GRAPHS

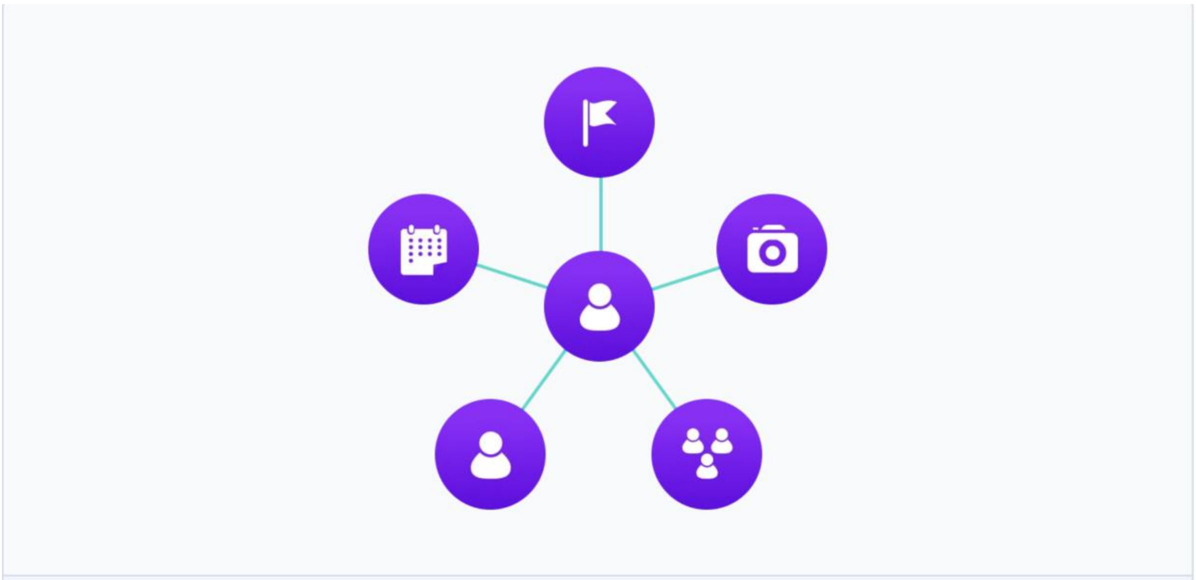
Definition

Graph Data Structure

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

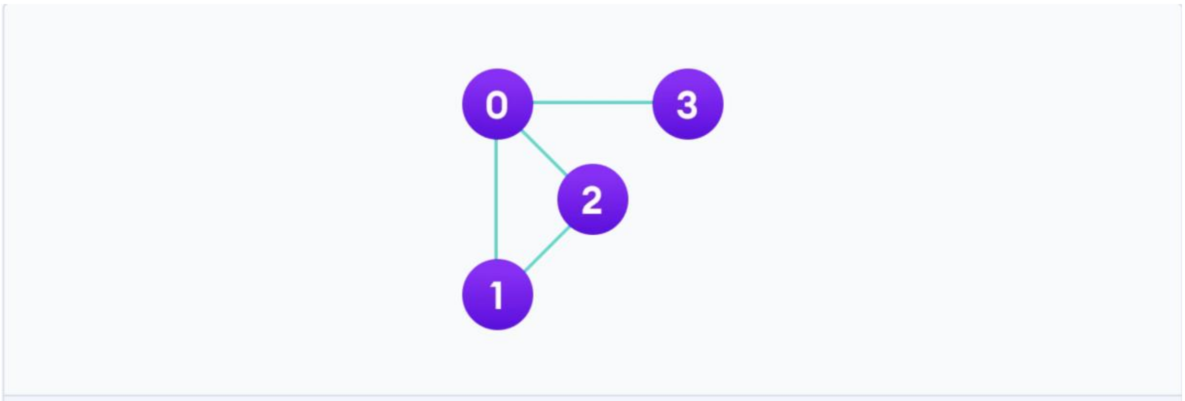


Example of graph data structure

All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)


