Shortest path algorithms complexity analysis and application to route planning

Fernando Lima Saraiva Filho
Computer Science Department
Federal University of São Paulo
São José dos Campos, Brazil
fernando.saraiva@unifesp.br

***Abstract*—Finding the shortest-path between two points is very important in route planning. Some of the most famous algorithms that solve this problem are Dijkstra and Bellman-Ford. In this work, we analyse their computational complexity and compare their execution time in different situations. Furthermore, we show the application of this kind of algorithm in a real world decision problem using data from a platform that provides free geographic data.**

***Index Terms*—dijkstra, bellman-ford, complexity analysis, shortest path**

I. Introduction

Efficient route planning is a fundamental necessity in many contexts. Although there may be many different paths connecting two points, it is, in general, desirable to find the shortest path in some sense, like distance or travel time. In this paper, two of the most famous algorithms of the literature to find minimum paths are discussed: the Dijkstra algorithm and the Bellman-Ford algorithm. We analyse their complexity, run some examples and calculate their execution time. Furthermore, we show the usefulness of shortest-paths algorithms applying them to a real world problem of choosing a place to live based on distances or time travels to points of interest in a city.

Dijkstra algorithm was introduced by the computer scientist Edsger Dijkstra in 1959 [2] and it solves the problem of finding a minimum path between two given nodes in a graph with no edges with negative weights. On the other hand, the Bellman-Ford's algorithm solves the same problem even when the graph has edges with negative weights [4].

There are many examples of applications of these algorithms in the literature. In [6], Wang *et al.* apply the Dijkstra algorithm for robot path planning. In their work, the shortest path is chosen in a situation in which obstacles must be avoided. In [7], Schambers *et al.* use the Bellman-Ford algorithm to plan driving routes of electric vehicles based on energy efficiency. This algorithm was chosen to approach this problem because it is possible to recharge the battery of an electric vehicle while driving and, in this way, the energy cost to travel a highway may be negative.

II. Data and Method

A. The data set

In this work, the object of study is a road map obtained from a graph centered in the Parque Santos Dumont in the city of São José dos Campos. The data of the road system was obtained from the OpenStreetMap (OSM) Project. OSM is a platform developed by volunteers to provide free geographic data about roads, trails, train stations and more around the world [8]. In Brazil, many governmental institutes provide data to the OSM project, such as: Instituto Brasileiro de Geografia e Estatística (IBGE), Instituto Nacional de Pesquisas Espaciais (INPE), Departamento Nacional de Infraestrutura de Transportes (DNIT) and many others [9].

The information available of the Project can be accessed directly through the website. However, it was developed a Python package, named OSMnx, that makes it possible to download the geospatial data from OpenStreetMap and to model, visualize, and analyze real world street networks [1]. This package is free and is built on top of other very useful Python libraries: NetworkX, Matplotlib, and Geopandas [1]. Using the OSMnx package, it is possible to extract the data specifying the region of interest. That region can be defined in different ways. In figure 4, one can see the road system of a map centered in the Parque Santos Dumont in the city of São José dos Campos and containing all the nodes within 20 km of the center of the graph, extracted with the function *graph_from_place* of the OSMnx package.

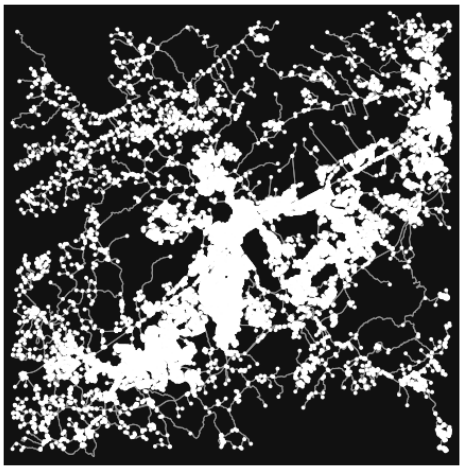

Fig. 1. Road System of the graph centered in the Parque Santos Dumont

When the function *graph_from_point* was used, it was

generated a graph with 25024 nodes and 57691 edges, and, in this context, the edges represent the roads and the nodes represent the intersection between the edges.

