**Graph**

**This topic includes:**

1. **Definition of graph**
2. **Graph terminology**
3. **Types of graph**
4. **Graph representation**
5. **Graph operation**
6. **Graph tranversal**
7. **Topological sorting**
8. **Spanning tree**
9. **Shortest path**

**1) Definition:** A Graph G = (V,E) is a data structure that consists of two sets: a finite set of vertices (V) (also called vertexs/nodes/points) and a set of edges (E) (also called links/lines) that connect the vertices.

* V(G) represents the set of vertices of the Graph.
* E(G) represents the set of edges of the Graph.

Some examples of graphs:

Diagram

Description automatically generated a)

V(G) = {0,1,2,3,4}

E(G) = {(0,1),(0,2),(1,3),(1,2),(1,4),(2,3)}

G = (V,E)

Diagram

Description automatically generated b)

V(G) = {0,1,2,3}

E(G) = {(0,1),(0,2),(1,2),(2,3)}

G = (V,E)

c)

Shape, circle

Description automatically generated

V(G) = {0,1,2,3,4,5}

E(G) = {(0,1),(0,2),(1,3),(1,4),(2,5)}

G = (V,E)

**2) Graph terminology:**

* Subgraph: A subgraph consists of a subset of graph’s vertices and edges.

Diagram

Description automatically generatedDiagram

Description automatically generated

Graph A One subgraph of graph A

* Adjacency: Two vertices are adjacent if they are connected by a edges.

A group of circles

Description automatically generated with low confidence

These pairs of vertex are adjacent: (1) & (0), (1) & (2), (2) & (3).

These pairs of vertex are adjacent: (0) & (1), (1) & (2), (0) & (2), (2) & (3)

Diagram

Description automatically generated

* Path: A sequence of edges that go from one vertex to another vertex is called path. A *simple path* is the path that passes through the same vertices of the graph just **one** time.

Diagram

Description automatically generated

There is a path that begins at vertex (0), then go to vertex (1), then (2), and ends at vertex (4).

There is also another path that begins at vertex (0) and ends at vertex (4):

Diagram

Description automatically generated

* Cycle: A cycle is a path that goes from a beginning vertex and ends at that vertex again. A *simple cycle* is a cycle that passes through the same vertices of the graph just **one** time (except the start-end vertex).

Diagram

Description automatically generated

The path is a cycle in the graph. (start at 0, end at 0)

The path is a cycle in the graph. (start at 2, end at 2)

Diagram

Description automatically generated

* Diagram

  Description automatically generatedLabel: Each edges of a graph can have a name/value called its label.

Each edge of graph has a name: a,b,c,d.

Diagram

Description automatically generated

Each edge of graph has a value: 100,250,50,10.

* Self edge (loop): A edge begin and end at the same vertex is called a self edge (or loop).

Diagram

Description automatically generatedDiagram

Description automatically generated

Some sefl edge (in red) of graphs

**3) Types of Graph:**

* Connected and Disconnected Graph: A connected graph is a graph whose each pair of vertices is connected by a path. Oppositely, a disconnected graph is a graph that has at least one pair of vertices not connnected by any path.

Example:

Diagram

Description automatically generatedDiagram

Description automatically generated

Disconnected Graph Connected Graph

* Complete Graph: A complete graph is a graph whose each pair of distinct vertices is connected by a edge. A complete graph is also a connected graph, but the converse is not true.

Diagram

Description automatically generatedDiagram

Description automatically generated

Complete Graph Not a Complete Graph

* Multigraph: A graph can’t have more than one edge between a pair of vertices. A multigraph allow multiple edges between a pair of vertices. So, a multigraph is **not** a graph.

Diagram

Description automatically generatedDiagram, shape

Description automatically generated

Multigraphs

* Diagram

  Description automatically generatedWeighted Graph: If the labels of graph’s edges represent numeric values, the graph is called weighted graph. (The value can be represented for some kind of common units such as: km/m, time,…)

Diagram

Description automatically generated

Weighted Graphs

* Undirected Graph and Directed Graph: In undirected graph, the graph’s edges are bi-directional, meaning that they do not point to any specific direction. Oppositely, in directed graph (digraph), each edge of graph points to one specific direction, and it’s called directed edges i.e an edge **(a,b)** doesn’t mean that there is also an edge **(b,a)**, **a** is adjacent to **b** but **b** is not adjacent to **a**. In digraph, the edge is also called the arc or the arrow.

Diagram

Description automatically generated

V(G) = {0,1,2}

E(G) = {<1,0>,<0,2>,<2,1>}

In this graph, each edge points to a single direction, edges:

The vertex (0) is adjacent to the vertex (1), but the vertex (1) isn’t adjacent to the vertex (0). Similar to vertex (2) and (0), (1) and (2).

Directed Graph

Diagram

Description automatically generated

V(G) = {0,1,2,3,4}

E(G) = {<0,1>,<1,2>,<2,1>,<2,3>,<3,0>,<2,4>,<4,3>}

In this graph, each edge points to a single direction, edges: (

The vertex (0) is adjacent to the vertex (1), but the vertex (1) isn’t adjacent to the vertex (0). The vertex (1) is adjacent to the vertex (2) and the vertex (2) is also adjacent to the vertex (1).

Directed Graph

Diagram

Description automatically generated

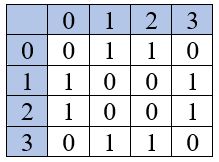
This is a undirected graph. Each edge doesn’t point to any specific direction.