Vertices and edges

In the graph,

V = {0, 1, 2, 3}

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

G = {V, E}

Graph Terminology

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

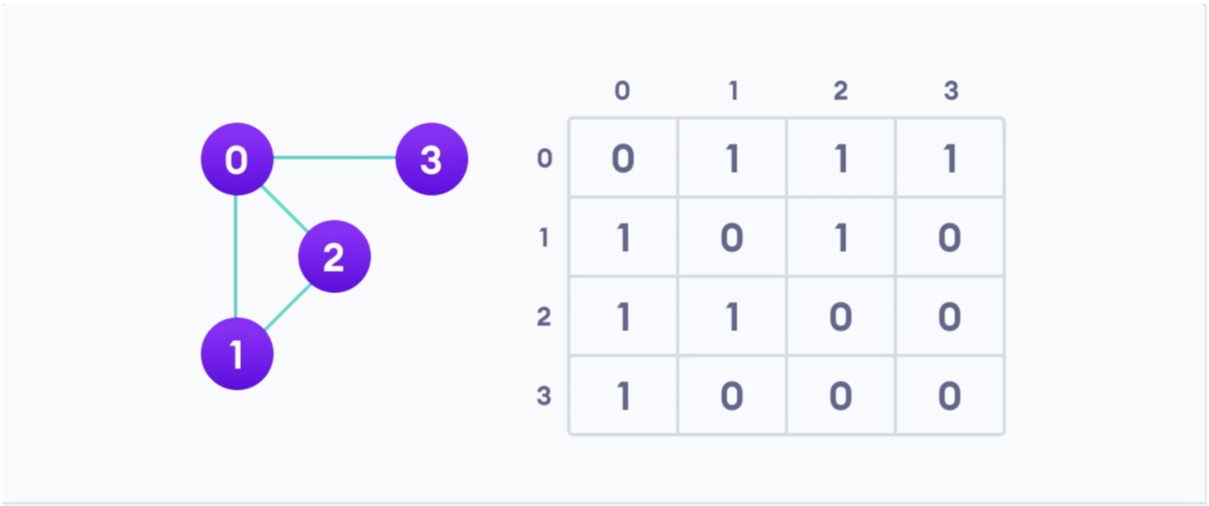
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

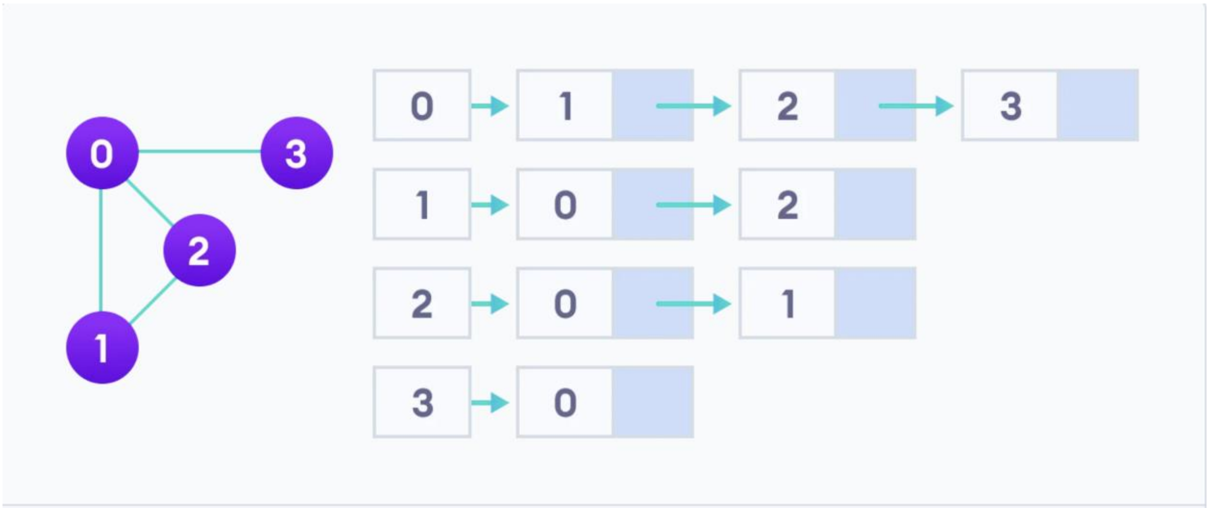
Graph adjacency matrix

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

Graph Operations

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements (vertex, edges) to graph
- Finding the path from one vertex to another