B. Mathematical Description

In the problem of the minimum path, we are given a weighted, directed graph $G(V, E)$, with a weight function $w : E \rightarrow \mathcal{R}$, that maps edges to real-valued weights [3]. In the case of graphs representing roads, these weights may represent the distance or the travel time between nodes. The weight $w(p)$ of a path $p =< v_0, v_1, ..., v_k >$ is the sum of the weights of the edges that constitute the path [3]:

$$w(p) = \sum w(v_{i-1}, v_i)$$

If there is a path from u to v, we define the shortest-path weight $\delta(u, v)$ from u to v by:

$$\delta(u, v) = min\{w(p) : u \xrightarrow{p} v\}$$

If there is not a path, we define $\delta(u, v) = \infty$

A minimum-path from vertex u to vertex v is defined, then, as any path p with with weight $w(p) = \delta(u, v)$ [3].

In this paper, we focus on the single-source shortest-paths problem, that is the problem of finding a shortest path from a given source vertex $s \in V$ to every vertex $v \in V$.

C. The algorithms

One of the main references used in this whole section is [3]. The algorithms used in this paper apply the technique of relaxation. For each vertex $v \in V$, there is an attribute $v.d$, that works as an upper bound of the weight of a minimum path from source s to v, and it is called shortest-path estimate.

The algorithm to initialize the estimates of shortest-path and predecessor for each vertex can be seen below [3].

Algorithm 1 Initialize-Single-Source (G, s)

```
for each vertex v ∈ G.V do
    v.d = ∞
    v.π = NIL
end for
s.d = 0
```

After the initialization, the predecessor of all vertices are considered null ($v.\pi = NIl$). Besides, the shortest path estimate is set to infinity to all vertices except to the source itself ($s.d = 0$). The initialization procedure is $\Theta(V)$, because the operation is performed one time for each vertex.

The process of relaxing an edge (u, v) occurs when it is tested whether it is possible to improve the shortest path to v. In affirmative case, the estimate $v.d$ and the predecessor $v.\pi$ are updated. In the pseudo-code in 2 from [3], we can see a relation step on edge (u, v) in $\mathcal{O}(1)$ time:

The two following algorithms presented in this section use these Initialize-Single-Source and Relaxing processes.

Algorithm 2 Relax (u, v, w)

```
if v.d > u.d + w(u, v) then
    v.d = u.d + w(u, v)
    v.π = u
end if
```

1) Bellman-Ford algorithm: The Bellman-Ford algorithm can solve the shortest-path problem in the general case in which it is possible to have edges with negative weights. This algorithms returns a boolean value that indicates whether or not there is a negative-weight cycle that is reachable from the source. That is necessary, because, if there is such negative cycle, no solution exists, because we could generate arbitrarily short paths by going through this loop multiple times. If there is no negative cycle, the algorithm finds the shortest paths and their weights.

The pseudo-code of the Bellman-Ford algorithm can be seen in 3 [3].

Algorithm 3 Bellman-Ford (G, w, s)

```
Initialize-Single-Source (G, s)
for I = 1 to |G.V| − 1 do
    for each edge (u, v) ∈ G.E do
        Relax (u, v, w)
    end for
end for
for each edge (u, v) ∈ G.E do
    if v.d > u.d + w(u, v) then
        return FALSE
    end if
end for
return TRUE
```

The initialization takes $\Theta(V)$ time, as we mentioned earlier. The first loop has $|V| - 1$ iterations and each iteration passes over all the edges and takes $\Theta(E)$ time. The final loop takes $\mathcal{O}(E)$. Therefore, the Bellman-Ford algorithm runs in time $\Theta(VE)$ [3]. After the i iterations of the main loop, the correct shortest-path distances of every path with at most i edges leaving the source were corrected calculated. If the graph has no negative cycles, the shortest path from the source to any other vertex is a simple path with at most $V - 1$ edges [4].

A complete proof of the correctness of the Bellman-Ford algorithm may be seen in [3].

Furthermore, a graph with V nodes has it most $E = \frac{V(V-1)}{2}$ edges, that is a quadratic quantity in V. Therefore, we can say that Bellman Ford has complexity $\mathcal{O}(V^3)$.

Our implementation of the Bellman Ford algorithm can be seen in appendix A.