Undirected Graph

**4) Graph representations:** There are two most commonly used ways to represent a graph:

* *Adjacency matrix:*
* An adjacency matrix is a 2D – array (A[][]), V is the number of vertices in the graph. In the matrix, A[i][j] = 1 means that there is an edges between vertex i and vertex j, A[i][j] = 0 means that there isn’t any edges between verter i and vertex j.
* Adjacency matrix is also used to reprensent a weighted graph. A[i][j] = W means that there is an edge between vertex i and vertex j and the value of that edge is equal to W. (so that if there isn’t a vertex from i to j, don’t use W = 0 because in some real experience, we need to use W = 0 to store a value of edge equal to 0. Use symbol: A[i][j] = ∞.
* In undirected graph, if (A[i][j] = 1), (A[j][i] = 1) also. But in directed graph, (A[i][j] = 1) doesn’t mean that (A[j][i] = 1).
* Examples:

Diagram

Description automatically generated

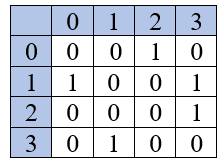
Adjacency matrix of this undirected graph

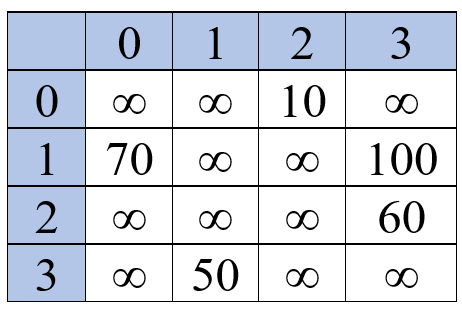
Diagram

Description automatically generated

Adjacency matrix of this directed graph

Adjacency of this undirected graph



Diagram

Description automatically generated

Adjacency matrix of this directed graph with edges’s values

Adjacency of this undirected graph

* Advantages of using adjacency matrix:
* Demonstrating a 2D – array is easy to follow and understand.
* Determine if two vertex are adjacent to each other, add an edge, delete an edge in a constant of time and these actions take effectively if you want to add/delete edges very frequently.
* Disadvantages of using adjacency matrix:
* It’s take the same memory even when the graph is spraser (always O()).
* It’s slow to add/delele a vertex or iterate over all edges because it’s a 2D – array.
* *Adjacency list:*
* An adjacency list represents a graph as an array of linked lists. The array size is equal to the number of graph’s vertices. The linked list in the array represents the vertex in the graph and each element in that linked listed represents the other vertices that form an edge with that vertex and vertex is adjacent to each element (especially in directed graph). Each element in each linked list can also carry a edge’s value.
* Examples: represent adjacency lists for some kinds of graph:Diagram

  Description automatically generatedA screenshot of a computer

  Description automatically generated with medium confidence

*Undirected graph*

**Index Vertex Adjacency linked lists**

A screenshot of a computer

Description automatically generated with medium confidence

Diagram

Description automatically generated

*Directed graph*

Explain the difference between undirected graph and directed graph in adjacency list representation of two example: in undirected graph: vertex B is adjacent to the vertex A and C, but in directed graph, because there is just one arc from A to B so A is adjacent to B but B isn’t adjacent to A. Vertex E, C and D is also follow this rule too.

Here is one more example of directed graph with edges’s value:

A screenshot of a computer

Description automatically generated with medium confidence

**Index Vertex Adjacency linked lists**

Diagram

Description automatically generated

*Directed graph with edges’s value*

* Advantages of using adjacency list:
* Memory used depends on number of edges and verticles.
* It’s fast to add/delete a vertex, a edge.
* It’s fast to iterate over all edges/nodes.
* Disadvantages of using adjacency list:
* It’s time consuming finding relationship between two nodes is less effective than adjacency-matrix’s.

**5) Graph operations:**

These are some basic operations of graph as data structure:

*(graph G, node x, node y, value val)*

* ***checkEmptyGraph(G):*** test whether the graph is empty.
* ***countVertices(G):*** get the number of vertices in a graph.
* ***countEdges(G):*** get the number of edges in a graph.
* ***adjacent(G,x,y):*** if the graph is undirected, test whether there is a edge between vertex ***x*** and vertex ***y***. If the graph is directed, test whether there is a edge start from vertex ***x*** and end at vertex ***y***.
* ***addVertex(G,x,val):*** add vertex ***x*** in the graph, if ***val*** isn’t in the graph.
* ***addEdge(G,x,y,val):*** if the graph is undirected, add a edge between vertex ***x*** and ***y*** with edge’s value is ***val***, if it doesn’t exist in the graph. If the graph is directed, add a edges start from vertex ***x*** and end at vertex ***y*** with edge’s value is ***val***, if it is not there before. With non – weighted graph, the variable val doesn’t been used.
* ***removeVertex(G,x):*** delete vertex ***x*** and every edges that connected with ***x*** in the graph, if ***x*** exists in the graph.
* ***removeEdge(G,x,y):*** delete the edges between vertex ***x*** and vertex ***y***, if it exists in the graph.
* ***setVertexValue(G,x,val):*** if vertex ***x*** exists, update its value to ***val***.
* ***setEdgeValue(G,x,y,val):*** if there is an edge between ***x*** and ***y***, update its value to ***val***. With non – weighted graph, this function is meanless.

Diagram, venn diagram

Description automatically generatedExample for graph’s operations: Directed – Weighted graph

* Note: this vertex means: vertex **0** has the value **150**.
* Action 1: start with empty graph G.

checkEmptyGraph(G) = True.