Important Algorithms

Depth First Search (DFS)

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

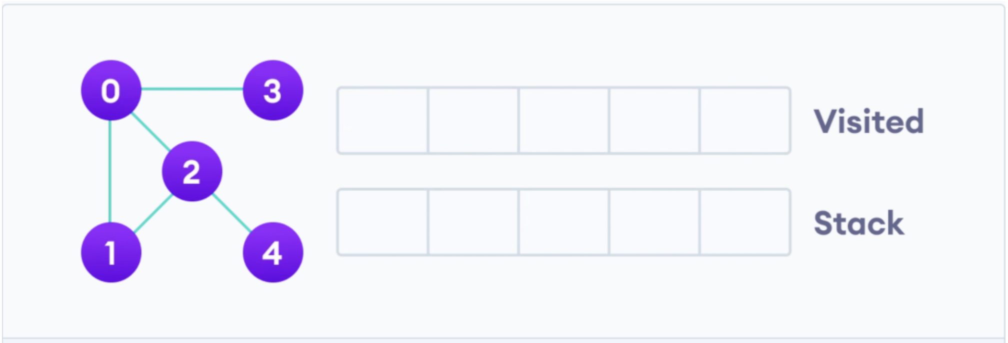
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

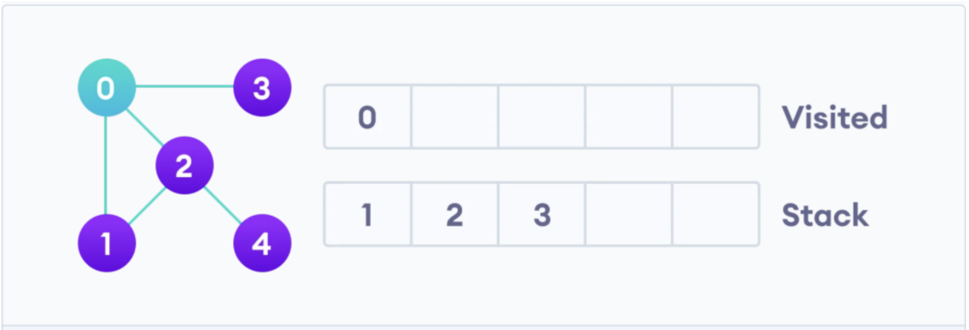
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



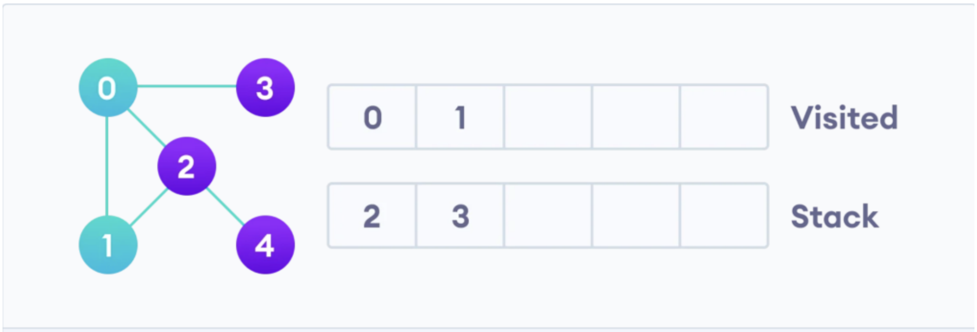
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



