Draft version November 15, 2023
Typeset using LaTeX default style in AASTeX631

Explanation of Dijkstra's shortest path algorithm*

Nguyen Quang Hung,[1] Vu Minh Nhat,[1] Dinh Dao Lan Vy,[1] Duong Phuong Giang,[1] Nguyen Tue Anh,[1] and Nguyen Gia Bao[1]

[1] *National Economic University*

ABSTRACT

We embark on exploring a practical application of the shortest path-finding algorithm. Widely used in routing and network management, finding the shortest path is a crucial problem in computer science and programming. We will construct a piece of programming utilizing a common graph algorithm to determine the shortest path between two points on a map or graph.

The program not only illustrates the real-world applicability of the algorithm but it also provides a profound insight into how graph algorithms can be integrated into practical applications to address routing issues. Navigating through the intricacies of graph theory, we'll delve into the implementation of a path-finding algorithm, shedding light on the underlying principles and mechanisms that power modern routing systems. With the flexibility of Manim library by 3Blue1Brown on Python, we aim to not only to have functional implementation but also to foster a comprehensive understanding of how algorithms play a pivotal role in solving real-world routing challenges by visualizing the algorithm in videos.

Keywords: Algorithm — Graph — Shortest Path — Visualization — Dijkstra — Maps

1. INTRODUCTION

Dijkstra's shortest path algorithm, a fundamental method in graph theory and computer science, plays a pivotal role in finding the shortest path between two nodes in a graph. As a critical component of many real-world applications, such as network routing and transportation planning, understanding Dijkstra's algorithm is crucial for both students and professionals in the field.

This paper aims to provide a comprehensive exploration of Dijkstra's algorithm through effective visualization techniques. By employing visual aids, we aim to enhance the clarity of the algorithm's mechanics and illustrate its step-by-step progression. Visual representations not only facilitate a better understanding of the algorithm's inner workings but also contribute to a more engaging and accessible learning experience.

In the following sections, we will first discuss existing approach before the appearance of Dijkstra's algorithm and its break-points, as well as tools that we use for the visualization. After that, we will delve into the core principles of Dijkstra's algorithm, breaking down its steps and intricacies. Real-life application of Dijkstra algorithm is also being discussed in Section 4 of this paper.

2. LITERATURE REVIEW

The problem of finding the shortest path is one of the most famous problems in computer science. There have been numerous approaches for the solutions, and each of them have unique advantages and disadvantages. Here we will briefly talk about general information and two basic solutions and go into the most important algorithm in the following section.

2.1. *Time and Space Complexity*

Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the size of its input. It provides an estimate of the maximum amount of time an algorithm will require to run based on the input

* Released on November, 15nd, 2023

size. When analyzing the time complexity of an algorithm, the focus is on the dominant term that contributes the most to the growth rate as the input size becomes large. This dominant term is what is used in the Big O notation. Big O notation describes the upper bound of the growth rate of an algorithm's running time in the worst-case scenario.

Some common time complexities and their notion:

1. $O(1)$: Constant Time Complexity
2. $O(log(n))$: Logarithmic Time Complexity
3. $O(n)$: Linear Time Complexity
4. $O(nlog(n)$: Linearithmic Time Complexity
5. $O(n^2)$: Quadratic Time Complexity
6. $O(2^n)$: Exponential Time Complexity
7. $O(n!)$: Factorial Time Complexity

Space complexity is a measure of the amount of memory space an algorithm requires to execute, expressed as a function of the input size. It evaluates the efficiency of an algorithm in managing memory resources. Space complexity analysis focuses on the amount of additional memory required by an algorithm, excluding the input space. It considers variables, data structures, and recursive call stacks that contribute to the overall memory usage. It notation is similar to Time complexity.

In this paper, for time and space complexity analysis, we will denote:

1. V: the number or vertices.
2. E: the number or edges between two nodes.

2.2. *Brute Force*

Most problems have the simplest solution, which is the Brute Force algorithm, which means you list all the possibilities that can happen and find the optimal solution among those possibilities. By using Depth First Search (DFS) approach, every possible path connected between 2 nodes will be calculated and store its minimum value. The **Algorithm 1** pseudocode is one of many example for the Brute Force approach.