* Diagram

  Description automatically generatedAction 2: Insert some vertices
* addVertex(G,0,150)
* addVertex(G,1,100)
* addVertex(G,2,50)
* addVertex(G,3,110)
* addVertex(G,4,100)

checkEmptyGraph(G) = False

Graph G

* Action 3: Insert some edges
* Diagram

  Description automatically generatedaddEdge(G,0,1,3300)
* addEdge(G,0,2,500)
* addEdge(G,0,3,3000)
* addEdge(G,1,2,2000)
* addEdge(G,4,1,1500)
* addEdge(G,3,4,3000)
* addEdge(G,2,0,500)
* addEdge(G,2,4,600)

adjacent(G,0,2) = True

adjacent(G,4,2) = False

countVertices(G) = 5

countEdges(G) = 8

* Diagram

  Description automatically generatedAction 4: Some other operations:
* removeVertex(G,2) = True
* removeEdge(G,0,4) = False

(because there is not a edge

from 04 in the graph)

* setVertexValue(G,0,500) = True
* setEdgeValue(G,0,1,5000) = True

**6) Graph traversal:**

* Graph traversal refers to the process of visiting each vertex in the graph. If the graph is not a connected graph, graph-tranversing will just visit a subgraph (the unconnected vertex will not be visited). Graph traversal may require that some vertices be visit more than one, so it’s important to remember that which vertex has already been visited in case that the graph has a loop, the traversing may be loop indefinitely.
* There are two most common ways to tranverse the graph: Depth-First Search (DFS) and Breath-First Seach (BFS). DFS and BFS is a basic procedure for many graph-related algorithms.
* **Depth-First Search (DFS):**
* DFS algorithm is a algorithm that tranverses the depth of a whole particular path before it tranverses its breadth’s path in a graph. In other words, DFS visits all child - vertices before visiting the sibling - vertices.
* DFS algorithm starts from a selected vertex (root) and then iteratively goes to its adjacent-unvisited vertex until at its position, it can’t find any more unvisited vertex (it has been go to the deepest vertex of that path), then, the vertex moves back to the previous vertex until it finds another adjacent-unvisited vertex of its current position’s vertex. And the above process is continuelly used for the new path which DFS just found. Until the position backs to the root vertex and the root vertex doesn’t have any unvisited – adjacent vertex.
* Remember to mark which vertex is visited because it’s very easy to get in a endless loop if DFS visits a vertex many times.
* With directed graph and unconnected graph, the DFS may be visit just a subgraph of the graph instead of the whole graph.
* In Adjacency List, the complexity of DFS is . In Adjacency Matrix, the complexity of DFS is . With V is the number of vertices, E is the number of edges in the graph.
* Pseudo-code of DFS using recursive function:



* Pseudo-code of DFS using stack structure:



* Examples for DFS algorithm:

Diagram

Description automatically generated

**Example 1:**

Connected – Undirected graph

Diagram

Description automatically generated

Step 1: Start DFS algorithm at vertex 0.

Go through the path:

Then stop because vertex 6 hasn’t any unvisited adjacency.

Visited vertex: 0,3,7,4,6

Unvisited vertex: 1,2,5

Diagram

Description automatically generated

Step 2: From vertex 6, back to vertex 4,7,3,0. Stop at vertex 0 because vertex 0 has another unvisited adjacent vertex: vertex 2

Go through the path:

Then stop because vertex 5 hasn’t any unvisited adjacency.

Visited vertex: 0,3,7,4,6,2,5

Unvisited vertex: 1

Diagram

Description automatically generated

Step 3: From vertex 5, back to vertex 2. Stop at vertex 2 because vertex 2 has another unvisited adjacent vertex: vertex 1.

Go through the path:

Then stop because vertex 1 hasn’t any unvisited adjacency.

Visited vertex: 0,3,7,4,6,2,5,1

Unvisited vertex: None

Diagram

Description automatically generated

Step 4: From vertex 1, back to vertex 2,0. Stop at vertex 0 because this is the root and vertex 0 hasn’t no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

All vertices of the graph has been visited because this is the connected – undirected graph.

Visited vertex: 0,3,7,4,6,2,5,1

Unvisited vertex: None

Diagram

Description automatically generated

**Example 2:**

Connected – Directed graph

Diagram, schematic

Description automatically generated

Step 1: Start DFS algorithm at vertex 0.

Go through the path:

Then stop because vertex 7 hasn’t any unvisited adjacency.

Visited vertex: 0,2,1,3,7

Unvisited vertex: 5,6,4

Diagram, schematic

Description automatically generated

Step 2: From vertex 7, back to vertex 3,1,2,0. Stop at vertex 2 because vertex 2 has another unvisited adjacent vertex: vertex 4.

Go through the path:

Then stop because vertex 4 hasn’t any unvisited adjacency.

Visited vertex: 0,2,1,3,7,4

Unvisited vertex: 5,6

Diagram, schematic

Description automatically generated

Step 3: From vertex 4, back to vertex 2,0. Stop at vertex 0 because this is the root and vertex 0 hasn’t no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

Some vertices (5,6) hadn’t been visited because this is a directed graph.

Visited vertex: 0,2,1,3,7,4

Unvisited vertex: 5,6

**Example 3:**

Diagram

Description automatically generated

Unconnected – Undirected graph

Diagram

Description automatically generated

Step 1: Start DFS algorithm at vertex 1.

Go through the path:

Then stop because vertex 5 hasn’t any unvisited adjacency.

Visited vertex: 1,3,7,4,2,5

Unvisited vertex: 6,0

Step 2: From vertex 5, back to vertex 2,4. Stop at vertex 4 because vertex 4 has another unvisited adjacent vertex: vertex 6.

Go through the path:

Then stop because vertex 6 hasn’t any unvisited adjacency.

Visited vertex: 1,3,7,4,2,5,6

Unvisited vertex: 0

Diagram

Description automatically generated

Diagram

Description automatically generated

Step 3: From vertex 6, back to vertex 4,7,3,1. Stop at vertex 1 because this is the root and vertex 1 hasn’t no more unvisited-adjacent vertex.

Now the DFS algorithm stops.

Some vertices (0) hadn’t been visited because this is an unconnected graph.

