MULTIMEDIA UNIVERSITY

TDA 3231: ALGORITHM DESIGN AND ANALYSIS

APPLICATIONS OF DIJKSTRA'S ALGORITHM

****

Trimester 2, 2018/2019

LECTURER : DR. CHEAH WOOI PING

GROUP : DA1C

NAME	STUDENT ID	SIGNATURE
Phuah Chee Woei	1141128570	
Nur Rufaidah Binti Omar	1151103909	
Lee Bing Jun	1141124191	

Table of Content

Background

Dijkstra thought about the shortest path problem when working at the Mathematical Center in Amsterdam in 1956 as a programmer to demonstrate the capabilities of a new computer called ARMAC (Knuth, 1977). His objective was to choose both a problem and a solution (that would be produced by computer) that non-computing people could understand. He designed the shortest path algorithm and later implemented it in ARMAC for a slightly simplified transportation map of 64 cities in the Netherlands. A year later, he came across another problem from hardware engineers working on the institute's next computer to minimize the amount of wire needed to connect the pins on the back panel of the machine. As a solution, he re-discovered the algorithm known as Prim's minimal spanning tree algorithm (known earlier to Jarník, and also rediscovered by Prim). Dijkstra published the algorithm in 1959, two years after Prim and 29 years after Jarník.

Introduction

Dijkstra’s "label algorithm" which was proposed in 1959 is one of the best shortest path algorithms. "Label algorithm" has a very wide range of applications, such as multi-point routing, surveying and mapping science, the shortest path of logistics and transport, the intelligent transportation system, the expressway network toll collection, and so on (Shu-Xi, 2012). Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. Dijkstra’s algorithm is also known as a single source shortest path algorithm. It is applied only on positive weights (Dcsa & Chhillar, 2014). It perform with more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

The Dijkstra algorithm uses labels that are positive integer or real numbers, which have the strict weak ordering defined. Interestingly, Dijkstra can be generalized to use labels defined in any way, provided they have the strict partial order defined, and provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the Generic Dijkstra shortest-path algorithm (Szcze, Jajszczyk, & Wo, 2018).

In other fields like artificial intelligence in particular, Dijkstra's algorithm or a variant of it is known as uniform cost search and formulated as an instance of the more general idea of best-first search. The two algorithms have many similarities and are logically equivalent. The most important similarity is that they expand exactly the same nodes and in exactly the same order. It is important to note that Dijkstra's algorithm in its basic form as usually is designed to find the shortest paths to all vertices of the graph (known as the single-source shortest path problem). However, with a small modification, it can be used to only find a shortest path to any given goal (a source target shortest-path problem) or to any set of goals. (Felner, 2011).

Problem Statement

Path finding generally refers to finding the shortest path between any two locations. Many existing algorithms are designed precisely to solve the shortest path problem such as Genetic, Floyd algorithm.

Dijkstra’s algorithm came to be because of the problem that exists with calculating the most efficient path from one point to another. It first started as a simple question as to how to traverse from a starting point to the destination point given that multiple points or nodes act as a middlemen between the path. The idea is to calculate the most efficient way to traverse the multiple nodes, with the idea of keeping the accumulated distance to a minimum in mind. Situations relating to this problem may be related to optimization of mapping systems, network architecture designs, and all other problems relating to distance between two points with multiple intermediaries in between.

The method proposed in this project is Dijkstra’s algorithm, which will find the shortest path solution in time efficiency and shortest distance. It also a variant of informed search, as the location of the starting and ending points are taken into account before it is widely used for finding the shortest path. The algorithm is a refinement of the shortest path algorithm that directs the search towards the desired goal. Dijkstra’s algorithm also is a simple and excellent method for path planning. It chooses one with the minimum cost and until the goal is found, the search will continue as it calculates all possible paths from the starting node to the goal, then choose the best solution by comparing the minimum distance of every permutation. It remains relevant because it is realistically fast and relatively easy to implement.

Project Objective

For the aim of this project to be achieved, the following objectives should be fulfilled:

- Finding the shortest path between nodes in a graph using Dijkstra’s algorithm
- To choose both a problem and a solution that non-computing people could understand by using a concept of algorithm
- To demonstrate the importance of Dijkstra’s algorithm

Project Scope

The project will focus only to demonstrate Dijkstra’s algorithm based on time to search the shortest path by calculating the minimum distances to get from one node to another given the starting point or node and also the ending node, and represent the result graphically by implementing the algorithm in Python.

Literature Review

In the intelligent transportation system, the calculation of the shortest path and the best path is an important link of the vehicle navigation function. Due to more and more real-time information to participate in the calculation, the calculation requires high efficiency for the algorithm. (Chuang Ruan, Jianping Luo, & Yu Wu, 2014) proposed a navigation system based on vehicle terminal, designed the overall framework of the system and gave the function of each module. The system can provide optimal service path for the user's choice and needs. The optimization of the design and application of Dijkstra algorithm saved storage space in the process of system operation, improved the accuracy of the navigation path, reduced the complexity of the Dijkstra algorithm and greatly improved the efficiency of the system.

(Galán-García, Aguilera-Venegas, Galán-García, & Rodríguez-Cielos, 2015) presented a new algorithm PEDA (Probabilistic Extension of Dijkstra's Algorithm) which introduces probabilistic changes in the weight of the edges and also in the decisions when choosing the shortest path. When PEDA is applied to traffic flow, more realistic simulations, in which the shortest path is not always chosen, are obtained. This more accurately simulates the more normal behavior of drivers. As an example of an application, an accelerated-time simulation of car traffic in a smart city was described. Through the application will produces more realistic simulations considering different drivers' behaviors.