2.3. *Bellman-Ford*

The Bellman-Ford algorithm (3) can be used to find the shortest starting path from a vertex $s \in V$ in the case of graph $G = (V, E, w)$ without negative cycles. This algorithm is quite simple: Initialize the distance labels $d[s] = 0$ and $d[v] = +\infty$, $\forall v \neq s$, then perform contraction along every arc of the graph. Just keep repeating that until no more label $d[v]$ can be minimized. More detail of the algorithm will be discusses in Appendix B

The pseudocode of Bellman-Ford **Algorithm 2**

2.4. *Manim library*

The Manim (1) library, created by a youtuber 3Blue1Brown is a library that support the visualization of math formula, graph, chart with various and comprehensive animation. There have been numerous projects or research that use Manim as a tool for explanation or education purpose. The youtube channel of 3Blue1Brown has thousands of interesting videos, explanation of mathematical concepts with Manim.

In comparison with other tools avaliable, we found Manim suit our need the most. Manim provides the flexibility to do different types of seamless animation, from basic animation such as moving or scaling element to morphing one elements to others with only a few lines of codes.

Even with the popularity of Manim in creating animation, we believe that it still have several "pain-points" in order to master the tools. First, Manim have a stunning capability to manifest the animation between Mobject (an object represent element in Manim), the document itself does not describe all the amaze it provides. Lots of our animations in this project were experienced by ourselves and others sources, which just a fraction of them were properly documented.

Algorithm 1: Shortest Path using DFS

```
Function ShortestPath($G, start, end$):
    $visited \leftarrow$ empty set
    $path \leftarrow$ empty list
    $shortestPath \leftarrow$ empty list
    Function DFS($currentNode$):
        Mark $currentNode$ as visited
        Append $currentNode$ to $path$
        if $currentNode = end$ then
            if $shortestPath$ is empty or length of $path$ < length of $shortestPath$ then
                $shortestPath \leftarrow$ copy of $path$
            end
            else
                for each neighbor $nextNode$ of $currentNode$ do
                    if $nextNode$ is not in $visited$ then
                        DFS($nextNode$)
                    end
                end
            end
        end
        Remove $currentNode$ from $path$
        DFS($start$)
    return $shortestPath$
```

Algorithm 2: Bellman-Ford Algorithm

```
Data: Graph $G$, source node $s$
Result: Shortest distances from $s$ to all other nodes
Initialization;
for each node $v$ in $G$ do
    $dist[v] \leftarrow \infty$;
    $visited[v] \leftarrow$ false;
end
$dist[s] \leftarrow 0$;
for $i \leftarrow 1$ to $|V(G)| - 1$ do
    for each vertex $v$ and $weight(u, v)$ in $E(G)[u]$ do
        if $distance[u] \neq \infty$ and $distance[u] + weight(u, v) < distance[v]$ then
            $distance[v] \leftarrow distance[u] + \text{weight}(u, v)$
        end
    end
end
for $i \leftarrow 1$ to $|V(G)| - 1$ do
    for each vertex $v$ and $weight(u, v)$ in $E(G)[u]$ do
        if $distance[u] \neq \infty$ and $distance[u] + weight(u, v) < distance[v]$ then
            return None
        end
    end
end
```

Secondly, unlike standard python library, Manim require a specific configuration before usable, which was not 'user-friendly' for beginner. Therefore, we choose to use Google Colab instead of local machine for the universal environment in Colab's vitual environment.

For visualization, we will use Manim to show nodes, directions and weights between them, store and illustrate how the elements and scores change after every iteration. The detail of illustration will be discussed further in the Section 3.4.

3. METHOLOGY

Bellman-Ford algorithm has to check all the given node and weight, therefore having the the time complexity of $O(VE)$. However, there is another algorithm named Dijkstra's algorithm can solve the similar problem with the time complexity of just $O((V + E) * log(V))$.