Visited vertex: 1,3,7,4,2,5,6

Unvisited vertex: 0

* **Breath-First Search (BFS):**
* Breath-First Search is a graph traversing algorithm that starts from a selected vertex (root), then visits all its unvisited adjacent vertices. After visiting all unvisited adjacent vertices, BFS moves towards the next-level unvisited adjacent vertex (it means that visiting the current vertex’s grandchildrens). This process repeats until there are no more unvisited adjacent vertices which can be visited.
* In other words, BFS algorithms visits all sibling – vertices before it goes to deeper layers. It is the opposite of DFS algorithm.
* Remember to mark which vertex is visited because it’s very easy to get in a endless loop if BFS visits a vertex many times or if the graph has a loop.
* With directed graph and unconnected graph, the BFS may be visit just a subgraph of the graph instead of the whole graph.
* In Adjacency List, the complexity of BFS is . In Adjacency Matrix, the complexity of BFS is . With V is the number of vertices, E is the number of edges in the graph.
* Pseudo-code of BFS using queue:



* Examples for BFS algorithm:

**Example 1:**

Diagram

Description automatically generated

Connected – Undirected graph

Diagram

Description automatically generated

Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices (3,2).

Go these next paths:

Then stop because vertex 0 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,3

Unvisited vertex: 1,5,7,4,6

Diagram

Description automatically generated

Step 2: From vertex 2, go to its unvisited adjacent vertices. (1,5)

Go these next paths:

Then stop because vertex 2 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,3,1,5

Unvisited vertex: 7,4,6

Diagram

Description automatically generated

Step 3: Back to select vertex 3 (sibling of vertex 2), go to its unvisited adjacent vertices. (7)

Go these next paths:

Then stop because vertex 3 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7

Unvisited vertex: 4,6

Diagram

Description automatically generated

Step 4: Because vertex 2 children don’t have any more unvisited adjacent vertex, continue with vertex 3 (sibling of vertex 2) child (vertex 7).

Go these next paths:

Then stop because vertex 7 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7,4

Unvisited vertex: 6

Diagram

Description automatically generated

Step 5: Because vertex 7 doesn’t have any sibling whose parent is vertex 3, continue with vertex 7 child (vertex 4)

Go these next paths:

Then stop because vertex 4 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,3,1,5,7,4,6

Unvisited vertex: None

Diagram

Description automatically generated

Step 6: Back to vertex 4, stop because vertex 4 doesn’t have any other unvisited adjacency vertex.

The algorithm now stops because there are no more unvisited vertex to visit.

Visited vertex: 0,2,3,1,5,7,4,6

Unvisited vertex: None

**Example 2:**

Diagram

Description automatically generated

Connected – Directed graph

Diagram

Description automatically generated

Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices. (2)

Go these next paths:

Then stop because vertex 0 hasn’t any more unvisited adjacency.

Visited vertex: 0,2

Unvisited vertex: 1,3,5,7,4,6

Diagram

Description automatically generated

Step 2: Because vertex 2 is the only child of vertex 0. So the next step continue to deal with children of vertex 2. (1,4)

From vertex 2, go to its adjacent vertices (1,4).

Go these next paths:

Then stop because vertex 2 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,1,4

Unvisited vertex: 3,5,7,6

Diagram

Description automatically generated

Step 3: From vertex 1, go to its adjacent vertices (3).

Go these next paths:

Then stop because vertex 1 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,1,4,3

Unvisited vertex: 5,7,6

Step 4: Next go to vertex 4 (sibling of vertex 1). But because vertex 4 doesn’t have any unvisited adjacent vertex, so then continue deal with children of vertex 1 (3).

Because vertex 3 is the only child of vertex 1, next step will deal with children of vertex 3. (5,7)

Go these next paths:

Then stop because vertex 3 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,1,4,3,5,7

Unvisited vertex: 6

Diagram

Description automatically generated with low confidence

A picture containing different

Description automatically generated

Step 5: From vertex 5, go to its unvisited adjacent vertices. (6)

Go these next paths:

Then stop because vertex 6 hasn’t any more unvisited adjacency.

Visited vertex: 0,2,1,4,3,5,7,6

Unvisited vertex: None

Step 6: Next go to vertex 7 (sibling of vertex 5). But because vertex 7 doesn’t have any unvisited adjacent vertex, so then continue to deal with children of vertex 5 (6). But because vertex 6 doesn’t have any unvisited adjacent vertex too, stop at vertex 6.

The algorithm now stops because there are no more unvisited vertex which can be visited.

Visited vertex: 0,2,1,4,3,5,7,6

Unvisited vertex: None

A picture containing different

Description automatically generated

Diagram

Description automatically generated

**Example 3:**

Unconnected – Undirected graph

Diagram

Description automatically generated

Step 1: Start BFS algorithm at vertex 0. Then go to its adjacent vertices. (3)

Go these next paths:

Then stop because vertex 0 hasn’t any more unvisited adjacency.

Visited vertex: 0,3

Unvisited vertex: 1,2,5,7,4,6

Diagram

Description automatically generated

Step 2: Because vertex 3 is the only child of vertex 0. So the next step continue to deal with children of vertex 3. (7)

From vertex 3, go to its adjacent vertices (7).

Go these next paths:

Then stop because vertex 3 hasn’t any more unvisited adjacency.

Visited vertex: 0,3,7

Unvisited vertex: 1,2,5,4,6

Diagram

Description automatically generated

Step 3: Because vertex 7 is the only child of vertex 3. So the next step continue to deal with children of vertex 7. (5,6)

From vertex 7, go to its adjacent vertices (5,6).

Go these next paths:

Then stop because vertex 3 hasn’t any more unvisited adjacency.

Visited vertex: 0,3,7

Unvisited vertex: 1,2,5,4,6

Step 4: Because vertex 5 and vertex 6 (children of vertex 7) don’t have any unvisited adjacent vertex, stop at vertex 5 and 6.

