DIJKSTRA ALGORITHM

Adolf A D'Costa - 903538404
Ananthula Lekha - 903872019
Sannithi Vinod Kumar - 903636939

Final Math's Project

APPLIED DISCRETE MATHEMATICS
(MTH5051)

INDEX

HISTORY OF DIJKSTRA ALGORITHM

What is the shortest way to travel from Rotterdam to Groningen? In general, it was given from city to city. It is the Algorithm for the shortest path given by him, which is designed in Twenty minutes. One Fein morning he was shopping in Amsterdam with his young fiancée, and they were tried, they sat down at the cafe terrace to drink coffee. And then designed the algorithm for the shortest path. It was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable; it is, in fact, quite nice. He learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became, to my great amazement, one of the cornerstones of my fame.

— Edger Dijkstra, In an interview with Philip L. Frana, communications of the ACM 2013 [i]

Dijkstra was thought about this method when he was working in Mathematical Center in Amsterdam in 1956. He is there for teaching the capabilities of the new computer known as ARMAC [ii]. His Goal is to find both question and solution (that would be produced by the computer) He designed the shortest path algorithm and later implemented it for ARMAC for a slightly simplified transportation map of 64 cities in the Netherlands (64, so that 6 bits would be sufficient to encode the city number) [iii]. A year later he came across with another problem from Hardware Engineers working on the institute next computers, As the solution for this he came up with the Algorithm known as Prim minimal spanning tree Algorithm (known earlier to Arnik and also rediscovered by Prim).[iv] [v] ,Dijkstra published the algorithm in 1959, two years after Prim and 29 years after Jarník [vi] [vii].

DIFFERENT ALGORITHMS FOR FINDING SHORTEST PATH:

BELLMAN - FORD ALGORITHM

HISTORY OF BELLMAN FORD ALGORITHM

The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively. ... Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.

This Algorithm was first proposed by Alfonso Shimbel (1995) but it is named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956. Respectively in 1957 Edward F. Moore also published the same algorithm, so it is also called as Bellman-Ford - Moore Algorithm

BELLMAN-FORD ALGORITHM

The Bellman-ford Algorithm is an algorithm that computes the shortest path from a single vertex to all the vertices in a weighted graph.[viii] It is slower than the Dijkstra Algorithm for the same problem but more easy to adapt but it can handle in which some weights are negative numbers.

Negative edge weights are found in Many graphs, this is the usefulness of the algorithm. If a graph contains negative cycles i.e. sum of edges come to negative values in graph, from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk round the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence. [ix][x]

HOW DOES IT WORKS:

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, we are able to guarantee that the end result is optimized . The Process of Bellman Ford Algorithm is shown in the below Figure 1 (bellman - ford) [xi]

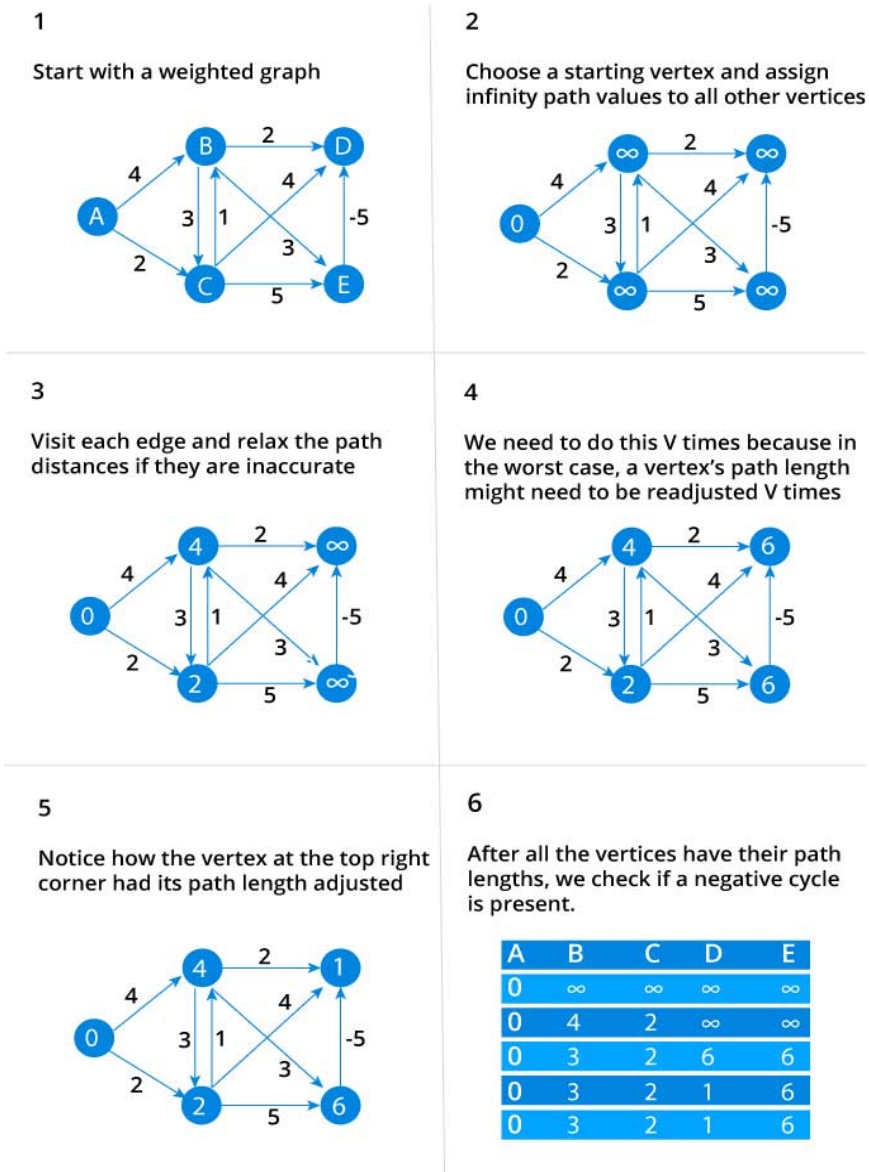


Figure 1 shows Bellman Ford algorithm

FLOYD - WARSHALL ALGORITHM

HISTORY OF FLOYD - WARSHALL ALGORITHM

The Floyd Warshall Algorithm is the example of the Dynamic Programming and was published in its currently recognized form by Robert Floyd in 1962 [xii] However, it is essentially the same as algorithms previously published by Bernard Roy in 1959 [xiii] and also by Stephen Warshall in 1952 [xiv] for finding the transitive closure of a graph,[xv] and it is closely related to Kleene's Algorithm (published in 1956) for converting a Deterministic finite Automation into a Regular expression [xvi]. The modern formulation of the algorithm as three nested for loops was first described by Peter in German, in 1962. [xvii] The Algorithm is also known as Floyd algorithm, the Roy Warshall algorithm, or the Roy - Floyd algorithm, or WFI algorithm.

Floyd-Warshall is extremely useful in networking, similar to solutions to the shortest path problem. However, it is more effective at managing multiple stops on the route because it can calculate the shortest paths between all relevant nodes. In fact, one run of Floyd-Warshall can give you all the information you need to know about a static network to optimize most types of paths. It is also useful in computing matrix inversions.

FLOYD - WARSHALL ALGORITHM

The Floyd - Warshall algorithm is an algorithm to find the shortest path in a weighted graph with positive and negative edge weights (but with no negative cycles). [xviii][xix] A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.

HOW DOES FLOYD - WARSHALL ALGORITHM WORK:

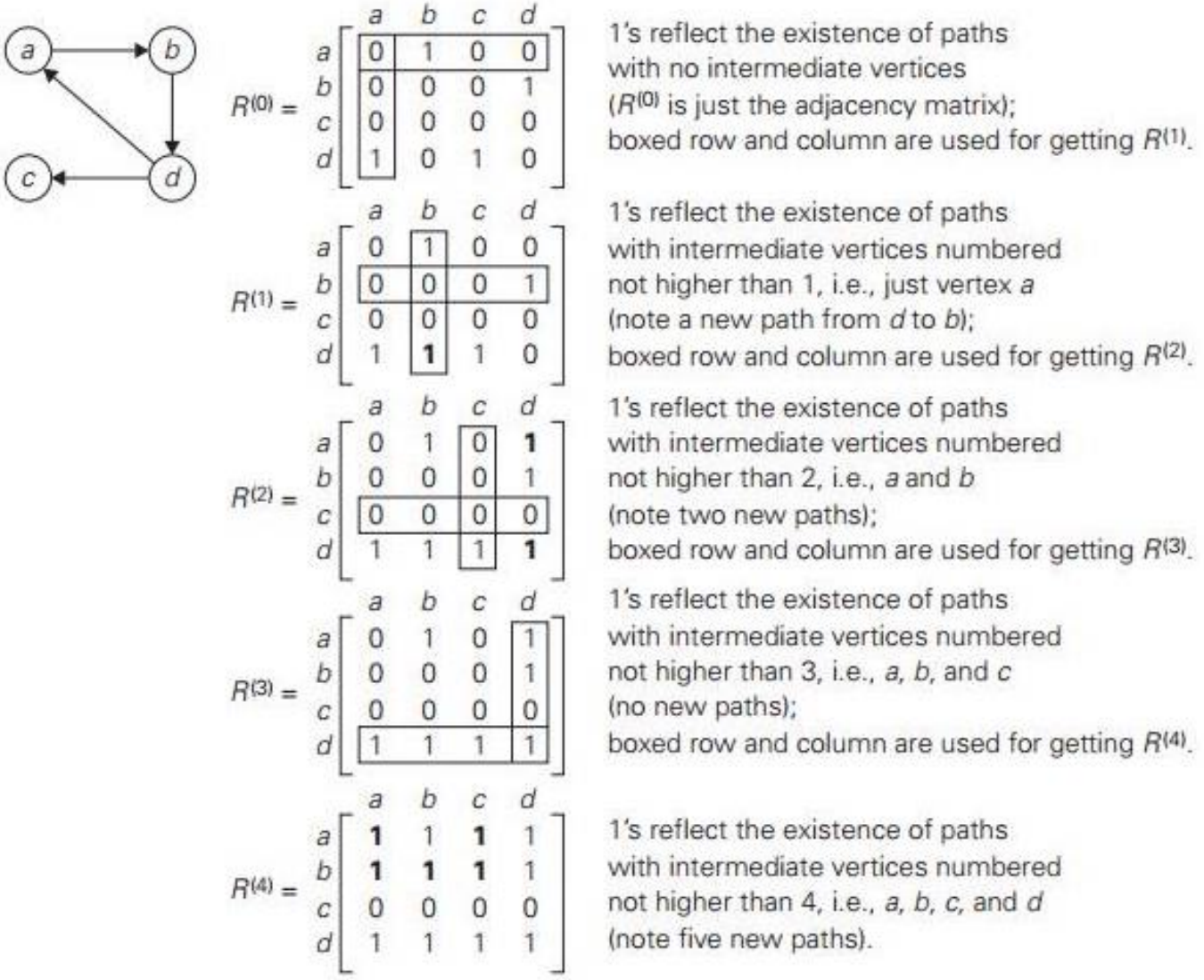


Figure 2 shows how Floyd- Warshall Algorithm works

PRIM'S ALGORITHM

HISTORY PRIM'S ALGORITHM

The algorithm was developed in 1930 by Czech Mathematician Vojtéch Jarnik and later rediscovered and republished by computer scientists Robert C. Prim in 1959 [xx] and Edger Dijkstra in 1953. Therefore, it is also sometimes called the Jarnik's algorithm, Prim's algorithm, Prim's Dijkstra algorithm or DJP algorithm.

ALGORITHM

The Other well-known algorithms for this problem include Kruskal Algorithm and Boruvka's algorithm. [xxi] these algorithms find the minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of Prim's algorithm only finds the minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph, it can also be used to find the minimum spanning forest. [xxii] However, for graphs that are sufficiently dense, Prim's algorithm can be made to run in linear time, meeting or improving the time bounds for other algorithms.