(Xu, Wen, & Zhang, 2015) proposed an Indoor optimal path planning based on Dijkstra Algorithm. The optimal path planning will providing optimal path information and the personal navigation path is the open issue for the pedestrian guided system. In order to solve the problems existing in the pedestrian guided system nowadays, such as the jagged paths which probable exist in the shortest path, and the lack of the considerations of different users' preferences generated by collecting and analyzing users' behavioral information, then optimize the paths in advance.

(Deng, Chen, Zhang, & Mahadevan, 2012) proposed a graded mean integration representation of fuzzy numbers to improve the classical Dijkstra algorithm based on the canonical representation of operations on triangular fuzzy numbers to handle the fuzzy shortest path problem. Compared with existing methods, the proposed method is more efficient due to the fact that the summing operation and the ranking of fuzzy numbers can be done is an easy and straight manner. The proposed method can be applied to real applications in transportation systems, logistics management, and many other network optimization problem that can be formulated as shortest path problem.

In emergency situations, finding suitable routes to reach destination is critical issue. The shortest path problem is one of the well-known and practical problems in computer science, networking and other areas. (Kai, Yao-Ting, & Yue-Peng, 2014) present an overview on shortest path analysis for an effective emergency response mechanism to minimize hazardous events. The proposed method are based on the integration of Geographic Information System (GIS) and also Dijkstra algorithm, web services and Asynchronous JavaScript and XML (Ajax) technologies to provide a web application for finding optimal routes from locations of specialized response team stations to incidents site so as to maximize their ability to respond to hazard incidents.

Methodology

Theory of Dijkstra's Algorithm

In some applications, it is useful to model data as a graph with weighted edges. These graphs are called "weighted graphs". Let's imagine that each node is a city, and each edge is an existing road between two cities. This means that you can drive from A to B directly.

However, you can't drive from A to D directly, as there's no road between those cities; instead, for example, you need to go from A to B and then from B to D. We may represent that time with weights that we assign to the roads (edges of the graph). In the example, let's assume that each number represent the amount of time in hours that it takes you to take a road. This would mean that going from A to B takes 3 hours if you use the road that connects them directly.

Dijkstra's Algorithm allows the calculation of the shortest path between one node and to another node in the graph. For example, calculate the shortest path between node C and the other nodes in a graph:

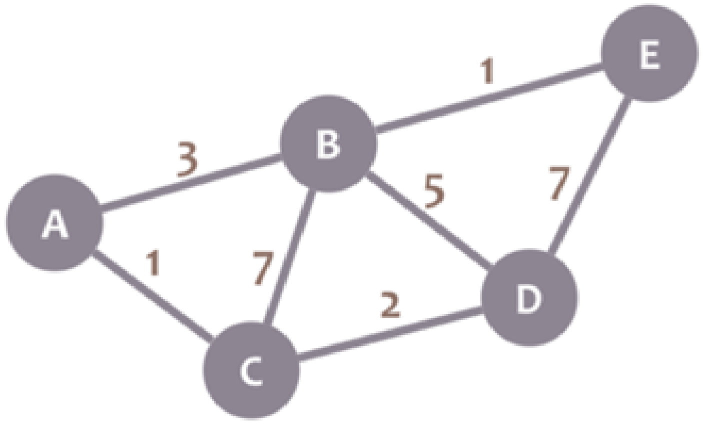


Figure 1

During the algorithm execution, mark every node with its minimum distance to node C (selected node). For node C, this distance is 0. For the rest of nodes, still don't know that minimum distance, it starts being infinity (∞):

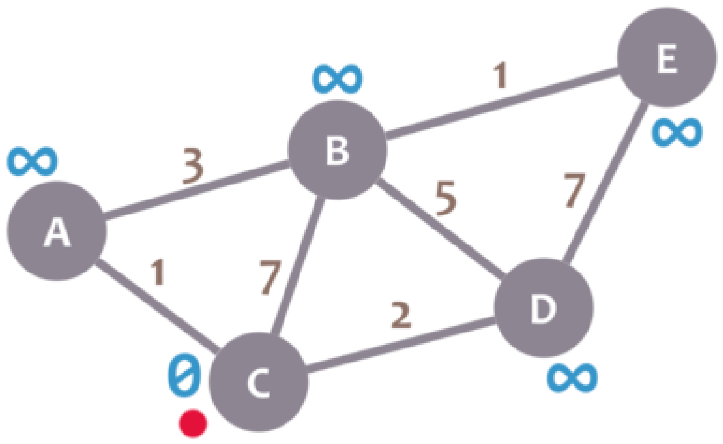


Figure 2

Then a current node need to preset. Initially, set it to C (our selected node). In Figure 2, mark the current node with a dot.