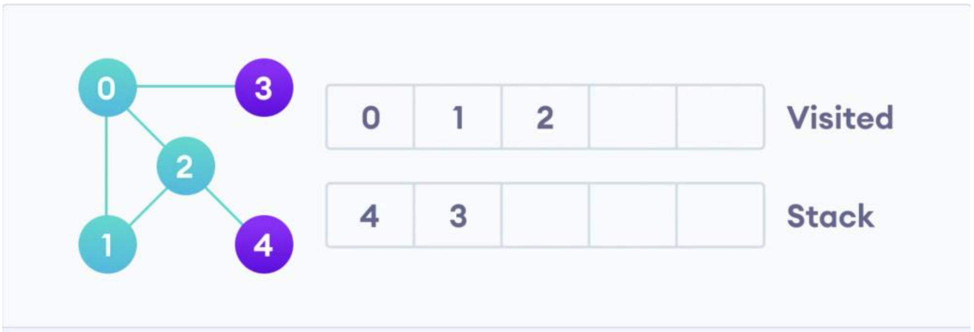
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e., 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

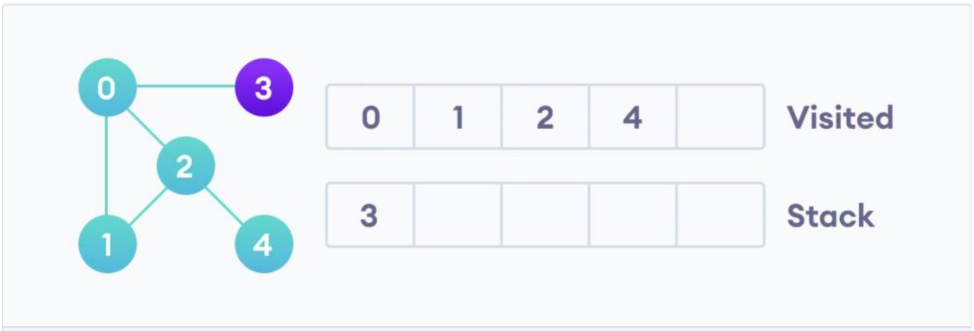


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

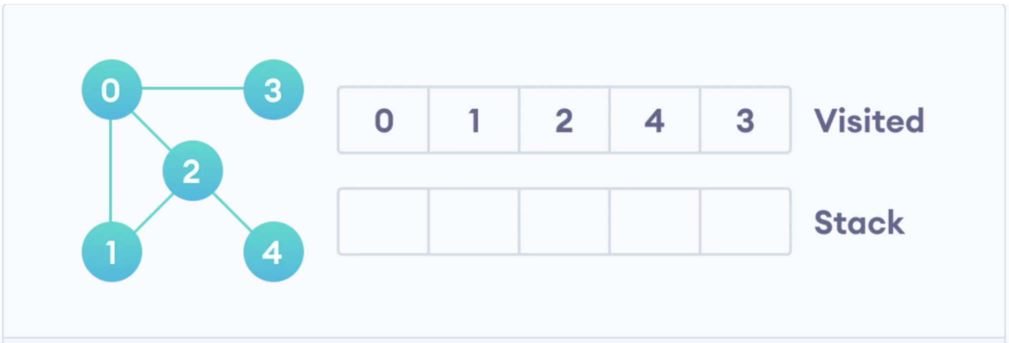


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

DFS Implementation in Java

```
// DFS algorithm in Java

import java.util.*;

class Graph {
  private LinkedList<Integer> adjLists[];
  private boolean visited[];

  // Graph creation
  Graph(int vertices) {
    adjLists = new LinkedList[vertices];
    visited = new boolean[vertices];

    for (int i = 0; i < vertices; i++)
      adjLists[i] = new LinkedList<Integer>();
  }

  // Add edges
  void addEdge(int src, int dest) {
    adjLists[src].add(dest);
  }

  // DFS algorithm
  void DFS(int vertex) {
    visited[vertex] = true;
    System.out.print(vertex + " ");

    Iterator<Integer> ite = adjLists[vertex].listIterator();
    while (ite.hasNext()) {
      int adj = ite.next();
      if (!visited[adj])
        DFS(adj);
    }
  }

  public static void main(String args[]) {
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    System.out.println("Following is Depth First Traversal");

    g.DFS(2);
  }
}
```

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Breadth First Search (BFS)

Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS Algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

