Dec 2025, India

# *Dijkstra's Algorithm*

Tutorial

Tutorial By

***Vikas Awadhiya***

Invented By

# Edsger W. Dijkstra

Tutorial By

***Vikas Awadhiya***

LinkedIn profile: https://in.linkedin.com/in/awadhiya-vikas

# 1 Introduction

Dijkstra's algorithm finds shortest path in a weighted graph. The algorithm finds the shortest path from a given source vertex to all other vertices in a graph. Dijkstra's algorithm can also be customized to find the shortest path from a given source vertex to a specific destination vertex by ending the process once the destination vertex is discovered.

The Dijkstra's algorithm doesn't find the shortest path directly rather find the shortest distance from source vertex to an individual vertex and the shortest path of a vertex can be found by backtracking the predecessor up to source vertex.

The Dijkstra's algorithm works with simple graph, multi-graph, connected graph, disconnected graph and all possible combination of mentioned graph variants with undirected and directed weighted graphs for example directed/undirected simple graph, directed/undirected multi-graph. The only constraint is, a graph must not have any edges with negative weights.

The time complexity of Dijkstra's shortest path algorithm is $O((V + E) \log V)$ where binary heap data structure is used to find the vertex at smallest distance among the undiscovered vertices. Here $V$ is number of vertices $|V|$, and $E$ is number of edges $|E|$ in a graph.

Dijkstra's algorithm belongs to a specific category of algorithms called greedy algorithms.

# 2 The Algorithm

Dijkstra's algorithm finds the shortest path from a given source vertex to every other vertex of a graph $G = (V, E, \omega)$, where $V$ is finite set of vertices, $E$ is finite set of edges and $\omega$ is a weight function where $\omega : E \to \mathbb{R}^+$ or simply $\forall e \in E,\ \omega(e) \geq 0$ ( here $\omega$ is a Greek alphabet omega in lower case, $\to$ means "map to" / "is a function from", $\mathbb{R}^+$ is real positive numbers, $\forall$ is read as "for all"/ "for every", $\in$ is read as "element of"/"belongs to" and $e$ represents an edge ).

In simple words Dijkstra's algorithms works with a weighted graph only if weights of all the edges are positive as expressed by expression $\forall e \in E,\ \omega(e) \geq 0$ and algorithm can't find shortest path in a weighted graph that has any edges with negative weights.

Dijkstra's algorithm can find the shortest path in a simple graph (doesn't have self-loop and multiple/parallel edges between two vertices), in a multi-graph (has self-loop and parallel edges between two vertices) as shown below in fig 1 and fig 2 respectively.

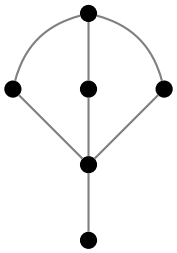

Fig 1 : Simple Graph

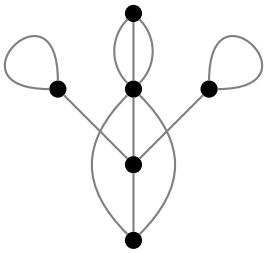

Fig 2 : Multi Graph

It also works with connected graph and disconnected graph (has more than one component) as shown below in fig 3 and fig 4 respectively

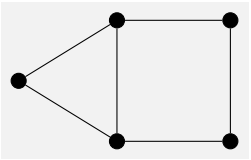

Fig 3 : Connected Graph

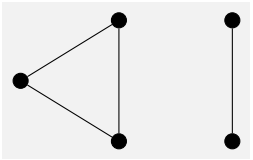

Fig 4 : Disconnected Graph

It works with both directed and undirected weighted graphs with all possible combinations of graph variants mentioned above. In this document a graph means a finite graph.

The last thing remains to explain before Dijkstra's algorithm explanation begins is, what is the shortest path in a weighted graph? To understand shortest path, it is first required to understand what is a path in a graph?

In a graph a walk $W$ is sequences of vertices and the edges where a vertex and an edge may occur more than once, it means a walk may visit a vertex and an edge multiple time. A graph is shown in fig 5 below and a walk $W = v_0, e_0, v_2, e_6, v_4, e_1, v_0, e_0, v_2, e_3, v_1$ is shown in fig 6 is highlighted by bold lines. The walk $W$ visits vertices $v_0$, $v_2$ and an edge $e_0$ twice.

When a walk $W$ has distinct vertices and edges or in other words if a walk visits any vertex and edge only once, $W$ is called a path (if only edges are distinct, $W$ is called a trail).

Weight of a path is sum of weights of all its edges and the shortest path from vertex $u$ to vertex $v$ is a path that has smallest weight compare to the weight of any other alternative path between these vertices.

