Inductive hypothesis: Let u be the last vertex added to Q. Let $Q' = Q \cup \{u\}$. Our I.H. is: for each $x \in Q'$, $d(x) = \delta(x)$

Using the I.H: By the inductive hypothesis, for every vertex in Q' that isn't u, we have correct distance assigned. We need only show that $d(u) = \delta(u)$ to complete the proof.

Suppose for a contradiction that the shortest path from s-to-u is P and has length

$$l(P) < d(u) \tag{1}$$

and at some leaves Q' (to get to u which is not in Q'). Let xy be the first edge along P that leaves Q'. Let P_x be the s-to-x subpath of P. Clearly:

$$l(P_x) + l(xy) \leq l(P)$$

Dijkstra's algorithm only works for non negative edge weights. If $l(xy) \geq 0$ we can rewrite it to be

$$l(P_x) < l(P) \tag{2}$$

Since $d(x)$ is the length of the shortest s-to-x path by I.H., $d(x) \leq l(P_x)$, giving us

$$d(x) + l(xy) \leq l(P_x) \tag{3}$$

Since y is adjacent to x, $d(y)$ must have been updated by the algorithm, so

$$d(y) \leq d(x) + l(xy) \tag{4}$$

Finally, since u was picked by the algorithm, u must have the smallest distance assigned -

$$d(u) \leq d(y) \tag{5}$$

Combining equation (1), (2), (3), (4), (5) we get,

$$d(u) \leq d(y) \leq d(x) + l(xy) \leq l(P_x) < l(P) < d(u)$$

Which implies $d(u) < d(u)$ - a contradiction. Therefore, no such shorter path P can exist and so $d(u) = \delta(u)$.

So the lemma holds, which proves that at the end of the algorithm, every vertex has been assigned the shortest distance from the source vertex.

3.2 Running Time

The algorithm takes different running time for different implementation for the abstract data structures used (graph and priority queue). But there is a fixed bottleneck for most common implementations. There are mainly 2 stages of the algorithm -

Initialization stage Initializing the *dist* and *prev* arrays, enqueuing the vertices

Iteration stage Iterating through the vertices one by one in increasing order of their priority and updating the *dist* and *prev* arrays and priorities of the vertices

For every implementation the iteration stage becomes the bottleneck. That is because the initialization stage simply iterates through the vertices in $O(|V|)$ running time ($|V|$ denotes the number of vertices and $|E|$ denotes the number of edges). But the iteration stage is costlier. So far we haven't found an implementation that can perform the iteration stage in $O(|V|)$).

There are mainly 2 operations that make the iteration stage the bottleneck of the whole algorithm. *extract_min* and *decrease_priority*. Different implementations handle these operations differently than one another. But we can generalize the overall running time in an abstract fashion.
Analyzing the bottleneck operations:

Extract Min This is called $\Theta(|V|)$ times because each vertex is enqueued exactly once in the initialization stage and never again

Decrease Priority This is called for each neighbor of a vertex. This is done by traversing the edges of a vertex. And since this traversal takes place for every single vertex, all the edges are traversed eventually. So *decrease_priority* is called $\Theta(|E|)$ times

Let, for a specific implementation, the running time of *extract_min* is T_{em} and the running time of *decrease_priority* is T_{dp}

So the overall running time,

$$T = \Theta(|V|) \times T_{em} + \Theta(|E|) \times T_{dp}$$

Different implementation have different T_{em} and T_{dp}

So far the use of a Fibonacci Heap is found to be the most optimal.