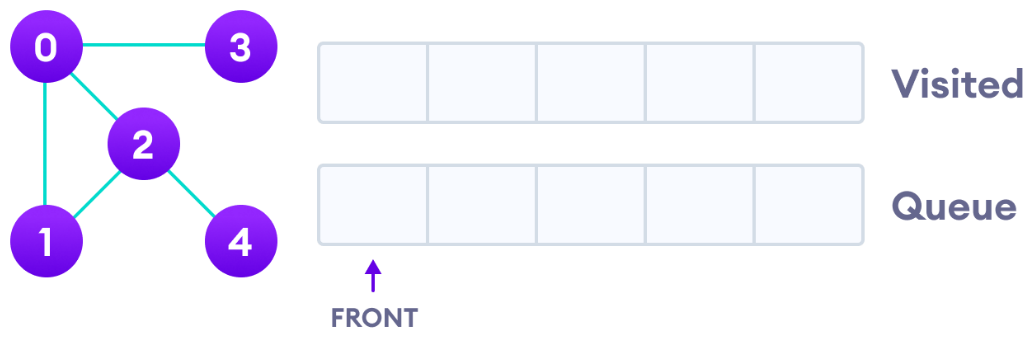
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

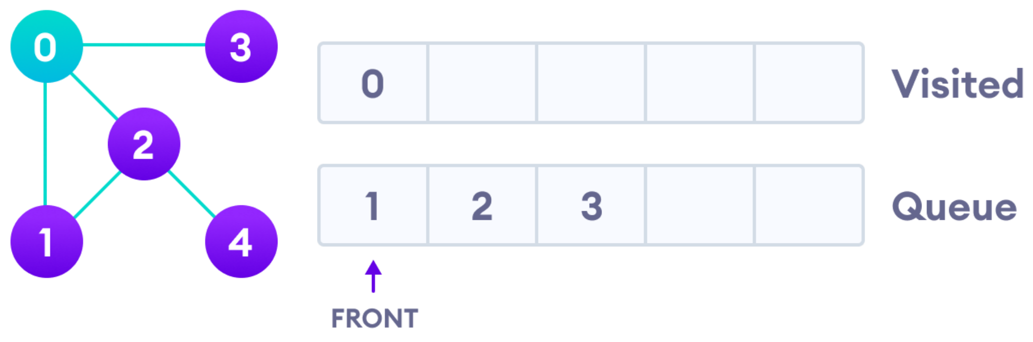
BFS Example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



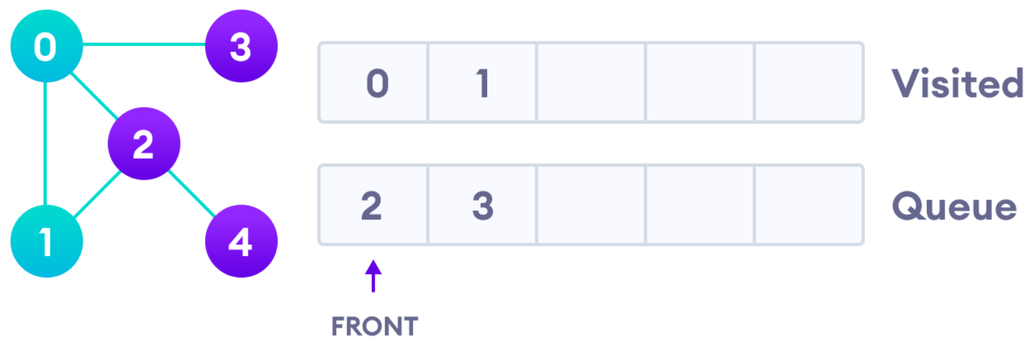
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

