Algorithms Fall 2019

Lecture : Single Source Shortest Path -Dijkstra's Algorithm

Imdad Ullah Khan

Contents

1 Weighted Graphs

So far, we have looked at graphs where edges are assumed to be uniformly weighted i.e. that their weights relative to other edges in the graph are the same, hence the absense of weights. However, there are certain situations where such a graph is unsuitable for modelling the problem. For example, consider modelling maps such that vertices represent nodes or locations and weights represent distances between them. There are several real-world scenarios which are modelled as weighted graphs. For instance:

- water networks with weights representing water capacity of pipes
- electrical circuits with weights representing resistance or maximum voltage or current
- computer or phone networks with weights representing length of wires between nodes

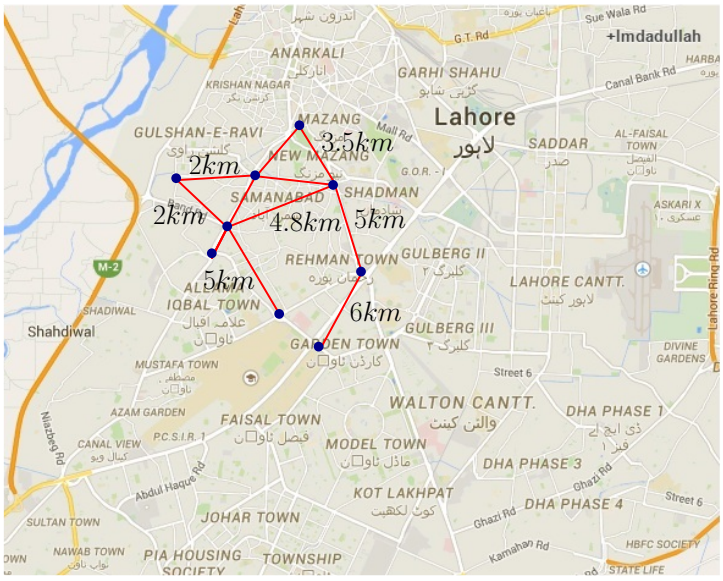


Figure 1: A canonical application of weighted graphs

Hence, we now define a new type of graph, one where edges have a weight associated with them, also known as the cost. A *weighted graph* is defined as a graph $G = (V, E)$ where V is the set of its vertices and E is the set of its edges, and a cost or weight function w, defined as $w : E \to \mathbb{R}$ where $e \in E$. Alternatively, we may define a set W of pairs (e, w) where $e \in E$ and w is the associated cost with e. Throughtout, we consider $|V| = n$ and $|E| = m$.

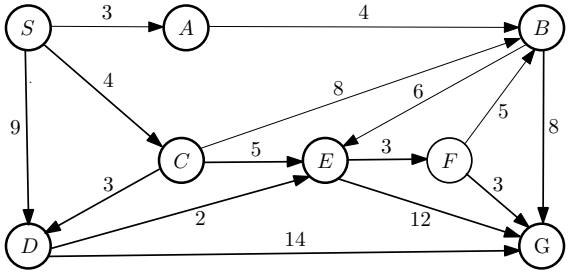
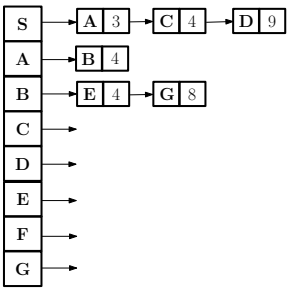


Figure 2: A weighted graph

Weighted graphs can be represented as adjacency list or adjacency matrix. The adjacency list of vertex u is now a list of (key,value) pairs where the key is the adjacent vertex v and the value is the weight of the edge (u, v). In the adjacency matrix M, the entry $M[i][j]$ is the weight of the edge (i, j). Note that for undirected graphs, the adjacency matrix would be symmetric, and for a graph where no vertex has a self-loop, the diagonal of the matrix of is all zeros as $w(u, u) = 0$.