Implementation	$\boldsymbol{T_{dp}}$	$\boldsymbol{T_{em}}$	$\boldsymbol{T}$
Singly Linked List	$\Theta(1)$	$\Theta(\|V\|)$	$\Theta(V^2)$
Binary Min Heap	$\Theta(log\|V\|)$	$\Theta(log\|V\|)$	$\Theta\left((E+V)log\|V\|\right)$
Fibonacci Heap	$\Theta(1)$	$\Theta(log\|V\|)$	$\Theta(E+Vlog\|V\|)$

Table 1: Running Time for Different Implementations

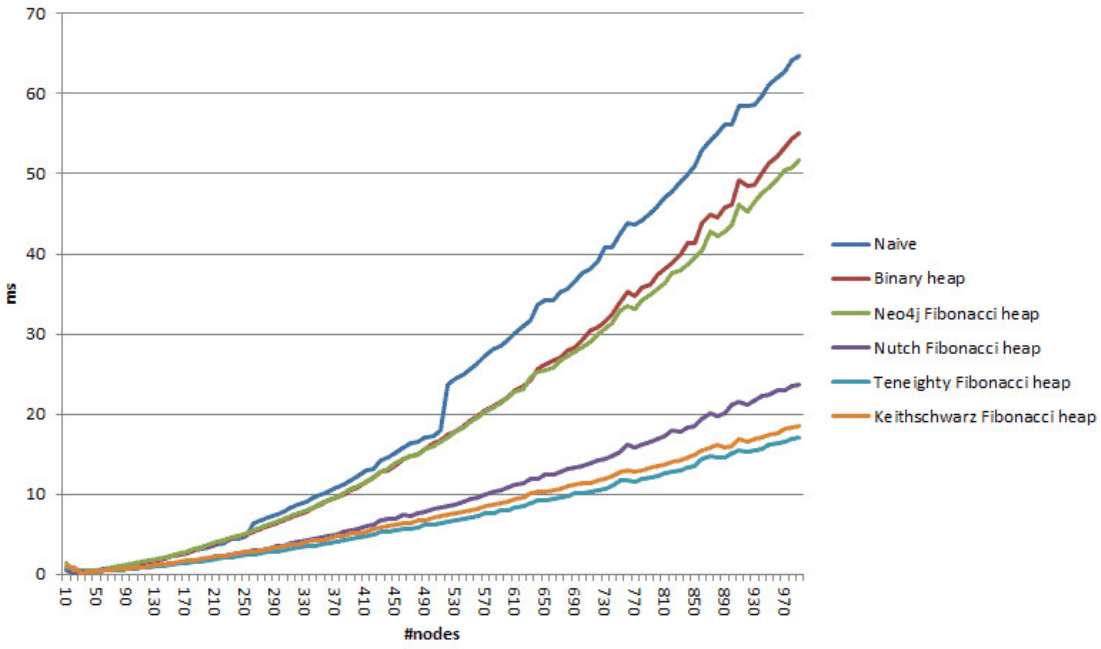

Figure 2: Benchmarking Different Implementations [2]

4 Drawbacks

4.1 Traversing Unnecessarily

Dijkstra's algorithm guarantees to produce the accurate result for both directed and undirected graphs. However, it may result in a longer running time for certain graphs.

For instance, in this graph, our objective is to determine the most efficient route from the starting point (S) to the destination (D).

