Jun 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

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.

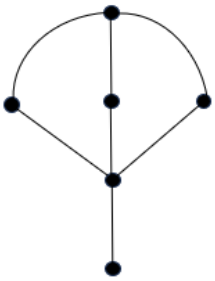
Dijkstra's algorithm belongs to a specific category of algorithms called greedy algorithms.

Dijkstra's Algorithm

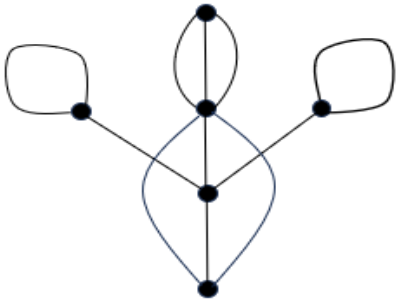
Dijkstra's algorithm finds the shortest path from a given source vertex to every other vertex of a graph G = (V, E, ω), where V is finite set of vertices, E is finite set of edges and ω is weight function where $\omega: E \to R^+$ or simply $\forall e \in E,\ \omega(e) \geq 0$ (here ω is Greek letter omega in lowercase, → means "maps to"/ "is a function from", R^+ is positive real number, ∀ is read as "for all"/ "for every" and ∈ 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 1.0 and fig 1.1 respectively.

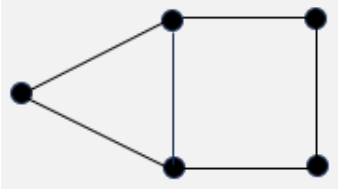


Simple Graph
Fig 1.0

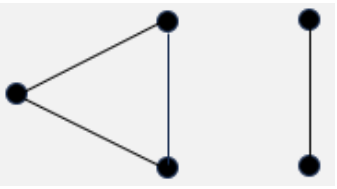


Multi Graph
Fig 1.1

It also works with connected graph and disconnected graph (has more than one component) as shown below in fig 2.0 and fig 2.1 respectively.



Connected Graph
Fig 2.0



Disconnected Graph
Fig 2.1

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 3.0 below and a walk W = $\underline{v_0}$, $\underline{e_0}$, $\underline{v_2}$, e_6, v_4, e_1, $\underline{v_0}$, $\underline{e_0}$, $\underline{v_2}$, e_3, v_1 is shown in fig 3.1 is highlighted by bold lines. The walk W visits vertices v_0, v_2 and an edge e_0 twice.

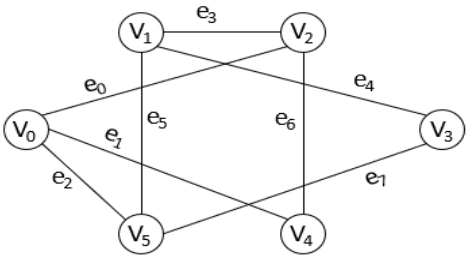


Fig 3.0

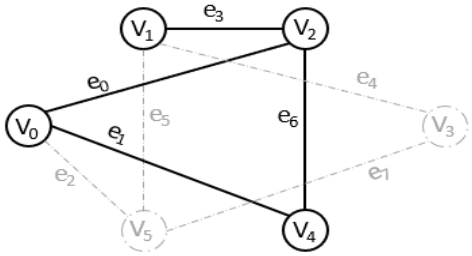


Fig 3.1

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. In a graph shown in fig 4.0 below on the left, the shortest path from vertex u to vertex v is highlighted by bold lines in fig 4.1 below, which has smallest weight.

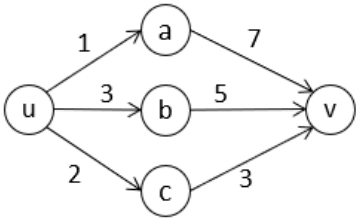


Fig 4.0

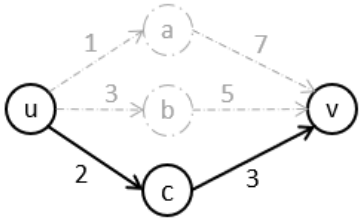


Fig 4.1

The Algorithm

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 (∞) 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 neighbouring vertices (vertices directly connected to this vertex). It is also called relaxation, if the reevaluated weight of a neighbouring 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 neighbouring 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 5.0 below. The vertices of a graph are usually labelled with numeric values but for batter explanation English alphabets are used this clearly distinguishes vertices from distances.

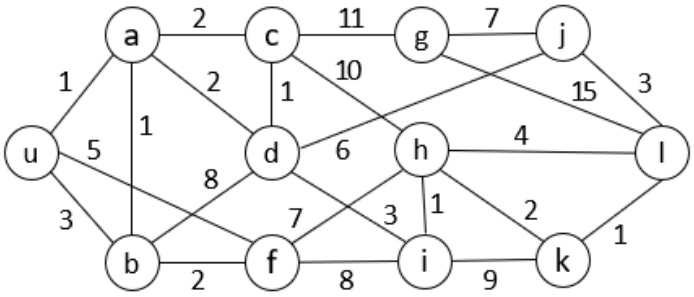


Fig 5.0

The undirected weighted graph shown in fig 5.0 above is a simple graph and in a simple graph {u, v} uniquely represents an edge between any vertex u and vertex v and ω(uv) represents its weight/distance.