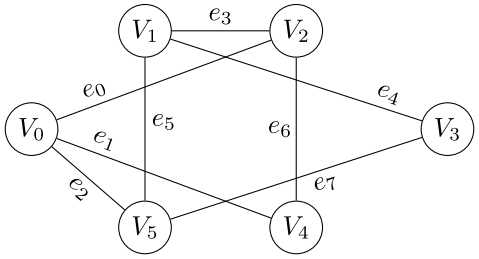


Fig 5

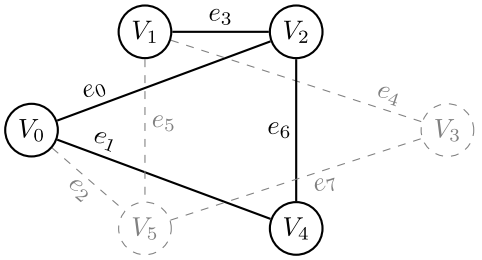


Fig 6

In a graph shown in fig 7 below on the left, the shortest path from vertex u to vertex v is highlighted by bold lines in fig 8 below, which has smallest weight.

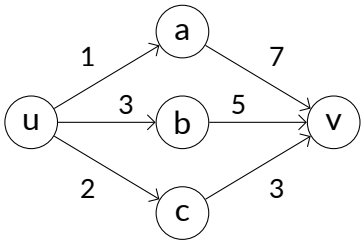


Fig 7

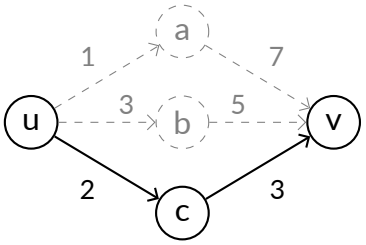


Fig 8

## 2.1 Algorithm's Steps

Dijkstra's algorithm on a graph $G$ with a given source vertex $u$ can be viewed as following steps,

1. Weight of source vertex $u$ is set to zero, because it is a starting vertex and weight of all other vertices are set to infinity ($\infty$) because initially shortest path is not known to any vertex.
2. Select a vertex of smallest weight (initially source vertex has smallest weight zero).
3. Reevaluate the weights of neighboring vertices (vertices directly connected to this vertex). It is also called relaxation, if the reevaluated weight of a neighboring vertex from this newly discovered vertex is less than its previously known weight then weight is updated and this vertex is set as predecessor vertex of neighboring vertex.
4. Go back to step 2 until all the vertices are not discovered (all the vertices of the component to which source vertex belongs).

In a weighted graph, weight may represent distance, time, cost or any measurable quantity and in the discussion of shortest path, term "distance" is most-often used compared to "weight" because distance is naturally associated with a path and a shortest path means shortest distance.

To understand these steps let's consider a graph as an example, shown in fig 9 below. The vertices of a graph are usually labeled with numeric values but for batter explanation English alphabets are used this clearly distinguishes vertices from distances.

The undirected weighted graph shown in fig 9 below is a simple graph and in a simple graph $\{u, v\}$ uniquely represents an edge between any vertex $u$ and vertex $v$ and $\omega(\mathrm{uv})$ represents its weight/distance.

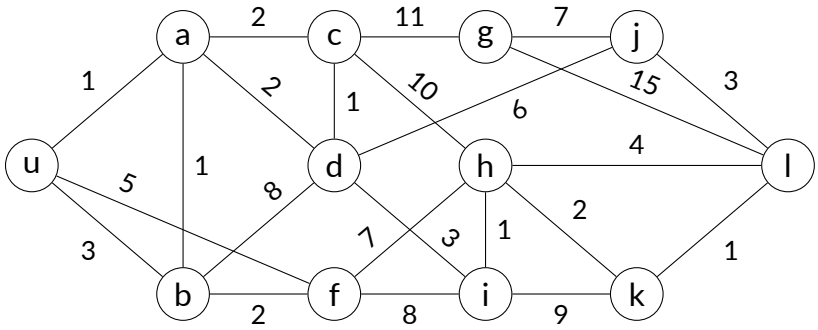


Fig 9 : Graph $G$

As a first step, distance to source vertex $u$ is set to zero and distance to other vertices are set to infinity as shown in fig 10, below on the left in bold letters.