2) Dijkstra algorithm: The Dijkstra algorithm solves the single-source shortest-paths problem for the case in which negative weights are not allowed. In this algorithm, we maintain a set of Unvisited nodes. The algorithm repeatedly selects the vertex u of the Unvisited Vertices Set with the minimum shortest-path estimate and relax all the edges of u and relax

them. After this process is completed, u is considered visited and no longer belongs to the Unvisited set. In order to have a more efficient implementation of the Dijkstra algorithm, in general it is used a min-priority queue Q of vertices, in which the keys are their d values [2]. A pseudo-code adapted from [4] can be seen below:

Algorithm 4 Dijkstra (G, w, s)

```
Initialize-Single-Source (G, s)
Q = G.V
while Q! = ∅ do
  u = EXTRACT-MIN(Q)
  for each vertex v ∈ G.Adj[u] do
    if v.d > u.d + w(u, v) then
      v.d = u.d + w(u, v)
      v.π = u
      DecreaseKey(v, d(v))
    end if
  end for
end while
```

In the algorithm 4, a slightly different process of relaxation happens. After the edge is relaxed, the function DecreaseKey is called, so that the Priority Queue remains correctly set.

The complexity of this algorithm depends upon how the Priority Queue (PQ) is implemented. If it is implemented like a Binary Heap, the functions *Insert*, *ExtractMin* and *DecreaseKey* have complexity $\mathcal{O}(\log V)$ [4]. The time to construct the binary heap is $\mathcal{O}(V)$ [3]. Besides, the functions *ExtractMin* is called V times and the function *DecreaseKey* is called at most E times. Therefore, the complexity of this algorithm is $\mathcal{O}((E + V).\log V)$, that can be simplified as $\mathcal{O}(E.\log V)$ for sparse graphs.

In [4], it is presented a variation of the Dijkstra algorithm, in which the priority queue is not initialized with all the nodes, only the source. The pseudocode of this variation can be seen in 5.

Algorithm 5 Dijkstra 2 (G, w, s)

```
Initialize-Single-Source (G, s)
Insert ((s, 0))
while Q! = ∅ do
  u = EXTRACT-MIN(Q)
  for each vertex v ∈ G.Adj[u] do
    if v.d > u.d + w(u, v) then
      v.d = u.d + w(u, v)
      v.π = u
      if v is in Q then
        DecreaseKey(v, d(v))
      else
        Insert(v, d(v))
      end if
    end if
  end for
end while
```

There is a proof in [4] that in this algorithm, each node is scanned at most once and each edge is relaxed at most once. Therefore, the complexity is the same as in the previous case: $\mathcal{O}(E.\log V)$. The running time can be improved to $\mathcal{O}(E + V\log(V))$ if Fibonacci Heap, a more complex implementation Priority Queue is used [4].

However, the implementation of priority queue from Python that we use in this work does not support the DecreaseKey operation. It only supports the operations Insert and ExtractMin. For this reason, we use a variation of the algorithm adapted from [5]. The pseudo-code can be seen in 6.

Algorithm 6 Dijkstra 3 (G, w, s)

```
Initialize-Single-Source (G, s)
Visited = ∅
Insert ((s, 0))
while Q! = ∅ do
  u = EXTRACT-MIN(Q)
  if u not in Visited then:
    Visited = Visited ∪ u
    for each vertex v ∈ G.Adj[u] do
      if v.d > u.d + w(u, v) then
        v.d = u.d + w(u, v)
        v.π = u
        Insert(v, d(v))
      end if
    end for
  end if
end while
```

In this variation, whenever a shorter distance is found to a node v, the item $(v, d(v))$ is inserted in the priority queue. In this way, the same node can be inserted many times, with decreasing keys. All instances of the node v in the heap should be ignored when extracted from the heap. We identify these instances to be skipped keeping a set of Visited nodes, that is one of the main differences of that last implementation. Thus, we avoid the use of DecreaseKey operation. According to [5], this implementation also achieves complexity $\mathcal{O}(E\log V)$.

The reason for this is that, in the main loop, the operation *Insert* is done at most E times and it has complexity $\mathcal{O}(\log V)$. Besides, the ExtractMin operation (that also has complexity $\mathcal{O}(\log V)$) is performed at most E times. Therefore, the overall complexity is $\mathcal{O}(E\log V)$.