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 6.0, below on the left in bold letters.

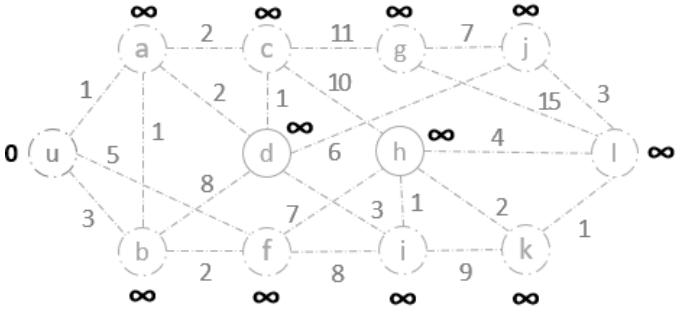


Fig 6.0

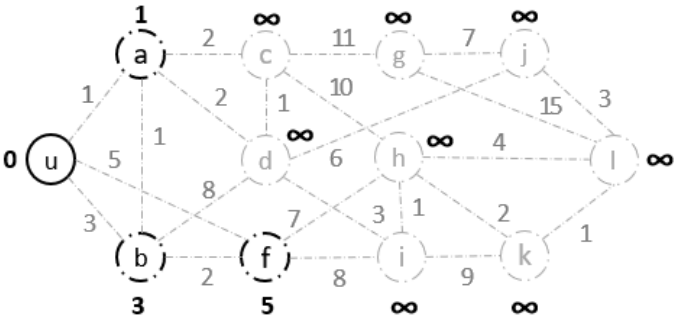


Fig 6.1

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

Now vertex u with smallest distance zero is selected/discovered. The neighbouring 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**, distance[u] + ω(ua) = 0 + 1 = 1 < ∞, hence its distance is relaxed. Similarly distances to other neighbouring vertices b and f are relaxed as highlighted in bold dashed boundary line in fig 6.1 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 7.0 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**.

Vertex b is reachable from vertex **a** by an edge {a, b} of unit distance and after re-evaluation distance becomes, distance[a] + ω(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 neighbouring vertices c and d are relaxed as highlighted in bold dashed boundary lines in fig 7.1 below on the right.

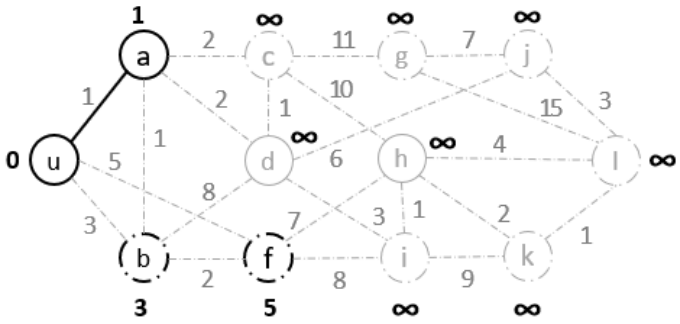


Fig 7.0

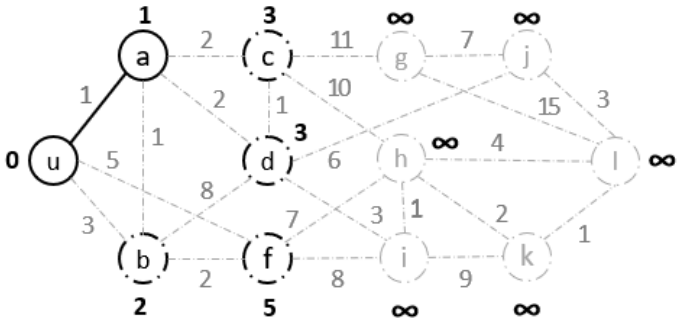


Fig 7.1

Now vertex b with smallest distance among the undiscovered vertices is selected/discovered as shown in fig 8.0 below on the left. The distance to vertex d reevaluated to distance[b] + ω(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 neighbouring vertex f is reevaluated to distance[b] + ω(bf) = 2 + 2 = 4 < 5, so distance is relaxed to 4 and predecessor changed to vertex b as shown in fig 8.1 below on the right.

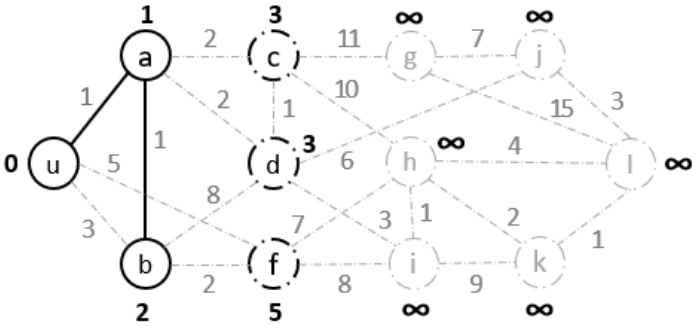


Fig 8.0

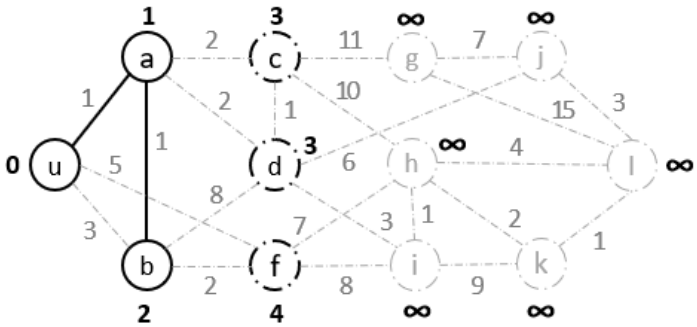


Fig 8.1

At this moment there are two vertices c and d have 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 9.0 below on the left. Vertex **a** is predecessor because the distance of vertex c last relaxed when vertex **a** was discovered.