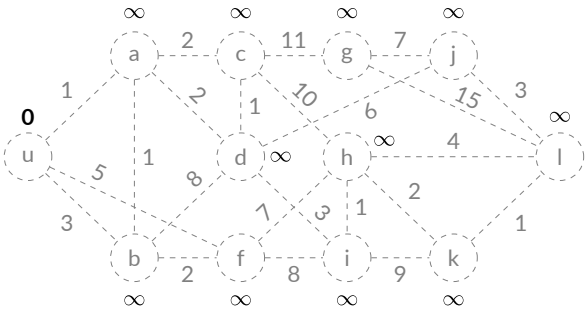

Fig 10

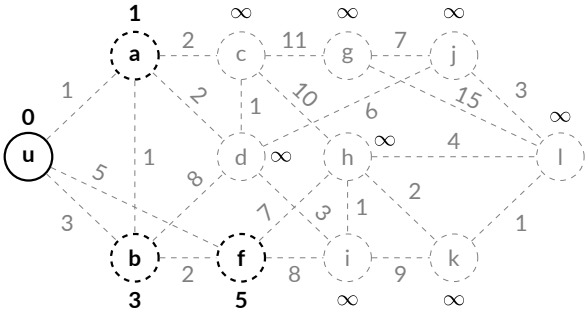

Fig 11

In each step a vertex with shortest distance is selected or discovered then the distances to its neighboring vertices are reevaluated to see if distance to the individual neighboring vertices decreases from this newly discovered vertex, it is also called distance relaxation.

Now vertex $u$ with smallest distance zero is selected/discovered. The neighboring vertex $a$ is directly reachable by an edge $\{u, a\}$ from vertex $u$ in unit distance and the overall distance is less than already known distance to vertex $a$, $\text{distance}[u] + \omega(\text{ua}) = 0 + 1 = 1 < \infty$, hence its distance is relaxed. Similarly distances to other neighboring vertices $b$ and $f$ are relaxed as highlighted in bold dashed boundary line in fig 11 above on the right.

Now the vertex with smallest distance among the undiscovered vertices is vertex $a$. So, vertex $a$ is selected/discovered as shown in fig 12 below on the left. Vertex $a$ is connected with vertex $u$ in the shortest path as highlighted by bold line because distance to vertex $a$ last relaxed when vertex $u$ was discovered hence vertex $u$ is the predecessor of vertex $a$.

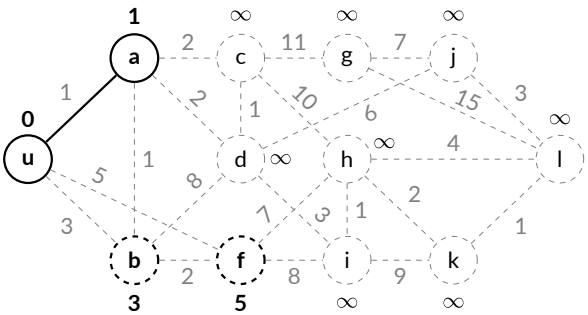

Fig 12

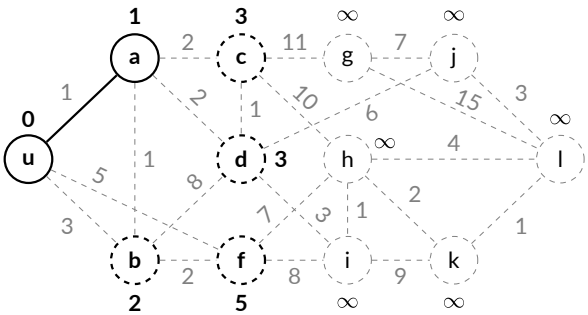

Fig 13

Vertex $b$ is reachable from vertex $a$ by an edge $\{a, b\}$ of unit distance and after re-evaluation

distance becomes, $\text{distance}[a] + \omega(\text{ab}) = 1 + 1 = 2 < 3$, and it is less than known distance to vertex $b$. So, the distance to vertex $b$ is relaxed again and the predecessor is also changed to vertex $a$. Similarly, the distances to other neighboring vertices $c$ and $d$ are relaxed as highlighted in bold dashed boundary lines in fig 13 above on the right.

Now vertex $b$ with smallest distance among the undiscovered vertices and it is selected/ discovered as shown in fig 14 below on the left. The distance to vertex $d$ reevaluated to $\text{distance}[b] + \omega(\text{bd}) = 2 + 8 = 10$ but it is greater than known distance $3$ to vertex $d$, so its distance remains unchanged or in other words distance to vertex $d$ is not relaxed further. Distance to other neighboring vertex $f$ is reevaluated to $\text{distance}[b] + \omega(\text{bf}) = 2 + 2 = 4 < 5$, so distance is relaxed to $4$ and predecessor changed to vertex $b$ as shown in fig 15 below on the right.

