Application of Shortest Path Algorithm Using Dijkstra's Algorithm

Thomas Yong Hee Song Jr.

Valor School

ABSTRACT

While current GPS systems do efficiently map routes from a starting point to a desired destination for different modes of transportation, it is the unfortunate case that these systems are not optimized and personalized for the user. For example, a user might have a limited amount of money or preferences for types of roads they want to travel. In addition, many times, route-finding systems fail to acknowledge the realistic logistics of the road when mapping the shortest distance, such as constantly changing road congestion or waiting times at traffic lights. This lack of personalization and accuracy makes it difficult and frustrating to utilize such services at times. Developing an algorithm to tackle these problems would help users to get to place faster while optimizing their journey based on their interests, facilitating travel from place to place. This paper ultimately aims to gain inspiration from, and apply, Dijkstra's Algorithm to various shortest path problem variants and formulate actual al gorithms that are tested for efficiency.

1. INTRODUCTION

Graph

In a mathematical context, graphs represent shapes consisting of vertices and the edges that connect select vertices. These objects can be utilized to represent many real life situations. Some examples include networks consisting of interconnected computers or cities and the roads that connect them. Depending on the amount of edges relative to the number of vertices, graphs can be divided into dense/sparse graphs. Furthermore, the two most basic methods of data storage for vertices and edges are adjacency matrices and adjacency lists. Since the efficiency of each storage method differs case by case for the structure of the graph, the paper utilizes both to tackle each problem. [1]

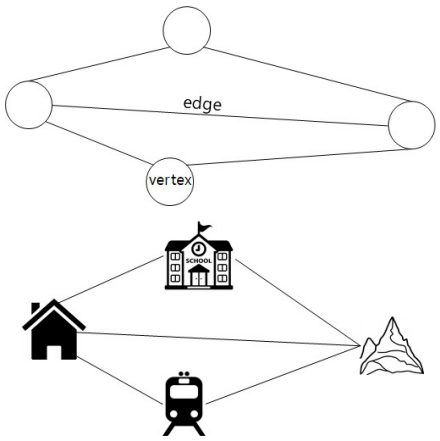

Figure 1: An example of graph modeling

Dijkstra's Algorithm

In order to solve the many shortest path problems presented in this paper, the famous path-finding algorithm – "Dijkstra's Algorithm" – was used extensively. Given a graph with vertices and weighted edges, Dijkstra's Algorithm can find the shortest path from one vertex to another. This can easily be translated to the situation at hand by converting a map into a weighted graph: places are represented as vertices, roads are represented as edges, and other factors such as distance or congestion can be input through the weights assigned to edges.

Design of the Paper

Section 2 of the paper is dedicated to proving the Dijkstra's algorithm, as well as explaining the numerous variations to be utilized in data-storage and algorithm building for the code. Since the density of the graph is determines which variation of Dijkstra's Algorithm is more efficient, two different methods were used: the "for loop" and the "priority queue." Section 3 presents said variations of the algorithm and explains how they were made to fit each shortest path problem. With each problem, the methods and results are presented. Finally, the conclusion compiles and brings all the results together.

2. DIJKSTRA'S ALGORITHM

2.1. OVERVIEW

Dijkstra's Algorithm maps a route by constantly updating the distance from one vertex to another. By default, all distances from the starting vertex are set to infinity regardless of the weights respective to each edge. The algorithm first looks at the neighboring nodes - if the newly calculated distance is smaller than the current distance, the distance is updated. Subsequently, the current node will be reallocated to the node that has the smallest assigned distance.

Again, the neighboring nodes are analyzed and if the newly calculated distance based on the weights is smaller than the current distance, the distance is updated and the current node is moved. This process is repeated until the destination becomes the current node. Alternatively, if a destination is not defined, the algorithm will continue to run until all distances have been updated.