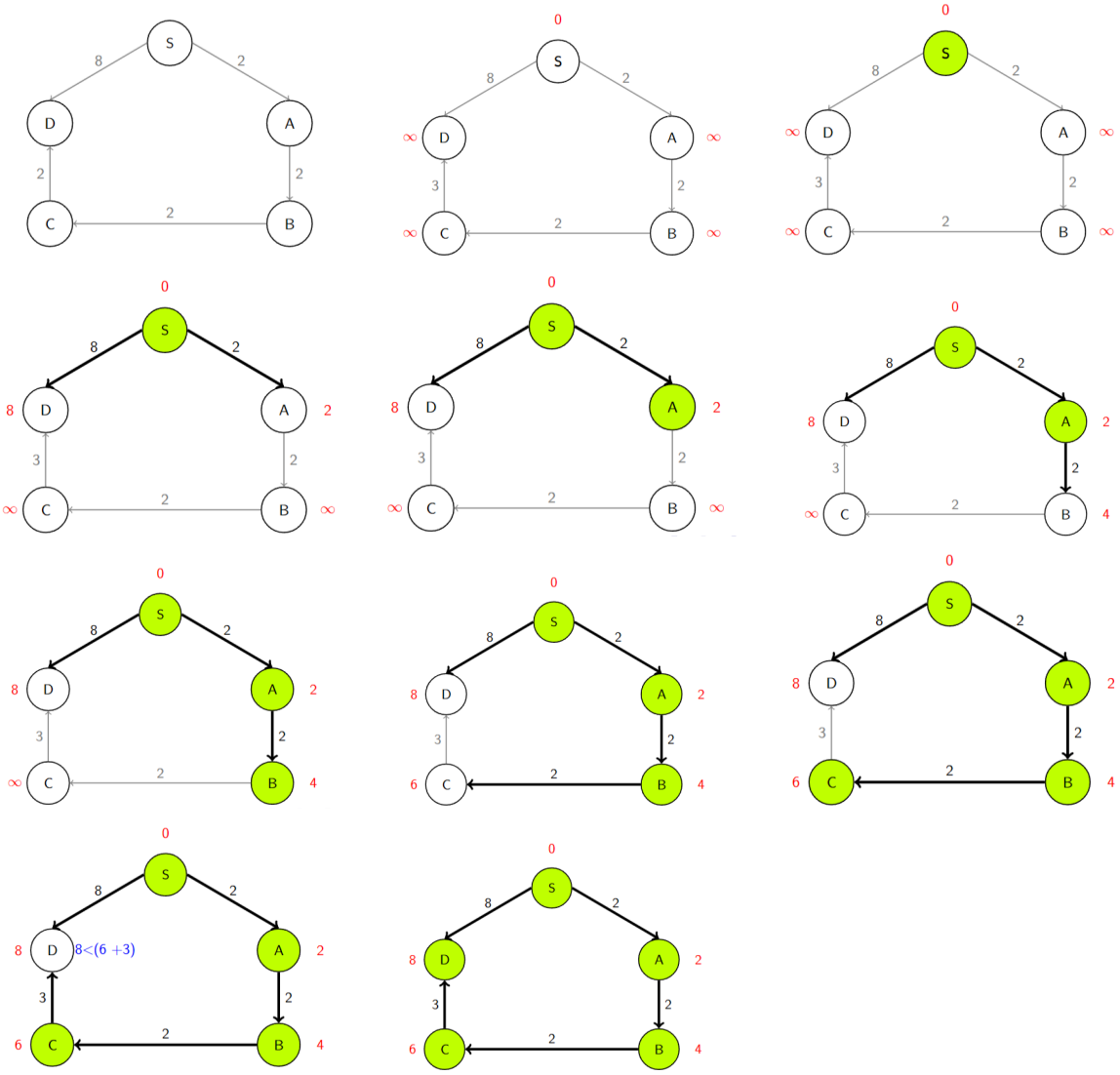

Figure 3: Extended traversal caused by *Dijkstra's* Algorithm

Even though the points are in close adjacency to each other, implementing *Dijkstra's* algorithm would necessitate traversing the entire graph to identify the shortest path.

4.2 Inability to Handle Negative-Weight Edges

The algorithm falls short in finding the shortest path between nodes or from a single node to all others. Due to its greedy nature, it selects the most favorable path available at each step. However, the presence of negative-weight edges introduces the possibility of a more optimal route utilizing edges that the algorithm may disregard during intermediate stages.

5 Specialized Variants

When arc weights are small integers(bounded by a parameter $\boldsymbol{C}$), specialized queues which ake advantage of this fact can be used to speed up *Dijkstra's* algorithm.

- The first algorithm of this type was **Dial's algorithm** [3] for graphs with positive integer edge weights, which uses a bucket queue to obtain a running time of $O(|E| + |V| * C)$.
- The use of a **Van Emde Boas tree** [5] as the priority queue brings the complexity to $O(|E| * loglogC)$.
- Another interesting variant based on a combination of a new radix heap and the well-known Fibonacci heap runs in time $O(|E| + |V|\sqrt{logC})$ which was properposed by **Ahuja et. al***(1990)* [1].
- Finally, the best algorithm in this case was given by **Thorup***(2000)* [4] which runs in $O(|E| * loglogV)$.