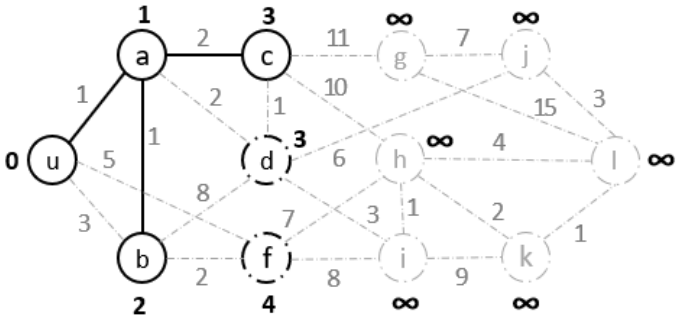


Fig 9.0

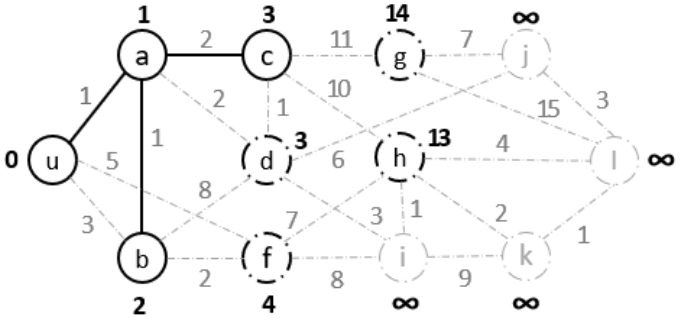


Fig 9.1

The distances to neighbouring vertices g and h are relaxed and reevaluated to 14 and 13 respectively as highlighted in bold dashed lines in fig 9.1 above on the right, but the distance to neighbouring vertex d remains unchanged because distance[c] + ω(cd) = 3 + 1 = 4 > 3.

Now vertex d has smallest distance among undiscovered vertices and it is selected next as shown in fig 10.0 below on the left. Distances to two neighbouring vertices are relaxed,

distance to vertex j is relaxed to distance[d] + ω(dj) = 3 + 6 = 9 < ∞, and distance to vertex **i** is relaxed to distance[d] + ω(di) = 3 + 3 = 6 < ∞, as highlighted by bold dashed lines in fig 10.1 below on the right.

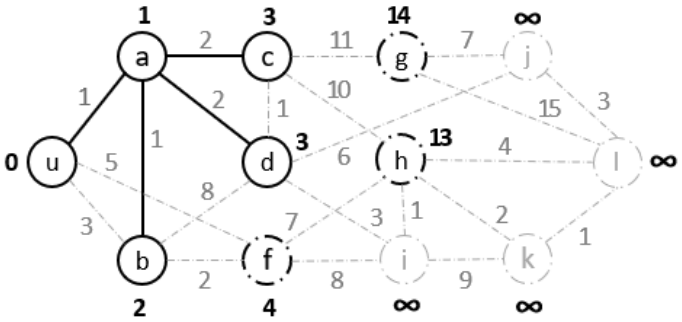


Fig 10.0

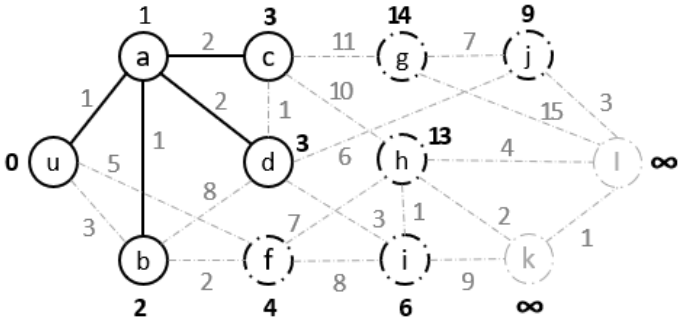


Fig 10.1

Now vertex f has smallest distance among undiscovered vertices and it is discovered next as shown in fig 11.0 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.