The algorithm now stops because there are no more unvisited vertex which can be visited.

Visited vertex: 0,3,7,5,6

Unvisited vertex: 1,2,4

Because this is a unconnected graph, there are some vertices that can’t be reached by BFS algorithm.

Diagram

Description automatically generated

**7) Topological Sorting:**

* Topological Sorting for Directed Acyclic Graph (a directed without cycle graph) is a linear ordering of vertices in the graph. The edges after sorting will point in one direction left or right.

Diagram

Description automatically generated

Diagram

Description automatically generated

Topological Sorted List

Unsorted Graph

* Topological Sorting steps:
* Step 1: Find a vertex that doesn’t have any successor – which means the vertex that doesn’t adjacent to any vertex in directed graph.
* Step 2: Remove from the graph this vertex and all edges link with this vertex. Add this vertex to the beginning of the sorted list.
* Step 3: Repeat step 2 until the graph empty. The sorted list now has all vertices of the old graph.

(In examples, we can draw the edges of vertices at each step in the sorted list for easier to understand implementation).

**Diagram

Description automatically generatedExamples 1:**

Unsorted Graph

Background pattern

Description automatically generated

Step 1: Start Topological algorithm at vertex 4 because vertex 4 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.

After delete 4



Ordered List

Background pattern

Description automatically generatedA picture containing clock, gauge

Description automatically generated

Ordered List

After delete 3

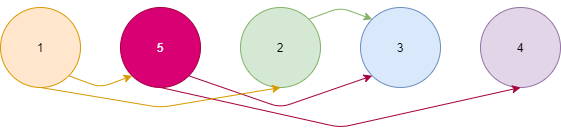
Order List

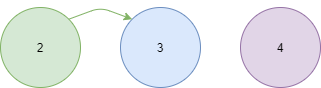
Step 2: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.

A picture containing pool ball, sport

Description automatically generatedA picture containing pool ball

Description automatically generatedA picture containing text, pool ball, vector graphics

Description automatically generatedA picture containing text

Description automatically generated

Ordered List

Ordered List

Ordered List

After delete 1

After delete 5

Step 5: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.

Order List

Order List

After delete 2

Order List

Step 4: Continually, Topological algorithm takes vertex 5 because vertex 5 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.

Step 3: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.

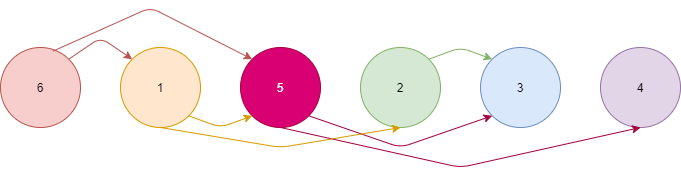
Step 6: Continually, Topological algorithm takes vertex 6 because vertex 6 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 6 in the beginning of the sorted list.

A picture containing diagram

Description automatically generated

After delete 6



Ordered List

Order List

Step 7: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

Now

After delete 0: Empty graph

A picture containing text

Description automatically generated

Ordered List

Order List

Step 8: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.

A picture containing text

Description automatically generated

**Examples 2:**

Background pattern

Description automatically generated

A picture containing text, pool ball, vector graphics

Description automatically generated

A picture containing text, pool ball, vector graphics

Description automatically generatedA picture containing diagram

Description automatically generatedA picture containing chart

Description automatically generated

After delete 4

After delete 5

Ordered List

Ordered List

Order List

Order List

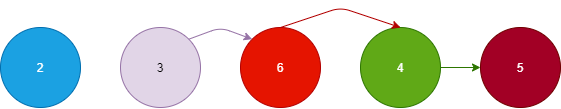
Step 2: Continually, Topological algorithm takes vertex 4 because vertex 4 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.

Step 1: Start Topological algorithm at vertex 5 because vertex 5 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.

A picture containing graphical user interface

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text, pool ball, table

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text, pool ball, vector graphics

Description automatically generated

After delete 2

After delete 3

After delete 6

Ordered List

Ordered List

Ordered List

Order List

Order List

Order List

Step 5: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.

Step 4: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.

Step 3: Continually, Topological algorithm takes vertex 6 because vertex 6 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 6 in the beginning of the sorted list.

A screenshot of a video game

Description automatically generated with low confidenceA screenshot of a video game

Description automatically generated with low confidenceA picture containing text

Description automatically generatedA picture containing diagram

Description automatically generated

After delete 1

After delete 0: Empty graph

Ordered List

Ordered List

Order List

Order List

Step 8: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.

Step 7: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

Step 6: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.

A picture containing text, pool ball

Description automatically generated **Example 3:**

A picture containing text, clock

Description automatically generatedA picture containing text, pool ball, table, vector graphics

Description automatically generatedA yellow circle with a black background

Description automatically generated with medium confidenceA picture containing text, pool ball, vector graphics

Description automatically generated

After delete 3

After delete 4

Step 2: Continually, Topological algorithm takes vertex 3 because vertex 3 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 3 in the beginning of the sorted list.

Ordered List

Ordered List

Step 1: Start Topological algorithm at vertex 4 because vertex 4 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 4 in the beginning of the sorted list.

A picture containing diagram

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text, pool ball, table, vector graphics

Description automatically generated

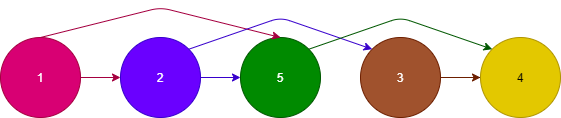
Step 4: Continually, Topological algorithm takes vertex 2 because vertex 2 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 2 in the beginning of the sorted list.

Step 3: Continually, Topological algorithm takes vertex 5 because vertex 5 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 5 in the beginning of the sorted list.