Now, check the neighbours of current node (A, B and D) in no specific order then begin with B and add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects current node with B (in this case, 7), so it obtain 0 + 7 = 7. Compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):

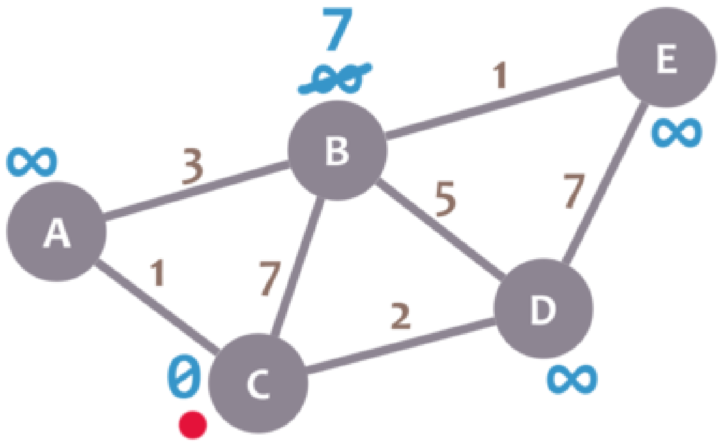


Figure 3

Now, check neighbour A with add 0 (the minimum distance of C, the current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. Compare that 1 with the minimum distance of A (infinity), and leave the smallest value:

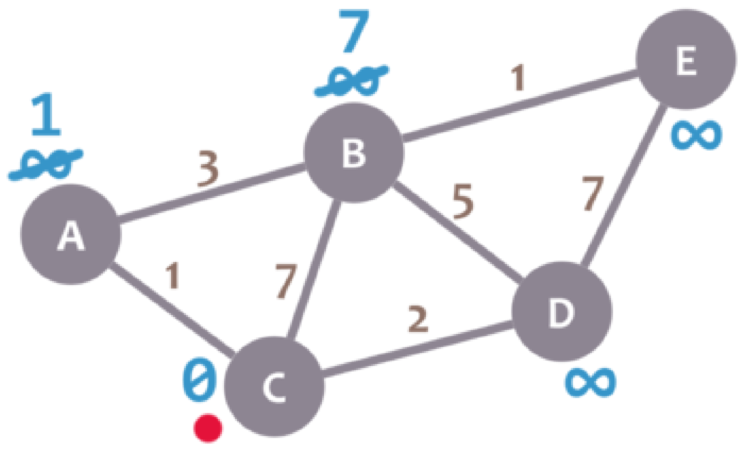


Figure 4

Repeat the same procedure for D:

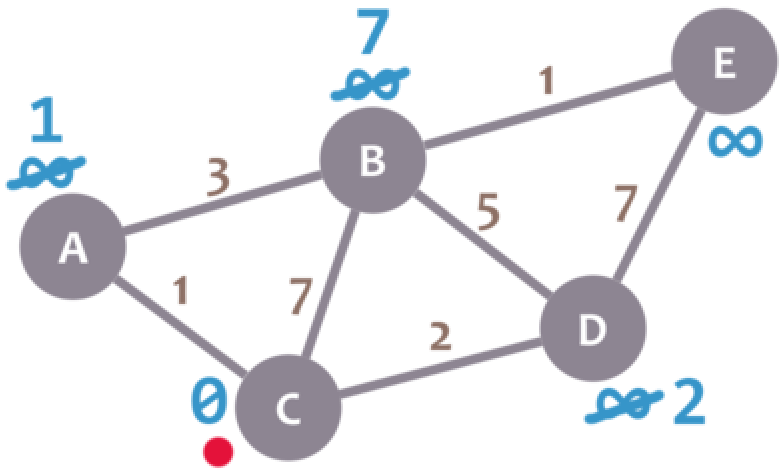


Figure 5

If have checked all the neighbours of C. Then mark it as visited and represent visited nodes with a check mark:

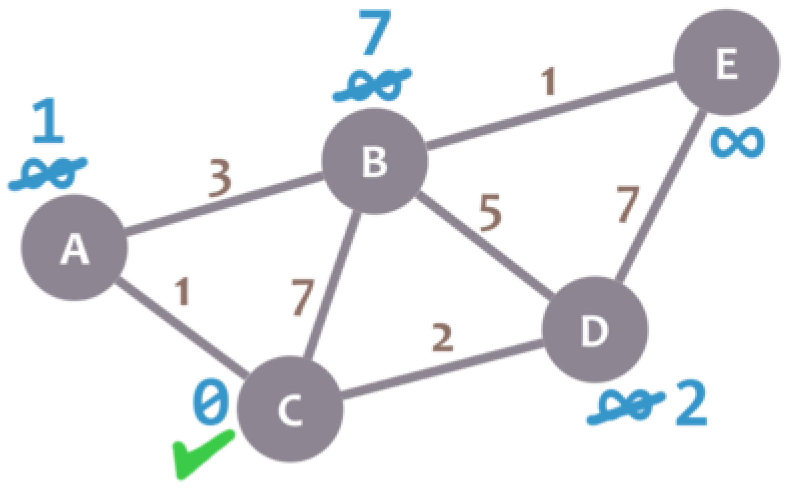


Figure 6

After that, need to pick a new current node. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A and mark it with the red dot:

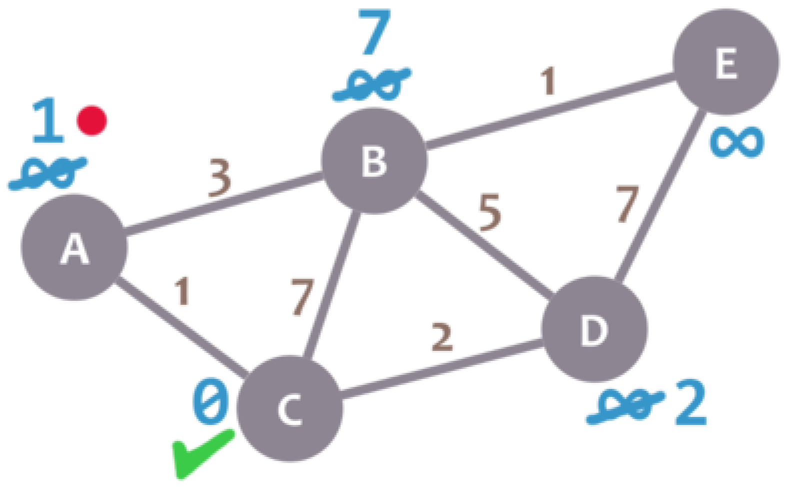


Figure 7

And now repeat the algorithm with check the neighbours of our current node, ignoring the visited nodes. This means only check B.