Dijkstra's algorithm (2) is a popular algorithm in computer science and graph theory for finding the shortest path between nodes in a graph, particularly in graphs with non-negative edge weights. The algorithm was conceived by computer scientist Edsger Dijkstra in 1956.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes 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

3.1. *Algorithm*

The step-by-step description of Dijkstra's algorithm:

1. Initialization:
   (a) Assign a tentative distance value to every node. Initialize the source node's distance to 0 and all other nodes' distances to infinity.
   (b) Create a set of unvisited nodes. Initially, this set contains all nodes.
2. Selection of the Current Node:
   (a) Set the current node to the source node.
3. Evaluation of Neighbors:
   (a) For each neighbor of the current node that is still in the set of unvisited nodes.
   (b) Calculate the tentative distance from the source node to the neighbor through the current node.
   (c) Compare the tentative distance to the current known distance for that neighbor.
   (d) If the tentative distance is shorter, update the neighbor's distance.
4. Mark the Current Node as Visited:
   (a) Once all neighbors have been evaluated, mark the current node as visited. It is now considered permanently labeled.
   (b) Select the Unvisited Node with the Shortest Distance:
   (c) Choose the unvisited node with the smallest tentative distance as the next "current node" and go back to step 3.
5. Repeat:
   (a) Repeat steps 2-4 until all nodes are visited or the destination node is reached.

The pseudocode of Original Dijkstra's Algorithm is at **Algorithm 3**

3.2. *Dijkstra's Algorithm using Priority Queue (A)*

Dijkstra'a Algorithm can be implemented with Priority Queue. With the power of Heap structure, priority queue can always sort all the elements in either increasing or decreasing order with the time complexity of $O(log(n)$ for every addition of new node. Therefore, extracting unvisited node with the minimum distance $dist[u]$ can be as easy as take the first element in the priority queue which cost only $O(1)$.

This algorithm will follow these **4 main steps**:

1. Enqueue Source Vertex:
   (a) Enqueue the source vertex with distance 0 into the priority queue.

Algorithm 3: Original Dijkstra's Algorithm

```
Data: Graph $G$, source node $s$
Result: Shortest distances from $s$ to all other nodes
Initialization;
for each node $v$ in $G$ do
    $dist[v] \leftarrow \infty$;
    $visited[v] \leftarrow$ false;
end
$dist[s] \leftarrow 0$;
while there are unvisited nodes do
    $u \leftarrow$ unvisited node with the minimum distance $dist[u]$;
    Mark $u$ as visited;
    for each neighbor $v$ of $u$ do
        if $dist[u] + weight(u, v) < dist[v]$ then
            $dist[v] \leftarrow dist[u] + \text{weight}(u, v)$;
        end
    end
end
```

2. While Priority Queue is Not Empty:
   (a) Dequeue the vertex with the smallest tentative distance from the priority queue.
3. For Each Neighbor of the Dequeued Vertex:
   (a) Calculate the tentative distance from the source to the neighbor through the dequeued vertex.
   (b) If the calculated distance is smaller than the current tentative distance, update the distance and enqueue the neighbor into the priority queue.
4. Repeat:
   (a) Continue the process until the priority queue is empty or the destination vertex is reached.

The pseudocode in **Algorithm** 4 demonstrate how Dijkstra's Algorithm works with priority queue

3.3. *Complexity of algorithm*

The algorithm have the overall time complexity of $O((V + E) * log(V))$ and space complexity $O(V + E)$
It's worth noting that if a dense graph is used, where E is close to V^2, and an adjacency matrix is employed, the time complexity may be $O((V^2) * log(V))$ and space complexity may be $O(V^2)$ due to the adjacency matrix itself. However, in practice, Dijkstra's algorithm is often applied to sparse graphs where E is proportional to V, and the time space complexity is dominated by $O((V + E) * log(V))$ and $O(V + E)$, respectively.

3.4. *Structure of the visualization*

Manim has a built-in function for visualization on graphs. However, we decided to build-up the graph from scratch, so we can easily adjust the graph, build and modify custom animation for the visualization **The main structure** of the visualization process.

1. Points and Paths construction:
   (a) class Point: Define a point layer to represent the vertices of the graph on video.
   (b) class DijkstraGraph: Create graphs and related elements such as points and lines.
2. CustomGraph Scene:
   (a) Graph Initialization: Create and display the initial graph. This include creating dots (nodes), the labels of the nodes, lines between nodes, the weight between two nodes and its scores.