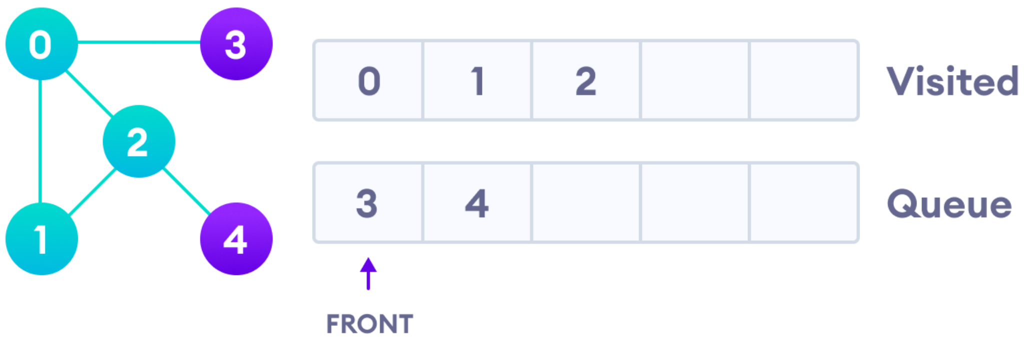


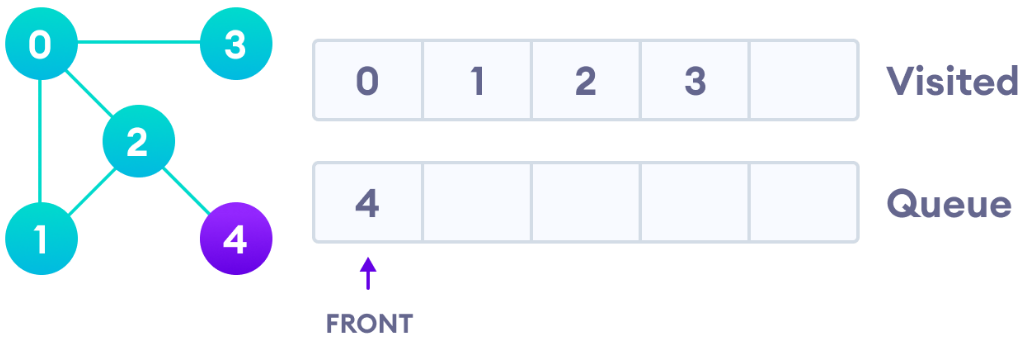
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e., 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

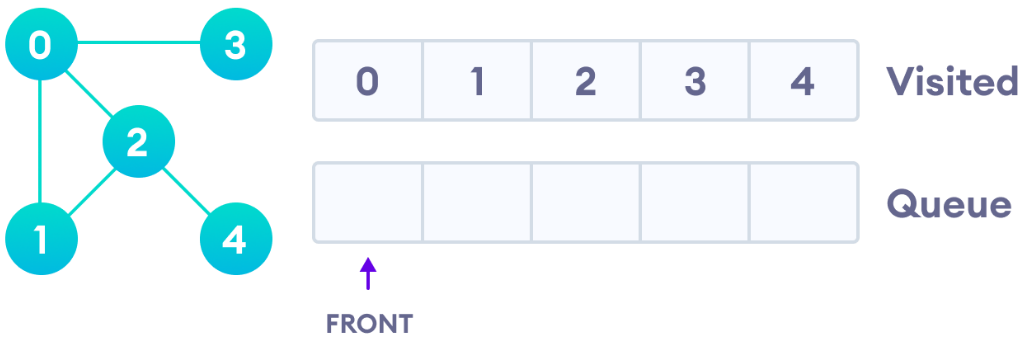


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e., 0 is already visited. We visit it.



Visit last remaining item in the stack to check if it has unvisited neighbours.

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS pseudocode

```
create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    mark and enqueue all (unvisited) neighbours of u
```

DFS Implementation in Java

```java
// BFS algorithm in Java

import java.util.*;

public class Graph {
  private int V;
  private LinkedList<Integer> adj[];

  // Create a graph
  Graph(int v) {
    V = v;
    adj = new LinkedList[v];
    for (int i = 0; i < v; ++i)
      adj[i] = new LinkedList();
  }

  // Add edges to the graph
  void addEdge(int v, int w) {
    adj[v].add(w);
  }

  // BFS algorithm
  void BFS(int s) {

    boolean visited [] = new boolean[V];

    LinkedList<Integer> queue = new LinkedList ();

    visited[s] = true;
    queue.add(s);

    while (queue.size() != 0) {
      s = queue.poll();
      System.out.print(s + " ");

      Iterator<Integer> i = adj[s].listIterator();
      while (i.hasNext()) {
        int n = i.next();
        if (!visited[n]) {
          visited[n] = true;
          queue.add(n);
        }
      }
    }
  }

  public static void main(String args[]) {
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal " + "(starting from
vertex 2)");

    g.BFS(2);
  }
}
```