For B, add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. Compare that 4 with the minimum distance of B (7) and leave the smallest value as 4.

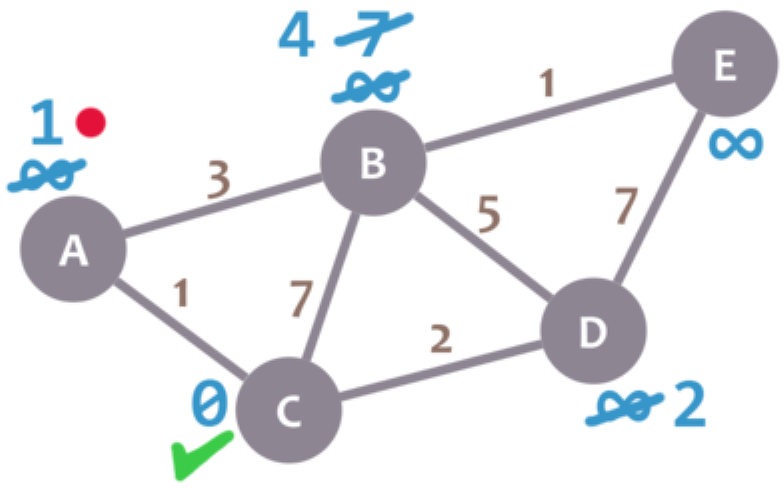


Figure 8

Afterwards, mark A as visited and pick a new current node as D, which is the non-visited node with the smallest current distance.

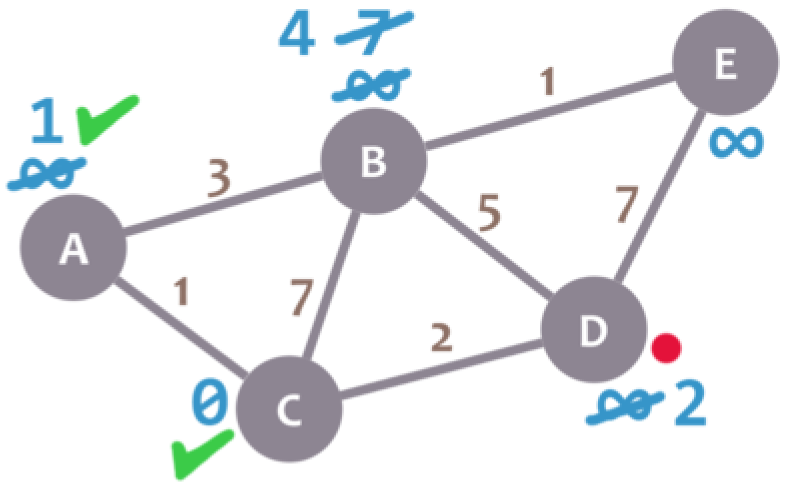


Figure 9

Repeat the algorithm again. This time, check B and E.

For B, it obtain 2 + 5 = 7. Compare that value with B's minimum distance (4) and leave the smallest value (4). For E, it obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

Then mark D as visited and set the current node to B.

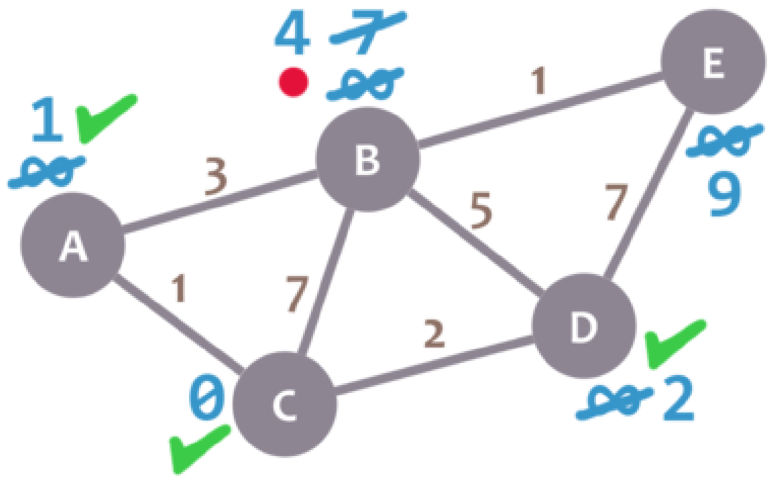


Figure 10

After that, only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, mark B as visited and set E as the current node.