It is said that the Dijkstra's algorithm has a greedy strategy because it always chooses the vertex with the minimum shortest-path estimate in the set of the unvisited vertices. Having a greedy strategy is not a guarantee of finding an optimal solution in general [3]. However, in this case, Dijkstra's algorithm indeed is able to find the optimal solution and a proof of this can be found in [3].

Our implementations of the Dijkstra's algorithm for the cases of distance and time travel can be seen in appendices B and C, respectively.

D. Experiments

The first experiment aims to show the difference between the execution times of the Dijkstra's and Bellman-Ford's algorithms. As we have shown in the previous section, these algorithms have different time complexities, which lead to very different execution times. In order to demonstrate that, we apply both algorithms in three different graphs, with different number of nodes and edges. All the graphs are centered in a point with coordinates (-23.19952, -45.89139) and are generated with the function *ox.graph.graph_from_point*. This function has a parameter *distance* which enables the choice of a desired radius around the point.

In the second experiment, we show the usefulness of shortest-path algorithms in the process of choosing a place to live. We consider a hypothetical situation in which a person wants to choose a place to live taking into account the distances or the time travels to the places he frequents the most: the university campus of Federal University of São Paulo (Unifesp), the Maranata Baptist Church (IBM), Department of Aerospace Science and Technology (DCTA) and a Super Market. The approximate coordinates of these places can be seen in the Table I.

TABLE I
MOST FREQUENTED PLACES

Place	Latitude	Longitude
Unifesp	-23.1638	-45.7932
IBM	-23.2700	-45.8189
DCTA	-23.2052	-45.8825
Market	-23.2158	-45.9060

We consider two different metrics in the application of the Dijkstra's algorithm: weights of the edges being length or time travel. In the first situation, we find the minimum distance between the nodes. In the second situation, we find the minimum time of travel between two nodes. In the calculation of the time travel, it is considered the length of the edge and the maximum speed allowed on the road.

However, not all edges have the maximum speed attribute. Then, we have to use the function osmnx.speed.add_edge_speeds. This function imputes free-flow speed for all edges with missing values, using the mean maxspeed value of the edges of the same highway type. This imputation, due to lack of data, is a source of imprecision in the analysis [10]. Posteriorly, we use the function osmnx.speed.add_edge_travel_times, that calculates free-flow travel time in seconds long each edge, using the length and the maxspeed information [10].

This person looks for a place to live in five different neighborhoods: Jardim Satélite (JS), Jardim São Dimas (JSD), Vila Adyana (VA), Jardim Aquarius (JA) and Jardim das Flores (JF). The approximate coordinates of the possible places to live can be seen in the table II.

In order to use those points in our algorithms, we first have to use the function *ox.distance.nearest_nodes* to find out which graph node is the nearest to the point to which we want to apply the algorithm.

TABLE II
PLACES IN ANALYSIS

Place	Latitude	Longitude
Jardim Satélite	-23.2309	-45.8782
Jardim São Dimas	-23.1976	-45.8878
Vila Adyana	-23.1974	-45.8909
Jardim Aquarius	-23.2249	-45.9106
Jardim das Flores	-23.1463	-45.7992

III. RESULTS AND DISCUSSION

In the Table III, we show the time of execution of the Dijkstra and of the Bellman Ford algorithms for three different graphs.

TABLE III
TIME OF EXECUTION

Graph	Nodes	Edges	Dijkstra TE	Bellamn Ford TE
Graph 1	515	918	0.008167	0.96644
Graph 2	1422	2829	0.016765	5.1636
Graph 3	2604	5320	0.05545	21.9506

It is remarkable to see that the increase of time of the Bellman Ford algorithm is much greater than the Dijkstra's. That happen because the complexity of Bellman Ford is $\mathcal{O}(EV)$ and, as we said before, we can also say that it is $\mathcal{O}(V^3)$, while the complexity of Dijkstra is $\mathcal{O}(E \log V)$.

In the Table IV, we show the minimum distance to each point of interest from each possible place that is being analysed calculated using the Dijkstra algorithm.