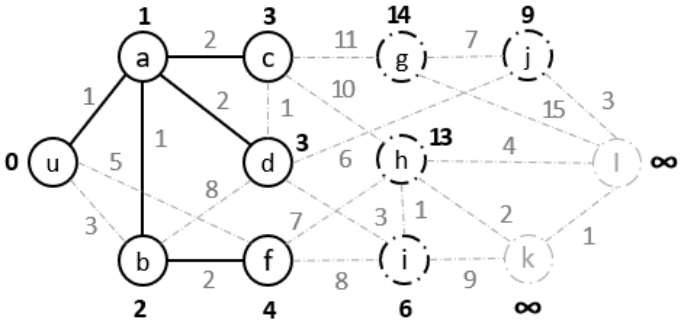


Fig 11.0

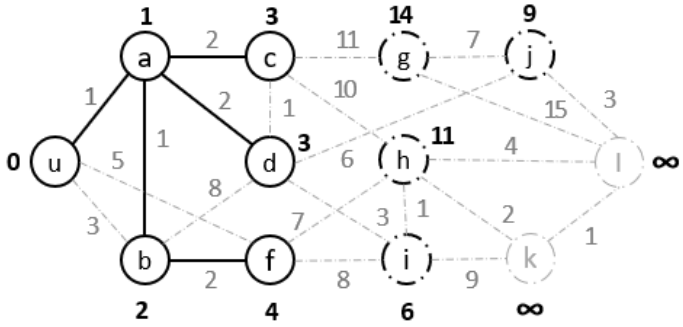


Fig 11.1

Distance to only neighbouring vertex h is relaxed as shown in fig 11.1 above on the right.

Now vertex **i** is selected/discovered as a vertex with the smallest distance among undiscovered vertices as shown in fig 12.0 below on the left.

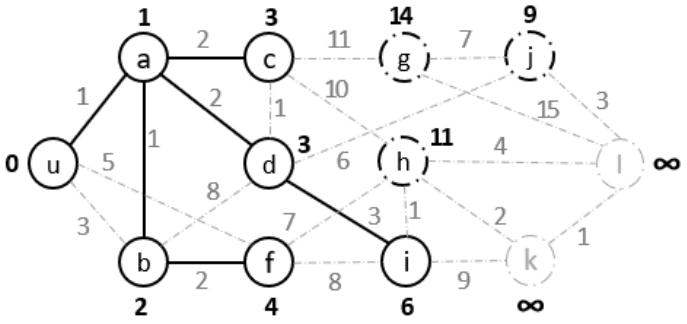


Fig 12.0

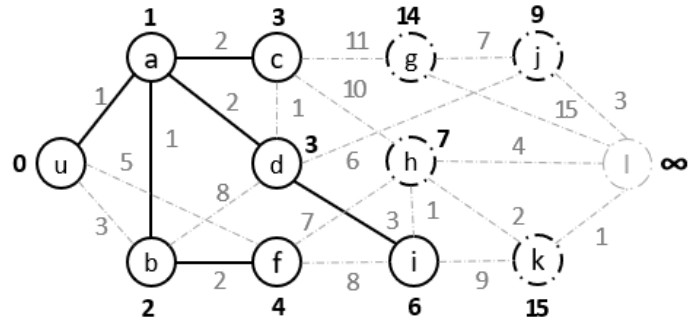


Fig 12.1

The distance to vertex h is further relaxed to distance[i] + ω(ih) = 6 + 1 = 7 < 11 and vertex **i** is set as predecessor. The distance to other neighbouring vertex k is reevaluated to distance[i] + ω(ik) = 6 + 9 = 15 < ∞, as shown in fig 12.1 above on the right.

Now vertex h is selected/discovered as the vertex with the smallest distance among undiscovered vertices as shown in fig 13.0 below on the left. The distance to neighbouring

vertex k is relaxed again to distance[h] + ω(hk) = 7 + 2 = 9 < 15, and vertex h is set as predecessor. The distance to other neighbouring vertex **l** is relaxed and reevaluates to distance[h] + ω(hl) = 7 + 4 = 11 < ∞, as shown in fig 13.1 below on the right.

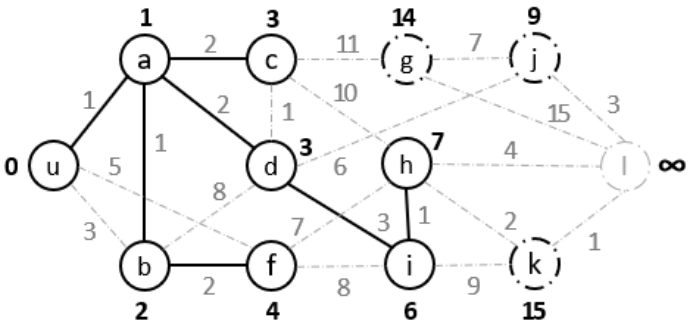


Fig 13.0

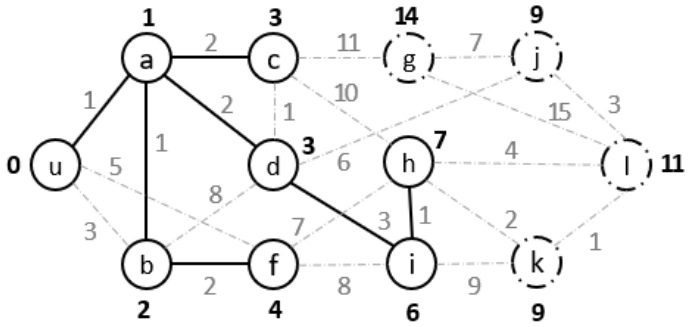


Fig 13.1

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 14.0 below on the left. It is connected to vertex d in the shortest path because its distance last relaxed when vertex d was discovered.