In computer science, prim's (also known as Jarnik's) algorithm is a greedy an algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

HOW DOES THE PRIM'S ALGORITHM WORK?

The idea behind the Prim's algorithm is simple, a spanning tree means all vertices must be connected. So, the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected

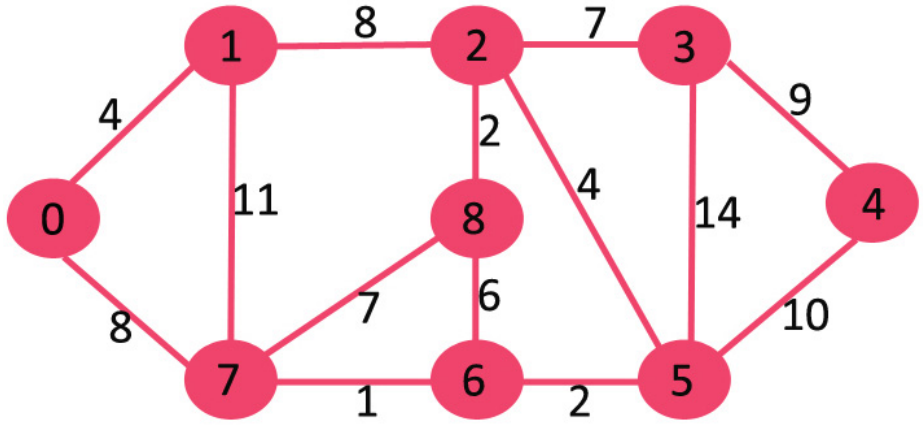
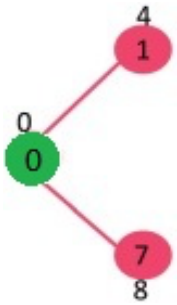


Figure 3 example for Prim's Algorithm

with the minimum weight edge to make it a Minimum Spanning Tree and example is given below.

The set mstSet is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

Figure 4 vertices included in MST in green color

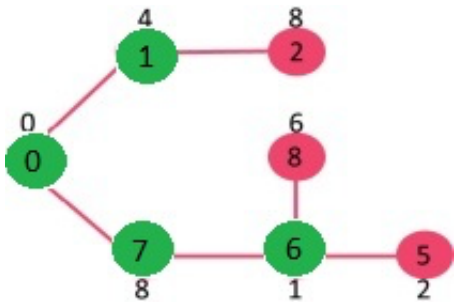


Figure 5 after updating the key values

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

We repeat the above steps until mstSet includes all vertices of given graph. Finally, we get the following graph.

A* SEARCH ALGORITHM

HISTORY OF A* ALGORITHM

A* was created as part of The Shakey Project which had the aim of building a mobile robot that could plan its own actions. Nils Nilsson originally proposed using the Graph Traverser algorithm [xxiii] for Shakey's path planning Graph Traverser is guided by a heuristic function h(n) the estimated distance from node g(n).

MOTIVATION

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances. We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)

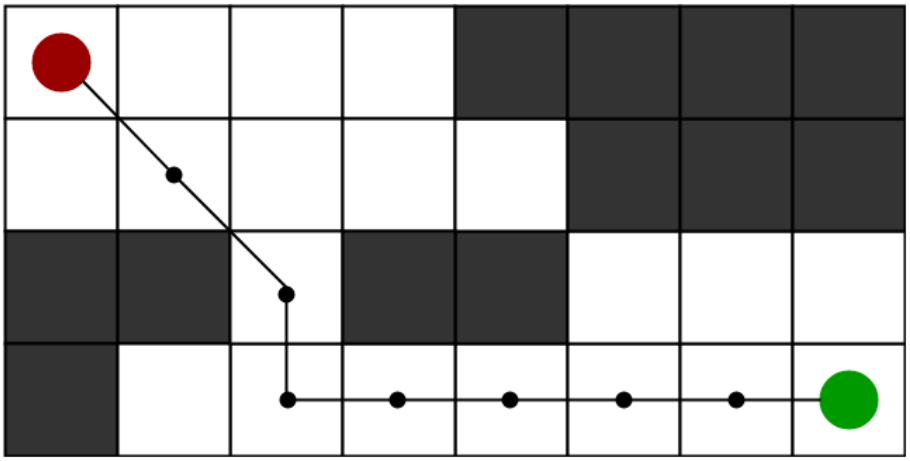


Figure 6 basic sample of algorithm

WHAT IS A* SEARCH ALGORITHM:

A* search Algorithm is one of the best and popular techniques used in path-finding and graph traversals.

WHY A* ALGORITHM?

Speaking which A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in the below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

HOW DOES A* ALGORITHM WORK

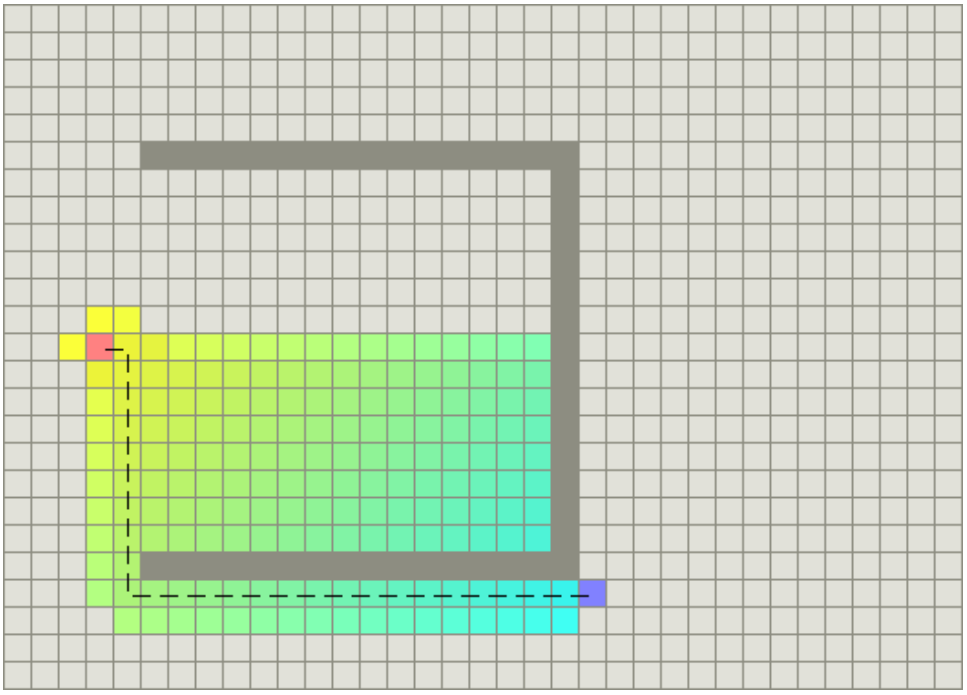


Figure 7 A algorithm example with graph*

A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the *exact cost* of the path from the starting point to any vertex n, and h(n) represents the heuristic *estimated cost* from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

KRUSKAL'S ALGORITHM

HISTORY OF KRUSKAL'S ALGORITHM

Kruskal studied mathematics at the University of Chicago. He was awarded a BS in 1948 and an MS in 1949 by Chicago. Joseph Bernard Kruskal Sr. died in 1950. Joseph Kruskal's mother Lillian married Harry C Oppenheimer in 1954. Kruskal married Rachel Solomon in 1953; they had two children, Joyce and Benjamin.

ALGORITHM

Kruskal's Algorithm is a minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.[xxiv] it is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connecting weighted graph adding increasing cost arcs at each step, this means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

This algorithm first appeared in Proceedings of the first American Mathematical Society, which is written by Joseph Kruskal [xxv] Other algorithms for this problem include Prims Algorithms, Reverse algorithm, and Boruvka algorithm.

HOW DOES KRUSKAL'S ALGORITHM WORK?

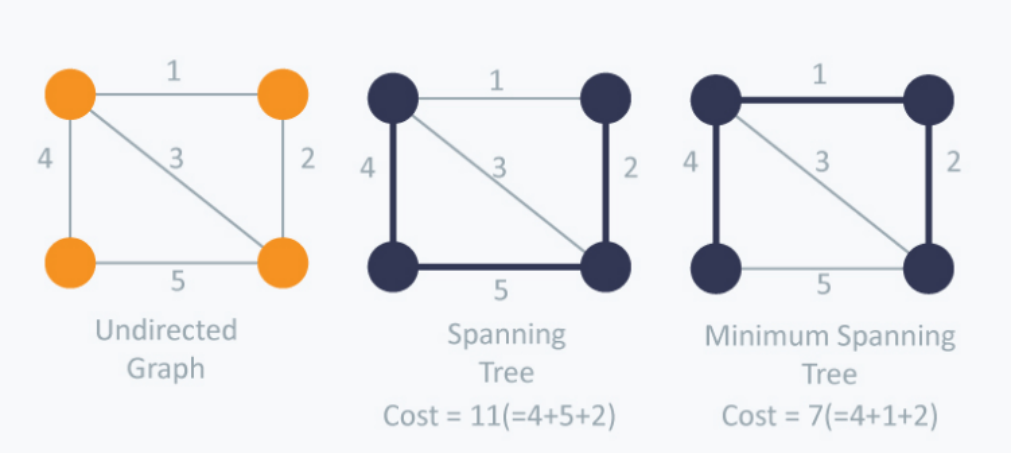


Figure 8 Kruskal algorithm using Greedy approach

Kruskal's algorithm uses the greedy approach for finding a minimum spanning tree. Kruskal's algorithm treats every node as an independent tree and connects one with another only if it has the lowest cost compared to all other options available.

- Sort the graph edges with respect to their weights,
- Start adding edges to the minimum spanning tree from the edge with the
- Smallest weight until the edge of the largest weight,

Only add edges which don't form a cycle—edges which connect only disconnected components.

DIJKSTRA'S ALGORITHM

HISTORY OF DIJKSTRA'S ALGORITHM

Dijkstra Algorithm or (Dijkstra Shortest path first algorithm) [xxvi] is an algorithm is normally used to find the shortest path between nodes in the graph, which may represent, for example, road networks. It was conceived by computer scientist Edger Dijkstra in 1956 and published three years later [xxvii] The algorithm exists in many variants; Dijkstra's original variant found the shortest path between

two nodes [xxviii] but a 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.

ALGORITHM

For a given source node in the graph, the algorithm finds the shortest path between that node and every other.[xxix] 196-206 It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

However, there are some ideas and definitions that are much harder to understand. These are the ones that you feel like you're supposed to know, but you haven't run across it enough to really comprehend it.

Topics that we feel like we're meant to know but never quite got around to learningare the most intimidating ones of all. The barrier to entry is so high, and it can feel impossibly hard to understand something that you have little to no context for. For me, that intimidating topic is Dijkstra's algorithm. I had always heard it mentioned in passing, but never came across it, so I never had the context or the tools to try to understand it.

HOW DOES THE ALGORITHM WORK?