TABLE IV
MINIMUM DISTANCES IN KM

Destination / Origin	JS	JSD	VA	JA	JF
Unifesp	16.871	13.270	13.711	17.191	6.431
IBM	7.957	12.715	13.156	14.907	22.775
DCTA	5.045	1.706	2.147	5.627	14.358
Market	7.512	4.233	3.804	1.859	18.964
Average Distance	9.346	7.981	8.205	9.896	15.632

Therefore, taking into account only the distance factor, Table IV leads to the following priority queue: $JSD > VA > JS > JA > JF$. It is important to mention that this analysis is a simplification. In order to be more realistic, it would be necessary to use a weighted average, so that more frequented places would have a greater importance to the decision.

In the Table V, we can see the minimum travel times to each point of interest from each possible place calculated using the Dijkstra considering the weights of the edges as the travel times.

Taking into account only the time factor, Table V leads to the same priority queue that Table IV did: $JSD > VA > JS > JA > JF$. However, it is important to mention that there is no guarantee that these two analysis,

TABLE V
MINIMUM TRAVEL TIMES IN MINUTES

Destination / Origin	JS	JSD	VA	JA	JF
Unifesp	14.11	11.03	11.57	14.80	8.79
IBM	4.93	9.22	9.17	10.18	20.28
DCTA	5.64	2.12	2.66	6.05	11.92
Market	7.01	4.66	4.09	2.03	16.37
Average Time	7.92	6.76	6.87	8.26	14.34

based on distance and time, will always lead to the same priority queue. Eventually, there may be longer paths that are faster because they go through roads that have higher maximum speed allowed.

IV. CONCLUSION

This work discussed the application of two very famous algorithms that tackle shortest-paths problem. We have showed that the complexity of Bellman-Ford's algorithm is significantly greater than Dijkstra's. On the other hand, Bellman-Ford's algorithm is able to find minimum paths efficiently even when the graph has edges with negative weights.

We demonstrated the use of the Dijkstra's algorithm to aid decision making in situations in which distances or time travels to multiple points in a city are important.

One limitation of our experiment is that we used a simple average of the distances or of the times travels to the points of interest. In order to do a more realistic analysis, it is important to use a weighted average to take into account frequency of visiting of each point.

One limitation of the use of the Bellman Ford algorithm is its complexity $\mathcal{O}(V^3)$, which makes unpractical that its application in problems with large graphs like the one we used in this work. In future work, it would be interesting to use a graphical processing unit (GPU) and a parallel implementation of Bellman-Ford, like in [7].

Other possible further improvement in to ameliorate the implementation of the algorithms analysed in this work. One way to make the Dijkstra algorithm more efficient is implementing the priority queue as a Fibonacci heap, instead of a binary heap.

Furthermore, in a future work, we can compare the performance of these algorithms with other shortest-paths algorithms, like Floyd-Warshall or A^*.

REFERENCES

[1] Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. Computers, Environment and Urban Systems 65, 126-139. doi:10.1016/j.compenvurbsys.2017.05.004

[2] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269–271.

[3] Cormen, T., Leiserson, C., Rivest, R., Stein, C., 2001. Introduction to Algorithms. MIT Press.

[4] Erickson, J. Algorithms.

[5] Brodal, G. S. (2022). Priority Queues with Decreasing. 11th International Conference on Fun with Algorithms Keys, Island of Favignana, Sicily, Italy.

[6] H. Wang, Y. Yu, and Q. Yuan, "Application of dijkstra algorithm in robot path-planning," in Proc. Second Int. Conf. Mechanic Automat. Control Eng., 2011, pp. 1067–1069.

[7] A. Schambers, M. Eavis-O'Quinn, V. Roberge, and M. Tarbouchi, "Route planning for electric vehicle efficiency using the Bellman-Ford algorithm on an embedded GPU," in Proc. 4th Int. Conf. Optim. Appl., Apr. 2018, pp. 1–6.

[8] https://www.openstreetmap.org/about

[9] https://wiki.openstreetmap.org/wiki/Contributors

[10] https://osmnx.readthedocs.io/en/stable/osmnx.html#osmnx.speed.add_edge_travel_times

APPENDIX A
BELLMAN FORD ALGORITHM IMPLEMENTATION

```
def bellman_ford(G,source):
  #Initialization
  distancias = {v: float('inf') for v in G.nodes()}
  previous_node = {v: [] for v in G.nodes()}
  distancias[source] = 0
  #Relaxing edges
  for i in range(G.number_of_nodes()-1):  #repeat V-1 times
    for edge in G.out_edges.data("length",default=0):
      u,v,w = edge[0],edge[1],edge[2]
      #u é o primeiro vértice, v é o segundo vértice e w é o peso
      if(distancias[v] > distancias[u] + w):
        distancias[v] = distancias[u] + w
        previous_node[v] = u

  #Verify whether there is a negative cycle
  for edge in G.out_edges.data("length",default=0):
      u,v,w = edge[0],edge[1],edge[2]
      if(distancias[v] > distancias[u] + w):
        return False

  return distancias, previous_node
```