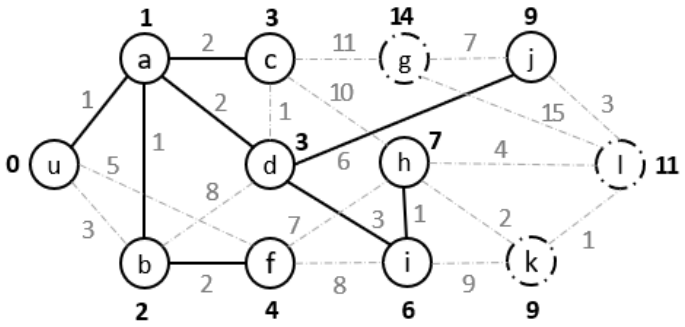


Fig 14.0

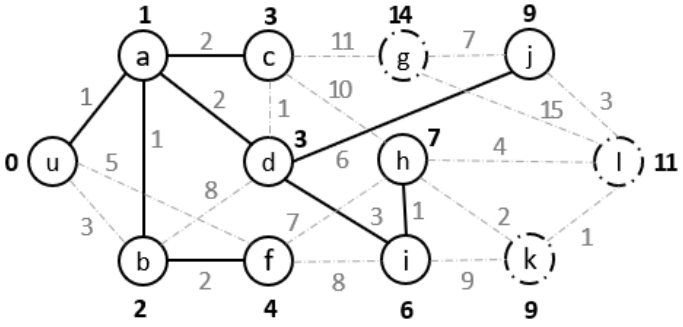


Fig 14.1

There is no distances relaxation to neighbouring vertices of vertex j, hence fig 14.0 and fig 14.1 are identical.

Now vertex k is selected/discovered as a vertex with smallest distance among the undiscovered vertices as shown in fig 15.0 below on the left. It is connected to vertex h in the shortest path because its distance last relaxed when vertex h was discovered.

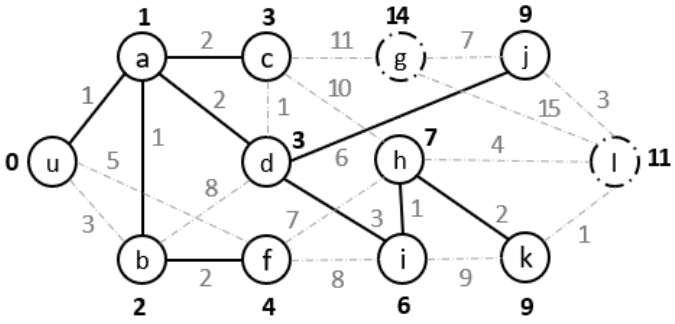


Fig 15.0

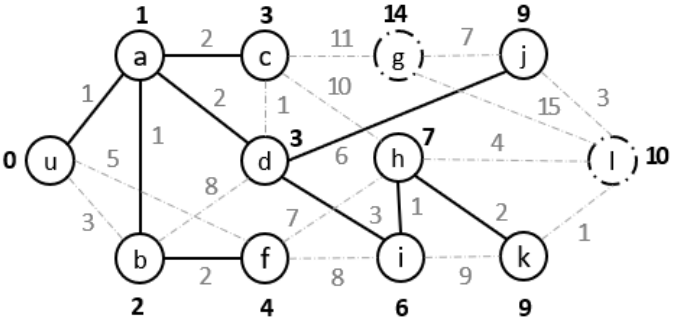


Fig 15.1

Distance of neighbouring vertex **l** is relaxed further and reevaluated to distance[k] + ω(kl) = 9 + 1 = 10 < 11 and vertex k is set as predecessor as shown in fig 15.1 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 16.0 below on the left.

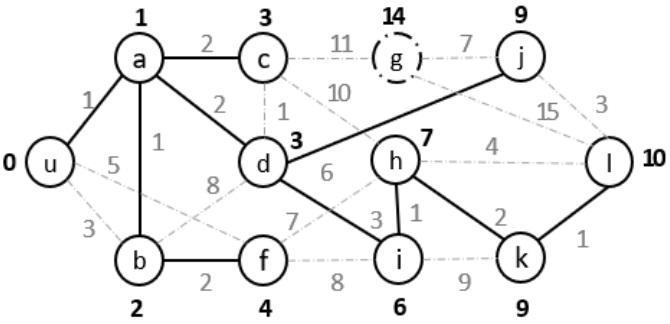


Fig 16.0

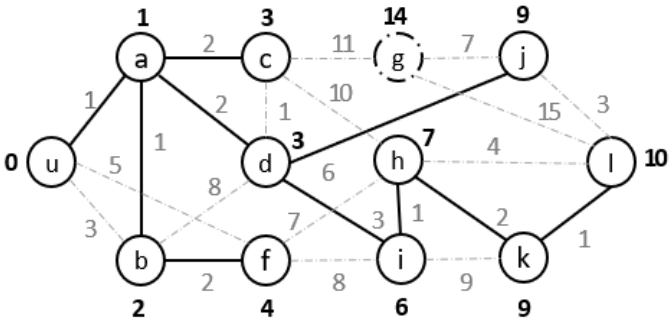


Fig 16.1

Vertex **l** has only one undiscovered neighbouring vertex g and no further relaxation happens to its distance, distance[l] + ω(lg) = 10 + 15 = 25 > 14.

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 17.0 below. Vertex g is connected to vertex c because vertex c is its predecessor and its distance last relaxed when vertex c was discovered.