2.2. PROOF OF ALGORITHM

The proof of the feasibility of the algorithm is quite simple through a proof by contradiction. Let's say that there is a set S of points whose shortest paths from the starting point have already been established. Suppose that there is a vertex x outside of it that, by Dijkstra's theorem is the next point to be registered so that the shortest path to vertex x should be one that is directly connected to set S. However, let us say that there is a vertex y such that an indirect path to vertex x through vertex y is the shorter path, thereby refuting the theorem. However, if this is true, vertex y would have been the next vertex to be registered rather than vertex x. Therefore, the contradiction is invalid. Although the proof is invalid when edges with negative weights are involved, since that is not the case for finding paths in real life, the proof is solid.

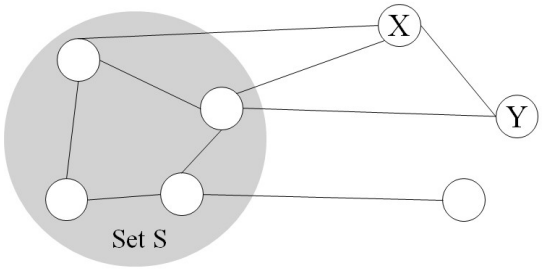

Figure 2: Proof of Dijkstra's algorithm

2.3. "FOR" LOOP VS. PRIORITY QUEUE

There are two main ways of constructing a python code for Dijkstra's Algorithm: employing the structure of a "for" loop or using a priority queue. As can be seen in the pseudo-code, the method of a "for" loop was used due to its superiority of time complexity in this case. Time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

When using the loop, for each vertex, the loop goes as such. It first searches among the vertices that do not have a set distance in order to find the current smallest value. This will result in a time complexity of V for each loop. Subsequently, after the vertex has been registered as the current vertex, each of its edges are evaluated in order to update them if a smaller distance is found. After all loops have been completed, this will result in a time complexity of E. Therefore, the final time complexity can be represented by $O(V^2+E)$.

On the other hand, the nature of a priority queue allows it to produce a time complexity of logE because it makes a tree of all the edges for each vertex. When the entire code has been run, the final time complexity will be O(ElogE). Since we are using maps of cities, it is important to note that the resulting graphs will be very dense. As a result, it follows that in such dense graphs, $V^2+E < ElogE$ since E is considerably bigger than V. Therefore, the "for" loop is a better choice in this case.

2.4. GRAPH STORAGE METHOD

When constructing the code in python, there were two ways to store the graph upon which the algorithm would operate on. The information pertaining to the graph, including the number of vertices, connections between vertices, and the weights of edges, could either be stored as an adjacency matrix or an adjacency graph. Using an adjacency list would mean creating a dictionary with each vertex being a key and the value being a list of pairs (connected vertex, weight of edge) to the corresponding vertex. An adjacency matrix would be a matrix with coordinates with vertices as values (v1, v2). If the two vertices aren't connected, the value in the matrix would be 0. If they are, the value would be the weight of the edge that connects them.

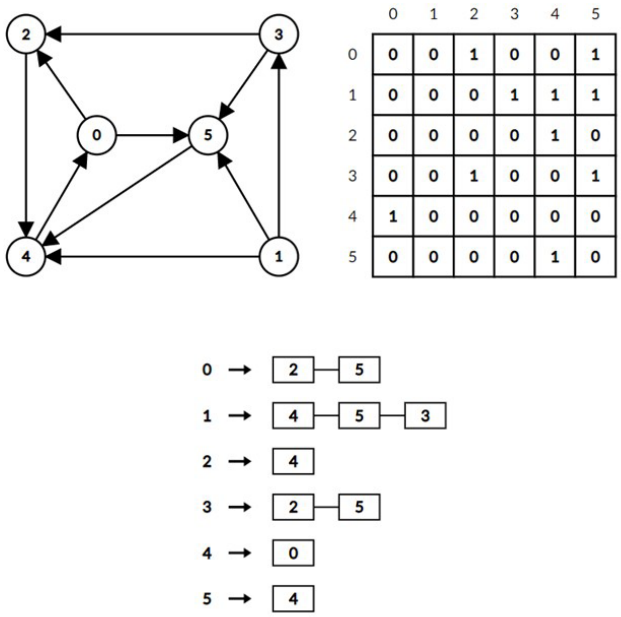

Figure 3: Adjacency Matrix(mid) and Adjacency List(right) of Sample Graph(left) [2]

Since running time is a crucial factor in choosing the optimal code, the method of creating an adjacency list was chosen because of its ability to run faster than an adjacency matrix. When the matrix is used, unnecessary information is used because vertex pairs that are not connected are included in the matrix. Therefore, the computer will have to go through extraneous data before it can access the vertices that are actually connected. On the other hand, the adjacency list simply stores data just for the vertices that are connected, eliminating much of the running time that is present when using a matrix. Although the time gap will be minimal if the graph is very dense, the fact remains that using an adjacency list always results in a shorter running time, making it the optimal choice.