Let's calculate the shortest path between node C and the other nodes in our graph:

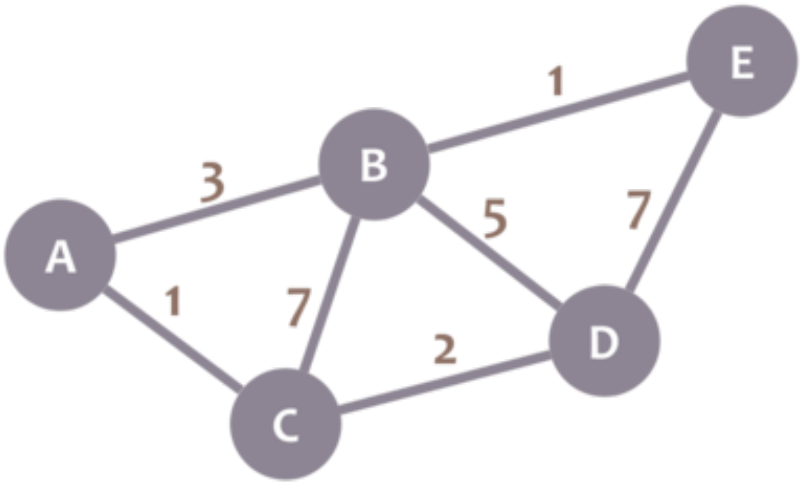


Figure 9 example for Dijkstra algorithm

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

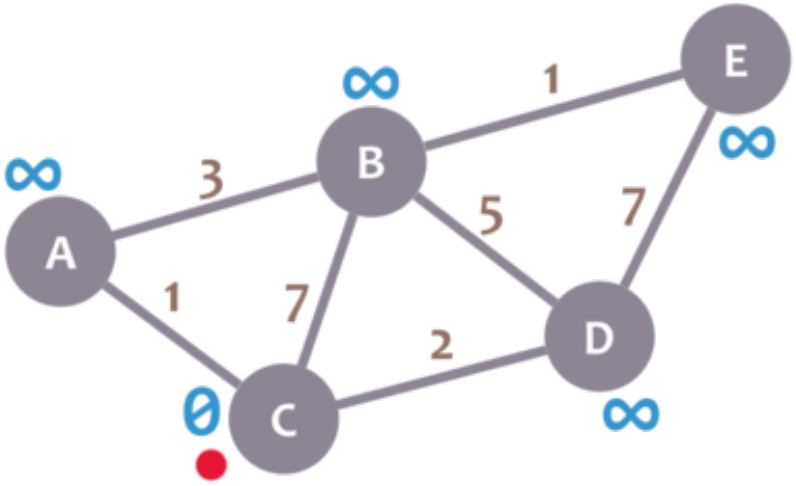


Figure 10 initially making the staring node with red dot

also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

Now, we check the neighbors of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We 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)

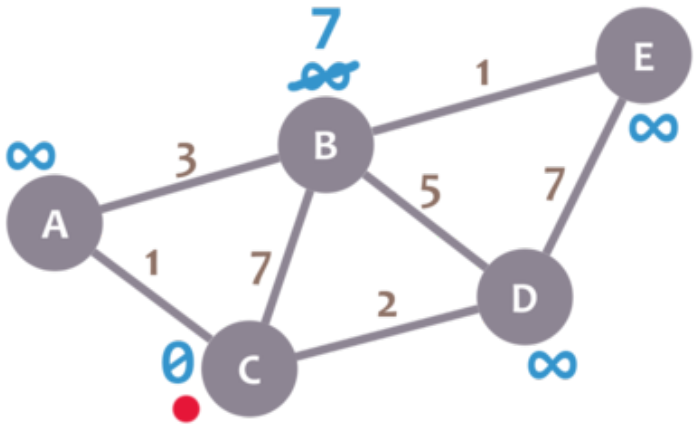


Figure 11 adding minimum distance from initial node

So far, so good. Now, let's check neighbor A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:

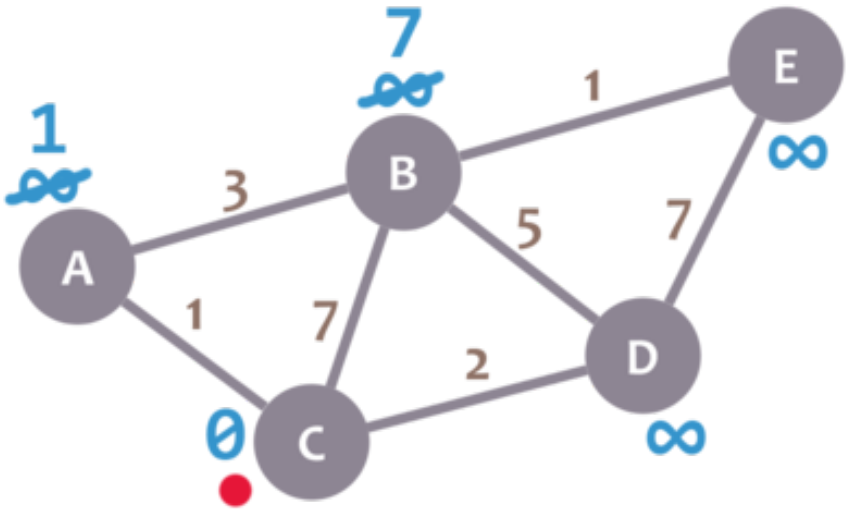


Figure 12 checking the neighbor and adding the values

OK. Repeat the same procedure for D

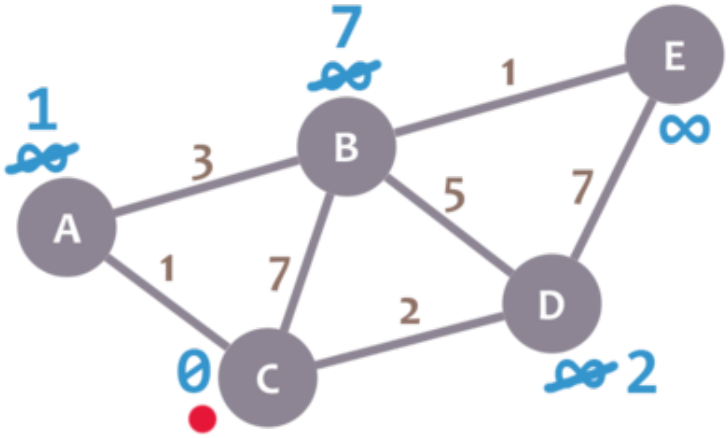


Figure 13 Repeating the same process

We have checked all the neighbors of C. Because of that, we mark it as visited. Let's represent visited nodes with a green check mark:

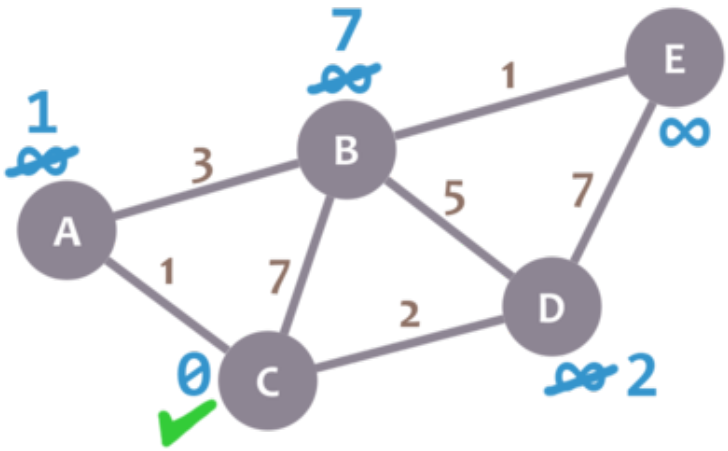


Figure 14 After representing the visiting nodes with green tick

We now 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. Let's mark it with the red dot again:

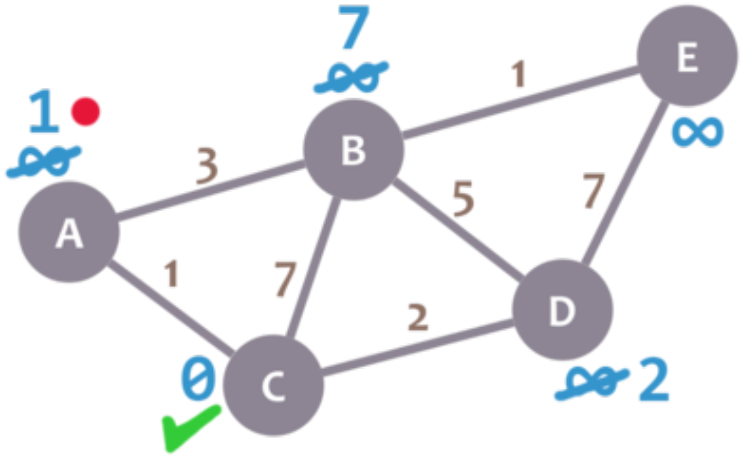


Figure 15 Picking another node

And now we repeat the algorithm. We check the neighbors of our current node, ignoring the visited nodes. This means we only check B.

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

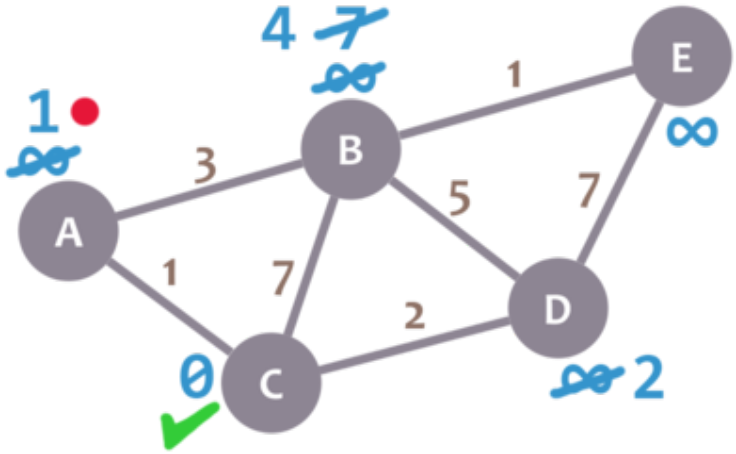


Figure 16 Adding the minimum distance from B

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

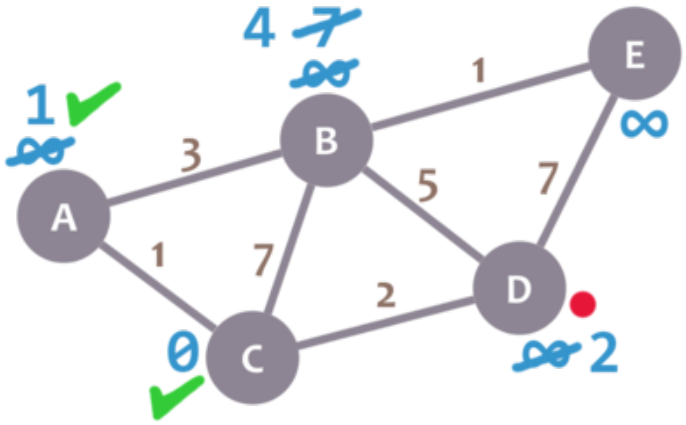


Figure 17 Picking the new node

We repeat the algorithm again. This time, we check B and E.

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

We mark D as visited and set our current node to B.

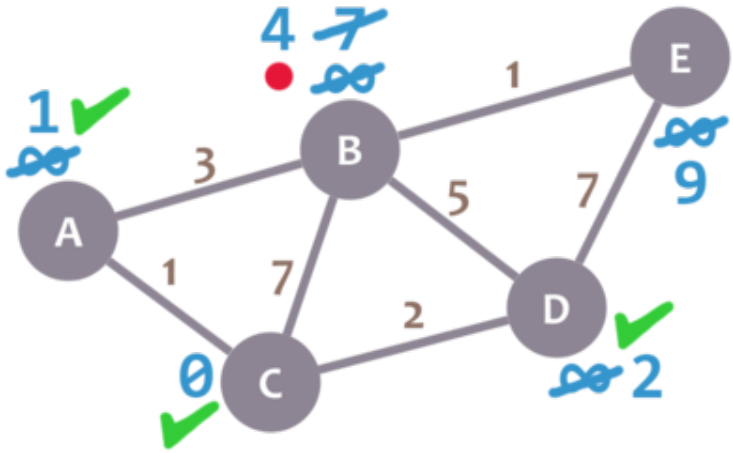


Figure 18 marking the D as visited

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

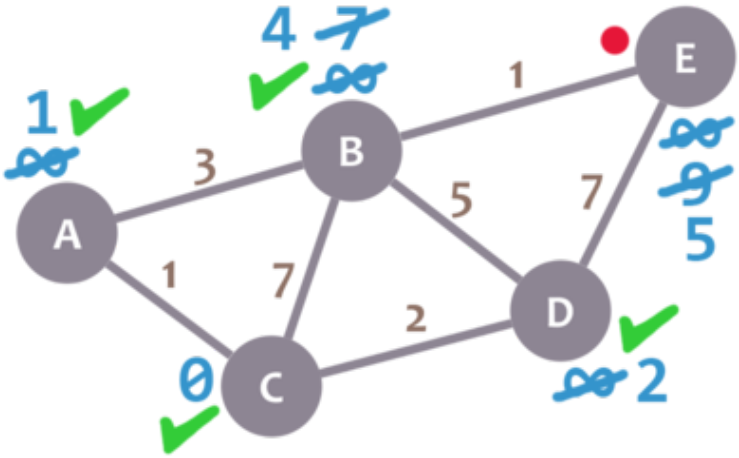


Figure 19 checking the node E

E doesn't have any non-visited neighbor's, so we don't need to check anything. We mark it as visited.