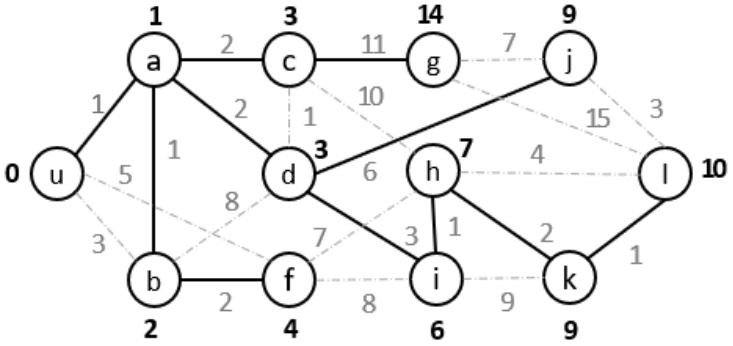


Fig 17.0

Algorithm ends here and the shortest path to each vertex from source vertex u is 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. In complexity discussion **log** means log base 2 unless otherwise stated.

The document explains the time and space complexities later in a separate section.

Negative Weight Constraint

To understand why Dijkstra's algorithm doesn't work with edges of negative weights let's consider a simple undirected graph as shown in fig 18.0 below on the left.

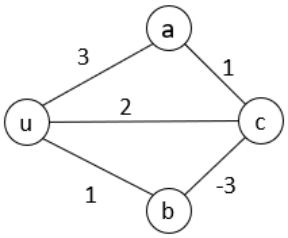


Fig18.0

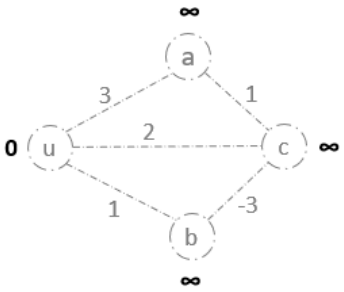


Fig18.1

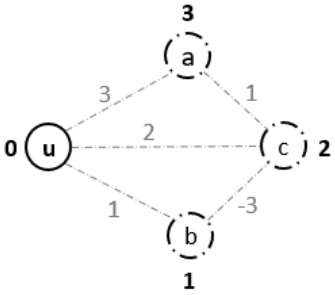


Fig18.2

The given source vertex is u and distance to it is set to zero and the distances to other vertices are set to infinity as shown in fig 18.1 above. Now vertex u is selected and distances to neighbouring vertex **a**, vertex b and vertex c are relaxed as shown in fig 18.2 above.

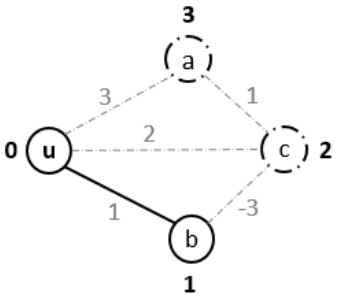


Fig19.0

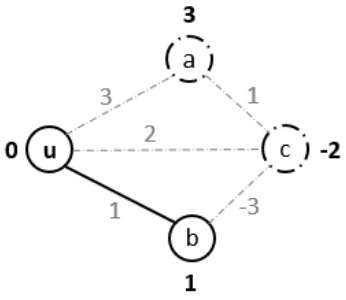


Fig19.1

Now vertex b with smallest distance is selected as shown in fig 19.0 above and distance to vertex c is relaxed again as shown in fig 19.1 above.

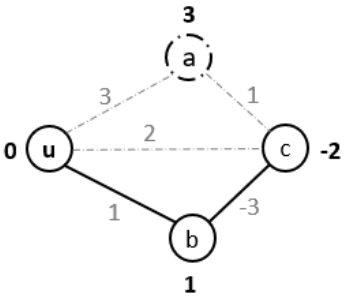


Fig20.0

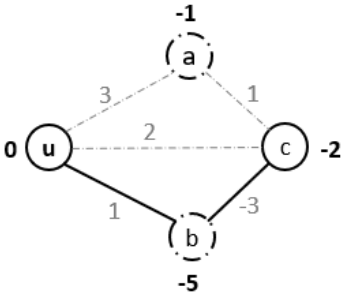


Fig20.1

Now vertex c is selected as shown in fig 20.0 above but then something unusual happens, distances to the already discovered neighbouring vertex b is relaxed again as shown in fig 20.1 above. The distance to vertex b is relaxed again because of negative weight of the edge and due to relaxation vertex b is now treated as an undiscovered vertex by algorithm. The distance to vertex b is smallest distance among undiscovered vertices and it is rediscovered and distances to its neighbouring vertices u and c are relaxed again. As a result, algorithm is trapped in the endless cycle of distance relaxation and rediscoveries of vertices b and c.

In this example problems arias due to distance relation of already discovered vertices and it seems vertex select/discover status can help to avoid it. So, lets reconsider this example with vertex select/discover status and gray colored filled node represents marked select/discover status of a vertex. Last time problem raised at vertex c, so let's focus from selection of vertex c and selection of vertex u and vertex b are same as those were last time.