3. APPLICATIONS AND EXPERIMENTS

Problem 1. Dijkstra

In this case, distance is the only limiting factor for paths. While finding a path from the starting point to the destination, the path with the shortest distance is to be found. The inputs for this problem will be the weight of the distance of the edge for each pair of connected vertices. Each time an edge is crossed and a vertex is reached, the weight will be added to the current distance to that vertex. The current distance to a vertex will always be the shortest one and will be updated accordingly, depending on novel paths to the vertex.

In order to test the efficiency of the two different types of graph storage method - matrix and list - as well as Dijkstra algorithms - "for" loop and priority queue - in terms of running time, each method was executed onto either a sparse or a dense graph. A subway map was used as an example of a sparse graph, and a complete graph with 3000 vertices was used for the dense graph.[3] Below are the running times ofe ach method.

	Adjacency Matrix		Adjacency List	
	"For" Loop	Priority Queue	"For" Loop	Priority Queue
Sparse Graph	3.238	2.456	3.398	2.245
Dense Graph	3.577	6.493	7.930	12.39

(units: seconds)

Table 1: Time Comparison between adjacency matrix/list and graph types

As can be seen in Table 1, the priority queue method was comparatively efficient in the case of a sparse graph, but the opposite is apparent in the case of a dense graph. This is a predictable result. Since the sparser the graph is, the more accurate is the approximation, V≈E, and thus the time complexity of O(ElogE) is better than that of O(V^2+E) (faster results, therefore smaller time complexity, is considered "better").

$$(V^2+E)\text{-}(ElogE)\approx(E^2+E)(ElogE)=E^2+E(1\text{-}logE)>0$$

since the domain of E is [0, ∞), all integers.

The four versions of codes that were used are included in the Appendix.

```
Algorithm Problem1
create priority queue Q
for each vertex v in Graph:
        dist[v] ← INFINITY

dist[source] ← 0
insert source to Q
while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u:
                cand ← dist[u] + length(u, v)
                if cand < dist[v]:
                        dist[v] ← cand
                        insert v to Q
```

Table 2: Pseudocode for Problem 1

Problems 2. Limited Money

Realistically, travelers possess a limited amount of money that they can use for transportation. Thus, we add a second factor to the algorithm, which is cost. While finding a path from the starting point to the destination, the amount of remaining money must always be greater than or equal to zero. In addition, the distance should be minimal while maintaining a cost lower than the threshold.

In addition to the initial weight of distance in the edges, a second weight of the cost for traversing each edge will be appended to the adjacency list. There will also be an additional required input of the initial cost that the person starts with at the starting point. The algorithm will be set so that each time an edge is crossed, the respective cost will be subtracted from the remaining cost at the current vertex, and a path will be deleted if the remaining cost becomes a negative value.

Since there are 2 limiting factors, although it does mean that there can be multiple stored paths at each vertex, this doesn't mean that all paths with non-negative remaining costs will be stored as valid. At each vertex, certain paths will be obviously "better" regardless of subsequent movements. If, for one path, the remaining cost is lower and the distance is greater than another at the same vertex, the former path will be deleted. The same goes for when one of the conditions remains true and the values are the same for the other. Otherwise, all paths will be stored as valid.

```
Algorithm Problem2
create priority queue Q
for each vertex v in Graph:
        for each integer m in 0~Money:
                dist[v][m] ← INFINITY

dist[source][0] ← 0
insert (source, 0) to Q
while Q is not empty:
        (u, t) ← vertex in Q with min dist[u]
        remove (u, t) from Q

        for each neighbor v of u:
                cand ← dist[u][t] + length(u, v)
                money ← t + cost(u, v)
                if cand < dist[v][money]:
                        dist[v][money] ← cand
                        insert (v, money) to Q
```

Table 3: Pseudocode for Problem 2

Problems 3. Minimal Transitions

The number of transitions that are made between subway lines, bus lines, or modes of transportation is taken into consideration in this case. This time, a second limiting factor of frequency of transitions of a path is added. While finding a path from the starting point to the destination, the algorithm must maintain a minimal number of transitions.