A picture containing icon

Description automatically generatedA picture containing diagram

Description automatically generated

After delete 1

After delete 2

After delete 5

Ordered List

Step 5: Continually, Topological algorithm takes vertex 1 because vertex 1 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 1 in the beginning of the sorted list.

Ordered List

Ordered List

A picture containing diagram

Description automatically generatedA picture containing diagram

Description automatically generated

Ordered List

Step 7: Now the Topological algorithm ends because the graph is empty.

Now the order list appears.

After delete 0: Empty graph

Step 6: Continually, Topological algorithm takes vertex 0 because vertex 0 doesn’t adjacent to any vertex.

Delete this vertex and its edges. Add 0 in the beginning of the sorted list.

* The Topological algorithm sorting can be implemented with stack data structure. Pseudo-code:



**8) Spanning tree:**

* Spanning tree is a subgraph of Graph which includes all vertices covered with minimum possible number of edges to form a tree. A spanning tree has **n** vertices and **n-1** edges.
* Every connected undirected graph has at least one spanning tree. A disconnected graph can’t draw any spanning tree.

A picture containing pool ball, table

Description automatically generatedA picture containing pool ball, table

Description automatically generated

Graph G

One spanning tree of G root 0

* With DFS algorithms, we can form a spanning tree by marked each edges which visited by DFS. Pseudo-code recursive DFS:



* Example implementations for Spanning tree with DFS:

Background pattern

Description automatically generated

A picture containing vector graphics

Description automatically generatedA group of blue and red circles

Description automatically generated with low confidenceA picture containing text

Description automatically generated

Step 3: Now DFS stops because all vertices are visited.

These marked edges with visited vertices form a tree. (root is 0)

Step 2: From vertex 1, back to vertex 3,2. Continue DFS at vertex 2 because it has adjacent vertex.(4)

Go to vertex 2,4,5:

Marked these edges (red lines).

Stop at vertex 5 because it doesn’t have any unvisited vertices.

Visited vertices: 0,2,3,1,4,5

Unvisited vertices: None

Step 1: Start DFS algorithm at vertex 0.

Go to vertex 2 (vertex 0’s adjacent), 3,1:

Marked these edges (red lines).

Stop at vertex 1 because it doesn’t have any unvisited vertices.

Visited vertices: 0,2,3,1

Unvisited vertices: 4,5

* A picture containing pool ball, table, vector graphics

  Description automatically generatedA picture containing pool ball, table, vector graphics

  Description automatically generatedA picture containing pool ball, table, vector graphics

  Description automatically generatedWith weighted graph, minimun spanning tree is a spanning tree that has less minimum total weight than all other spanning trees which can form in a same graph. In real life, this kind of spanning tree of weighted graph solves many problems about distance, traffic, data,..

Graph G

Spanning tree cost: 79

Minimum spanning tree cost: 49

* We can form a minimun spanning by using Prim’s algorithms.
* Step 1: Start Prim’s algorithm at the selected root vertex. Add it in the tree as the root.
* Step 2: Select the least cost edge which begin with a vertex in the current tree and end at the vertex not in the tree.
* Step 3: Add the end vertex and the edge chosen in the tree.
* Step 4: Repeat step 2 until all vertices of the graph are visited.
* Examples of Prim’s algorithm:

**Example 1:**

Background pattern

Description automatically generated

Step 1: Start DFS algorithm at vertex 0.

Now choose the edge that has least cost: cost = 2.

Visited vertices: 0,3

Unvisited vertices: 1,2,4,5,6,7,8

Background pattern

Description automatically generated

Background pattern

Description automatically generated

Step 2: Continually, choose the edge that has least cost among the vertices in the tree (0,3):

cost = 1.

Visited vertices: 0,3,4

Unvisited vertices: 1,2,5,6,7,8

Background pattern

Description automatically generated

Step 3: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4):

cost = 3.

Visited vertices: 0,3,4,1

Unvisited vertices: 2,5,6,7,8

A picture containing background pattern

Description automatically generatedA picture containing background pattern

Description automatically generatedBackground pattern

Description automatically generated

Step 6: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8):

cost = 2.

Visited vertices: 0,3,4,1,8,7,2

Unvisited vertices: 5,6

Step 4: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1):

cost = 4.

Visited vertices: 0,3,4,1,8

Unvisited vertices: 2,5,6,7

Step 5: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8):

cost = 4.

Visited vertices: 0,3,4,1,8,7

Unvisited vertices: 2,5,6

A picture containing text

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text

Description automatically generated

Minimum spanning tree

Step 9: Now the Prim’s algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

Visited vertices: 0,3,4,1,8,7,2,5,6

Unvisited vertices: None

Step 8: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8,5):

cost = 8.

Visited vertices: 0,3,4,1,8,7,2,5,6

Unvisited vertices: None

Step 7: Continually, choose the edge that has least cost among the vertices in the tree (0,3,4,1,8,2):

cost = 1.

Visited vertices: 0,3,4,1,8,7,2,5

Unvisited vertices: 6

**Background pattern

Description automatically generated** **Example 2:**

Background pattern

Description automatically generatedBackground pattern

Description automatically generatedA picture containing schematic

Description automatically generatedA picture containing background pattern

Description automatically generatedBackground pattern

Description automatically generated

Step 2: Continually, choose the edge that has least cost among the vertices in the tree (0,3):

cost = 3.

Visited vertices: 0,3,1

Unvisited vertices: 2,4,5

Step 1: Start DFS algorithm at vertex 0.

Now choose the edge that has least cost: cost = 1.

Visited vertices: 0,3

Unvisited vertices: 1,2,4,5

Step 4: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1,5,4):

cost = 4.