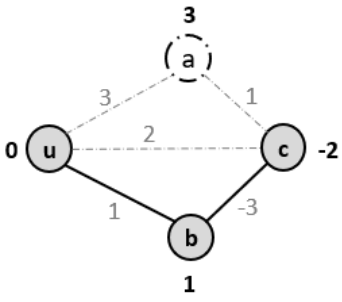


Fig22.0

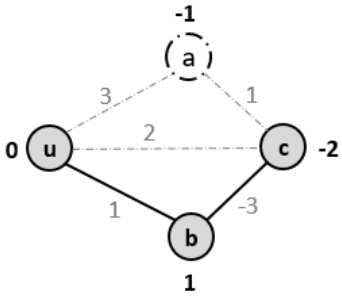


Fig22.1

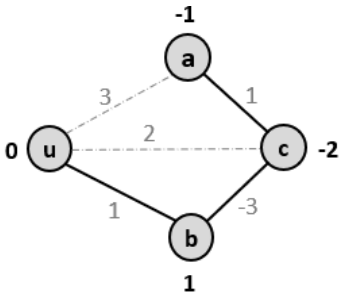


Fig22.2

Now vertex c is selected/discovered as shown in fig 22.0 above. This time distance to neighbouring vertex **a** is only relaxed as show in fig 22.1 above and marked select/discover status prevent further relaxation of the distance to vertex b. Now vertex **a** is selected as shown in fig 22.2 above and no further distance relaxation happens.

It seems select/discover status solves the problem but actually it doesn't rather it introduce the problem if used with edges of negative weight. Let consider another simple undirected graph with u as source vertex as shown in fig 23.0 below on the left.

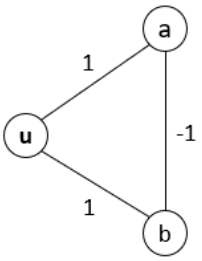


Fig23.0

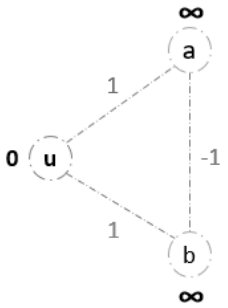


Fig23.1

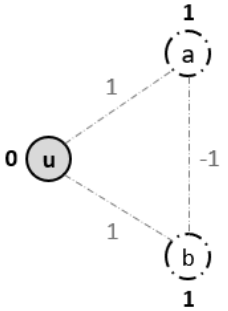


Fig23.2

The distance to vertex u is set to zero and distances to other vertices are set to infinity as shown in fig 23.1 above. Now vertex u is selected/discovered and the distances to vertex **a** and vertex b are relaxed as shown in fig 23.2 above on the right. At this moment distance to both vertices, vertex **a** and vertex b, are same and any one of them can be selected.

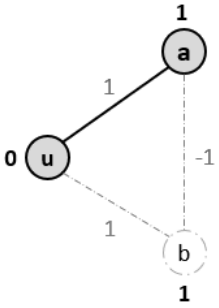


Fig24.0

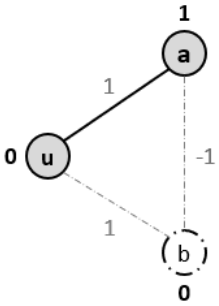


Fig24.1

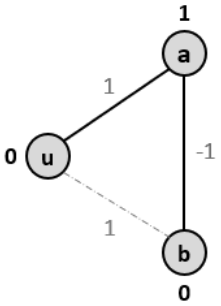


Fig24.2

Now let's consider vertex **a** is selected as shown in fig 24.0 above and distance to neighbouring vertex b is relaxed again as shown in fig 24.1 above. Then vertex b is selected and distance to vertex a is not relaxed further because vertex **a** is already marked as selected as shown in fig 24.2 above and this leads to incorrect evaluation of shortest path of vertex **a**.

In this example the shortest path of only one vertex either vertex **a** (if vertex b is selected first) or vertex b (if vertex **a** is selected first) can be evaluated correctly because of select/discover status. The correct shortest paths of vertex **a** and vertex b is shown below in fig 25.0 and fig 25.1 respectively. Both of these paths can't be shown in single figure because vertex b is predecessor of vertex **a** in shortest path from vertex u to vertex **a** and in shortest path from vertex u to vertex b, vertex **a** is the predecessor of vertex b as shown below.

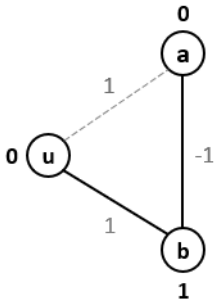


Fig 25.0

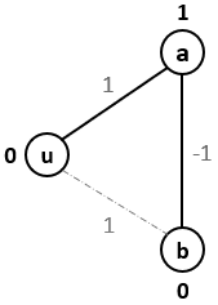


Fig 25.1

These paths are valid because in shortest path from vertex u to vertex **a**, the vertices and edges belong to the path are only visited once and that is same for the shortest path from vertex u to vertex b.

Dijkstra's algorithms look to all directly accessible undiscovered vertices from discovered vertices and a vertex v at nearest distance is selected. This newly discovered vertex v is at greater or at equal distance from source vertex u compare to previously discovered vertices (that's why those vertices were selected earlier).