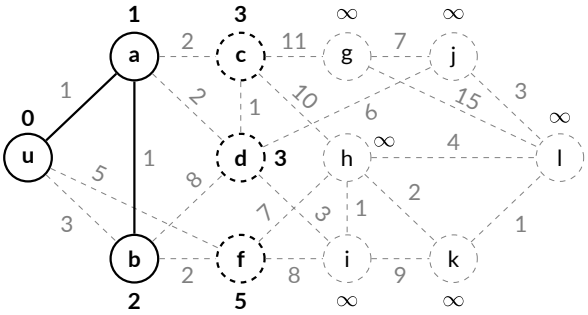


Fig 14

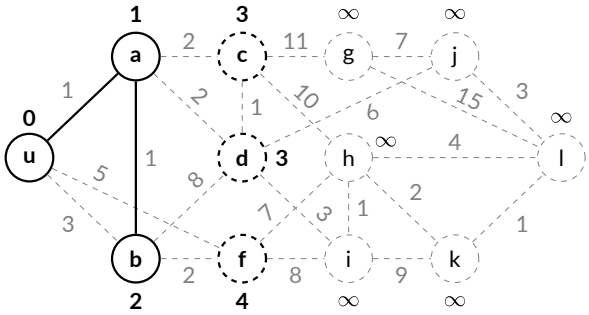


Fig 15

At this moment there are two vertices $c$ and $d$ have the smallest distance among the undiscovered vertices and any one can be selected. Let's consider vertex $c$ is discovered and connected to its predecessor vertex $a$ in the shortest path as shown in fig 16 below on the left. Vertex $a$ is predecessor because the distance of vertex $c$ last relaxed when vertex a was discovered.

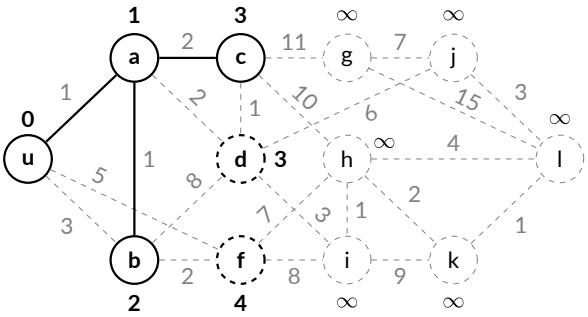


Fig 16

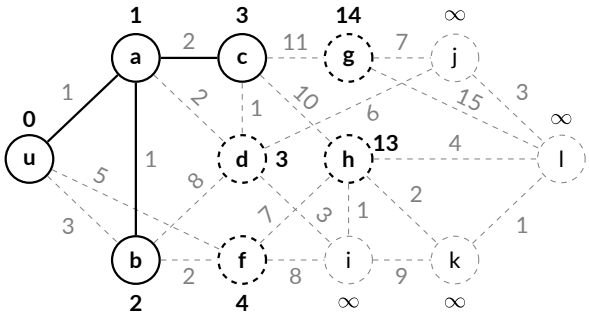


Fig 17

The distances to neighboring vertices $g$ and $h$ are relaxed and reevaluated to $14$ and $13$ respectively as highlighted in bold dashed lines in fig 17 above on the right, but the distance to neighboring vertex $d$ remains unchanged because $\text{distance}[c] + \omega(\text{cd}) = 3 + 1 = 4 > 3$.

Now vertex $d$ has smallest distance among undiscovered vertices and it is selected next as shown in fig 18 below on the left. Distances to two neighboring vertices are relaxed, distance to vertex $j$ is relaxed to $\text{distance}[d] + \omega(\text{dj}) = 3 + 6 = 9 < \infty$, and distance to vertex $i$ is relaxed to $\text{distance}[d] + \omega(\text{di}) = 3 + 3 = 6 < \infty$, as highlighted by bold dashed lines in fig 19 below on the right.

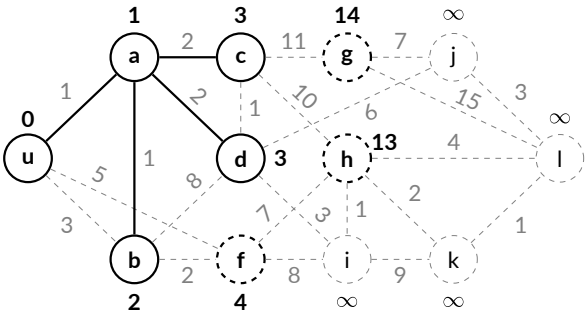


Fig 18

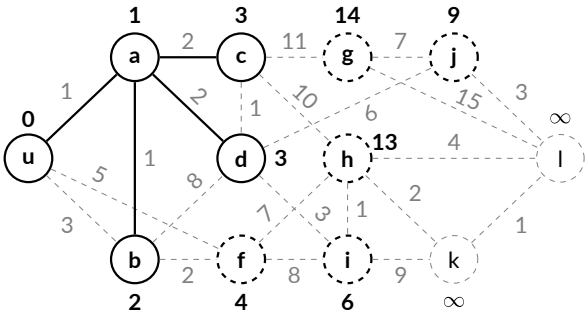


Fig 19

Now vertex $f$ has smallest distance among undiscovered vertices and it is discovered next as shown in fig 20 below on the left. It is connected to vertex $b$ its predecessor in the shortest path because distance of vertex $f$ was last relaxed when vertex $b$ was discovered.