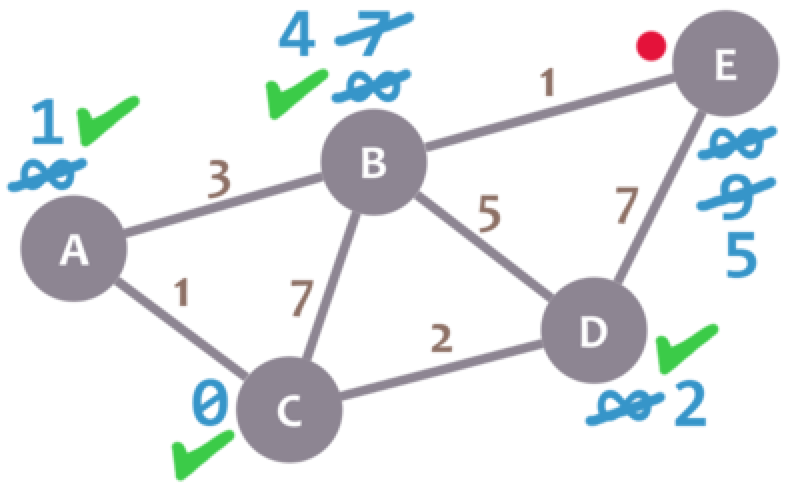


Figure 11

E doesn't have any non-visited neighbours, so don't need to check anything and mark it as visited.

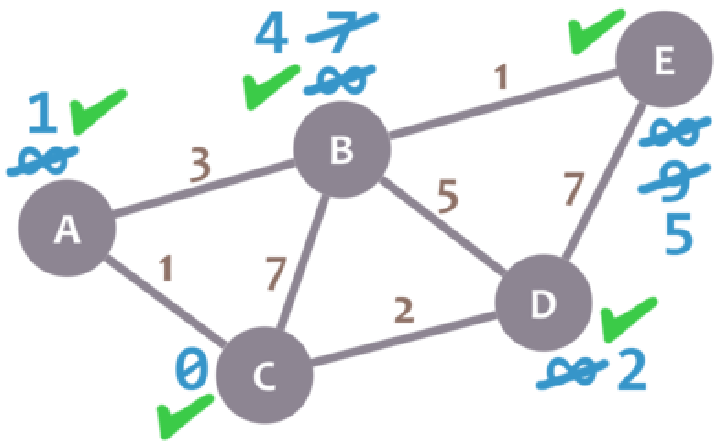


Figure 12

As there are not univisited nodes, The minimum distance of each node and now actually represents the minimum distance from that node to node C (the node picked as our initial node).

Pseudo-code of Disjkstra algorithm

```
procedure Dijkstra's(G)                       //Weighted graph,with all weights positive
    dist[s] ← 0                               //initially distance to source vertex is Zero
    for all v ∈ V − {s} do
        dist[v] ← ∞                           //set all other distances to infinity
    end for
    S ← ø                                     //S, the set of visited vertices is initially empty
    Q ← V                                     //Q, the queue initially contains all vertices
    while Q 6= ø do                           //while the queue is not empty
        u ← mindistance(Q, dist)              //select element of Q with the min.distance
        S ← S ∪ {u}                           //add u to list of visited vertices
        for all v ∈ neighbors[u] do
            if dist[v] > dist[u] + w(u, v) then          //if new shortest path found
            d(v) ← d(u) + w(u, v)                        //set new value of shortest path
            end if
        end for                               //if desired,add traceback code
    end while
    return dist
end procedure
```

Implementation

In this section, the discussion of how the implementation of Dijkstra's Algorithm will take place. The programming language of choice is the Python programming language, version 3.6.6 64-bit. The reason Python was chosen is because the ease of use, flexibility and understandability of the language. The entire built of it all is pretty simple, consisting of only two files. The dijkstras-algov1.py file, the first file, is used to calculate the shortest path needed using a Python implementation of Dijkstra's Algorithm. The graphFunctions.py file is used to automate the entire process in obtaining the relevant vertices and edges needed to plot the shortest path as well as the graph specified. Before going into the main function, all the other functions are called will be looked into first.

Code

graphFunctions.py

Set up the project

Import the necessary libraries

```
from collections import namedtuple, deque
from pprint import pprint as pp
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
```

The above code is to import all the libraries needed to execute the functionalities needed for the calculations of the shortest path, processing the data to be suitable for the graph, and drawing the graph.

Set the path to the GraphViz Layout

```
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/'
```

This step is not necessary, as it only enables the usage of a certain layout, the graphviz layout, which is replaceable by the spring layout. However, it should be kept in mind that the graphviz layout is the best one to plot out graphs with minimal overlaps.

Define the variables

```
inf = float('inf')
Edge = namedtuple('Edge', 'start, end, cost')
```

A process to define the variables before the execution of the code, where inf is of type float, and Edge is a tuple of size 2.

Define the class Graph

Specify the built-in initialise function

```
class Graph():
    def __init__(self, edges):
        self.edges = edges2 = [Edge(*edge) for edge in edges]
        self.vertices = set(sum(([e.start, e.end] for e in edges2), []))
```

We initialise the class Graph to accept the edges and vertices that have should be passed in a list to create a Graph object.

Define the Dijkstra's Algorithm function

```
    def dijkstra(self, source, dest):
        assert source in self.vertices
        dist = {vertex: inf for vertex in self.vertices}
        previous = {vertex: None for vertex in self.vertices}
        dist[source] = 0
        q = self.vertices.copy()
        neighbours = {vertex: set() for vertex in self.vertices}
        for start, end, cost in self.edges:
            neighbours[start].add((end, cost))
        print('NEIGHBOURS')
        pp(neighbours)

        while q:
           u = min(q, key=lambda vertex: dist[vertex])
           q.remove(u)
           if dist[u] == inf or u == dest:
               break
           for v, cost in neighbours[u]:
               alt = dist[u] + cost
               if alt < dist[v]:                            # Relax (u,v,a)
                   dist[v] = alt
                   previous[v] = u
        print('SMALLEST VALUE')
        pp(previous)
        s, u = deque(), dest
        while previous[u]:
            s.appendleft(u)
            u = previous[u]
        s.appendleft(u)
        return s
```

This function, which is a part of the class Graph, will calculate the shortest path needed from the start node to the destination node. It calculates the total current distance explored and compares with the previous one, with the smaller distance being chosen as part of the shortest path.