If there is an edge from this newly discovered vertex v to an already discovered vertex d and as explained above the distance of the shortest path from source vertex u to vertex d is less than or equal to the distance of shortest path from source vertex u to this newly discovered vertex v then adding the positive weight of edge {v, d} to already greater or equal distance can't make the distance of this alternative path to the vertex d smaller than the distance of the already know shortest path from source vertex u to vertex d. That's why Dijkstra's algorithm works with edges of positive weights.

In Dijkstra's algorithm, once a vertex is selected/discovered, distance to it never changes. This fact is used in implementation to improve the performance by using select/discover status, which reduces the relaxation efforts which otherwise anyway going to fail for already discovered vertices because distances to them can't change with edges of positive weights.

Complexity Analysis

A complete directed graph helps to understand time complexity of Dijkstra's algorithm. It is denoted by K_n, here n is number of vertices. In a complete graph each vertex is directly connected to all other vertices. If a graph has V vertices than each vertex has V – 1 edges and in a complete undirected graph there are V(V – 1) / 2 edges. It is divided by two because in undirected graph an edge connecting two vertices is counted twice in the edge count of both of these vertices. In complete directed graph each edge has V – 1 outdoing edges and total number of edges in the directed graph is V(V – 1), here divide by two is not required.

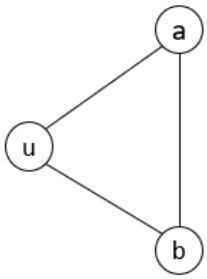


Fig 26.0

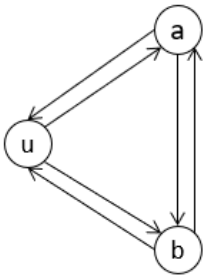


Fig 26.1

The triangle is an example of complete undirected graph represented by K_3 as shown in fig 26.0 above on the left and has 3(3-1)/2 = 3 edges. A complete directed graph also represented by K_3 is shown in fig 26.1 above on the right has 6 edges. Both of these graphs are also simple graphs because even in a directed complete graph there is only one edge between any two vertices in one direction.

In binary min heap (priority queue), insert/push of an element requires O(log n) time complexity and the key modification of an element also requires logarithmic time complexity. The extract/pop of top element requires O(log n) complexity.

The asymptotic analysis is a method to analyse how the time and space required by an algorithm increase as input size increases. In asymptotic analysis big O notation represents upper bound or worst scenario and performance of algorithm can't be worse than this. The big O notation ignores the constants for example a time complexity $ax^2 + bx + c$ in big O notation is $O(n^2)$ because quadratic term is dominating factor in complexity than linear and constant part bx + c.

Let's consider a complete directed graph with V number of vertices. The algorithm begins with selecting the source vertex and distances to all V – 1 other neighbouring vertices are relaxed and vertices with distances are inserted in priority queue and now it has V – 1 vertices.

The algorithm extracts top element/vertex of priority queue which represents selection/discovery of a vertex at smallest distance from source vertex among all undiscovered vertices. It takes O(log V) time and not O(log (V – 1)). Now onwards ignore the constant in V – 1. So, the extraction/selection of a vertex requires O(log V) time complexity.

After vertex selection/discovery, in the worst scenario distance to all V – 1 neighbouring vertices are relaxed and to modify the key (distance) of an entry in priority queue requires O(log V) time and to modify the keys of V – 1 entries requires V × O(log V) = O(V log V) time complexity. Modification in key of an element in priority queue is not always available and in case of distance relaxation a new entry of vertex with relaxed distance has to inserted in priority queue. So, time complexity to relax the distances to V – 1 vertices is O(V log V).

In real world scenario initially the number of undiscovered vertices are V – 1 and as the algorithm proceeds and discovers the vertices, the entries in priority queue decrease (may have duplicate entries of few vertices with relaxed weight) but because big O notation represents upper bound or worst scenario, time complexity of extraction of a vertex from priority queue is always considered as O(log V). So, a vertex can be discovered only once and, in a graph, there are V number of vertices so the overall complexity of selection/discovery of all vertices is V × O(log V) = O(V log V).

After the selection/discovery of a vertex (extraction of top element of priority queue) it is considered that distances to all neighbouring vertices are relaxed but it doesn't happen always and not even in a complete graph where each vertex is neighbour of all other vertices because few vertices are already discovered but again with big O notation, worst scenario is considered. So, after each vertex extraction/selection distances to all vertices are relaxed in O(V log V) time complexity and overall complexity of relaxation after selection/discovery of all V vertices is V × O(V log V) = $O(V^2 \log V)$.

The time complexity of Dijkstra algorithm is,

Time Complexity = Complexity of selection of all vertices + Overall complexity of distance relaxation
= $O(V \log V) + O(V^2 \log V)$

In a complete directed graph, each vertex has V – 1 outgoing edges, so number of edges in complete graph is E = V (V – 1) but big O notation ignores constant and E becomes E = $O(V^2)$ and it finally leads to most common form of time complexity of Dijkstra's algorithm.

Time Complexity = $O(V \log V) + O(V^2 \log V)$
= $O(V \log V) + O(E \log V)$
= $O((V + E) \log V)$

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 relaxion to vertices is bound to E.

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 = $O(V^2)$.

This is a basic explanation of time and space complexities of Dijkstra's algorithm.

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 under MIT License at: https://github.com/vikasawadhiya/Dijkstra-Algorithm

June 2025, India