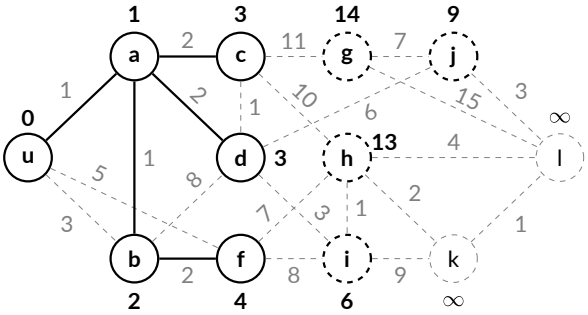


Fig 20

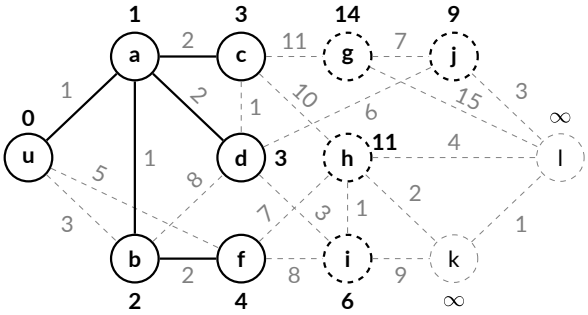


Fig 21

The distance to only neighboring vertex $h$ is relaxed as shown in fig 21 above on the right. Now vertex $i$ is selected/discovered as a vertex with the smallest distance among undiscovered vertices as shown in fig 22 below on the left.

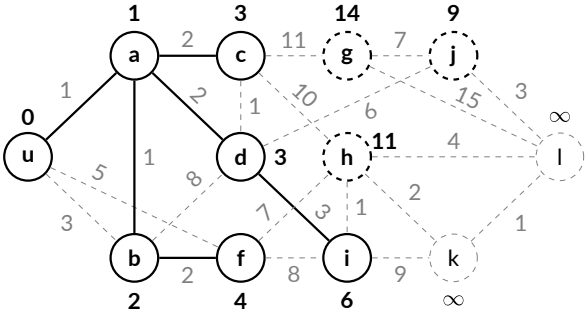


Fig 22

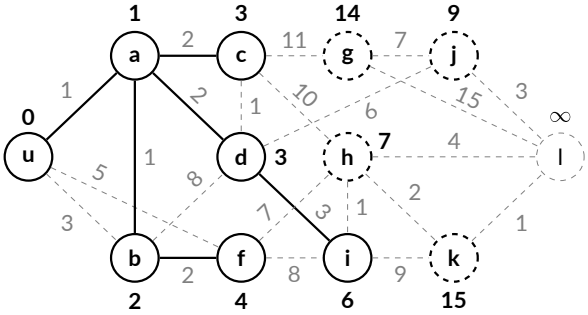


Fig 23

The distance to vertex h is further relaxed to $\text{distance}[i] + \omega(\text{ih}) = 6 + 1 = 7 < 11$ and vertex $i$ is set as predecessor. The distance to other neighboring vertex $k$ is reevaluated to $\text{distance}[i] + \omega(\text{ik}) = 6 + 9 = 15 < \infty$, as shown in fig 23 above on the right.

Now vertex $h$ is selected/discovered as the vertex with the smallest distance among undiscovered vertices as shown in fig 24 below on the left. The distance to neighboring vertex $k$ is relaxed again to $\text{distance}[h] + \omega(\text{hk}) = 7 + 2 = 9 < 15$, and vertex $h$ is set as predecessor. The distance to other neighboring vertex $l$ is relaxed and reevaluates to $\text{distance}[h] + \omega(\text{hl}) = 7 + 4 = 11 < \infty$, as shown in fig 25 below on the right.