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O (V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph

6. In minimum spanning tree

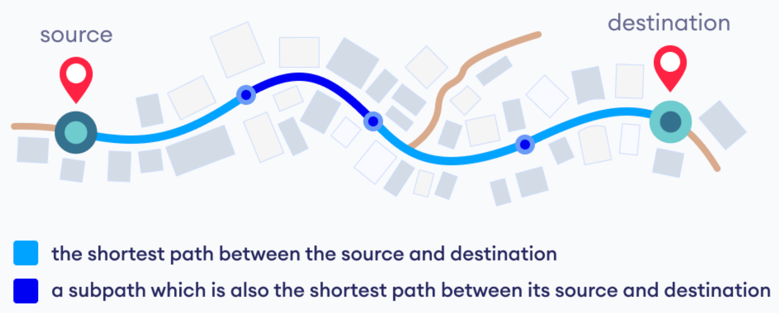
Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any sub path `B -> D` of the shortest path `A -> D` between vertices A and D is also the shortest path between vertices B and D.

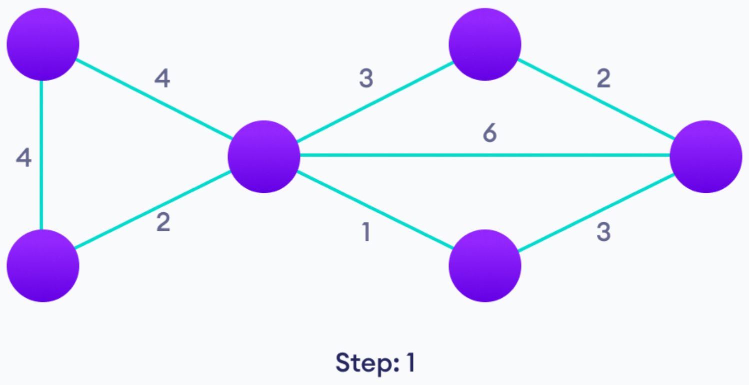

source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

Each sub path is the shortest path

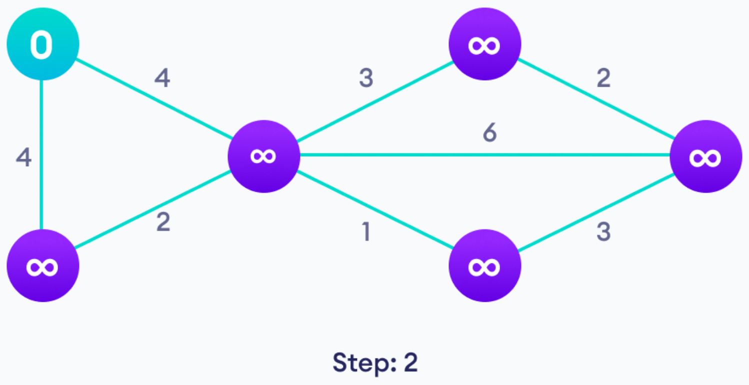
Dijkstra used this property in the opposite direction i.e., we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest sub path to those neighbours.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