	S	A	B	C	D	E	F	G
S	0	3	0	4	9	0	0	0
A	0	0	4	0	0	0	0	0
B	0	0	0	0	0	6	0	8
C	⋮							
D								
E								
F								
G								

Figure 3: Adjacency List and Adjacency Matrix of weighted graph in Figure 2

1.1 Weighted paths and shortest paths

We motivated the use of weighted graphs in modelling maps. An important problem then is to find the shortest path between two vertices, as we do for example in Google Maps.

Previously, a path was defined as a sequence of vertices with no repetition such every two consecutive pair of vertices is an edge in the graph and the length of the path was defined as the number of edges in path. The shortest path from s to t was one with the least number of edges between s and t.

For weighted graphs, we consider the total weight of the path, $C(p)$. This is simply determined by adding the weights of all edges $e \in U$. For a path $p = v_0, v_1, \ldots, v_k, C(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The shortest path from s to t in a weighted graph is a path with least total weight. The weight of the shortest path from s to t is called the *distance* from s to t denoted by $d(s, t)$.

In the example graph in Figure 4, where if the graph was unweighted, the shortest path between nodes S and G would have been S, D, G and its length would have been 2. However, since the graph is weighted, the shortest path from S to G is a path with least total weight among all paths from S to G. As shown in the figure, the paths p_1 and p_2 are of equal weight and have the least weight. The important thing to note here is that the shortest path is not necessarily unique. *There can be multiple shortest paths between two vertices.*

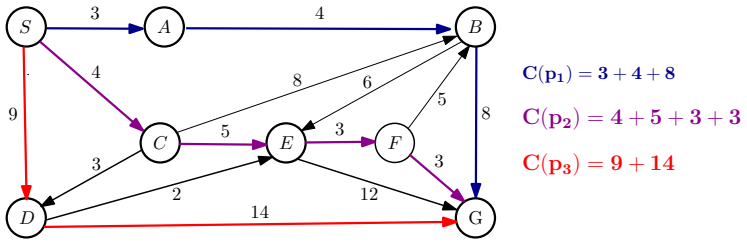


Figure 4: Paths of varying lengths from S to G

2 Single Source Shortest Path Problem

The single source shortest path problem is defined as follows. Given a weighted graph G and a source vertex $s \in V$, find a shortest path from s to all vertices $v \in V$.

Previously, we discussed that the single source shortest path problem can be solved by BFS for unweighted graphs, i.e. graphs with unit weights, and proved that at level k of the BFS tree, there are exactly those vertices that are at distance k away from the source vertex.

In order to use BFS to solve the single source shortest path problem for weighted graphs, one can replace each edge e by directed path of $w(e)$ unit weight edges, as shown in Figure 5.

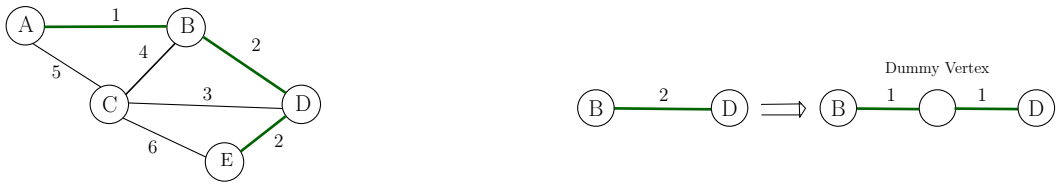


Figure 5: Using BFS to solve single source shortest path problem for weighted graphs

This seems like a correct solution at first, but what if weights are not integers or negative? This solution would not work then. Even if we limit the input to graphs having positive integer weights, what if the weights are of the order of millions? An edge of weight 1 million would have to be replaced by 1 million dummy vertices and edges. This would blow up the size of the graph and given that running time of BFS is $O(n + m)$, this is an infeasible algorithm. Therefore, we need another solution to the single source shortest path problem, i.e. Dijkstra's algorithm for shortest path.