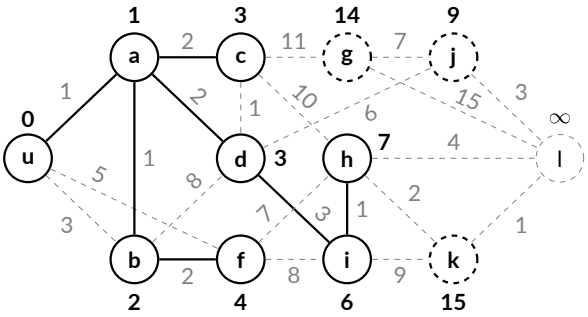


Fig 24

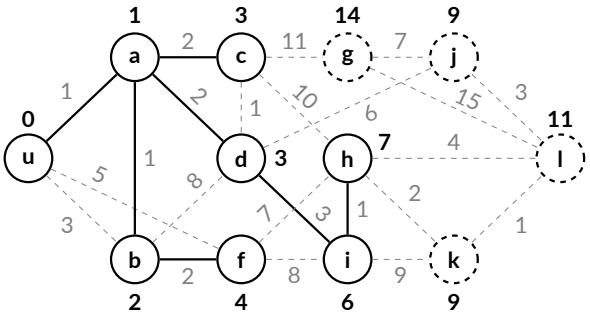


Fig 25

At this moment two vertices, vertex $j$ and vertex $k$ have smallest distance among undiscovered vertices and anyone can be selected. Let's consider vertex $j$ is selected/discovered as shown in fig 26 below on the left. It is connected to vertex $d$ in the shortest path because its distance last relaxed when vertex $d$ was discovered.

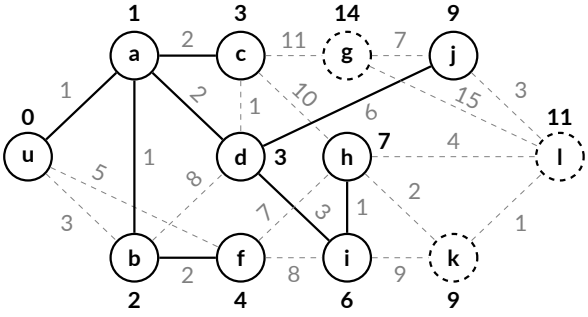


Fig 26

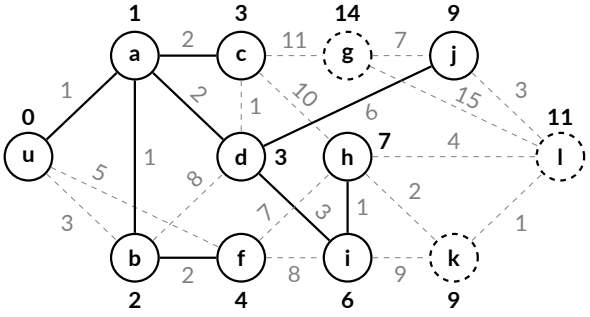


Fig 27

There is no distances relaxation to neighboring vertices of vertex $j$, hence fig 26 and fig 27 are identical.

Now vertex $k$ is selected/discovered as a vertex with smallest distance among the undiscovered vertices as shown in fig 28 below on the left. It is connected to vertex $h$ in the shortest path because its distance last relaxed when vertex $h$ was discovered.

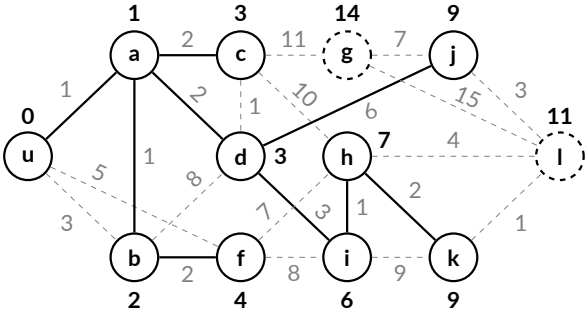


Fig 28

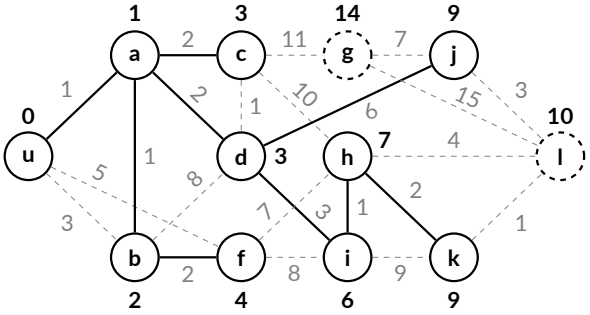


Fig 29

Distance of neighboring vertex $l$ is relaxed further and reevaluated to $\text{distance}[k] + \omega(\text{kl}) = 9 + 1 = 10 < 11$ and vertex $k$ is set as predecessor as shown in fig 29 above on the right.

Now only two undiscovered vertices remain and the vertex $l$ with smallest distance among them is selected/discovered as shown in fig 30 below on the left.

Vertex $l$ has only one undiscovered neighboring vertex $g$ and no further relaxation happens to its distance, $\text{distance}[l] + \omega(\text{lg}) = 10 + 15 = 25 > 14$, that's why fig 30 and fig 31 are identical.