For this case, there will be a second required input of the "type" of each edge in addition to the distance. The type of the edge will be an indicator of transition because if the type of the previous edge is different from the subsequent edge in respect to the vertex, it means that a transition was made. However, there will not be two methods of storage for the paths because the number of transitions will not be stored separately from the distance of the path. Instead, the algorithm will be structured so that a transition is extremely harmful to the path. Each time a transition is made in a path, an extremely large number, one that is much greater than the weights of the edges will be added to the current distance. As a result, a path with a smaller number of transitions will always be superior to the one with more transitions.

However, these paths will not necessarily be deleted. Sometimes, transitions will be necessary to get from the starting point to the destination. In this case, the extremely large number will have to be added to all paths, resulting in comparison among these large values. Transitions can be thought of as levels, because paths will always be compared among those with the same number of transitions.

```
Algorithm Problem3
create priority queue Q
for each vertex v in Graph:
        dist[v] ← INFINITY

dist[source] ← 0
insert source to Q
while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u:
                cand ← dist[u] + length(u, v)
                if v is different line:
                        cand ← cand + 1000000
                if cand < dist[v]:
                        dist[v] ← cand
                        insert v to Q
```

Table 4: Pseudocode for Problem 3

The algorithm was tested on an actual subway map of seoul to determine optimality between a for-loop and a priority queue, as well as a matrix and an adjacency list. JSON was utilized to extract information from a data file containing the information for the lines and stops of the subway system in Seoul. Once a graph was constructed using JSON, either as a matrix or an adjacency list, the algorithm was executed to compare the running time between the different methods. Specifically, the algorithm was tested on several examples of routes between one station and another, such as Apgujeong Station to Seoul Forest Station. As can be seen, it displays the route that takes the minimum amount of transitions.

Figure 4: Seoul subway map [4]

Problems 4. Traffic lights

A major influential factor in travelling is the periodical delay that results from the changing of traffic lights. Due to traffic lights, an edge can be passed through during a certain portion of a period, the period being the length of a cycle from the start of a green light to the end of a green light. If an edge connecting two vertices is not passable at a certain moment, there will be a waiting time that adds on to the inherent weight of the edge. As a result, the algorithm takes into account the waiting times at given moments, influencing the path that it creates. An edge that is shorter than another may be deemed longer if there is a waiting time that is present.

An important assumption is made in running the algorithm, which is, upon the start of the path, every traffic light starts at a green light. However, it is important to note that this assumption does not influence the algorithm's accuracy when the assumption does not hold true, because if it is able to perform normally in subsequent steps, it means that it is able to perform normally in non-uniform traffic-lights. The assumption is made simply for convenience when constructing the algorithm.

```
Algorithm Problem4()
create priority queue Q
for each vertex v in Graph:
        dist[v] ← INFINITY

dist[source] ← 0
insert source to Q
while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u:
                cand ← dist[u] + length(u, v)
                cand ← wait until green light
                if cand < dist[v]:
                        dist[v] ← cand
                        insert v to Q
```

Figure 4: Seoul subway map [4]

Problems 5. Congestion Updates

Traffic situations are subject to change from the moment a path is determined due to fluctuating congestion on the while traveling. A traffic accident that occurs may lead to a drastic increase in the time it takes to get through a road. The path should be continuously updated based upon the changing values of the edges leading to unvisited vertices. Input of new weights of the edges can potentially produce a path different from the current one.

The algorithm will not terminate after it is run at the starting point; it will continue to perform reruns based upon new inputs in weights of edges. If there are no modifications made to the weights, then the path is unchanged. However, if the weight of an edge is changed, the Dijkstra Algorithm is run again, with the current vertex as the new starting point and the same destination as before. Thus, based upon the change in the traffic situations, a new optimal path will be formed. In order to accomplish this, the Dijkstra Algorithm itself will be contained inside a "while loop."