Algorithm 4: Dijkstra's Algorithm with Priority Queue

```
Data: Graph $G$, source node $s$
Result: Shortest distances from $s$ to all other nodes
Initialization;
for each node $v$ in $G$ do
    $dist[v] \leftarrow \infty$;
    $visited[v] \leftarrow$ false;
end
$dist[s] \leftarrow 0$;
PriorityQueue $PQ \leftarrow$ initialize with $(s, 0)$;
while $PQ$ is not empty do
    $(u, d) \leftarrow PQ$.dequeue();
    if $visited[u]$ then
        continue;
    end
    Mark $u$ as visited;
    for each neighbor $v$ of $u$ do
        if $dist[u] + weight(u, v) < dist[v]$ then
            $dist[v] \leftarrow dist[u] + \text{weight}(u, v)$;
            $PQ$.enqueue($(v, dist[v])$);
        end
    end
end
```

(b) Effect Functions: Define vertex and line lighting, coloring and blinking effects.

(c) Iterate Node Function: Run a loop to scan points, update the table and pass in specific animation.

(d) Visualize Priority Queue Function: Display the priority queue and the elements in it.

(e) Visualizing Update: Pop out current values during one sub-iteration, making comparison and update the scores.

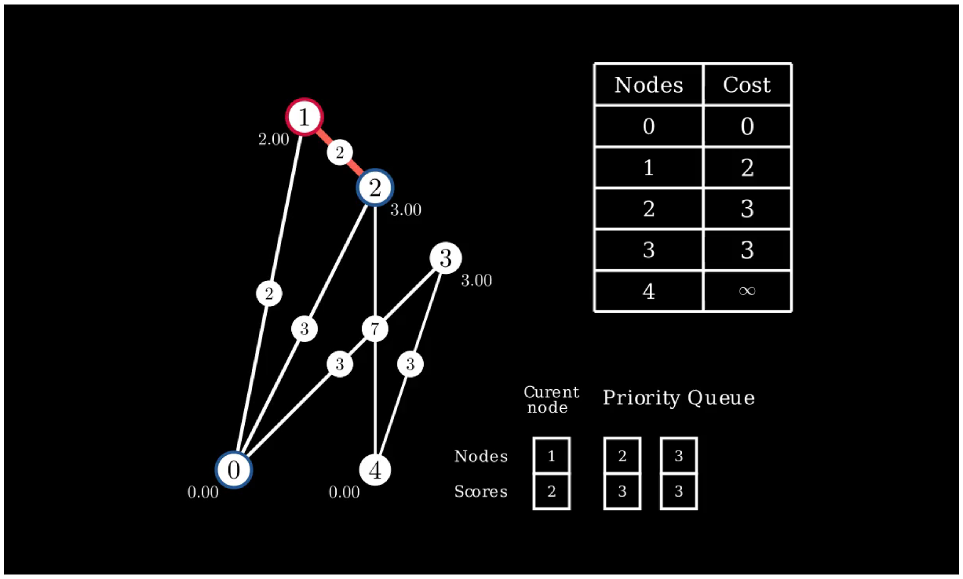
3. Inheritation from CustomGraph class:

    (a) Create another animation of the graph by just rewrite the method $node_init$ with specific configuration about background, nodes and direction of the nodes (either a directed graph or undirected graph)

The visualization is constructed parallelly with the process of Dijkstra's algorithm, and is demonstrated in **Figure** 1. Firstly, the undirected graph will be displayed first with some vertices fading in on the screen, in this case, 5 nodes from *0* to *4*, with lines and weights (which are pink-coloured) between each node. The costs of each node will be set as default, with the cost of node *0* is *0* and that of the others is ∞. Secondly, the initial table will be displayed, with two headers are respectively *Nodes* and *Cost*. At this point, the cost of all nodes will be set as *0* and the visualization of the Priority Queue will appear : two columns named *Priority Queue* and *Current Node*, two rows named *Score* and *Nodes*, and boxes containing numbers that represent those values. From now on, the visualization will be rendered alongside with the process of the Dijkstra algorithm.