Define the function to draw the graph

Create a directed graph

```
def drawGraph(graphItem, totalNodes, path, startNode, endNode, weight):
    G = nx.DiGraph()
```

The drawGraph function takes in a total of 6 parameters, which are the information related to the edges, total vertices, shortest path, starting node, ending node, and the weight of each edge of the graph. A directed graph is then initialised.

Define empty lists for the path

```
    pStartNode = []
    pEndNode = []
    pWeights = []
```

This is to hold the values of the shortest path, for later use.

Generate the nodes both sides

```
    for idx, val in enumerate(path):
        pEndNode.append(val)
        pStartNode.append(val)
        if idx == 0:
            pEndNode.pop()
    pStartNode.pop()
```

Append the start nodes and end nodes to allow the plotting of the graph through the shortest path both ways. This is so that plotting the graph from node n to node m, and node m to node n possible.

Combine the start and end nodes

```
    pathEdges = list(zip(pStartNode, pEndNode))
```

Combine all the start and end nodes into two lists.

Obtain path weights

```
for item in graphItem:
    for pathEdge in pathEdges:
        if (pathEdge[0] == item[0] and pathEdge[1] == item[1]):
            pWeights.append(item[2])
```

Compare the weights of the path and the entire graph to obtain the shortest path's weights.

Obtain the path's total weight

```
totalWeight = 0
for pWeight in pWeights:
    totalWeight += pWeight
print('Total weight:', totalWeight)
```

Iterate through the entire path to obtain the total weight of the path.

Set the logic for black edges

```
blackEdges = [edge for edge in G.edges() if edge not in pathEdges]
```

This is to be used later, when plotting the graph, to draw black edges on the graph if the edge is not part of the shortest path.

Add the edges

```
for i in range(totalNodes):
    G.add_edge(startNode[i], endNode[i], weight=weight[i])
```

Define a for loop to programmatically add all the edges into the directed graph defined before, with the starting node, ending node, and weight of each edge.

Keep the edge labels

```
edge_labels = dict([((u, v,), d['weight']) for u, v, d in G.edges(data=True)])
```

This particular line of code will keep the edge labels, which are the start nodes, end nodes, and weights in a dictionary, which will be used to draw the graph later.

Specify the positioning of the graph

```
pos = nx.nx_pydot.graphviz_layout(G, prog='neato')
```

As discussed earlier, the algorithmic positioning of the graph can be defined here. Instead of using the graphviz layout, the spring layout can also be used. Look into the documentation in the code for more.

Draw the graph

```
    nx.draw(G, pos, edge_color='black', width=1, linewidths=1, node_size=500,
        node_color='pink', labels={node:node for node in G.nodes()}, arrows=False)
```

Draw the graph G, with the algorithmic positioning pos, with a line width of 1, a node colour of pink, labels that have been defined in the nodes, and no arrows.

Draw the labels

```
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
        font_color='black')
```

This line plots the labels that have been previously defined in edge labels in the font colour black.

Draw the edges

```
    nx.draw_networkx_edges(G, pos, edgelist=pathEdges, edge_color='red',
            arrows=True, arrowsize=24)
    nx.draw_networkx_edges(G, pos, edgelist=blackEdges, arrows=False)
```

Draw the shortest path determined by the algorithm with the edge colour red, and arrows of size 24. Draw the entire graph after that with black edges and no arrows.

Show the graph

```
    showGraph()
```

Show the graph.

Define the swapNodes function

Extend the start and end nodes

```
def swapNodes(sStartNode, sEndNode, weight):
    startNodes = []
    endNodes = []

    startNodes.extend(sStartNode)
    endNodes.extend(sEndNode)
    startNodes.extend(sEndNode)
    endNodes.extend(sStartNode)
    weight += weight

    return startNodes, endNodes, weight
```

By adding new lists startNodes and endNodes, we swap the contents of the variables, and then extend them so that both the start nodes and end nodes will contain each other’s information.

Define the show graph function

Show the graph

```
def showGraph():
    plt.axis('off')
    plt.show()
```

Remove the axis that is usually shown in the window, and show the graph.

dijkstras-algo.py

Set up the project

Import the files

```
from graphFunctions import Graph, namedtuple, drawGraph, swapNodes
```

Import the functions that we have defined in graphFunctions.

Input the values

Specify the start and end nodes

```
start_node = 'a'
dest_node = 'h'
```

Assuming we have nodes spanning from ‘a’ to ‘h’, we specify the start and end node as above.

Plot the graph

```
graphItem = [('a', 'b', 9),  ('a', 'f', 14),  ('a', 'g', 14), ('b', 'c', 23),
                ('c', 'f', 18), ('c', 'd', 6), ('c', 'h', 19), ('c', 'e', 2),
                ('d', 'e', 11), ('d', 'h', 6), ('e', 'f', 30), ('e', 'g', 20),
                ('e', 'h', 16), ('f', 'g', 5), ('g', 'h', 44)]
```

Plot out the graph with the parameters (‘startnode’, ‘endnode’, ‘weight’). It is necessary that the graph is correctly plotted to have the algorithm work properly.