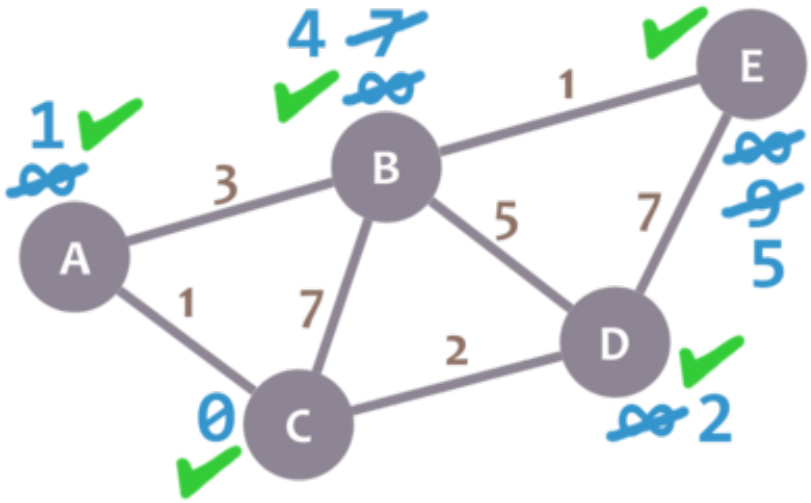


Figure 20 Minimum distance by using Dijkstra algorithm

As there are not unvisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

Here's a description of the algorithm:

1. Mark your selected initial 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 C.
3. For each neighbor N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it's 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. If there are non-visited nodes, go to step 2

RULES FOR DIJKSTRA ALGORITHM

Dijkstra Algorithm is unique for many reasons, which we'll soon see as we start to understand how it works. But the one that has always come as a slight surprise is the fact that this algorithm isn't just used to find the shortest path between two specific nodes in a graph data structure. Dijkstra's algorithm can be used to determine the shortest path from one node in a graph to every other node within the same graph data structure, provided that the nodes are reachable from the starting node.

This algorithm will run until all vertices in the graph have been visited. This means that the shortest path between any 2 nodes can be saved and looked up after.

Here This algorithm will continue to run until all of the reachable vertices in a graph have been visited, which means that we could run Dijkstra's algorithm, find the shortest path between any two reachable nodes, and then save the results somewhere. Once we run Dijkstra's algorithm just once, we can look up our results from our algorithm again and again—without having to actually run the algorithm itself! The only time we'd ever need to re-run Dijkstra's algorithm is if something about our graph data structure changed, in which case we'd end up re-running the algorithm to ensure that we still have the most up-to-date shortest paths for our particular data structure.

The abstracted rules are as follows:

- Every time that we set out to visit a new node, we will choose the node with the smallest known distance/cost to visit first.

- Once we've moved to the node we're going to visit, we will check each of its neighboring nodes

- For each neighboring node, we'll calculate the distance/cost for the neighboring nodes by summing the cost of the edges that lead to the node we're checking from the starting vertex.

- Finally, if the distance/cost to a node is *less than* a known distance, we'll update the shortest distance that we have on file for that vertex.

REAL LIFE APPLICATIONS OF DIJKSTRA'S ALGORITHM

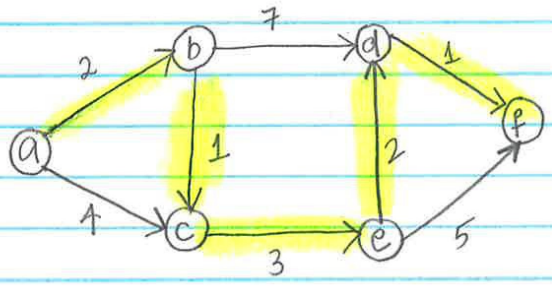
- It is used to It is used in finding Shortest Path.
- It is used in geographical Maps.
- To find locations of Map which refers to vertices of graph.
- Distance between the location refers to edges.
- It is used in IP routing to find Open shortest Path First.
- It is used in the telephone network. It's also called A* algorithm.

PRACTICAL EXAMPLE 1

THEORETICALLY EXPLAINED

Dijkstra's Algorithm

(1)



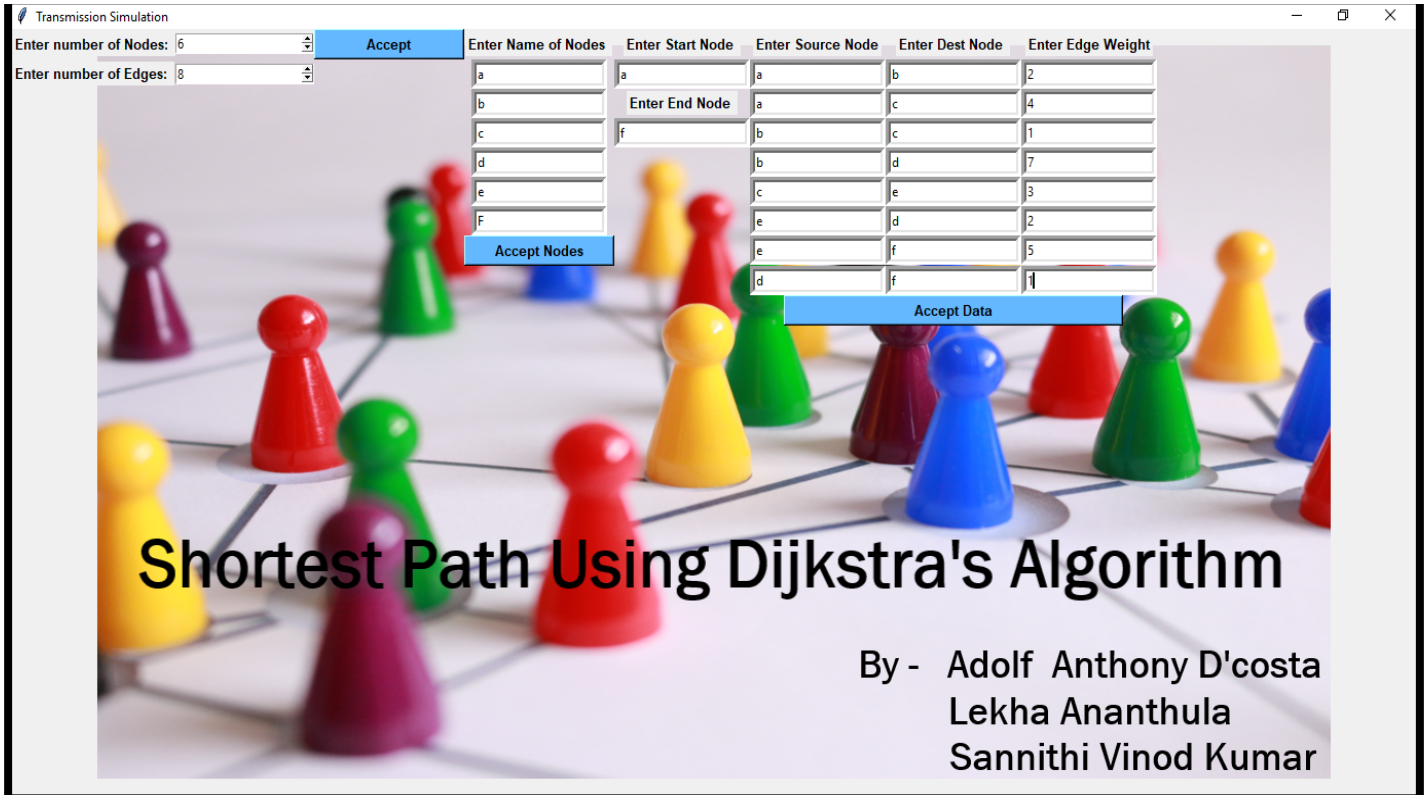
Start node - 'a' End node - 'f'

Shortest path

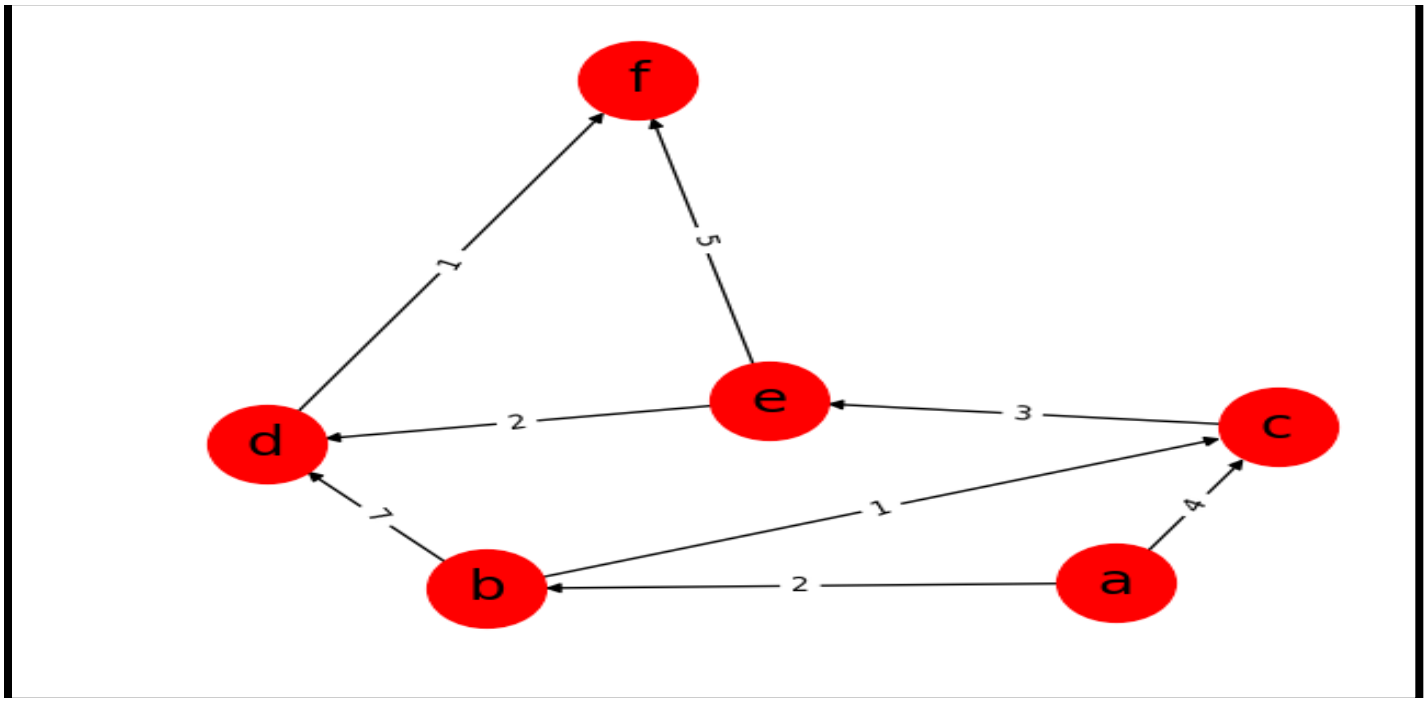
$a \xrightarrow{2} b \xrightarrow{1} c \xrightarrow{3} e \xrightarrow{2} d \xrightarrow{1} f$

The cost of it is '9' units.