As the iterate node function runs a loop to scan through each node, the *current node* will be red-circled and its *neighbor nodes* will be blue-circled. If the lines' color is blue, it represents the possibility of choosing a path from one node to another. If the lines' color is red, it means that we are analyzing the cost of this path. When the graph updates the cost at its node, the table also updates the values of the *Cost* column that equal to the costs of the graph. At the same time, the priority queue display will be updated: If the calculated cost is smaller than the current cost, updating the cost and enqueuing the node into the priority queue (in the video, the node will be added at the right-end of the queue). Consequently, dequeuing the node out of the priority queue (the node at the *current* node' area will disappear in the video, replaced by the node next to it) and continuing the process until the priority queue is empty. The video ends when the algorithm ends.

4. REAL-WORD APPLICATION



Nodes	Cost
0	0
1	2
2	3
3	3
4	∞

	Curent node	Priority Queue	
Nodes	1	2	3
Scores	2	3	3

Figure 1. Example of Dijkstra's Shortest path Algorithm visualization with Manim

In the ever-evolving social landscape, traffic management and optimization have become crucial challenges. The problem of finding the shortest path on a congested road poses significant challenges for transportation systems and traffic management. In reality, road congestion can result from various factors such as traffic jams due to increased vehicles, construction projects, or emergency events. To address this issue, the Dijkstra algorithm is a flexible and efficient choice for determining the shortest path between two points on a congested road. This is essential for improving the quality of life and enhancing the flexibility of transportation systems. Google Maps (4) is one the most popular navigating service, allowing users to find locations, show real-time traffic condition and shortest path between two locations on the maps by using Dijkstra algorithm and A* algorithm (an algorithm similar to Dijkstra).

4.1. *Building the Graph*

To simulate a congested road, we utilize a weighted graph, where vertices represent points on the road, and edges represent road segments. The weight of an edge indicates the distance between two points. **Figure 2** is an example for mapping the points

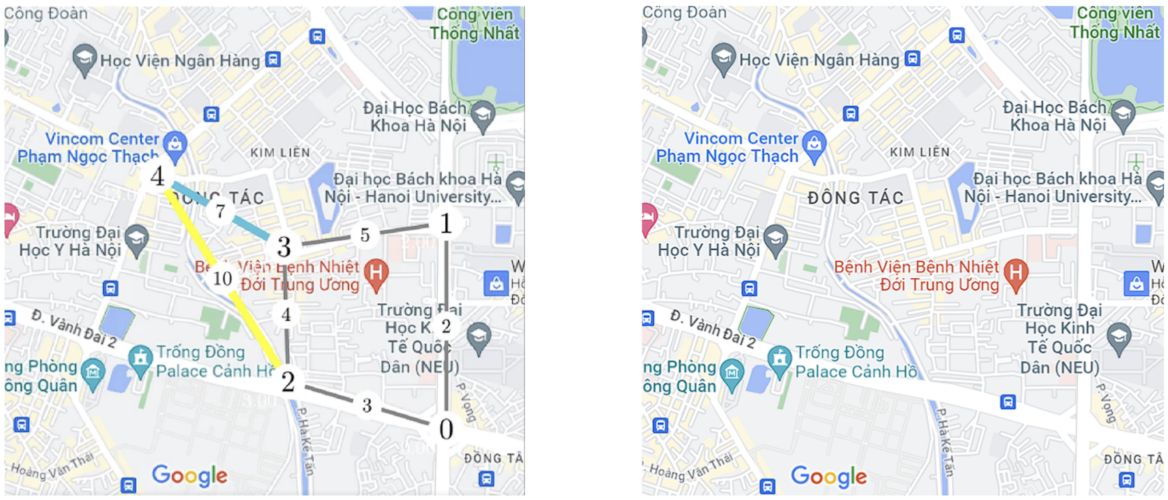

Công Đoàn
Công viên Thống Nhất
Học Viện Ngân Hàng
Đại Học Bách Khoa Hà Nội
Vincom Center Phạm Ngọc Thạch
KIM LIÊN
ĐỒNG TÁC
Đại học Bách khoa Hà Nội - Hanoi University...
Trường Đại Học Y Hà Nội
Bệnh Viện Bệnh Nhiệt Đới Trung Ương
Trường Đại Học Kinh Tế Quốc Dân (NEU)
Đ. Vành Đai 2
Trống Đồng Palace Cảnh Hồ
Google
P. Hà Kế Tấn
P. Vọng
Hoàng Văn Thái
0
1
2
3
4
5
7
10