This method presents some limitations to the functionality of the program. First of all, there is a loss of predictability because it is impossible to determine when the values of the weights will change. In addition, since a car cannot turn around in the middle of the road, if a new path is calculated, it will have to travel to the point it was initially going towards before a new path can be implemented.

Problems 6. Personalized Path

The users should be able to personalize the path finder. People prefer certain types of roads over others, such as scenic roads along the coastline rather than roads that go through alleyways, or travel by bus rather than transit by subway. The path should be able to fit itself to the optimalities so that preferred roads are maximized. Simply put, based on the input of optimal types of roads, optimal paths should change.

Since it is possible for a path to grow interminably long if it chooses to prioritize going through optimal types of roads, an upper bound is set before a path is calculated by running the normal Dijkstra Algorithm. This shortest path will be multiplied by a factor of 1.25 and set as the upper limit. As a result, paths will be deleted as soon as their distance goes over more than 25% of the minimal possible distance.

Within this limitation, types of roads will be "ranked" based upon the input of the user so that a smaller number indicates a greater preference. The preference score will be set as a second parameter that will be minimized during the construction of the path. If it were the case that bigger scores represented greater preference and the purpose had been to maximize the preference score, constructing an algorithm would not nearly be possible because maximizing distance of a path is an unsolved NP-Hard Problem. [5]

It is important to note that a problem occurs when the method of adding preference scores is recruited, because it can cause unpreferred roads to be chosen over preferred roads in some cases where the unpreferred path actually ends up with a lower preference score. For example, let's say road type 1 has a score of 1 and road type 2 has a score of 2. In getting from point A to B, there is a path that only uses type 1 roads and another that only uses type 2 roads. However, there are 8 short type 1 roads compared to just 3 long type 2 roads. Therefore, even if the distance is the same and the former should obviously be the intended result, the type 2

roads are chosen because they have a combined score of 6, which is lower than the score of 8.

In response to this issue, a solution is to multiply the distance by the preference score. As a result, there is a weight that is added to the score themselves. Longer distances will cover for the fact that less roads need to be taken. For the depicted example, the type 2 roads will now have a score is much higher, because each of the roads are longer.As a result of the algorithm, the user will be presented with several options that fit the upper limit and maximize preferred roads.

4. CONCLUSION

Using for-loop and priority queue variations of the Dijkstra Algorithm, as well as manipulating the density of the graph through numerous examples, I was able to inspect the most efficient combination of methods for each case. As predicted through theoretical calculations of time complexity, it was discovered that the denser the graph, the more efficient a for-loop method was; vice versa, the sparser the graph, the more efficient a priority queue method was.

In addition, I modified the Dijkstra's Algorithm to create an algorithm that fits the needs of each path problem. Using such algorithms, I was able to confirm full functionality for each example.

Finally, if I had the chance, I think it would be very interesting to delve deeper into the real-time updates pertinent to problem 5. As of now, every time an update is needed, the Dijkstra's Algorithm is rerun. It would be a noteworthy accomplishment to discover a way to decrease the time complexity and I anticipate that such a method could be applied in many other problems.

5. REFERENCES

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), Introduction To Algorithms (2nd ed.), MIT Press, p. 599-602

[2] Graph representation, CS Academy
URL https://csacademy.com/lesson/graph_representation/

[3] Seoul Metro Subway map data of.
URL https://data.seoul.go.kr/dataList/OA-15442/S/1 datasetView.do

[4] Seoul Subway Map
URL http://seoulsublet.com/subway-metro-map/

[5] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), Introduction To Algorithms (2nd ed.), MIT Press, p. 978