3 Dijkstra's Algorithm

Next, with the definitions of weighted graphs, shortest paths in such graphs, and motivation for a solution other than BFS for the single source shortest path problem in weighted graphs, we now discuss an algorithm for it. The algorithm is known as Dijkstra's algorithm, proposed by **Edsger W. Dijkstra**. It works for a weighted graph with non-negative weights, and builds a shortest-path tree rooted at s, provided the vertices are connected i.e. have atleast one path between them.

Assumptions:

Given a weighted graph $G = (V, E)$ and two vertices $s, v \in V$ where s is the source vertex, and v is to be the destination vertex, the problem is to find the shortest path from s to t. We make the following assumptions:

- All vertices are reachable from s, otherwise there would be no shortest path from s to the unreachable vertex, i.e. distance $= \infty$. By running a BFS/DFS rooted at s, we can determine if v is reachable from s, $\forall vinV$. If so, then we proceed as it is. If not, we can dimply remove the unreachable vertices.

- All edge weights are non-negative. This is required for Dijkstra's algorithm to work. Bellman-Ford algorithm deals with negative weights.

3.1 Dijkstra's Algorithm for Distances

Initialization Steps:

Initially, we only consider on finding the distances, not the paths, from s to all other vertices.

We set up an array d, which holds the cost for the shortest path from s to each vertex in G. Initially, all vertices have their current cost set to ∞, except for s, which has its cost set to 0, since the shortest path from s to itself has a cost of 0. Hence we have, for a n vertex graph:

$$d[1...n], d[i] = \infty, d[s] = 0$$

We also set up two sets R and $\bar{R}$. R is the set of vertices for whom the current shortest path is known, whereas $\bar{R}$ is the set of vertices for whom the shortest path is currently not determined. Initially, R contains only s, and $\bar{R}$ contains all other vertices.

Iterative Steps:

One vertex is iteratively added to R from $\bar{R}$ until $\bar{R} = \emptyset$ and $R = V$. The main question now is, which vertex from $\bar{R}$ should be added to R? The vertex closest to s should be added. Such a vertex must be at the 'frontier' of $\bar{R}$. Why must this be true?

Let v the closest to s in $\bar{R}$, and consider the shortest path $s, \ldots, u, v$. Since $w(uv) \geq 0$, u must be closer to s than v. This implies that u must be in R, because otherwise it will contradict v being closest to s in $\bar{R}$. This shows that v is only one edge away from R, and that edge is (u, v). Therefore, we restrict the search of the closest vertex to s to 'single edge extensions' of paths to $u \in R$.

In order to determine the closest vertex in $\bar{R}$ to s, the algorithm assigns a score to each *crossing edge*. An edge $e = (u, v)$ is crossing if $u \in R$ and $v \in \bar{R}$. Score is defined as: $d(u) + w(u, v)$ for $(u, v) \in E, u \in R, v \notin R$. The vertex added to R in each iteration is a frontier vertex adjacent through a crossing edge with minimum score.

Pseudocode:

Algorithm Dijkstra's Algorithm for distances from s to all vertices

$d[1 \dots n] \leftarrow [\infty \dots \infty]$
$d[s] \leftarrow 0 \quad R \leftarrow \{s\}$
while $R \neq V$ **do**
  Select $e = (u, v)$, $u \in R, v \notin R$, with minimum $d[u] + w(uv)$
  $R \leftarrow R \cup \{v\}$
  $d[v] \leftarrow d[u] + w(uv)$

Example Run:

