AMRITA VISHWA VIDYAPEETHAM ETTIMADAI, COIMBATORE

EMERGENCY VEHICLE DISPATCH SYSTEM USING DIJKSTRA'S ALGORITHM

Department	CEN
Course	21AIE212
Semester	4
Instructor	Dr. Vineet

GROUP NUMBER – 7

GROUP MEMBERS

SL NO	**NAME**	**ROLL NUMBER**
1	DINESH KUMAR M.R.	CB.EN.U4AIE20011
2	K.S.PAVAL	CB.EN.U4AIE20047
3	SHREYA SANGHAMITRA	CB.EN.U4AIE20066
4	N.T. SHRISH SURYA	CB.EN.U4AIE20067

PROJECT REPORT SUBMITTED FOR THE END SEMESTER EXAMINATION OF
21AIE212
ON 21.07.2022

EXTERNAL EXAMINER INTERNAL EXAMINER

ACKNOWLEDGMENT

We would like to thank our professor Dr. Vineet who gave us his valuable suggestions and ideas when we were in need of them to work on our project “Emergency Vehicle Dispatch System”.

We are also grateful to our university, Amrita Vishwa Vidyapeetham for giving us the opportunity to work on this project. Last but not the least, we would like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

INDEX

INTRODUCTION

Dijkstra's Algorithm is a graph algorithm for finding the shortest path from a source node to all other nodes in a graph (Single Source Shortest Path). It is a type of greedy algorithm. It only works on weighted graphs with positive weights. It has a time complexity of O(V^2) (a graph is called positively weighted if all of its edges have only positive weights).

WORKING OF DIJKSTRA'S ALGORITHM:

Highlights:

1. Greedy Algorithm
2. Relaxation

Dijkstra's Algorithm requires a graph and source vertex to work.The algorithm is purely based on greedy approach and thus finds the locally optimal choice(local minima in this case) at each step of the algorithm.

In this algorithm each vertex will have two properties defined for it-

- Visited property:-
    - This property represents whether the vertex has been visited or not.

    - We are using this property so that we don't revisit a vertex.
    - A vertex is marked visited only after the shortest path to it has been found.
- Path property:-
    - This property stores the value of the current minimum path to the vertex. Current minimum path means the shortest way in which we have reached this vertex till now.
    - This property is updated whenever any neighbour of the vertex is visited.
    - The path property is important as it will store the final answer for each vertex.

Initially all the vertices are marked unvisited as we have not visited any of them. Path to all the vertices is set to infinity excluding the source vertex. Path to the source vertex is set to zero(0).

Then we pick the source vertex and mark it visited.After that all the neighbours of the source vertex are accessed and relaxation is performed on each vertex. Relaxation is the process of trying to lower the cost of reaching a vertex using another vertex.

In relaxation, the path of each vertex is updated to the minimum value amongst the current path of the node and the sum of the path to the previous node and the path from the previous node to this node.

Assume that p[v] is the current path value for node v, p[n] is the path value upto the previously visited node n, and w is the weight of the edge between the current node and previously visited node(edge weight between v and n)

Mathematically, relaxation can be represented as: p[v]=minimum(p[v], p[n] + w)

Then in every subsequent step, an unvisited vertex with the least path value is marked visited and its neighbour's paths updated.

The above process is repeated till all the vertices in the graph are marked visited.