Problem 1 solved using code



Graph 1 plotted by the program

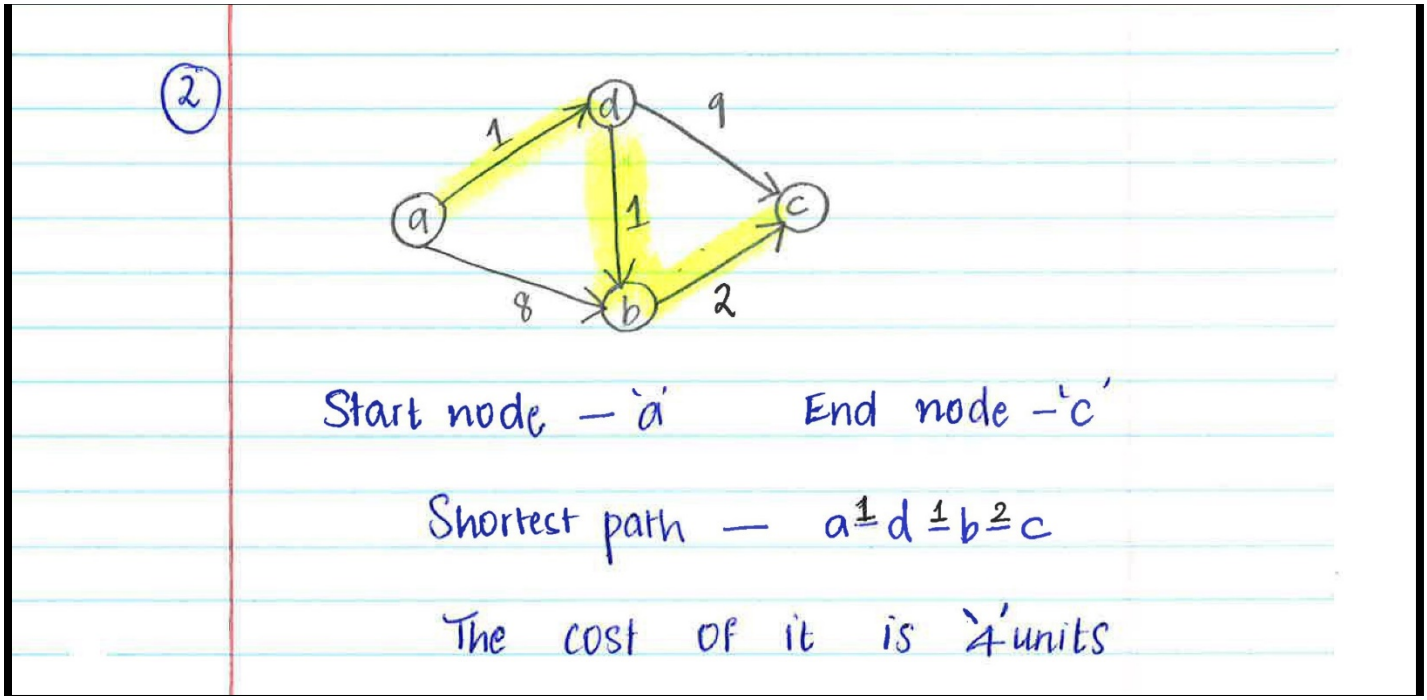


Result 1 from the program

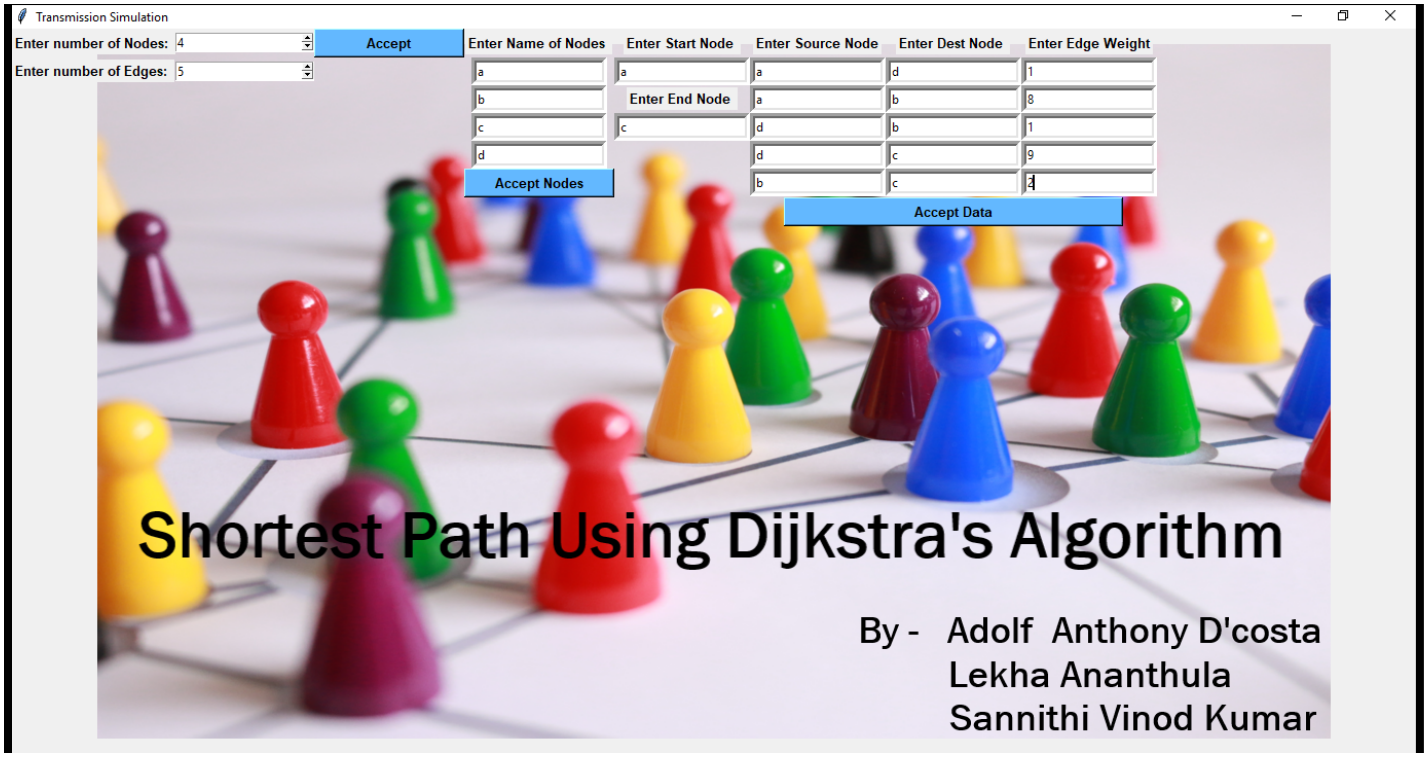
```
1 C:\Users\Adolf\AppData\Local\Programs\Python\Python37-32\
  python.exe "C:/Users/Adolf/Desktop/Maths Project/Maths
  Project.py"
2 [{'path': ['a', 'b', 'c', 'e', 'd', 'f'], 'weight': 9}, {'
  path': ['a', 'b', 'c', 'e', 'f'], 'weight': 11}, {'path
  ': ['a', 'b', 'd', 'f'], 'weight': 10}, {'path': ['a', 'c
  ', 'e', 'd', 'f'], 'weight': 10}, {'path': ['a', 'c', 'e
  ', 'f'], 'weight': 12}]
```