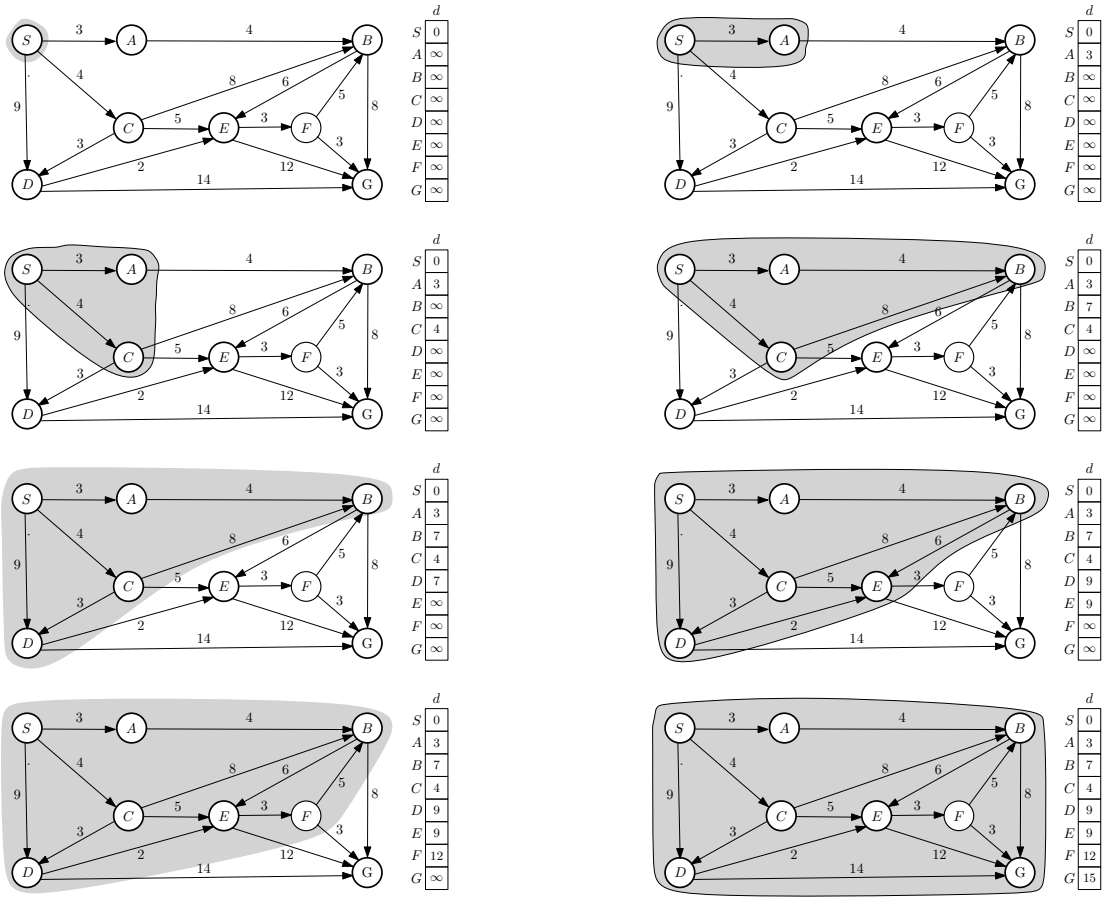


Figure 6: Example Run of Dijkstra's Algorithm for Distances

3.2 Dijkstra's Algorithm for Shortest Path

In order to find the shortest paths, and not just distances from s to all other vertices, we need to do some extra book-keeping, i.e. record predecessor relationships so that the source of each edge used is known. We set up another array P, which holds the vertex via which we came to that particular vertex. For all vertices, this is set to *null* at the start. To produce the shortest path to a vertex t from s, one can simply backtrack through the array P, starting at $P[t]$ and moving to the index indicated till s is reached. The pseudocode is as follows:

Algorithm Dijkstra's Algorithm for Shortest Paths from s to all vertices

$d[1 \dots n] \leftarrow [\infty \dots \infty]$
$P[1 \dots n] \leftarrow [null \ \dots \ null]$
$d[s] \leftarrow 0 \quad R \leftarrow \{s\}$
while $R \neq V$ **do**
  Select $e = (u, v)$, $u \in R, v \notin R$, with minimum $d[u] + w(uv)$
  $R \leftarrow R \cup \{v\}$
  $d[v] \leftarrow d[u] + w(uv)$
  $P[v] \leftarrow u$ ▷ Predecessor is the vertex whose path is single-edge extended