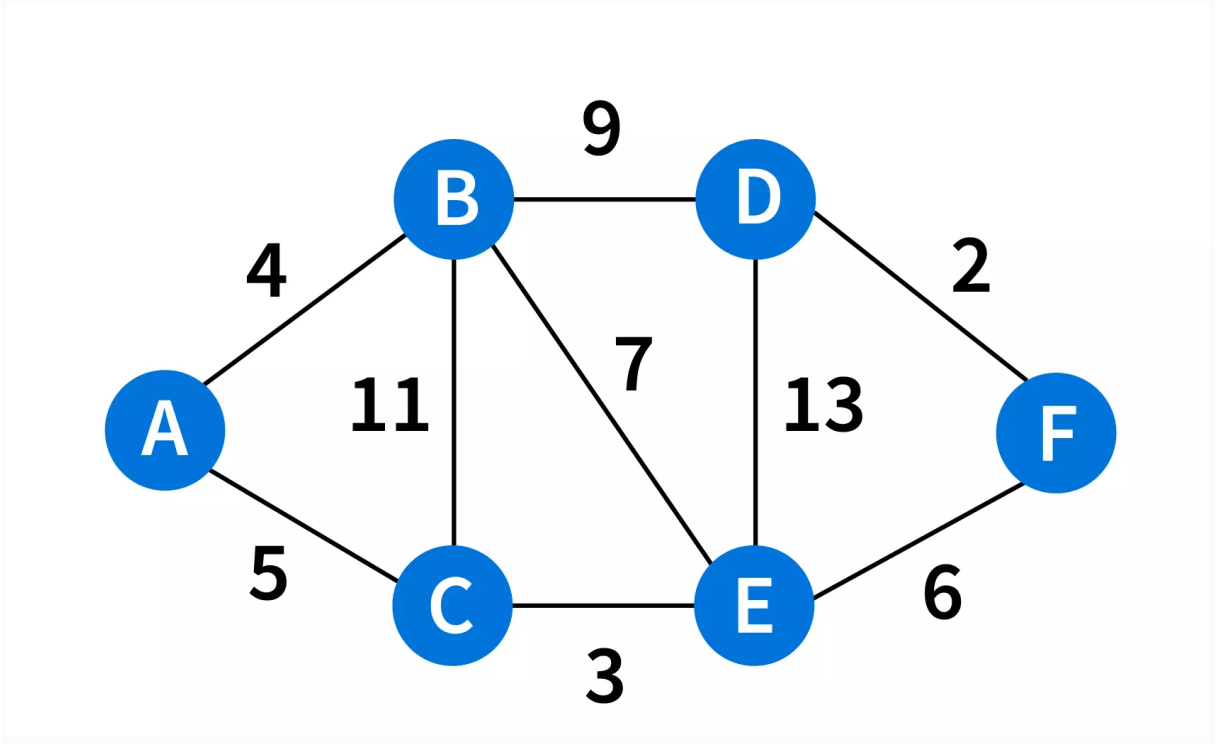
Whenever a vertex is added to the visited set, the path to all of its neighbouring vertices is changed according to it.

If any of the vertex is not reachable(disconnected component), its path remains infinity. If the source itself is a disconnected component, then the path to all other vertices remains infinity.

ALGORITHM

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node, let's say C.
3. For each neighbour N of the current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. Go to step 2 if there are any nodes that are unvisited.

EXAMPLE:



Let's assume the below graph as our input with the vertex A being the source.

- Initially all the vertices are marked unvisited.
- The path to A is 0 and for all the other vertices it is set to infinity.
- Now the source vertex A is marked as visited.Then its neighbours are accessed(only accessed and not visited).
- The path to B is updated to 4 using relaxation as the path to A is 0 and path from A to B is 4, so min((0+4),∞) is 4.

- The path to C is updated to 5 using relaxation as the path to A is 0 and path from A to C is 5, so min((0+5),∞) is 5.Both the neighbours of A are relaxed so we move ahead.
- Then the next unvisited vertex with the least path is picked and visited. So vertex B is visited and its unvisited neighbours are relaxed.After relaxing path to C remains 5, path to E becomes 11 and path to D becomes 13.
- Then vertex C is visited and its unvisited neighbour E is relaxed. While relaxing E, we find that the path to E via is smaller than its current path, so the path to E is updated to 8.All neighbours of C are now relaxed.
- Vertex E is visited and its neighbours B,D and F are relaxed. As only vertex F is unvisited only, F is relaxed. Path of B remains 4, path to D remains 13 and path to F becomes 14(8+6).
- Then vertex D is visited and only F is relaxed. .The path to vertex F remains 14.
- Now only vertex F is remaining so it is visited but no relaxations are performed as all of its neighbours are already visited.
- As soon as all the vertices become visited the program stops.

The final paths we get are:

- A=0(source)
- B=4(A->B)
- C=5(A->C)
- D=13(A->B->8
- E=8(A->C->E)
- F=14!(A->C->E->F)

PSEUDOCODE

```
function Dijkstra(Graph, source):
  for each vertex v in Graph:
     distance[v] = infinity
 distance[source] = 0
 G = the set of all nodes of the Graph


 while G is non-empty:
     Q = node in G with the least dist[ ]
     mark Q visited
     for each neighbour N of Q:
          alt_dist = distance[Q] + dist_between(Q, N)
          if alt-dist < distance[N]
               distance[N] := alt_dist
 return distance[ ]
```

EMERGENCY VEHICLE DISPATCH SYSTEM:

The algorithm should find the shortest distance between the nodes and allocate vehicles by processing the requests one by one. Given that there are three different types of emergency vehicles:

1. Ambulance

2. Fire Truck

3. Police car

Also assume that every request only needs one emergency vehicle.