Define the main function

Gather information

```
def main():
    totalNodes = len(graphItem) * 2
    startNode = [item[0] for item in graphItem]
    endNode = [item[1] for item in graphItem]
    weight = [item[2] for item in graphItem]
```

Save the information from the graph into separate variables for easy processing later.

Pre-process the graph values

```
extendedNodes = swapNodes(startNode, endNode, weight)
newGraphItem = list(zip(extendedNodes[0], extendedNodes[1], extendedNodes[2]))
```

Call on the swapNodes function to obtain the two way edges of the graph, then save it as a list inside the newGraphItem variable.

Save the values in other variables

```
newStartNode = extendedNodes[0]
newEndNode = extendedNodes[1]
newWeight = extendedNodes[2]
```

Save into the new variables to be passed onto the graph function later.

Calculate the shortest path

```
graph = Graph(newGraphItem)
path = graph.dijkstra(start_node, dest_node)
print('Shortest path:', path)
```

Create a new graph with the function defined previously in graphFunctions.py, and calculate the shortest path using Dijkstra's Algorithm. Print the shortest path.

Draw the graph

```
drawGraph(newGraphItem, totalNodes, path, newStartNode, newEndNode, newWeight)
```

Call the drawGraph function and pass the values relevant to draw the graph.

Big-O Notation

The running time of this particular algorithm is decided by its number of edges and vertices. In the worst case scenario, where the number of edges outnumber the number of vertices, it is O(E log V).

Real-World Implementation

Routing Systems

In routing systems, the goal is always wanting the best route to send a packet from source router to destination router. Source router is a default router of the source host. While destination router is a default router of the destination host. By giving a set of routers with links connected to it, Dijkstra’s Algorithm find a good path from the two points. Good path here means low cost in term of length, speed, money, and etc.

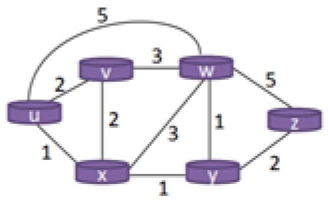


Figure 1 Graph R

Figure 1 above is example in lecture slide of Cornell University; Networking: Routing Algorithms. Given graph R where the problem is to find a least cost path from source to all the nodes and assuming that: -

- Node U, V, W, X, Y, Z represents routers.
- Edges with value represent links and its cost.
- Node U is a source node.

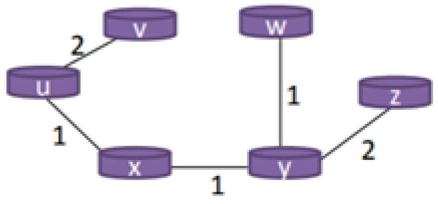
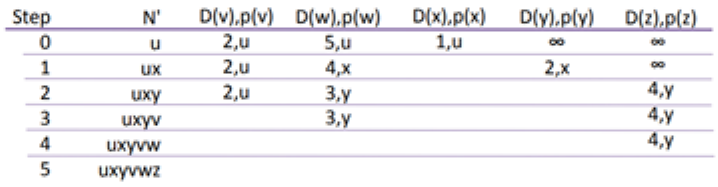


Figure 2 Shortest-Path Tree From U



Step	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					

Figure 3 Working Steps of Dijkstra's Algorithm

After run the problem through Dijkstra’s Algorithm, the least cost path as in the Figure 2.

Global Positioning System (GPS)

Dijkstra’s Algorithm can have Global Positioning System (GPS) as its new functionality. GPS is a satellite-based system that can be used in navigation to locate the positions anywhere on the earth (Singal & Chhillar, 2014). GPS can be used in different ways like to determine the position of locations, navigate from one location to another, create digitized maps and to determine the distance between two points. At this point, by implementing GPS in this algorithm, it can be beneficial for parents to look after their children, delivery services, fire services and others.

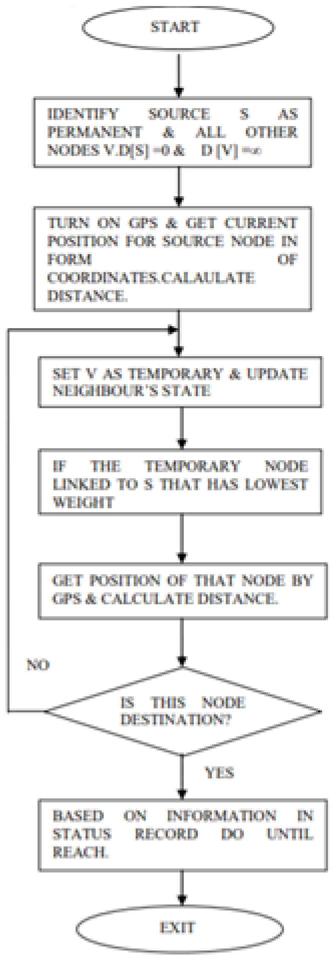

START
IDENTIFY SOURCE S AS PERMANENT & ALL OTHER NODES V.D[S] =0 & D [V] =∞
TURN ON GPS & GET CURRENT POSITION FOR SOURCE NODE IN FORM OF COORDINATES.CALAULATE DISTANCE.
SET V AS TEMPORARY & UPDATE NEIGHBOUR'S STATE
IF THE TEMPORARY NODE LINKED TO S THAT HAS LOWEST WEIGHT
GET POSITION OF THAT NODE BY GPS & CALCULATE DISTANCE.
NO
IS THIS NODE DESTINATION?
YES
BASED ON INFORMATION IN STATUS RECORD DO UNTIL REACH.
EXIT

Figure 4 has shown the idea how Dijkstra’s Algorithm and GPS work together. Basically, the GPS will give input to the algorithm in coordinate form. All the coordinates given to the algorithm will be analyzed in order to find the shortest path.