PRACTICAL EXAMPLE 2

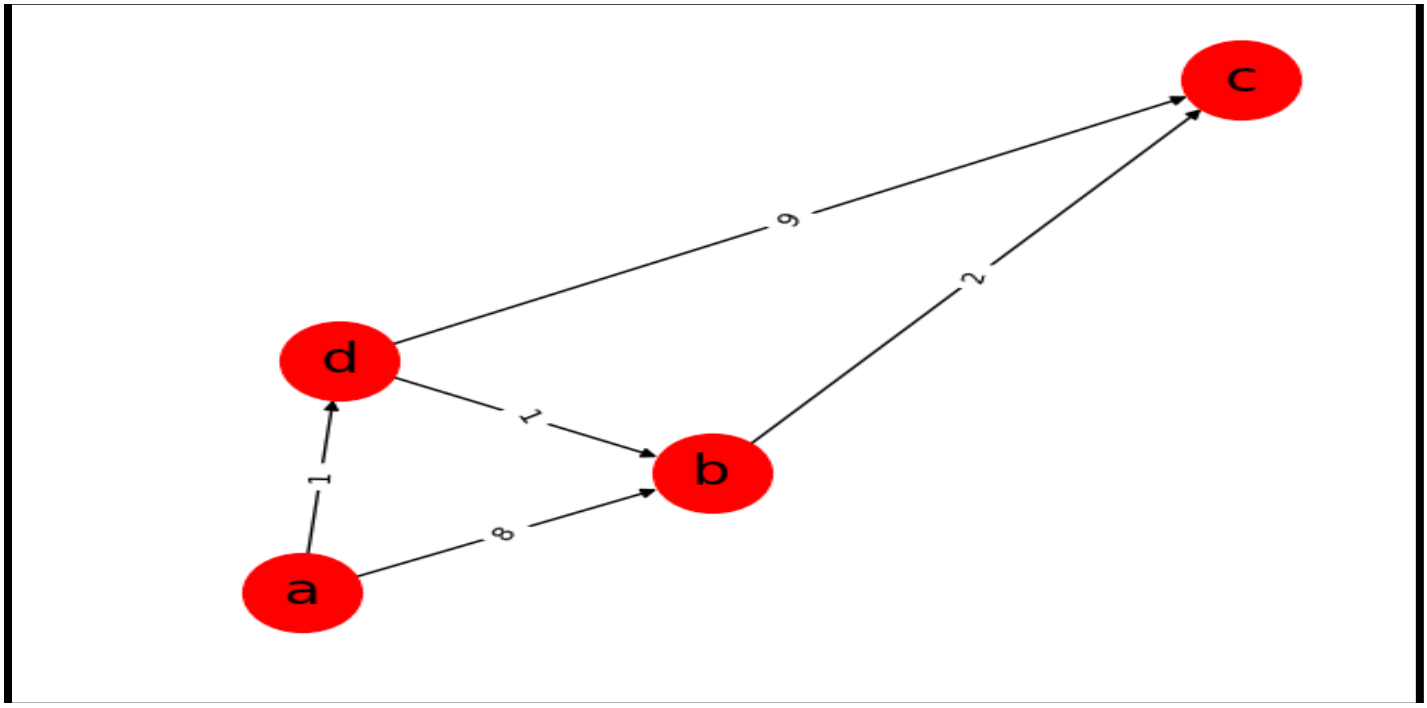
THEORETICALLY EXPLAINED



Problem 2 solved using code



Graph 2 plotted by the program

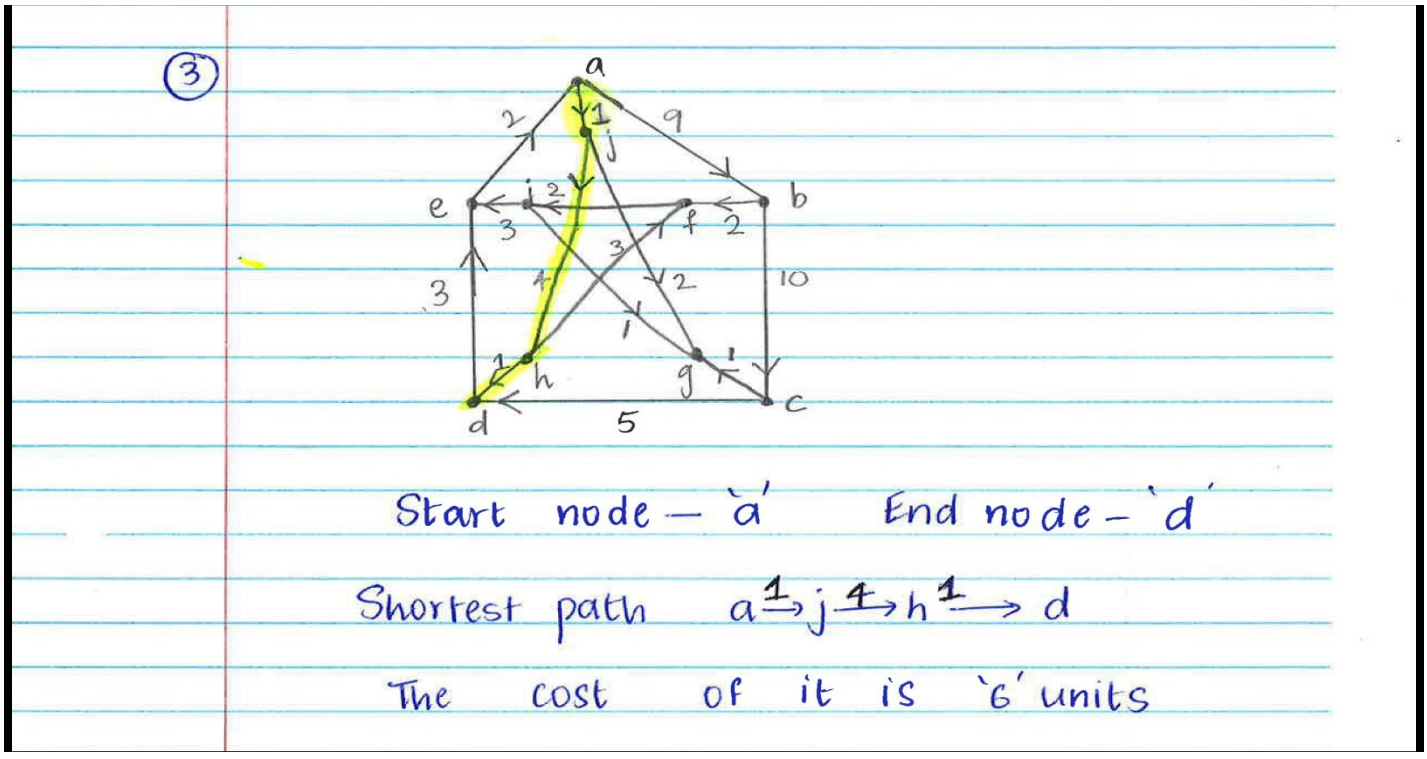


Result 2 from the program

```
1 C:\Users\Adolf\AppData\Local\Programs\Python\Python37-32\
  python.exe "C:/Users/Adolf/Desktop/Maths Project/Maths
  Project.py"
2 [{'path': ['a', 'd', 'b', 'c'], 'weight': 4}, {'path': ['a
  ', 'd', 'c'], 'weight': 10}, {'path': ['a', 'b', 'c'], '
  weight': 10}]
```