Graph Construction:

GRAPH CONSTRUCTION:

Here, we have considered zip codes as nodes and distance between the zip codes as weights. And implemented the solution in python with json data.

Sample data:

Request data

"requests":

[{

"id": 1,

"vehicle_type": 2,

"zipcode": 64167,

"vehicle_id": null,

"distance": 0

},

Vehicles data

"vehicles":

[{

"id": 1,

"type": 1,

"zipcode": 64149

},

Distance between nodes

"distances":

[{

"zipcode1": 64149,

"zipcode2": 64150,

"distance": 4

},

ATTRIBUTES CONSIDERED:

Zip codes as nodes

Distances as the edges weight with connected vertices

Vehicle type: among the above three types

IMPLEMENTED SOLUTION:

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalised. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialise all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn't include all vertices

a) Pick a vertex u which is not there in sptSet and has minimum distance value.

b) Include u to sptSet.

c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

CODE:

Dijkstra.py

```
import math

import heapq

class Vertex:

    def __init__(self, node): # node is the vertex id

        self.id = node # set the id

        self.adjacent = {} # set the adjacent vertices
```

```
        self.distance = math.inf # set the distance to
infinity
        self.visited = False # set the visited flag to false
        self.previous = None # set the previous vertex to none

    def add_neighbor(self, neighbor, weight=0):
# neighbor is the vertex id, weight is the weight of the edge
        self.adjacent[neighbor] = weight
# add the neighbor and weight to the adjacent vertices

    def get_connections(self): # get the adjacent vertices
        return self.adjacent.keys()
# return the keys of the adjacent vertices
    def get_id(self): # get the vertex id
        return self.id
    def get_weight(self, neighbor):
# get the weight of the edge between this vertex and the given
neighbor
        return self.adjacent[neighbor]

    def set_distance(self, dist):
# set the distance to the given distance
        self.distance = dist

    def get_distance(self): # get the distance
        return self.distance
```

```
    def set_previous(self, prev): # set the previous vertex
        self.previous = prev

    def set_visited(self): # set the visited flag to true
        self.visited = True

    def __str__(self): # print the vertex id
        return str(self.id) + ' adjacent: ' + str([x.id for x
in self.adjacent])

    def __lt__(self, other):
# compare the distance to the other vertex
        return self.id < other.id


class Graph:
    def __init__(self): # initialize the graph
        self.vert_dict = {}
# set the vertices to an empty dictionary
        self.num_vertices = 0
# set the number of vertices to 0

    def __iter__(self): # iterate through the vertices
        return iter(self.vert_dict.values())
```

```
    def add_vertex(self, node): # add a vertex to the graph
        self.num_vertices = self.num_vertices + 1
 # increment the number of vertices
        new_vertex = Vertex(node)
# create a new vertex
        self.vert_dict[node] = new_vertex
# add the vertex to the dictionary
        return new_vertex

    def get_vertex(self, n):
# get the vertex with the given id
        if n in self.vert_dict:
# if the id is in the dictionary
            return self.vert_dict[n] # return the vertex
        else:
            return None

    def add_edge(self, frm, to, cost = 0):
# add an edge to the graph
        if frm not in self.vert_dict:
# if the from vertex is not in the dictionary
            self.add_vertex(frm)
# add the vertex if it is not in the graph
        if to not in self.vert_dict:
# if the to vertex is not in the dictionary
```

```
            self.add_vertex(to)

        self.vert_dict[frm].add_neighbor(self.vert_dict[to],
cost) # add the edge to the graph
        self.vert_dict[to].add_neighbor(self.vert_dict[frm],
cost)

    def get_vertices(self): # get the vertices
        return self.vert_dict.keys()

    def set_previous(self, current):
        self.previous = current
# set the previous vertex to the current vertex

    def get_previous(self, current):
        return self.previous  # return the previous vertex

    def reset_vertices(self):
# reset the visited flag for all vertices
        for id, vertex in self.vert_dict.items():
# iterate through the vertices
            vertex.distance = math.inf
            vertex.visited = False
            vertex.previous = None
```

```
def shortest(v, path):
# get the shortest path to the given vertex
    if v.previous: # if the previous vertex is not none
        path.append(v.previous.get_id())
# add the previous vertex to the path
        shortest(v.previous, path)
# get the shortest path to the previous vertex
    return



def dijkstra(aGraph, start):
# get the shortest path from the start vertex to all other
vertices
    start.set_distance(0)
    unvisited_queue = [(v.get_distance(), v) for v in aGraph]
# create a priority queue of the unvisited vertices


    heapq.heapify(unvisited_queue)
# heapify the priority queue


    while len(unvisited_queue):
# while there are unvisited vertices
        uv = heapq.heappop(unvisited_queue)
# get the closest vertex
        current = uv[1]
        current.set_visited() # set the visited flag to true
```

```
        for next in current.adjacent:


# iterate through the adjacent vertices
            if next.visited:
                continue
            new_dist = current.get_distance() +
current.get_weight(next) # get the new distance


            if new_dist < next.get_distance():
# if the new distance is less than the current distance
                next.set_distance(new_dist)
# set the new distance
                next.set_previous(current)
# set the previous vertex
                # print('updated : current = %s next = %s
new_dist = %s' \
                    # % (current.get_id(), next.get_id(),
next.get_distance()))
            # else:
                # print('not updated : current = %s next = %s
new_dist = %s' \
                    # % (current.get_id(), next.get_id(),
next.get_distance()))


        while len(unvisited_queue):
# while there are unvisited vertices
```

```
            heapq.heappop(unvisited_queue)
# pop the closest vertex


        unvisited_queue = [(v.get_distance(), v) for v in
aGraph if not v.visited]
# create a priority queue of the unvisited vertices
        heapq.heapify(unvisited_queue)
# heapify the priority queue
```