6. APPENDIX

5.1. adjacency list + for loop

```
from heapq import *

INF = 10**10
vertexNum, edgeNum = input().split()
vertexNum, edgeNum = int(vertexNum), int(edgeNum)

### Input ########################################
# adjacency list:
# vertex id should be an integer between 1 ~ V
##################################################
adj = [[] for _ in range(vertexNum+1)]
for i in range(edgeNum):
    a, b, w = [int(i) for i in input().split()]

    adj[a].append((b, w))
    adj[b].append((a, w))

startVertex = int(input())

### Dijkstra's algorithm ########################
# Using adjacency list + for loop
##################################################

dist = [INF] * (vertexNum+1)
visited = [False] * (vertexNum+1)

dist[startVertex] = 0


while True:
    curDist = INF

    for v in range(vertexNum):
        if not visited[v] and dist[v] < curDist:
            cur = v
            curDist = dist[v]

    if curDist == INF:
        break

    visited[cur] = True
    for nxt, w in adj[cur]:
        nxtDist = curDist + w
        if nxtDist < dist[nxt]:
            dist[nxt] = nxtDist
```

5.2. adjacency list + priority queue

```
from heapq import *

INF = 10**10
vertexNum, edgeNum = input().split()
vertexNum, edgeNum = int(vertexNum), int(edgeNum)

### Input ########################################
# adjacency list:
# vertex id should be an integer between 1 ~ V
##################################################
adj = [[] for _ in range(vertexNum+1)]
for i in range(edgeNum):
    a, b, w = [int(i) for i in input().split()]

    adj[a].append((b, w))
    adj[b].append((a, w))

startVertex = int(input())

### Dijkstra's algorithm ########################
# Using priority queue + adjacency list
##################################################
```

```
dist = [INF] * (vertexNum+1)
pq = []

dist[startVertex] = 0
heappush(pq, (0, startVertex))

while pq:
    curDist, cur = heappop(pq)

    if curDist > dist[cur]:
        continue

    for nxt, w in adj[cur]:
        nxtDist = curDist + w
        if nxtDist < dist[nxt]:
            dist[nxt] = nxtDist
            heappush(pq, (nxtDist, nxt))
```

5.3. adjacency matrix + for loop

```
from heapq import *

INF = 10**10
vertexNum, edgeNum = input().split()
vertexNum, edgeNum = int(vertexNum), int(edgeNum)

### Input ######################################
# adjacency matrix:
# vertex id should be an integer between 1 ~ V
################################################
adj = [[INF]*(vertexNum+1) for _ in range(vertexNum+1)]
for i in range(edgeNum):
    a, b, w = [int(i) for i in input().split()]

    adj[a][b] = adj[b][a] = w

startVertex = int(input())

### Dijkstra's algorithm #######################
# Using adjacency matrix + for loop
################################################

dist = [INF] * (vertexNum+1)
visited = [False] * (vertexNum+1)

dist[startVertex] = 0


while True:
    curDist = INF

    for v in range(vertexNum):
        if not visited[v] and dist[v] < curDist:
            cur = v
            curDist = dist[v]

    if curDist == INF:
        break

    visited[cur] = True
    for nxt in range(1, vertexNum+1):
        w = adj[cur][nxt]
        nxtDist = curDist + w
        if nxtDist < dist[nxt]:
            dist[nxt] = nxtDist
```

5.4. adjacency matrix + priority queue

```
from heapq import *

INF = 10**10
vertexNum, edgeNum = input().split()
vertexNum, edgeNum = int(vertexNum), int(edgeNum)

### Input ######################################
# adjacency matrix:
# vertex id should be an integer between 1 ~ V
################################################
adj = [[INF]*(vertexNum+1) for _ in range(vertexNum+1)]
for i in range(edgeNum):
    a, b, w = [int(i) for i in input().split()]

    adj[a][b] = adj[b][a] = w

startVertex = int(input())

### Dijkstra's algorithm #######################
# Using priority queue + adjacency matrix
################################################

dist = [INF] * (vertexNum+1)
pq = []

dist[startVertex] = 0
heappush(pq, (0, startVertex))

while pq:
    curDist, cur = heappop(pq)

    if curDist > dist[cur]:
        continue

    for nxt in range(1, vertexNum+1):
        w = adj[cur][nxt]
        nxtDist = curDist + w
        if nxtDist < dist[nxt]:
            dist[nxt] = nxtDist
            heappush(pq, (nxtDist, nxt))
```