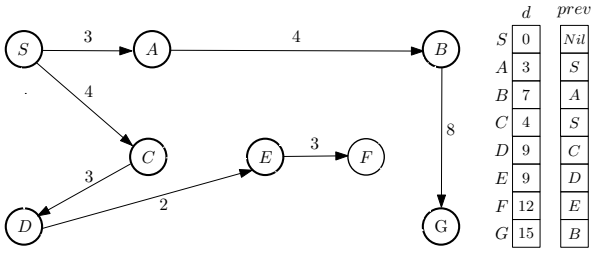


Figure 7: Example Run of Dijkstra's Algorithm for Shortest Path

As evident from Figure 7, Dijkstra's implicitly builds a tree when predecessor relationships are recorded. Note that all vertices are connected, there is no cycle and there are $n - 1$ edges, i.e. all the properties that define a tree.

3.3 Proof of Correctness

We can clearly see that Dijkstra algorithm produces the correct shortest paths in the above example, but obviously this doesn't mean that it would always work correctly. Consider the following example in Figure 8.

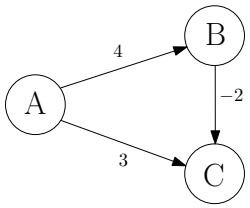


Figure 8: Dijkstra with source A adds greedily the vertex C and B to R in this order and selects the paths AC and AB as shortest paths from S to B and C. While clearly the shortest path from A to C is the path ABC of total weight 2.

This immediately gives us that if this algorithm is correct, its proof of correctness better use critically the fact that there are no negative weight edges.

Correctness of Dijkstra's algorithm follows from the following theorem.

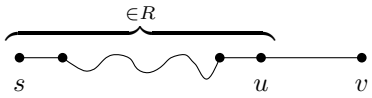
Theorem 1. *In each iteration i of the Dijkstra's algorithm, $\forall\ u \in R$, $d(u)$ is equal to the length of a shortest path from s to u.*

Proof. The proof is by induction on iteration i.

Base case is trivially true, since for $i = 0$, only $s \in R$ and $d(s) = 0$ which is correct, since the graph has no negative weight edges so there cannot be a path from s to s of length shorter than 0.

For the **Inductive hypothesis** assume that Dijkstra's algorithm is correct upto iteration i.

Inductive step: Suppose we added v to R in the ith iteration due to the edge (u, v) for some $u \in R$. The algorithm would set $d(v) = d(u) + w(uv)$. We will show that $d(v)$ is the length of a shortest path from s to v. Let P be the path from s to v that is selected by the algorithm, i.e $P = s \ldots, \ldots, u, v$. By inductive hypothesis we have that $l(P) = d(u) + w(uv)$.

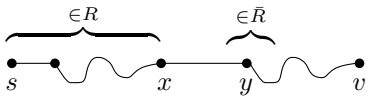


The Path P selected by the algorithm

Note that the prefix of the path from s to u are all vertices in R at this stage of the algorithm and only $v \in \bar{R}$.

We will show that every other path from s to v is at least as long as P

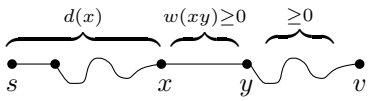
Let P' be any other path in G from s to v, classifying vertices of P' according to R and $\bar{R}$ at this iteration of the algorithm, we know that $s \in R$ and $v \in \bar{R}$, hence this path starts in R and must cross to $\bar{R}$ (may be many times). Consider the first time it crosses R, i.e. let (x, y) be the first edge in P' such that $x \in R$ and $y \in \bar{R}$.