Fig. 2. Bellman Ford Implementation

In the beginning, the initialization is done, as described in this work. The first loop performs the relaxing of all edges $V-1$ times. This is sufficient to guarantee that the correct shortest-path distances of every path with at most $V-1$ edges were corrected calculated. The second loop is performed in order to guarantee that there are no negative cycles.

APPENDIX B
DIJKSTRA'S ALGORITHM FOR DISTANCE IMPLEMENTATION

```
def dijkstra_meu(G,source):
  #Initialization
  distancias = {v: float('inf') for v in G.nodes()}
  previous_node = {v: [] for v in G.nodes()}
  distancias[source] = 0
  #Criando pilha de prioridade com os vértices do grafo
  from queue import PriorityQueue
  unvisited = PriorityQueue()
  #Vamos apenas o nosso nó fonte na pilha de prioridade
  unvisited.put((distancias[source],source))
  #Vamos criar um conjunto para armazenar o conjunto dos nós visitados
  visited = set()
  #Enquanto a lista dos não visitados não estiver vazia, vamos repetir o loop abaixo
  while (not unvisited.empty()):
    #Para extrair o vértice em Q com distância mínima, basta extrair normalmente,
    #pois Q é uma fila de prioridade
    current_node = unvisited.get()[1]   #usamos o índice 1 para pegar o nó e não a distância
    if(current_node not in visited):
      visited.add(current_node)
      #Realizar o seguinte loop para cada vizinho do current_node
      for neighbor in G._adj[current_node]:
        u,v,w = current_node, neighbor, G._adj[current_node][neighbor][0]['length']
        if(distancias[v] > distancias[u] + w):
          distancias[v] = distancias[u] + w
          previous_node[v] = u
          #now, add the neighbor in the priority queue
          unvisited.put((distancias[v],v))
  return distancias,previous_node
```

Fig. 3. Dijkstra for Distance Implementation

APPENDIX C
DIJKSTRA'S ALGORITHM FOR TIME TRAVEL IMPLEMENTATION

```python
def dijkstra_tempo(G,source):
  #Initialization
  distancias = {v: float('inf') for v in G.nodes()}
  previous_node = {v: [] for v in G.nodes()}
  distancias[source] = 0
  #Criando pilha de prioridade com os vértices do grafo
  from queue import PriorityQueue
  unvisited = PriorityQueue()
  #Vamos apenas o nosso nó fonte na pilha de prioridade
  unvisited.put((distancias[source],source))
  #Vamos criar um conjunto para armazenar o conjunto dos nós visitados
  visited = set()
  #Enquanto a lista dos não visitados não estiver vazia, vamos repetir o loop abaixo
  while (not unvisited.empty()):
    #Para extrair o vértice em Q com distância mínima, basta extrair normalmente,
    #pois Q é uma fila de prioridade
    current_node = unvisited.get()[1]   #usamos o índice 1 para pegar o nó e não a distância
    if(current_node not in visited):
      visited.add(current_node)
      #Realizar o seguinte loop para cada vizinho do current_node
      for neighbor in G._adj[current_node]:
        u,v,w = current_node, neighbor, G._adj[current_node][neighbor][0]['travel_time']
        if(distancias[v] > distancias[u] + w):
          distancias[v] = distancias[u] + w
          previous_node[v] = u
          #now, add the neighbor in the priority queue
          unvisited.put((distancias[v],v))
  return distancias,previous_node
```

Fig. 4. Dijkstra for Time Travel Implementation

In those implementations, we use Dijkstra 3, a variation that avoids the use of DecreaseKey operation, because the Priority Queue built-in of Python does not support this operation. Then, instead of performing DecreaseKey, we simple add the nodes to the Priority Queue, whenever a shorter distance is found to a vertex v. In order not to extract multiple copies of the same node, we maintain a set of visited nodes.