Visited vertices: 0,3,1,5,4

Unvisited vertices: 2

Step 5: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1,5,4,2):

cost = 2.

Visited vertices: 0,3,1,5,4,2

Unvisited vertices: None

Step 3: Continually, choose the edge that has least cost among the vertices in the tree (0,3,1):

cost = 3.

Visited vertices: 0,3,1,5

Unvisited vertices: 2,4

A picture containing schematic

Description automatically generated

Step 6: Now the Prim’s algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

Visited vertices: 0,3,1,5,4,2

Unvisited vertices: None

Minimum spanning tree

**Example 3:**

**Background pattern

Description automatically generated**

Background pattern

Description automatically generated

Step 1: Start DFS algorithm at vertex 5.

Now choose the edge that has least cost: cost = 6.

Visited vertices: 5

Unvisited vertices: 0,1,2,4,6

A picture containing graphical user interface

Description automatically generatedA picture containing background pattern

Description automatically generatedA picture containing background pattern

Description automatically generated

Step 4: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1):

cost = 7.

Visited vertices: 5,3,0,1,4

Unvisited vertices: 2,6

Step 3: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0):

cost = 7.

Visited vertices: 5,3,0,1

Unvisited vertices: 2,4,6

Step 2: Continually, choose the edge that has least cost among the vertices in the tree (5,3):

cost = 5.

Visited vertices: 5,3,0

Unvisited vertices: 1,2,4,6

A picture containing circle

Description automatically generatedA picture containing circle

Description automatically generatedA picture containing circle

Description automatically generated

Minimum spanning tree

Step 7: Now the Prim’s algorithm stops because the graph has no more unvisited vertex.

The marked edges (red lines) form a minimum spanning tree.

Visited vertices: 5,3,0,1,4,2,6

Unvisited vertices: None

Step 6: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1,4,2):

cost = 9.

Visited vertices: 5,3,0,1,4,2,6

Unvisited vertices: None

Step 5: Continually, choose the edge that has least cost among the vertices in the tree (5,3,0,1,4):

cost = 5.

Visited vertices: 5,3,0,1,4,2

Unvisited vertices: 6

**9) Shortest path:**

* In unweighted graph, a shortesh path between two vertices is the path that has least edges. By this we can use BFS algorithms.
* In weighted graph, a shortest path between two vertices is the path that costs smallest weight.
* We can use E. Dijkstra’s algorithm to find shortest path from one vertex to all other vertex in the graph.
* E. Dijkstra’s algorithm steps:
* Step 1: E. Dijkstra’s algorithm includes one visited vertices set. Firstly, choose a start vertex. Mark it vertex as visited in the set. Mark its edge’s weight to all other vertices. (if there isn’t a edge, mark it as a simpol such as: )
* Step 2: Find a other vertex that is unvisited. Mark it as visited. Now update the path from the start vertex to other vertice in the graph by using edges of visited vertices set. (in examples, I will use a table to marked every update for easy understand).
* Step 3: Repeat step 2 until all vertices of the graph is written in the visited vertices set.
* Pseudo-code of E. Dijkstra’s algorithm:



**Example 1:**

**Background pattern

Description automatically generated**

Start at root is vertex 0. Find shortest path from 0 to all other vertices.

**Step 1:** Mark all edges’s weight from vertex 0 to all other vertices.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |

The visited vertex will be marked as red character.

**Step 2:** Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 2. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | **3** | 2 | **3** |  |  |  |  |

Now the path from vertex 0 to vertex 4 is smaller than the old one weight (, so update it. The path from vertex 0 to vertex 2 appears so update it also.

**Step 3:** Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 2. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | **4** |  |  |  |

Now, the path from vertex 0 to vertex 5 appears so update it.

**Step 4:** Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 3. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | **10** |  |

Now, the path from vertex 0 to vertex 7 appears so update it.

**Step 5:** Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 3. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | 10 |  |
| 0,1,3,2,4 | 0 | 2 | 3 | 2 | 3 | 4 | **6** | 10 | **6** |

Now, the path from vertex 0 to vertex 6 and 8 appears so update it.

**Step 6:** Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 4. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | 10 |  |
| 0,1,3,2,4 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |

Nothing update by using vertex 5.

**Step 7:** Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 6. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | 10 |  |
| 0,1,3,2,4 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5,8 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | **7** | 6 |

Now the path from vertex 0 to vertex 7 is smaller than the old one , so update it.

**Step 9:** Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 6. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | 10 |  |
| 0,1,3,2,4 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5,8 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 7 | 6 |
| 0,1,3,2,4,5,8,6 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 7 | 6 |

Nothing update by using vertex 6.

**Step 9:** Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 7. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 2 |  | 2 | 5 |  |  |  |  |
| 0,1 | 0 | 2 | 3 | 2 | 3 |  |  |  |  |
| 0,1,3 | 0 | 2 | 3 | 2 | 3 | 4 |  |  |  |
| 0,1,3,2 | 0 | 2 | 3 | 2 | 3 | 4 |  | 10 |  |
| 0,1,3,2,4 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 10 | 6 |
| 0,1,3,2,4,5,8 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 7 | 6 |
| 0,1,3,2,4,5,8,6 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 7 | 6 |
| 0,1,3,2,4,5,8,6,7 | 0 | 2 | 3 | 2 | 3 | 4 | 6 | 7 | 6 |

Nothing update by using vertex 7.

**Step 10:** Now Dijkstra’s algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.

**Example 2:**

Background pattern

Description automatically generated

Start at root is vertex 0. Find shortest path from 0 to all other vertices.

**Step 1:** Mark all edges’s weight from vertex 0 to all other vertices.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |

The visited vertex will be marked as red character.

**Step 2:** Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 4. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | **12** |  |  |  |  | 8 |  |

Now the path from vertex 0 to vertex 2 appears so update it.