Any other path P' from s to v, where(x, y) is the first crossing edge from R

We argue about each piece of this path separately

- The segment of P' from s to x is a shortest path from s to x, as $x \in R$ and all other vertices on P' before x are also in R, hence by inductive hypothesis the length of the prefix of P' from s to x is $d(x)$.
- $w(xy) \geq 0$, since all edges have non-negative weights
- the length of the sub-path (suffix) of P' from y to v is at least 0. This also follows from non-negativity of edges. This is the critical use of non-negativity of edge weights.
- $l(P') \geq d(x) + w(xy)$



To finish the proof we use how the algorithm selected the edge (u, v). At this iteration, since $x \in R$ and $y \in \bar{R}$ and also $u \in R$ and $v \in \bar{R}$. So certainly the edge (x, y) was a candidate to be selected as a crossing edge. The algorithm selected a crossing edge minimizing the score, so the fact that (u, v) was selected and (x, y) was not mean that $d(u) + w(uv) \leq d(x) + w(xy)$.

Combining this with the above we get that $l(P') \geq d(x) + w(xy) \geq d(u) + w(uv) = l(P)$, hence $l(P') \geq l(P)$ for any path P' from s to v. □

3.4 Runtime

We now analyze the runtime of this implementation of Dijkstra's algorithm. We can see that the while loop will run for $O(n)$ time, since it will stop when $R = V$, and a vertex is added to R at each vertex. Inside the while loop, we try to find the edge that gives us the least path cost. Since we will look at all the edges in the worst case to find the minimum, we would $O(m)$ time to do so. Overall, this gives us a total asymptotic time bound of $O(nm)$.

4 Dijkstra's Algorithm: Vertex-Centric Implementation

In the simple implementation of Dijkstra's, as discussed above, repeatedly finding the minimum is expensive. Since there can be atmost $O(n^2)$ edges in the graph, the total runtime may be of the order (On^3). We attempt to reduce the time spent in repeatedly finding the minimum by using a vertex centric approach, i.e. store information at the vertices instead of the edges alone.

Each vertex in $\bar{R}$ is assigned a key which is the length of the current best single edge extension, Then, the vertex in $\bar{R}$ closest to s is the vertex with the smallest key. Since a vertex may be the target of several edges, the search space for the minimum is reduced. The key is also easy to update; when adding a new vertex v to R, keys of only the neighbors of v in $\bar{R}$ must be updated. The keys of the rest of the vertices in $\bar{R}$ remain unchanged.

Example:

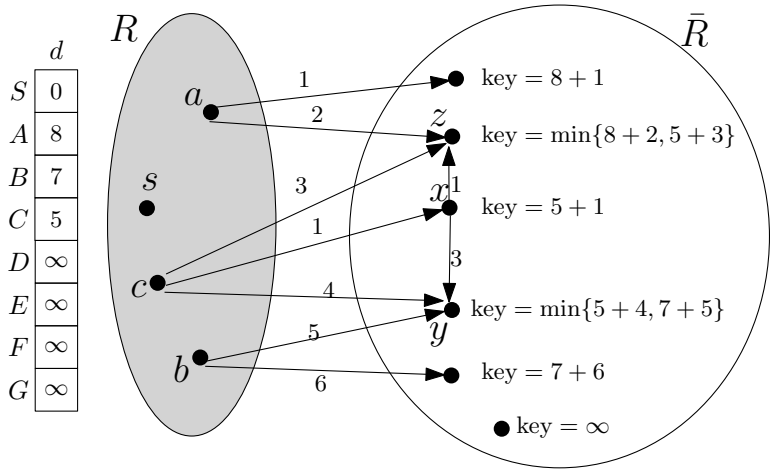


Figure 9: An example state in the Vertex-Centric approach to Dijkstra's.

Consider the state during a run of Dijkstra's implemented using vertex centric approach shown in Figure 9. The vertex currently with the minimum key is x, and hence x would be the vertex to be added to R in this iteration. Then, only keys of vertices adjacent to x need to be updated. The new key of y would be $\min(5+4, 7+5, 6+3)$ and the new key of z would be $\min(8+2, 5+3, 6+1)$. The vertex that would be added in the next iteration is z as it would have the minimum key, i.e. 7.