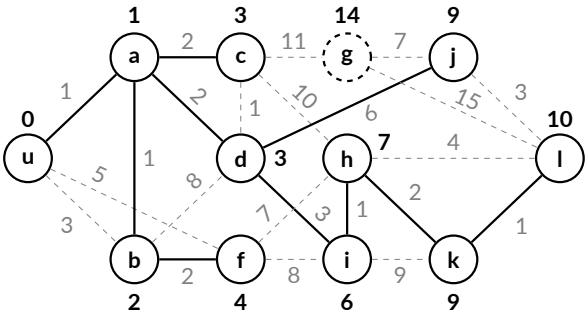


Fig 30

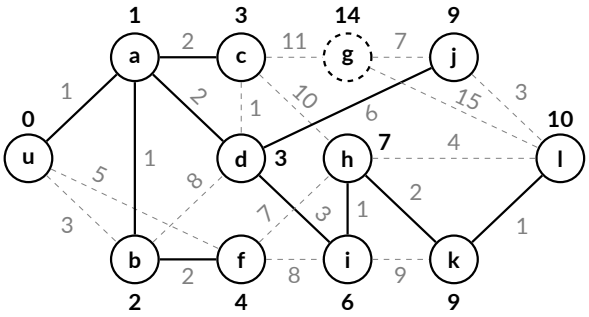


Fig 31

Now vertex $g$ is selected/discovered as a last remains undiscovered vertex and it is connected to vertex $c$ in the shortest path as shown in fig 32 below. Vertex $g$ is connected to vertex $c$ because vertex $c$ is its predecessor and its distance last relaxed when vertex $c$ was discovered.

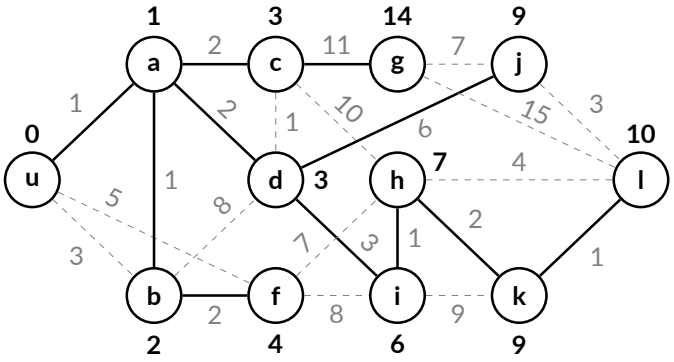


Fig 32

Algorithm ends here and the shortest paths to all vertices from source vertex $u$ are found.

In implementation the max value of a value type (for example unsigned int or unsigned long) is used to represents infinite value. The binary min heap (priority queue) data structure is usually used to find the vertex with smallest distance among the undiscovered vertices.

The time complexity of Dijkstra's algorithm where binary min-heap is used to find the vertex with smallest distance, is $O((V + E) \log V)$, here $V$ is number of vertices $|V|$, and $E$ is number of edges $|E|$ in a graph.

# 3 Complexity Analysis

The complete graphs $K_n$ help to understand time and space complexity of Dijkstra's algorithm. A complete graph $K_n$ is a graph where each vertex is neighbor of every other vertex or in other words each vertex is directly connected to every other vertex and has degree $d(V-1)$ (number of incident edges). The binary heap (min heap) data-structure is generally used to implement Dijkstra's algorithm and it requires $O(\log n)$ time complexity for insert(push) and remove(pop) operations.

Algorithm begins with source vertex and relaxes the distance of all other $V-1$ vertices (because each vertex is directly connected with every other vertex). The min-heap is populated in $O(V)$ time with $V-1$ entries where each entry represents the relaxed distance of vertex from source vertex. The linear time complexity of initial step is non dominating and next steps define the complexity of algorithm.

Now min-heap/priority-queue has $V-1$ edges and an insert or a remove operation requires $O(\log V)$ time complexity. Big $O$ complexity analysis represents worst scenario of asymptotic analysis and ignores the constant.

Algorithm in each step removes top element (vertex) of min-heap and insert $V-1$ entries in min-heap as result of distance relaxation. In practice, $V-1$ distance relaxations don't happen in each step because number of selected vertices increases as algorithm proceeds but to analyze the worst scenario, $V-1$ distance relaxations are assumed in each step.

To pop an element require $O(\log V)$ time and to push $V-1$ elements in heap requires $O(V \log V)$ time, and the time complexity of a step is,

Time Complexity of a step = complexity to pop an element + complexity to push $V-1$ entries
= $O(\log V) + O(V \log V)$

Algorithm performs this step $V-1$ times to select all vertices and the overall time complexity can be found by multiply the $O(V)$ with the time complexity of a single step as follows,

Time Complexity = $V \times$ Time complexity of a step
= $O(V \ \log V) + O(V^2 \log V)$
But in a complete undirected graph, edges $E = V(V-1) * \frac{1}{2} = O(V^2)$, constant ignored,
= $O((V+E) \log V)$

This is the most common form of time complexity of Dijkstra's algorithm and it is applicable to all types of graphs and highlights the distance relaxations of vertices are bound to $E$.