Figure 2. Example of mapping nodes for Dijkstra Algorithm in real map

Based on a map from the National Economics University to Vincom Pham Ngoc Thach in **figure 2**, there are five points:

- Point 0: National Economics University **(starting point)**
- Point 1: Phuong Mai intersection intersecting Giai Phong
- Point 2: Truong Chinh Street intersecting Riverside Road
- Point 3: Phuong Mai intersection intersecting Luong Dinh Cua
- Point 4: Vincom Pham Ngoc Thach **(destination point)**

We have the formula $s = v * t$
Assuming a motorcycle travels with a speed of $v = 20km/h$ on a clear road, it covers distances (*weights*):

- Point 0 $\rightarrow$ Point 1: $500m = 0.5km$ ($t = 1.5$ minutes)
- Point 0 $\rightarrow$ Point 2: $300m = 0.3km$ ($t = 0.9$ minutes)
- Point 1 $\rightarrow$ Point 3: $450m = 0.4km$ ($t = 1.2$ minutes)
- Point 2 $\rightarrow$ Point 3: $600m = 0.6km$ ($t = 1.8$ minutes)
- Point 2 $\rightarrow$ Point 4: $1000m = 1km$ ($t = 3$ minutes)
- Point 3 $\rightarrow$ Point 4: $500m = 0.5km$ ($t = 1.5$ minutes)

Applying Dijkstra's algorithm to find the shortest path, we can observe that the shortest path from Point 0 to Point 4 is: $0 \rightarrow 2 \rightarrow 4$, which cost only 3.9 minutes
Other paths:

- $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 : 1450$ ($t = 4.2$ minutes)
- $0 \rightarrow 2 \rightarrow 4 : 1300$ ($t = 3.9$ minutes)
- $0 \rightarrow 2 \rightarrow 3 \rightarrow 4 : 1400$ ($t = 4.2$ minutes)

However, travel time on a road segment is not always a constant. If the travel time from Point 2 to Point 4 increases due to traffic congestion (e.g., from 3 minutes to 9 minutes), then the distance from Point 2 to Point 4 will change. In this scenario, the shortest path from the starting point to the destination is recalculated:

- Point 0 - Point 1: $500m = 0.5km$ ($t = 1.5$ minutes)
- Point 0 - Point 2: $300m = 0.3km$
- Point 1 - Point 3: $450m = 0.4km$
- Point 2 - Point 3: $600m = 0.6km$
- Point 2 - Point 4: $3km$ ($t = 9$ minutes)
- Point 3 - Point 4: $500m$

Hence, the shortest path from the starting point to the destination changes to $0 \rightarrow 2 \rightarrow 3 \rightarrow 4$, which took 4.2 minutes when the road segment from Point 2 to Point 4 is congested.

4.2. *Conclusion*

The problem of finding the shortest path on a congested road requires a combination of algorithmic techniques and in-depth understanding of traffic conditions. Utilizing the Dijkstra algorithm not only provides an efficient solution but also opens up new avenues for research and development in the field of traffic management and transportation. Integrating real-time data and smart transportation systems can offer a more comprehensive solution, contributing to the improvement of performance and safety in the transportation system in the future.

In 2021, Google Map's algorithm has changed to Graph Neural Network (GNN) (5) for faster and more accurate at evaluating the traffic condition, using GNN architecture to find and predict shortest paths, rather than mathematical algorithm. However, Dijkstra algorithm still play an important role in the development of optimizing the complexity of algorithms in daily life.