Main.py

```
import sys
import json
import dijkstra



if __name__ == '__main__':

    with open(sys.argv[1], 'r') as data_file:  # open the file
        datastore = json.load(data_file)
# load the data from the file

    distances = datastore['distances'] # get the distances
    requests = datastore['requests']   # get the requests
    vehicles = datastore['vehicles']   # get the vehicles
```

```
    for vehicle in vehicles:
        vehicle['available'] = True
# set all vehicles to available

    g = dijkstra.Graph() # create a graph

    for distance in distances:
        if not distance['zipcode1'] in g.get_vertices():
# if the first zipcode is not in the graph
            g.add_vertex(distance['zipcode1'])
# add the first zipcode to the graph

        if not distance['zipcode2'] in g.get_vertices():
# if the second zipcode is not in the graph
            g.add_vertex(distance['zipcode2'])
# add the second zipcode to the graph

        g.add_edge(distance['zipcode1'], distance['zipcode2'],
distance['distance']) # add the edge to the graph

    """
    print('Graph data:') # print the graph data
    for v in g:
        for w in v.get_connections():
# for each vertex in the graph
            vid = v.get_id() # get the vertex id
```

```
            wid = w.get_id()

            print('( %s , %s, %3d)'  % ( vid, wid,
v.get_weight(w)))

# print the vertex id, the vertex id of the neighbor, and the
weight of the edge

    """

    def get_vehicle(vehicle_type, zipcode):

        available_vehicles = [v for v in vehicles if v['type']
== vehicle_type and v['available']]

# get all available vehicles of the given type


        if len(available_vehicles) > 0:

# if there are available vehicles

            g.reset_vertices()  # reset the vertices

            place = g.get_vertex(zipcode)

# get the vertex of the given zipcode

            dijkstra.dijkstra(g, place)

# run dijkstra's algorithm on the graph


            for av in available_vehicles:

                av['distance'] =
g.get_vertex(av['zipcode']).get_distance()

# get the distance of the vehicle from the given zipcode


            available_vehicles = sorted(available_vehicles,
key=lambda k: k['distance'])

# sort the available vehicles by distance
```

```
        return available_vehicles

    for request in requests: # for each request

        ev = get_vehicle(request['vehicle_type'],
request['zipcode'])

# get the available vehicles for the request


        if len(ev) > 0: # if there are available vehicles

            vehicles[vehicles.index(ev[0])]['available'] =
False # set the vehicle to unavailable

            request['vehicle_id'] = ev[0]['id']

# set the vehicle id

            request['distance'] = ev[0]['distance']

# set the distance

            print(request)



    # print(vehicles)

    # print(requests)
```