Google Maps

Dijkstra’s Algorithm was influential in modern day navigation systems (Lanning, Harrell, & Wang, 2014). The most popular navigation system is Google Maps. The combination of GPS and Dijkstra’s Algorithm make Google Maps not just a simple world map. It can give direction for several ways of transportation from driving to walking to biking. The application can also calculate the best route by minimizing its estimated arrival time and shortest distance to the destination. Moreover, Google Maps goes further by providing details like street views and images of the nearest place of the destinations to its users.

However, it would be false to presume that the entirety of Google Maps is based on the simple Dijkstra Algorithm. As a multi-billion dollar company, Google Maps would require far more advanced algorithms and systems to be able to process possibly millions of queries in a matter of seconds. This would of course incur the space-time tradeoff that Google would consider when it comes to the computation and memory the maps require on their cloud platform.

Creating a digital map is an arduous process, as it requires the tagging of every interest point. Taking that into account, Google Maps can be presumed to have some sort of hierarchical superstructure built along with its tagged points of interest to maximise its decision-making ability and at the same time, reduce computation complexity. Google would probably opt for more efficient algorithms, such as the A* search when performing computations at a lower level.

Waze

Waze is different from Google maps, an application that was discussed above. Google Maps is data-based, while Waze is community-driven, where users of their application will report events such as traffic jams, hazards along the road, and any accidents that may have occurred. Waze is built on top of Google Maps, but the routes that it chooses is entirely based on the most efficient route according to the events reported by its users. Waze will then direct its users to avoid unnecessary blockages or tolls, in favour of the user’s preferences. It could also be said that Waze does not entirely, as more efficient algorithms exist. The most probable and closest to their implementation would be the A* search, where it can avoid less desirable routes.

Reference

Chuang Ruan, Jianping Luo, & Yu Wu. (2014). Map navigation system based on optimal Dijkstra algorithm. In *2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems* (pp. 559–564). IEEE. https://doi.org/10.1109/CCIS.2014.7175798

Cornell University. (2016). Networking: Routing Algorithm. Retrieved from http://www.cs.cornell.edu/courses/cs4410/2016su/slides/lecture24.pdf

Dcsa, P. S., & Chhillar, R. R. S. (2014). *Dijkstra Shortest Path Algorithm using Global Positioning System*. *International Journal of Computer Applications* (Vol. 101). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.800.5178&rep=rep1&type=pdf

Deng, Y., Chen, Y., Zhang, Y., & Mahadevan, S. (2012). Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment. *Applied Soft Computing*, *12*(3), 1231–1237. https://doi.org/10.1016/J.ASOC.2011.11.011

Felner, A. (2011). *Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm*. Retrieved from www.aaai.org

Galán-García, J. L., Aguilera-Venegas, G., Galán-García, M. Á., & Rodríguez-Cielos, P. (2015). A new Probabilistic Extension of Dijkstraâ€™s Algorithm to simulate more realistic traffic flow in a smart city. *Applied Mathematics and Computation*, *267*, 780–789. https://doi.org/10.1016/j.amc.2014.11.076

Kai, N., Yao-Ting, Z., & Yue-Peng, M. (2014). Shortest Path Analysis Based on Dijkstra's Algorithm in Emergency Response System. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, *12*(5), 3476–3482. https://doi.org/10.11591/telkomnika.v12i5.3236

Knuth, D. E. (1977). A generalization of Dijkstra's algorithm. *Information Processing Letters*, *6*(1), 1–5. https://doi.org/10.1016/0020-0190(77)90002-3

Lanning, D. R., Harrell, G. K., & Wang, J. (2014). Dijkstra's Algorithm and Google Maps. *2014 ACM Southeast Regional Conference.* United States: Association for Computing Machinery, Inc. Retrieved from http://delivery.acm.org.proxyvlib.mmu.edu.my/10.1145/2640000/2638494/a30-lanning.pdf?ip=203.106.62.29&id=2638494&acc=ACTIVE%20SERVICE&key=69AF3716A20387ED%2EE854CB4DB8D6D408%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1547709875_b6863aa6ed1ae39287cbdd40

Shu-Xi, W. (2012). The Improved Dijkstra's Shortest Path Algorithm and Its Application. *Procedia Engineering*, *29*, 1186–1190. https://doi.org/10.1016/j.proeng.2012.01.110

Singal, P., & Chhillar, R. (2014). Dijkstra Shortest Path Algorithm using Global Positioning System. *International Journal of Computer Applications*, Volume 101-No.6. Retrieved from https://pdfs.semanticscholar.org/1540/945b4b16d9b485e410ff4a6730a7c103f399.pdf

Szcze, I., Jajszczyk, A., & Wo, zena. (2018). *Generic Dijkstra for Optical Networks*. Retrieved from https://arxiv.org/pdf/1810.04481.pdf

Xu, Y., Wen, Z., & Zhang, X. (2015). Indoor optimal path planning based on Dijkstra Algorithm, (Meita), 309–313. Retrieved from http://www.atlantis-press.com/php/download_paper.php?id=25838483