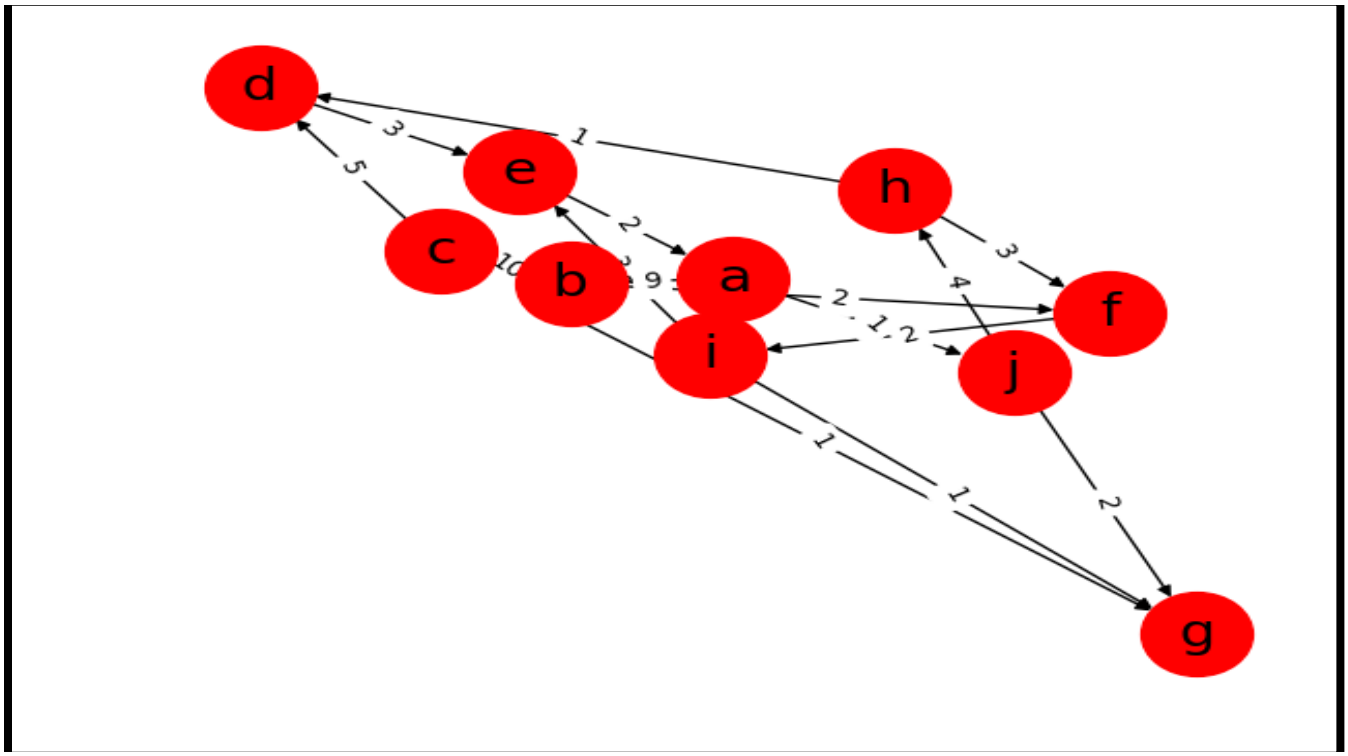
PRACTICAL EXAMPLE 3

THEORETICALLY EXPLAINED



Problem 3 solved using code

Graph 3 plotted by the program

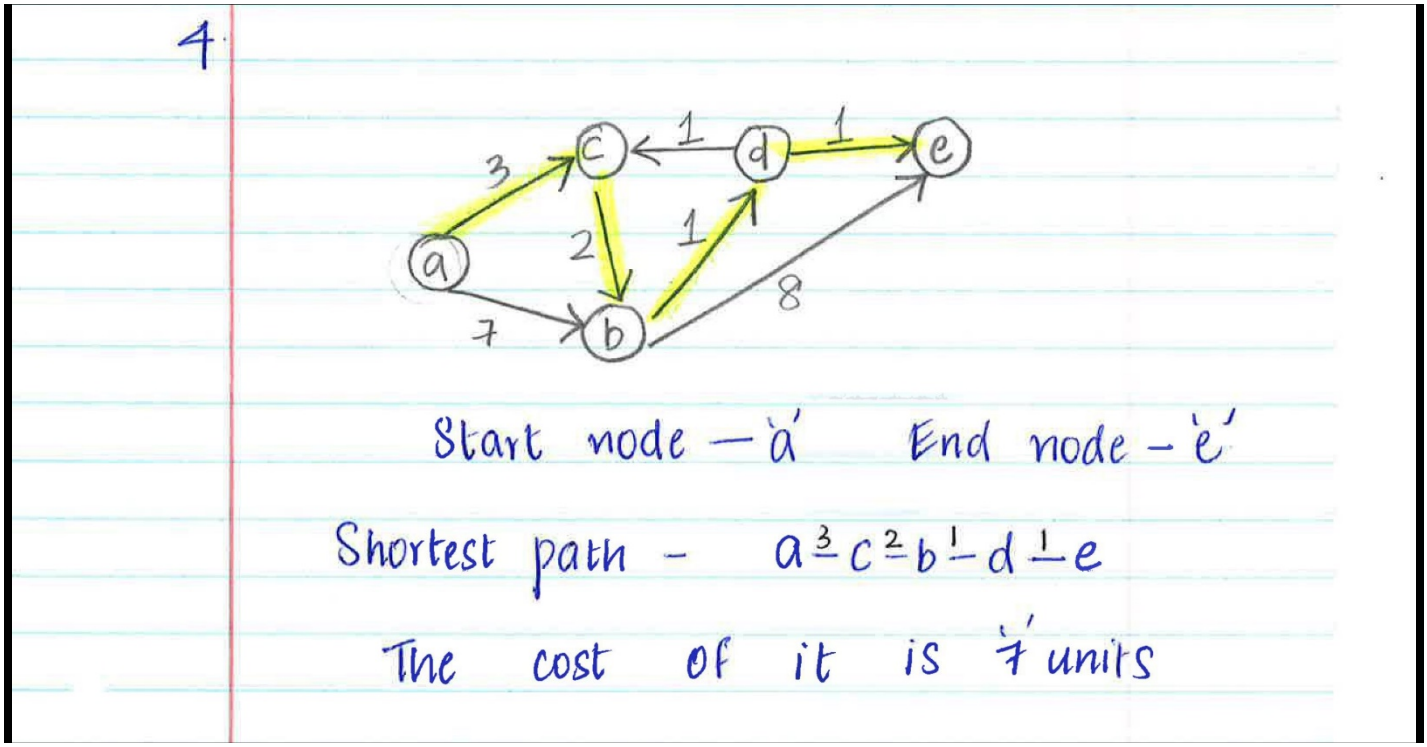


Result 3 from the program

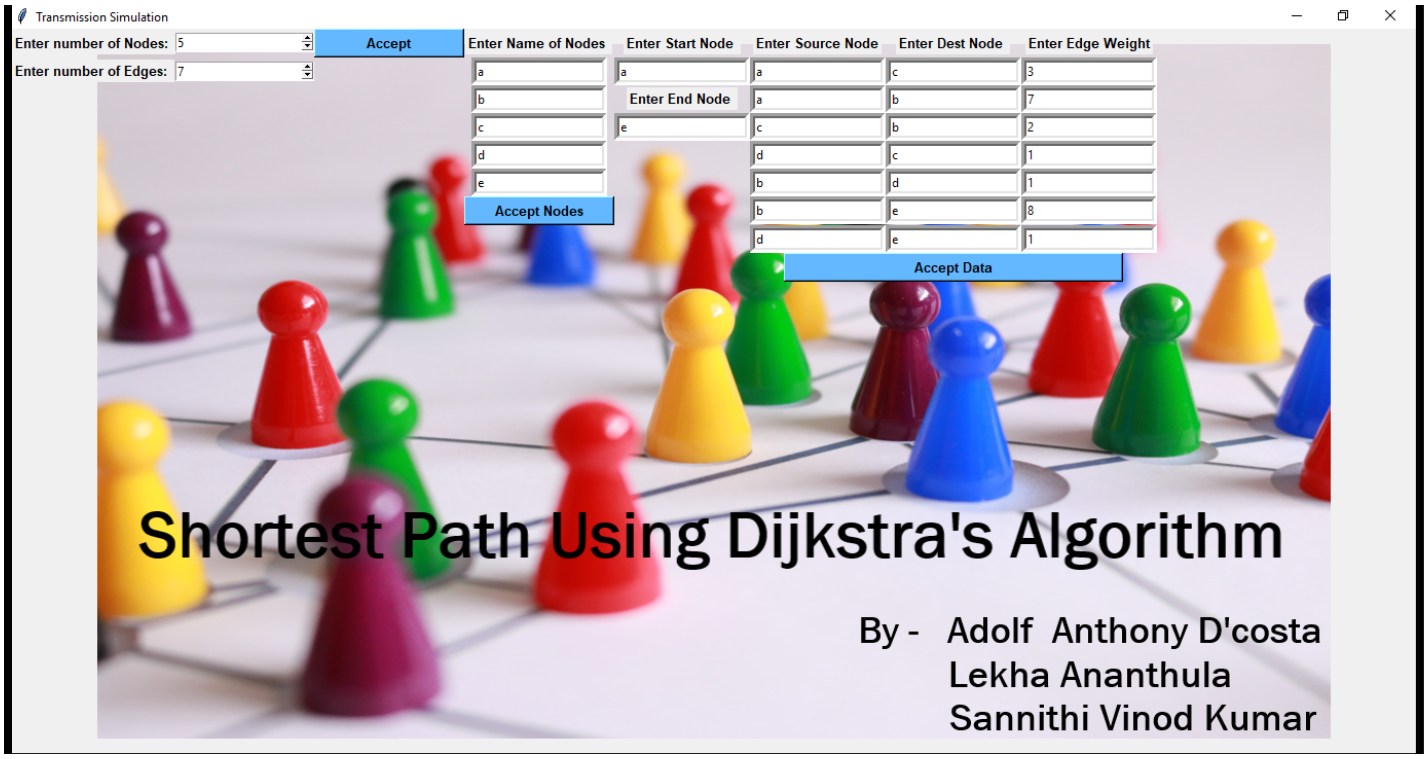
```
C:\Users\Adolf\AppData\Local\Programs\Python\Python37-32\
python.exe "C:/Users/Adolf/Desktop/Maths Project/Maths
Project.py"
[{'path': ['a', 'b', 'c', 'd'], 'weight': 24}, {'path': ['
a', 'j', 'h', 'd'], 'weight': 6}]
```

PRACTICAL EXAMPLE 4

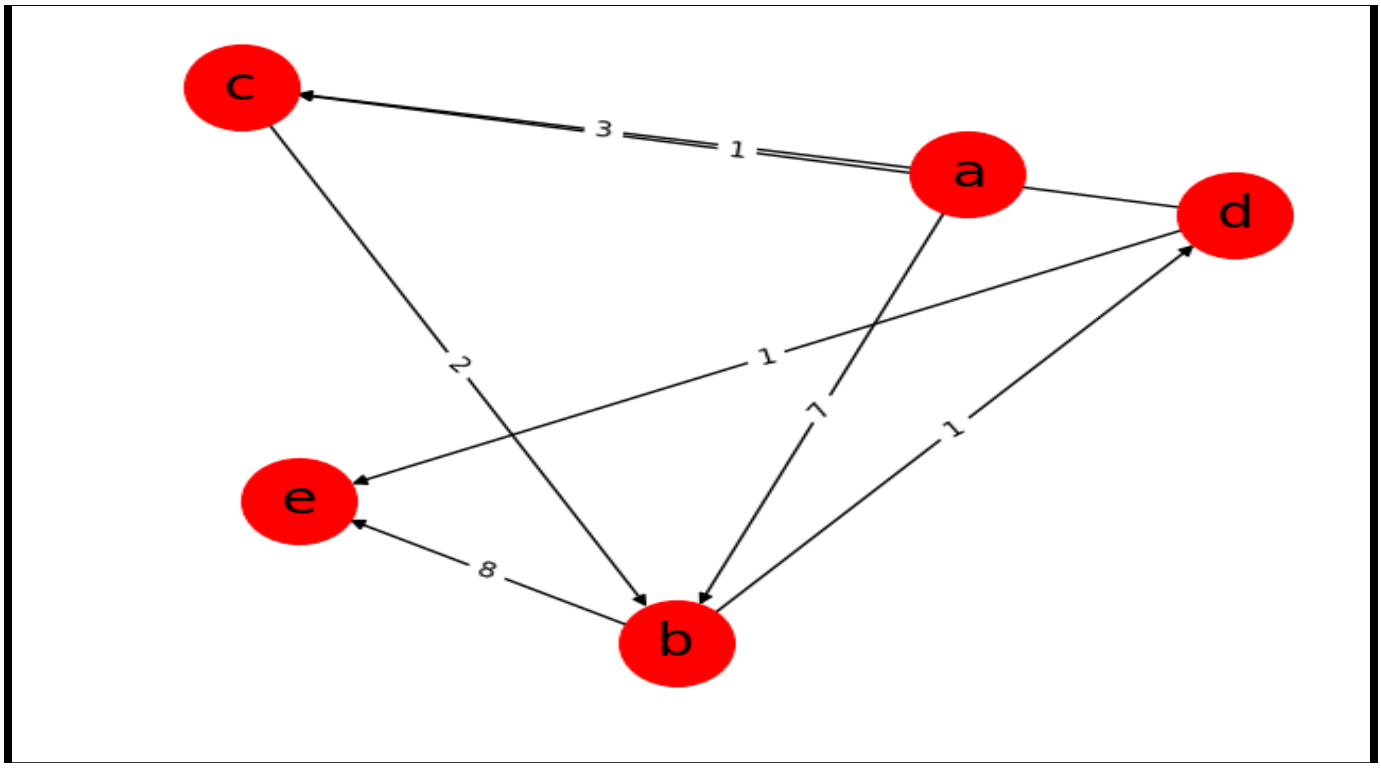
THEORETICALLY EXPLAINED



Problem 4 solved using code



Graph 4 plotted by the program

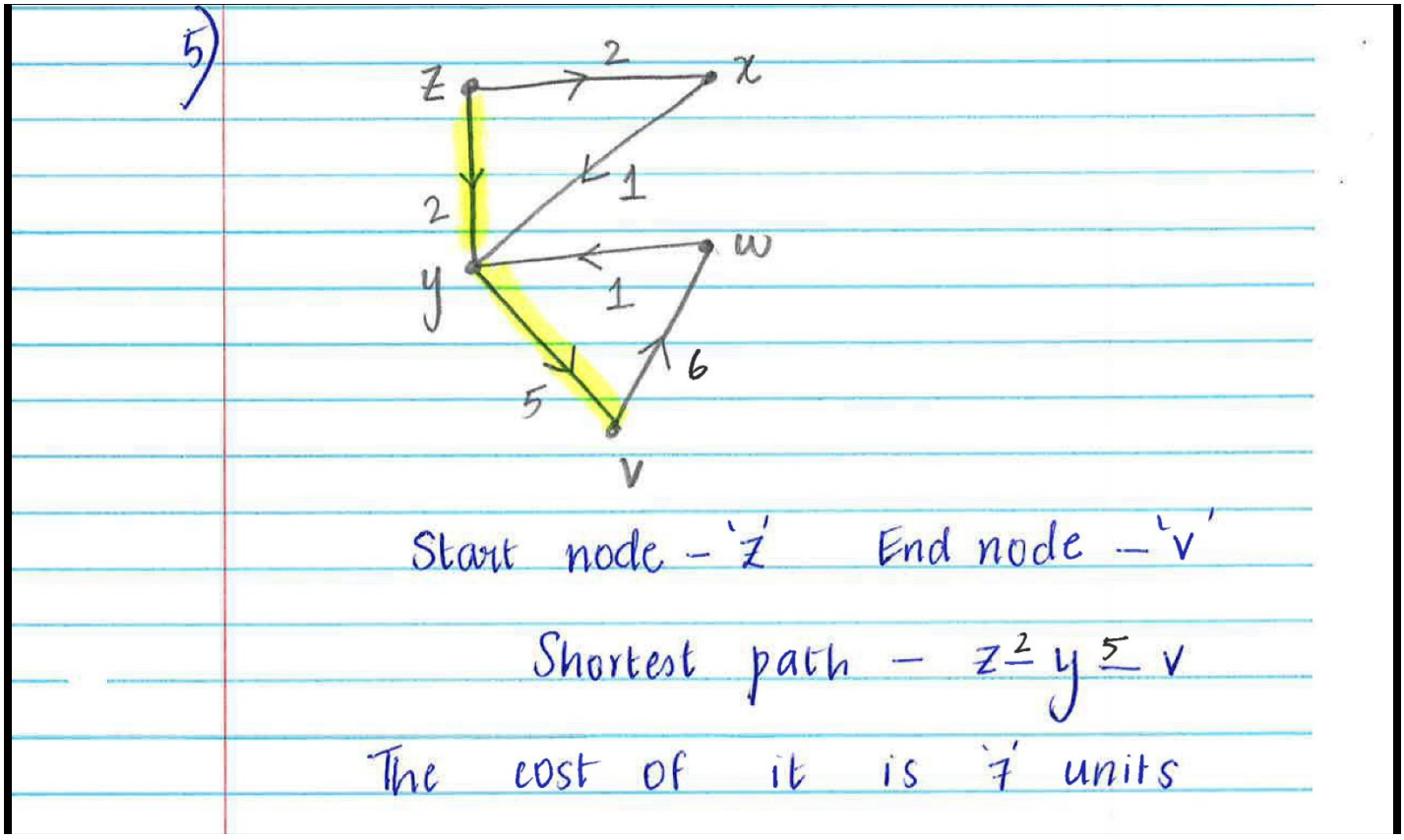


Result 4 from the program

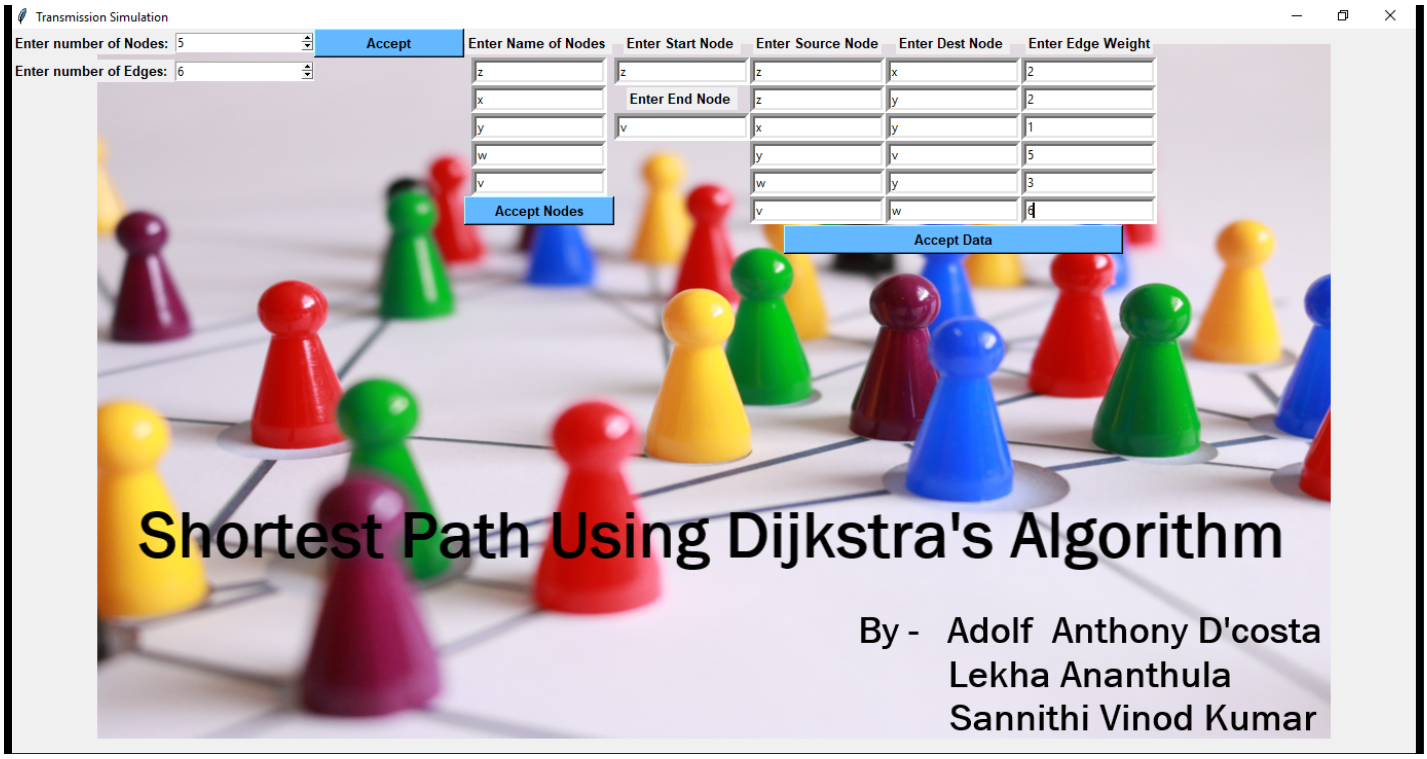
```
C:\Users\Adolf\AppData\Local\Programs\Python\Python37-32\
python.exe "C:/Users/Adolf/Desktop/Maths Project/Maths
Project.py"
[{'path': ['a', 'c', 'b', 'd', 'e'], 'weight': 7}, {'path
': ['a', 'c', 'b', 'e'], 'weight': 13}, {'path': ['a', 'b
', 'd', 'e'], 'weight': 9}, {'path': ['a', 'b', 'e'], '
weight': 15}]
```