OUTPUT:

```
Command Prompt
Microsoft Windows [Version 10.0.22000.795]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd Desktop/DAA

C:\Users\Admin\Desktop\DAA>python main.py data.json
{'id': 1, 'vehicle_type': 2, 'zipcode': 64167, 'vehicle_id': 41, 'distance': 9}
{'id': 2, 'vehicle_type': 3, 'zipcode': 64160, 'vehicle_id': 22, 'distance': 5}
{'id': 3, 'vehicle_type': 1, 'zipcode': 64150, 'vehicle_id': 9, 'distance': 2}
{'id': 4, 'vehicle_type': 1, 'zipcode': 64152, 'vehicle_id': 14, 'distance': 4}
{'id': 5, 'vehicle_type': 2, 'zipcode': 64157, 'vehicle_id': 16, 'distance': 6}
{'id': 6, 'vehicle_type': 3, 'zipcode': 64159, 'vehicle_id': 25, 'distance': 0}
{'id': 7, 'vehicle_type': 3, 'zipcode': 64158, 'vehicle_id': 23, 'distance': 0}
{'id': 8, 'vehicle_type': 2, 'zipcode': 64152, 'vehicle_id': 7, 'distance': 0}
{'id': 9, 'vehicle_type': 1, 'zipcode': 64164, 'vehicle_id': 32, 'distance': 6}
{'id': 11, 'vehicle_type': 1, 'zipcode': 64166, 'vehicle_id': 29, 'distance': 13}
{'id': 12, 'vehicle_type': 2, 'zipcode': 64154, 'vehicle_id': 12, 'distance': 6}
{'id': 13, 'vehicle_type': 2, 'zipcode': 64151, 'vehicle_id': 10, 'distance': 4}
{'id': 14, 'vehicle_type': 3, 'zipcode': 64167, 'vehicle_id': 43, 'distance': 9}
{'id': 15, 'vehicle_type': 1, 'zipcode': 64165, 'vehicle_id': 26, 'distance': 18}
{'id': 16, 'vehicle_type': 2, 'zipcode': 64153, 'vehicle_id': 24, 'distance': 5}
{'id': 17, 'vehicle_type': 2, 'zipcode': 64152, 'vehicle_id': 3, 'distance': 9}
{'id': 18, 'vehicle_type': 3, 'zipcode': 64164, 'vehicle_id': 36, 'distance': 6}
{'id': 20, 'vehicle_type': 1, 'zipcode': 64165, 'vehicle_id': 19, 'distance': 23}
{'id': 21, 'vehicle_type': 2, 'zipcode': 64163, 'vehicle_id': 33, 'distance': 0}
{'id': 22, 'vehicle_type': 3, 'zipcode': 64150, 'vehicle_id': 5, 'distance': 0}
{'id': 23, 'vehicle_type': 3, 'zipcode': 64160, 'vehicle_id': 30, 'distance': 11}
{'id': 24, 'vehicle_type': 2, 'zipcode': 64153, 'vehicle_id': 21, 'distance': 9}
{'id': 25, 'vehicle_type': 1, 'zipcode': 64161, 'vehicle_id': 20, 'distance': 9}

C:\Users\Admin\Desktop\DAA>
```

TIME COMPLEXITY

1. First thing we need to do is find the unvisited vertex with the smallest path. For that we require $O(V)$ time as we need to check all the vertices.
2. Now for each vertex selected as above, we need to relax its neighbours which means to update each neighbour's path to the smaller value between its current path or to the newly found. The time required to relax one neighbour comes out to be of order of O(1) (constant time).
3. For each vertex we need to relax all of its neighbours, and a vertex can have at most V-1 neighbours, so the time required to

update all neighbours of a vertex comes out to be [O(V) * O(1)] = O(V)

So now following the above conditions, we get:

- Time for visiting all vertices =O(V)
- Time required for processing one vertex=O(V)
- Time required for visiting and processing all the vertices = O(V)*O(V) =O(V^2)
- So the time complexity of dijkstra's algorithm using adjacency matrix representation comes out to be O(V^2)

SPACE COMPLEXITY

Space complexity of adjacency matrix representation of the graph in the algorithm is also O(V^2) as a V*V matrix is required to store the representation of the graph. An additional array of V length will also be used by the algorithm to maintain the states of each vertex but the total space complexity will remain O(V^2).

PROOF OF CORRECTNESS

Base Case:

- S = set of visited vertices, d(s) = dijkstra's solution, BPD(s) = best possible distance
- |S| = 1 since S can only grow in size, the only time |S|=1 is when the set S consists only of the source vertex.