5. CONCLUSION

In conclusion, Dijkstra Algorithm play an importance role in not just computer scientific field but also in daily life. Through the power of visualization, we've demystified the algorithm, providing an engaging, real-time representation of its step-by-step execution. The dynamic interplay of colors and animations has not only made the learning experience accessible but has also bridged the gap between theoretical understanding and practical implementation, encouraging both novices and seasoned practitioners to carry the newfound insights into their coding endeavors, fostering a deeper appreciation for the efficiency and elegance of Dijkstra's algorithm in solving real-world shortest path problems.

REFERENCES

[1]Eertmans, J. *'Manim slides: A python package for presenting manimcontent anywhere'*. (2023) , Journal of Open Source Education, 6(66), p. 206. doi:10.21105/jose.00206.

[2]Edsger W. Dijkstra, Laurent Beauguitte, Marion Maisonobe. E.W. Dijkstra *'A Note on Two Problems in Connexion with Graphs.'* 1959, Numerische Mathematik 1, p. 269271 Version bilingue et commentée. 2021. ffhal-03171590f

[3]Bellman R *'On a routing problem'* Q Appl Math 16(1):87–90

[4]Mehta, H., Kanani, P. and Lande, P. *'Google maps'* (2019) International Journal of Computer Applications, 178(8), pp. 41–46. doi:10.5120/ijca2019918791.

[5]Derrow-Pinion, A. et al. *'Eta prediction with graph neural networks in Google Maps'* (2021), Proceedings of the 30th ACM International Conference on Information amp;amp; Knowledge Management [Preprint]. doi:10.1145/3459637.3481916.

APPENDIX

A. PRIORITY QUEUE

A.1. *Heap*

A heap is a tree-like data structure that forms a complete tree and satisfies the heap invariant. The heap invariant states that, if A is a parent node of B, then A is ordered with respect to B for all nodes A and B in a heap. This means a parent node's value is always greater than or equal to the value of the child nodes for all nodes in a heap. Or the value of the parent node is less than or equal to the value of the child node for all nodes in a heap.

Additionally, there are two types of heap data structures, termed Max Heap and Min heap. The Max heap is a tree-like structure in which the parent node's value is greater than the value of the child node. The diagram given below represents a binary max heap having the highest value at its root node.

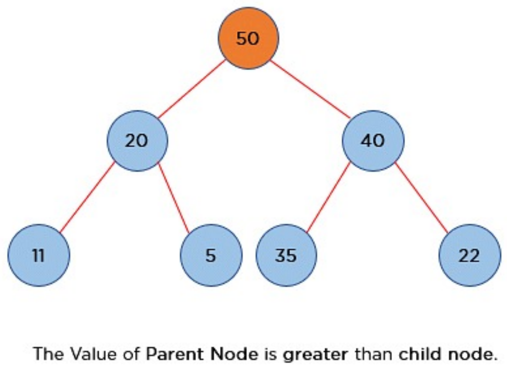


Figure 3. An Example of Max heap

The Min heap is a tree-like structure in which the parent node's value is smaller than the value of the child node. The tree diagram given below shows a binary heap tree having the smallest value at its root node.

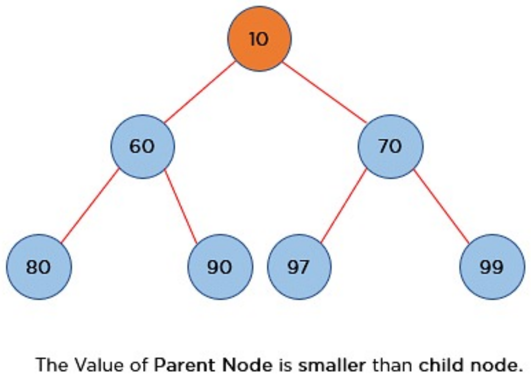


Figure 4. An Example Image of Min heap

A.2. *Priority Queue*

In computer science, a priority queue is an abstract data-type similar to a regular queue or stack data structure. Each element in a priority queue has an associated priority. In a priority queue, elements with high priority are served before elements with low priority. In some implementations, if two elements have the same priority, they are served in the same order in which they were enqueued. In other implementations, the order of elements with the same priority is undefined.