PRACTICAL EXAMPLE 5

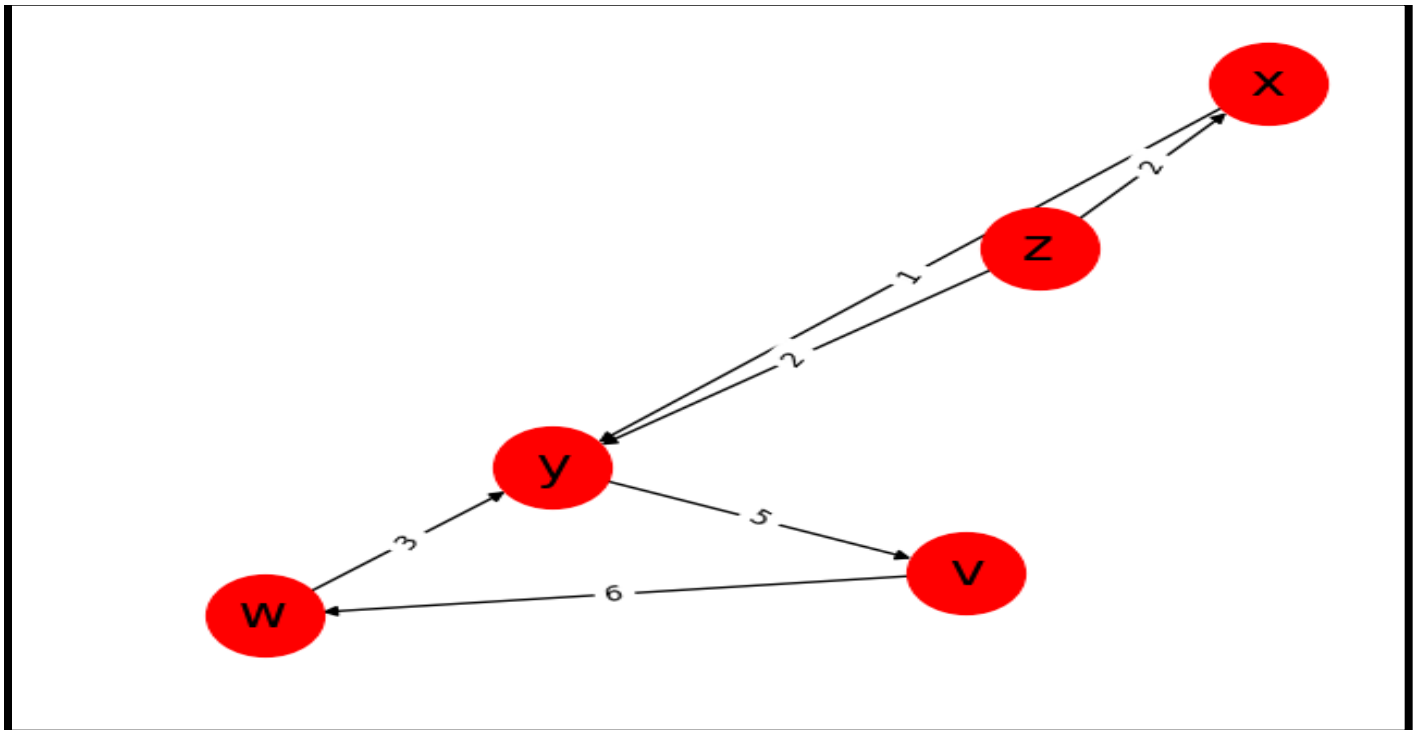
THEORETICALLY EXPLAINED



Problem 5 solved using code



Graph 5 plotted by the program



Result 5 from the program

```
C:\Users\Adolf\AppData\Local\Programs\Python\Python37-32\
python.exe "C:/Users/Adolf/Desktop/Maths Project/Maths
Project.py"
[{'path': ['z', 'x', 'y', 'v'], 'weight': 8}, {'path': ['z
', 'y', 'v'], 'weight': 7}]
```

EXTERNAL LINKS

- Oral history interview with Edsger W. Dijkstra https://conservancy.umn.edu/handle/11299/107247
- Implementation of Dijkstra's algorithm using TDD, http://blog.cleancoder.com/uncle-bob/2016/10/26/DijkstrasAlg.html
- Graphical explanation of Dijkstra's algorithm step-by-step on an example http://www.gilles-bertrand.com/2014/03/disjkstra-algorithm-description-shortest-path-pseudo-code-data-structure-example-image.html
- Shimbel, A. (1955). *Structure in communication nets*. Proceedings of the Symposium on Information Networks. New York, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn. pp. 199–203
- Bellman, Richard (1958). "On a routing problem". *Quarterly of Applied Mathematics*. **16**: 87–90 https://mathscinet.ams.org/mathscinet-getitem?mr=0102435
- Paper P-923. Santa Monica, California: RAND Corporation. http://www.rand.org/pubs/papers/P923.html
- Analytic Algorithmics and Combinatorics (ANALCO12), Kyoto, Japan. pp. 41–47 http://adsabs.harvard.edu/abs/2011arXiv1111.5414B
- Clear visual A* explanation, with advice and thoughts on path-finding. http://theory.stanford.edu/~amitp/GameProgramming/
- Variation on A* called Hierarchical Path-Finding A* (HPA*). https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf

REFERNCES

[i] Frana , Philip (Aug 10)" An interview with Edger Dijkstra ".communications of the ACM. https://dl.acm.org/citation.cfm?doid=1787234.1787249

[ii] The ARMAC Unsung Heroes in Dutch Computing History. 2007. Archived from the original http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html

[iii] Frana , Philip (Aug 10)" An interview with Edger Dijkstra ".communications of the ACM. 41-47 https://dl.acm.org/citation.cfm?doid=1787234.1787249

[iv] Dijkstra, Edsger W., https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD841a.PDF

[v] Tarjan , Robert Endre (1983), *Data Structures and Network Algorithms*, CBMS_NSF Regional Conference Series in Applied Mathematics, **44**, Society for Industrial and Applied Mathematics, p. 75, "The third classical minimum spanning tree algorithm was discovered by Jarník and rediscovered by Prim and Dikstra; it is commonly known as Prim's algorithm."

[vi] Prim, R.C. (1957). "Shortest connection networks and some generalizations" (PDF). *Bell System Technical Journal*. **36**: 1389–1401. doi:10.1002/j.1538-7305.1957.tb01515.x. Archived (PDF) from the original on 18 July 2017.

[vii] V. Jarník: *O jistém problému minimálním* [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti, 6, 1930, pp. 57–63. (in Czech)

[viii] Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm". *Digraphs: Theory, Algorithms and Applications* (First ed.). ISBN 978-1-84800-997-4.

[ix] Bang-Jensen, Jørgen; Gutin, Gregory (2000). "Section 2.3.4: The Bellman-Ford-Moore algorithm" http://www.cs.rhul.ac.uk/books/dbook/

[x] Kleinberg, Jon; Tardos, Éva (2006). *Algorithm Design*. New York: Pearson Education, Inc.

[xi] https://www.programiz.com/dsa/bellman-ford-algorithm

[xii] Robert Floyd (1962 June) "Algorithm 97: Shortest Path". *Communications of the ACM*. **5** (6): 345. doi:10.1145/367766.368168

[xiii] Roy, Bernard (1959). "Transitivité et connexité". *C. R. Acad. Sci. Paris*. **249**: 216–218

[xiv] Warshall, Stephen (January 1962). "A theorem on Boolean matrices". *Journal of the ACM*. **9** (1): 11–12. doi:10.1145/321105.321107

[xv] Weisstein, Eric W. "Floyd-Warshall Algorithm". *MathWorld*

[xvi] Kleene, S. C. (1956). "Representation of events in nerve nets and finite automata". In C. E. Shannon and J. McCarthy. *Automata Studies*. Princeton University Press. pp. 3–42.

[xvii] Ingerman, Peter Z. (November 1962). "Algorithm 141: Path Matrix". *Communications of the ACM*. **5** (11): 556. doi:10.1145/368996.369016

[xviii] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8. See in particular Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565 and Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.

[xix] Kenneth H. Rosen (2003). Discrete Mathematics and Its Applications, 5th Edition. Addison Wesley. ISBN 978-0-07-119881-3

[xx] Prim, R. C. (November 1957), "Shortest connection networks And some generalizations", *Bell System Technical Journal*, **36** (6): 1389–1401, doi:10.1002/j.1538-7305.1957.tb01515.x

[xxi] Tarjan, Robert Endre (1983), "Chapter 6. Minimum spanning trees. 6.2. Three classical algorithms", *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, **44**, Society for Industrial and Applied Mathematics, pp. 72–77.

[xxii] Kepner, Jeremy; Gilbert, John (2011), *Graph Algorithms in the Language of Linear Algebra*, Software, Environments, and Tools, **22**, Society for Industrial and Applied Mathematics, p. 55, ISBN 9780898719901

[xxiii] Doran, J. E.; Michie, D. (1966-09-20) http://rspa.royalsocietypublishing.org/content/294/1437/235

xxiv Cormen, Thomas; Charles E Leiserson, Ronald L Rivest, Clifford Stein (2009). *Introduction To Algorithms* (Third ed.). MIT Press. p. 631. ISBN 0262258102.

[xxv] Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". *Proceedings of the American Mathematical Society*. **7**: 48–50. doi:10.1090/S0002-9939-1956-0078686-7

[xxvi] "OSPF Incremental SPF"

[xxvii] Richards, Hamilton. "Edsger Wybe Dijkstra". *A.M. Turing Award*. Association for Computing Machinery. Retrieved October 16, 2017. “At the Mathematical Centre a major project was building the ARMAC computer. For its official inauguration in 1956, Dijkstra devised a program to solve a problem interesting to a nontechnical audience: Given a network of roads connecting cities, what is the shortest route between two designated cities?”

[xxviii] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs" (PDF). *Numerische Mathematik*. **1**: 269–271. doi:10.1007/BF01386390

[xxix] Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" (PDF). *Algorithms and Data Structures: The Basic Toolbox*. Springer. doi:10.1007/978-3-540-77978-0