- S = { source } and d(s) = BPD(s)

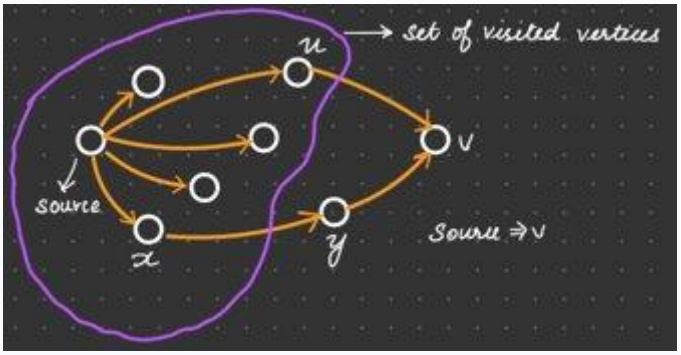
Inductive Hypothesis:

Let f be an element added to S, S' = S U { f }

Our hypothesis is that, for each element in S', x , d(x) = BPD(x)

Whenever we add a vertex i to the set of visited vertices S, we have the optimal distance from the source to that particular vertex i .

We need to show that d(x) = BPD(x)



- When |S| = n, we assume that the condition for optimal distances to all these vertices are true .
- All vertices inside the set have optimal distances from source
- Now, we need to add a node v to the set of visited vertices.
- We claim that

  d[v]>BPD[v] ------(1) (Statement)

- Therefore, we can say that the path to v through y is better. We need to add y to the set.

  BPD[x]+l (x,y) + … = BPD[v]

  … represents the possible intermediate vertices

$$BPD[x]+l(x,y) <= BPD[v] \quad \text{-----(2)}$$

The next thing we need to consider is the relaxation that might have happened when we added x to the set

$$d[y] <= d[x] + l(x,y) \quad \text{------(3)}$$

And finally, since we arrive to vertex y after x,

$$d[y] >= d[x] \quad \text{------(4)}$$

We now have 4 equations:

1. $d[v]>BPD[v]$
2. $BPD[x]+l(x,y) <= BPD[v]$
3. $d[y] <= d[x] + l(x,y)$
4. $d[y] >= d[x]$

Since x is already in the set of visited nodes, BPD[x]=d[x], so equation (2) can be updated

1. $d[v]>BPD[v]$
2. $d[x]+l(x,y) <= BPD[v]$
3. $d[y] <= d[x] + l(x,y)$
4. $d[y] >= d[x]$

LHS of (2) == RHS of (3)

1. d[v]>BPD[v]

2. d[y] <= BPD[v]

3. d [y] >= d[x]

Equation (2) and (1) can now be combined

1. d[y] <= d[v]

2. d [y] >= d[x] (We know that this is true)

Since we use a greedy algorithm, it needs to add it to the set before it adds y.

d[y]<d[v] & d[y]>=d[v]

This implies that

d[v]<d[v]

This cannot happen

Therefore, our contradiction is proved wrong

So, we prove that whenever we add a new node to the set, it is the best possible distance to that node.

Therefore, d[v] is optimal for all v

ADVANTAGES OF DIJKSTRA'S ALGORITHM:

- Once it has been carried out, one can find the least weight path to all permanently labeled nodes.
- A new diagram is not needed for each pass.
- Dijkstra's algorithm has an order of n2 so it is efficient enough to use for relatively large problems.

DISADVANTAGES OF DIJKSTRA'S ALGORITHM:

- The major disadvantage of the algorithm is the fact that it does a blind search thereby consuming a lot of time and wasting necessary resources.
- Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

RELATED ALGORITHMS:

A* algorithm is a graph/tree search algorithm that finds a path from a given initial node to a given goal node It employs a "heuristic estimate" h(x) that gives an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate. It follows the approach of breadth first search.

The Bellman Ford algorithm computes single-source shortest paths in a weighted digraph. It uses the same concept as that of Dijkstra's algorithm but can handle negative edges as well. It has a better running time than that of Dijkstra's algorithm.

Prims algorithm finds a minimum spanning tree for a connected weighted graph. It implies that it finds a subset of edges that form a tree where the total weight of all the edges in the tree is minimized. It is sometimes called the DJP algorithm or Jarnik algorithm.

CONCLUSION:

Therefore, we have managed to implement and analyse Dijkstra's algorithm in the context of applying it in an Emergency Vehicle Dispatch system.

Dijkstra's algorithm finds the shortest path between two nodes. Here we found the distance between the nodes from which the request is raised to all other nodes. Found shortest distance to all other nodes. Then we sorted the nodes in increasing order of distance and checked node with availability. The first node with the availability (requested vehicle) is the dispatcher node.