**Step 3:** Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 8. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | **15** |  |  |  | 8 | **9** |

Now, the path from vertex 0 to vertex 3 and vertex 8 appears so update it.

**Step 4:** Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 9. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | **11** | 8 | 9 |

Now, the path from vertex 0 to vertex 6 appears so update it.

**Step 5:** Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 11. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | 11 | 8 | 9 |
| 0,1,7,8,6 | 0 | 4 | 12 | 15 | **21** | **25** | 11 | 8 | 9 |

Now, the path from vertex 0 to vertex 4 and 5 appears so update it.

**Step 6:** Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 12. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | 11 | 8 | 9 |
| 0,1,7,8,6 | 0 | 4 | 12 | 15 | 21 | 25 | 11 | 8 | 9 |
| 0,1,7,8,6,2 | 0 | 4 | 12 | **14** | 21 | **19** | 11 | 8 | 9 |

Now the paths from vertex 0 to vertex 3 and vertex 5 are smaller than the old one , so update it.

**Step 7:** Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 14. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | 11 | 8 | 9 |
| 0,1,7,8,6 | 0 | 4 | 12 | 15 | 21 | 25 | 11 | 8 | 9 |
| 0,1,7,8,6,2 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |

Nothing update by using vertex 3.

**Step 8:** Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 19. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | 11 | 8 | 9 |
| 0,1,7,8,6 | 0 | 4 | 12 | 15 | 21 | 25 | 11 | 8 | 9 |
| 0,1,7,8,6,2 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3,5 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |

Nothing update by using vertex 5.

**Step 9:** Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 21. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 4 |  |  |  |  |  | 8 |  |
| 0,1 | 0 | 4 | 12 |  |  |  |  | 8 |  |
| 0,1,7 | 0 | 4 | 12 | 15 |  |  |  | 8 | 9 |
| 0,1,7,8 | 0 | 4 | 12 | 15 |  |  | 11 | 8 | 9 |
| 0,1,7,8,6 | 0 | 4 | 12 | 15 | 21 | 25 | 11 | 8 | 9 |
| 0,1,7,8,6,2 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3,5 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |
| 0,1,7,8,6,2,3,5,4 | 0 | 4 | 12 | 14 | 21 | 19 | 11 | 8 | 9 |

Nothing update by using vertex 4.

**Step 10:** Now Dijkstra’s algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.

**Example 3:**

Background pattern

Description automatically generated

Start at root is vertex 0. Find shortest path from 0 to all other vertices.

**Step 1:** Mark all edges’s weight from vertex 0 to all other vertices.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |

The visited vertex will be marked as red character.

**Step 2:** Choose smallest path weight of a vertex which unvisited: vertex 1 with cost 20. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | **30** |  | **120** |

Now the path from vertex 0 to vertex 9 appears so update it. The path from vertex 0 to vertex 7 is smaller than the old one , update it also.

**Step 3:** Continually, choose smallest path weight of a vertex which unvisited: vertex 7 with cost 30. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | **40** | **60** | 30 |  | 120 |

Now the paths from vertex 0 to vertex 5 and vertex 6 appear , update them.

**Step 4:** Continually, choose smallest path weight of a vertex which unvisited: vertex 2 with cost 40. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | **90** | 120 |

Now, the path from vertex 0 to vertex 8 appears so update it.

**Step 5:** Continually, choose smallest path weight of a vertex which unvisited: vertex 5 with cost 40. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |

Nothing changes by using vertex 5.

**Step 6:** Continually, choose smallest path weight of a vertex which unvisited: vertex 3 with cost 50. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5,3 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | **80** |

Now the path from vertex 0 to vertex 9 is smaller than the old one so update it.

**Step 7:** Continually, choose smallest path weight of a vertex which unvisited: vertex 6 with cost 60. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5,3 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6 | 0 | 20 | 40 | 50 | **130** | 40 | 60 | 30 | 90 | 80 |

Now the path from vertex 0 to vertex 4 appears so update it.

**Step 8:** Continually, choose smallest path weight of a vertex which unvisited: vertex 9 with cost 80. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5,3 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6 | 0 | 20 | 40 | 50 | 130 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9 | 0 | 20 | 40 | 50 | **110** | 40 | 60 | 30 | 90 | 80 |

Now the path from vertex 0 to vertex 4 is smaller than the old one so update it.

**Step 9:** Continually, choose smallest path weight of a vertex which unvisited: vertex 8 with cost 90. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5,3 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6 | 0 | 20 | 40 | 50 | 130 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9 | 0 | 20 | 40 | 50 | 110 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9,8 | 0 | 20 | 40 | 50 | 110 | 40 | 60 | 30 | 90 | 80 |

Nothing update by using vertex 8.

**Step 11:** Continually, choose smallest path weight of a vertex which unvisited: vertex 4 with cost 110. Add it in visited set and update shortest path.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| VisitedSet/Vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 20 | 40 | 50 |  |  |  | 80 |  |  |
| 0,1 | 0 | 20 | 40 | 50 |  |  |  | 30 |  | 120 |
| 0,1,7 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 |  | 120 |
| 0,1,7,2 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 120 |
| 0,1,7,2,5,3 | 0 | 20 | 40 | 50 |  | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6 | 0 | 20 | 40 | 50 | 130 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9 | 0 | 20 | 40 | 50 | 110 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9,8 | 0 | 20 | 40 | 50 | 110 | 40 | 60 | 30 | 90 | 80 |
| 0,1,7,2,5,3,6,9,8,4 | 0 | 20 | 40 | 50 | 110 | 40 | 60 | 30 | 90 | 80 |

Nothing update by using vertex 4.

**Step 10:** Now Dijkstra’s algorithm stop because all vertices are in the visited set. The last line of table is the smallest weight path that from vertex 0 to other vertices.