Pseudocode:

Algorithm Vertex-Centric Dijkstra's for distances from s to all vertices

```
d[1...n] ← [∞...∞]
d[s] ← 0
while R ≠ V do
    Select v ∈ R̄ with minimum d[v]        ▷ key of each vertex is its distance from the s
    R ← R ∪ {v}
    for each z ∈ N(v) ∩ R̄ do
        if d[z] > d[v] + w(vz) then
            d[z] ← d[v] + w(vz)
```

Runtime:

Analyzing the runtime of the vertex-centric Dijkstra's using the pseudocode above, we see that the outer while loop still runs $O(n)$ times. However, finding the minimum score vertex takes $O(n)$ time, as opposed to the $O(m)$ time to find the minimum score crossing edge in the previous implementation. Since we need to update the neighbors of the added vertex only, it takes $O(n)$ time at max. Therefore, the overall runtime is $O(n^2 + m) = O(n^2)$. This is better than the last implementation, specially for dense graphs where the number of edges is large (about n^2). However, quadratic runtime is still infeasible for practical purposes and large graphs. As always, we ask if we can do any better. As it turns out, we can do much better using a data structure more suited to our problem of repeatedly finding the minimum key vertex, which is expensive, namely the **min-heap**.

5 Dijkstra's Algorithm: Heap Implementation

We have discussed two implementation of Dijkstra's algorithm so far, a simple and a vertex-centric approach, with time complexities $O(nm)$ and $O(n^2)$ respectively. We observe that much of the expense in both these implementations arises from the search of a minimum value repeated in each outer loop iteration. Reducing the time required by this expensive operation would reduce the overall time complexity of the algorithm. In order to do so, we use a data structure designed specifically to provide fast and efficient retrieval of the minimum of a set of values.

Recall the **Priority Queue or Heap ADT** with operations Insert, ExtractMin, and DecreaseKey each taking $O(\log n)$ time. We now set up the usage of min-heap for implementing Dijkstra's Algorithm.

The idea follows from the vertex-centric approach. Each vertex v is assigned a key which is the current known distance from the source s to v. The n vertices are inserted into a min-heap $\mathcal{H}$. In each iteration, ExtractMin($\mathcal{H}$) is used to get the vertex with the minimum key, suppose u, which is to be added to R. Next, the neighborhood $\{v|(u,v) \in E\}$ of the extracted vertex u is traversed. The key of a neighboring vertex v is updated if the length of the new path $p = s, \cdots, u, v$ is less than the current key value of v using the operation DecreaseKey($\mathcal{H}$,v,$len(p)$).

Pseudocode:

Algorithm Dijkstra's Algorithm for distances from s to all vertices using MinHeap

```
d[1...n] ← [∞...∞]
d[s] ← 0
H ← Initialize(V, d)                    ▷ make a heap with all vertices and keys as d[·]
while R ≠ V do
    v ← extractMin(H)
    for each z ∈ N(v) ∩ R̄ do
        if d[z] > d[v] + w(vz) then
            DecreaseKey(H, z, d[v] + w(vz))
    R ← R ∪ {v}
```

Runtime:

In order to analyze the runtime of this heap-based implementation of Dijkstra's, we will count the number of calls to the ExtractMin and DecreaseKey operations as the rest of the work in the algorithm takes constant time.

- In total, there are n ExtractMin operations
- On extracting v, there are $O(deg(v))$ DecreaseKey operations
- In total, there are $\sum_{v \in V} deg(v) = m$ DecreaseKey operations.
- Each ExtractMin takes $O(\log n)$ time
- Each DecreaseKey takes $O(\log n)$ time
- In total, the runtime is $O(n \log n) + O(m \log n) = O((n+m) \log n)$

Therefore, the overall runtime of the heap-based Dijkstra algorithm is much better than the previous two implementations.