While priority queues are often implemented using heaps, they are conceptually distinct from heaps. A priority queue is an abstract data structure like a list or a map; just as a list can be implemented with a linked list or with an array, a priority queue can be implemented with a heap or another method such as an unordered array.

B. BELLMAN FORD

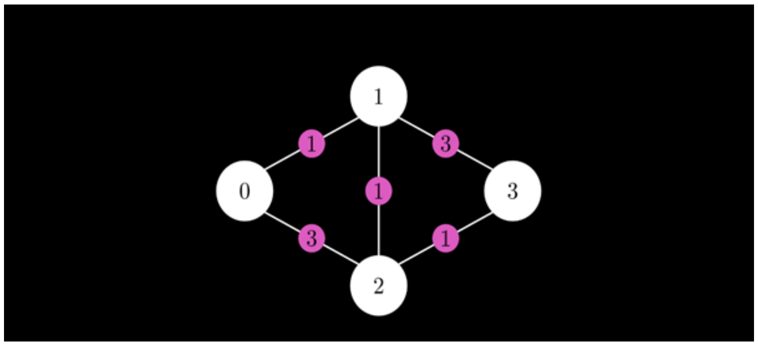


Figure 5. An Example Image of Bellman Ford

In the given example, let's consider the edge [0, 1]. We observe that $d[1] = d[0] + w[0, 1] < d[1] = \infty$ initially, so we update $d[1] = 1$. Moving on to examine the edge [0, 2], we see that $d[2] = d[0] + w[0, 2] < d[2] = \infty$, so we update $d[2] = 3$. We continue this process until none of the labels can be updated anymore. In this way, $d[1]$ represents the shortest path from 0 to 2, $d[2]$ from 0 to 2, and $d[3]$ from 0 to 3.

For backtracking, we can introduce a scaling factor. For instance, for $d[3]$, the scaling factor is 2, indicating that the shortest path to 3 must pass through 2. Similarly, for $d[2]$, the scaling factor is 1, and for $d[0]$, there is no scaling factor. This process is repeated until none of the labels can be updated further.

B.1. *Animate Bellman-Ford*

1. Display a graph with vertices and weighted edges.
2. Display the Bellman-Ford algorithm.
3. Display a table with squares and corresponding numbers to represent the algorithm.
4. Compute and display the steps of the algorithm, simultaneously applying effects on the graph.
5. Display the final result of the algorithm.

B.2. *Handling Negative Cycles in Shortest Path Algorithms*

In the analysis of the provided example, it becomes evident that continuously traversing through vertices such as 3, 4, 5 results in a path with a total weight of -1. Consequently, with a sufficient number of iterations, a path shorter than the original one can always be found. This poses a challenge for Dijkstra's algorithm, as its fundamental principle is to always choose the vertex with the smallest label (i.e., the nearest vertex from the source) to determine the best path.

When a negative cycle is present, this principle no longer holds true, as a path with a negative cycle can continuously decrease its value. As a result, the algorithm may loop infinitely, testing paths with decreasing costs. However, Bellman-Ford can handle this situation. After traversing all vertices and edges in the first iteration, it repeats the

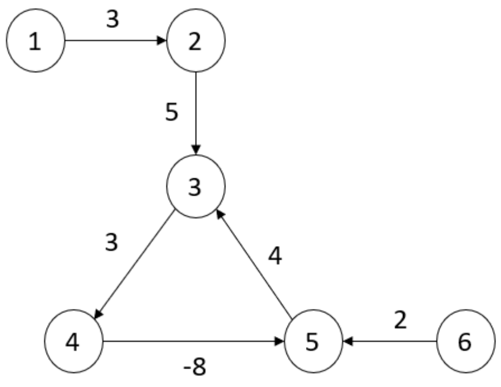


Figure 6. An Example Image of Handling Negative Cycles

process in subsequent iterations. If nothing changes, it indicates the presence of a shortest path. On the other hand, if the shortest path continues to change, it implies the graph has a negative cycle, and there is no shortest path.

Bellman-Ford iterates a finite number of times, specifically equal to the number of edges multiplied by the number of vertices. This ensures that the algorithm can always stop, providing a reliable mechanism for handling negative cycles.