The worst scenario considers, insertion of $V-1$ entries in each step but the push/pop operation of min-heap still requires $O(\log V)$ time because of power property of $\log$ function. At each step $V-1$ entries are inserted and with $V-1$ steps, in worst case min-heap has $V \times V = V^2$ entries (ignoring contains) but $\log_x(y^n) = n \log_x y$ and time complexity of push/pop operation remains same to $O(\log(V^2)) = O(2 \log V) = O(\log V)$ and constant $2$ ignored.

The space complexity of Dijkstra's algorithm when adjacency list is used to represent the sparse graph is $O(V+E)$ that is combine of $O(V)$ space for vertices in graph representation and another $O(V)$ for priority queue's entries but $O(2V) = O(V)$ and $O(E)$ space for edges but in dense graph like complete graph space complexity is $O(V^2)$ because of $E = O(V^2)$.

# 4 Negative Weight Constraint

A graph with edges of negative weight may have negative weight cycles and a negative weight cycle has sum of the weights of it's edges less then zero as shown in fig 33 below on the left.

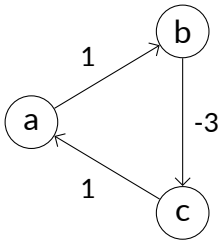


Fig 33 : Negative weight cycle

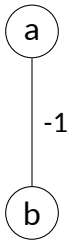


Fig 34

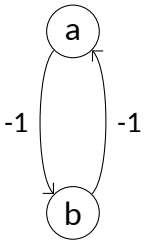


Fig 35

There is no shortest path solution exist for a graph that has a negative weight cycle. In the graph shown in fig 33 above if vertex $a$ is consider as source vertex then the distance of vertices $b$ and $c$ decrease with each loop of cycle, an endless process that's why there is no shortest path.

It means a shortest path can not be found for an undirected graph that has edges of negative weight because an undirected edge is used in both directions and form a negative weight cycle. An undirected graph shown in fig 34 above in middle is equivalent of a directed graph with negative weight cycle shown in fig 35 above on the right.

The Dijkstra's algorithm doesn't work with edges of negative weight even the directed graph doesn't have negative weight cycles because distance of a selected/discovered vertex can not be relaxed further and Dijkstra's algorithm stops once the destination vertex is discovered.

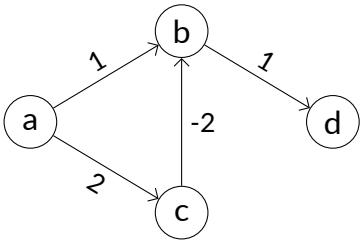


Fig 36

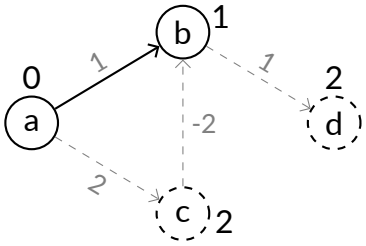


Fig 37

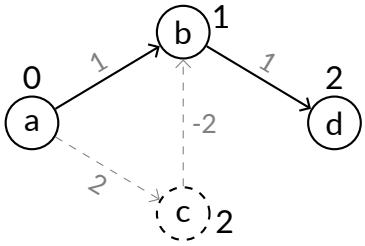


Fig 38

Let's consider a graph shown in fig 36 above on the left and vertex $a$ as source vertex and vertex $d$ as destination vertex. Algorithm begins with selecting the source vertex and relaxes the distances of vertex $b$ and vertex $c$ then it selects vertex $b$ and relaxes the distance of vertex $d$. Now vertex $d$ has smaller distance from source and algorithm selects it but it is the destination vertex and it's discovery stops the algorithm and algorithm never process remaining vertex $c$.

To work in this case algorithm can not stop at the discovery of vertex $d$ to let the vertex $c$ to be discovered and then it is also required to allow the distance relaxation of already discovered vertex $b$. Algorithm uses select status to mark the selected vertices (that is not must with a graph of only positive weight edges but improve performance by avoiding distance reevaluation) this status mechanism can be removed and algorithm also requires to wait for all vertices to be discovered.

But with these changes, algorithm will no longer be the Dijkstra's algorithm and will become an inefficient procedure.

# 5 Implementation

The C++ implementation of the Dijkstra's algorithm is provided under the MIT License. It also contains the `main.cpp` file to demonstrate its usage.

Code available at: https://github.com/vikasawadhiya/Dijkstra-Algorithm

December 2025, India