6 Applications

Dijkstra's algorithm is commonly used in various domains such as computer networking, transportation systems, and geographic information systems (GIS). In this section, we explore and discuss different application scenarios to illustrate how the algorithm plays a crucial role in solving practical problems within these contexts.

6.1 Routing in Computer Networks

- *Dijkstra*'s algorithm, utilized in internet routing, dynamically adapts to varying network conditions, facilitating efficient data transmission by continually recalculating optimal routes.
- It enables routers to prioritize paths with minimal latency, ensuring swift and reliable communication between devices connected to the network.

6.2 Transportation and Logistics

- Logistics companies rely on *Dijkstra*'s algorithm to properly plan delivery routes, optimizing the sequence of stops to minimize fuel consumption and reduce overall transportation costs.

- By considering real-time traffic data and delivery constraints, the algorithm aids in dynamic route adjustments, enhancing the efficiency of last-mile delivery operations.

- Its application in logistics contributes to timely and cost-effective transportation solutions, impacting industries ranging from e-commerce to traditional shipping services.

6.3 Network Routing Protocols

- Open Shortest Path First (OSPF), a widely used routing protocol, employs *Dijkstra*'s algorithm to compute the shortest paths between routers, promoting efficient data flow within complex network infrastructures.

- The algorithm's ability to adapt to changing network topologies enhances OSPF's resilience, ensuring rapid convergence and minimizing network downtime in response to link failures.

6.4 Geographic Information Systems (GIS)

- Geographic Information Systems leverage *Dijkstra*'s algorithm to analyze spatial data, providing valuable insights for urban planners by determining optimal routes for emergency evacuation in the event of natural disasters.

- The algorithm's consideration of varying terrain and obstacles enhances GIS applications' accuracy in predicting evacuation times and suggesting safe pathways for affected populations.

6.5 Robotics and Autonomous Vehicles

- Autonomous vehicles utilize *Dijkstra*'s algorithm for path planning, allowing them to navigate complex environments with real-time adjustments to avoid obstacles and reach destinations efficiently.

- The algorithm's incorporation into robotics enables adaptive decision-making, as vehicles dynamically calculate the optimal routes based on current conditions, enhancing safety and operational effectiveness.

6.6 Telecommunication Network Design

- Telecommunication engineers employ *Dijkstra*'s algorithm to plan fiber-optic cable routes efficiently, reducing signal transmission latency and optimizing the overall design of communication networks.

- By considering factors such as cable length and transmission speeds, the algorithm aids in designing networks that meet the demands of high-speed data transfer and communication reliability.

6.7 Game Development

- In video game development, *Dijkstra*'s algorithm ensures that non-player characters (NPCs) navigate game environments seamlessly, enhancing the overall gaming experience by providing realistic and adaptive character movements.
- The algorithm's integration into pathfinding algorithms allows game developers to create engaging and challenging environments where NPCs intelligently navigate obstacles and follow optimal routes.

7 Conclusion

In conclusion, Dijkstra's algorithm is a flexible and basic tool in computer science and engineering, with applications in a variety of domains. Its versatility in routing situations, resource management, and geographical analysis emphasises its importance in optimising pathways and effectively navigating complicated networks. The investigation of numerous use cases reveals that Dijkstra's algorithm plays a critical role in improving efficiency, dependability, and flexibility across a wide range of real-world systems.

As technology advances, Dijkstra's algorithm's continuous significance in routing protocols, logistics, autonomous systems, and other areas demonstrates its long-term effect. Future research and development may look at upgrades and adaptations to broaden the algorithm's capabilities and handle future difficulties in industries like telecommunications, transportation, and gaming.

References

[1] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, 37(2):213–223, 1990.

[2] W. b. baeldung. Understanding time complexity calculation for dijkstra algorithm, Nov 2022.

[3] R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633, 1969.

[4] M. Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[5] P. van Emde Boas. Van emde boas tree. *Van Emde Boas Tree*, 1974.