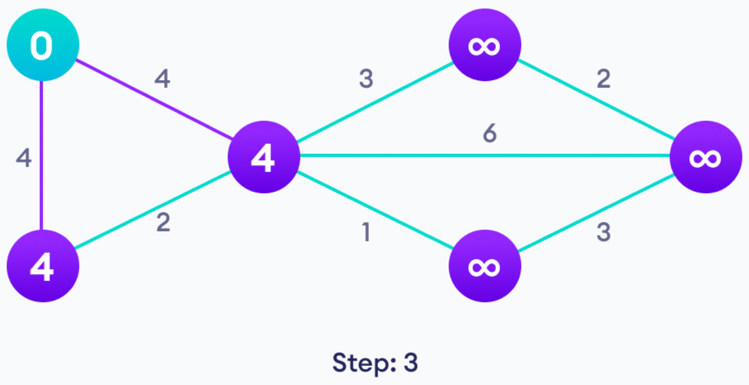
Example of Dijkstra's algorithm



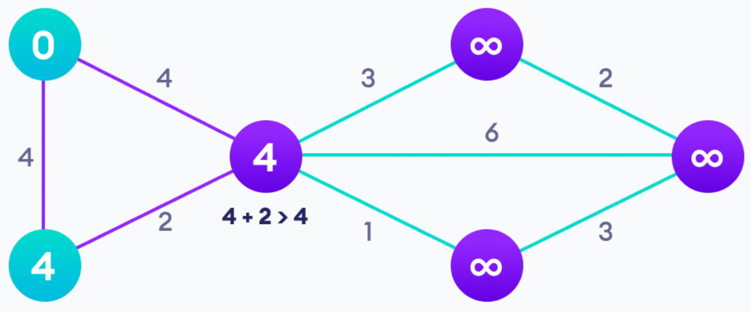
Start with a weighted graph



Choose a starting vertex and assign infinity path values to all other devices

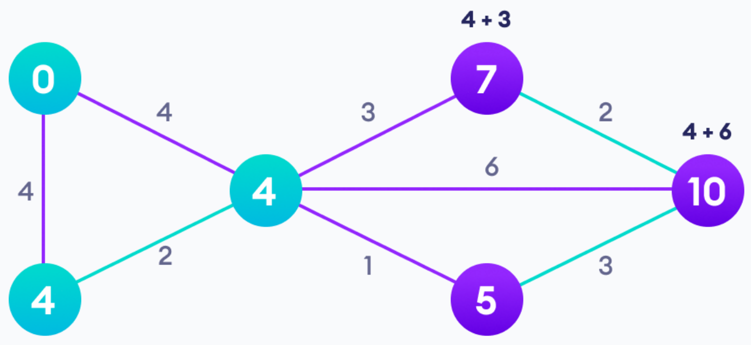


Go to each vertex and update its path length



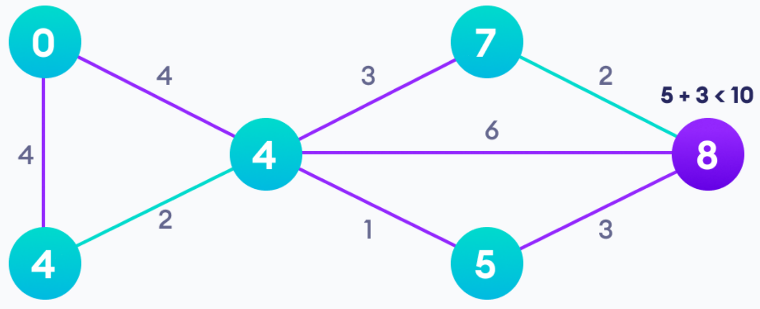
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



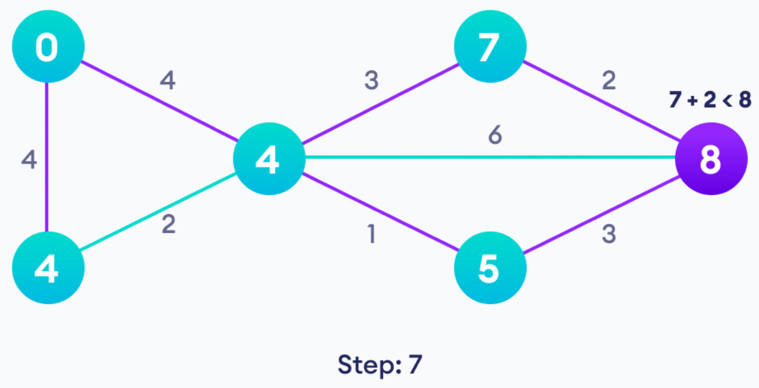
Step: 5

Avoid updating path lengths of already visited vertices

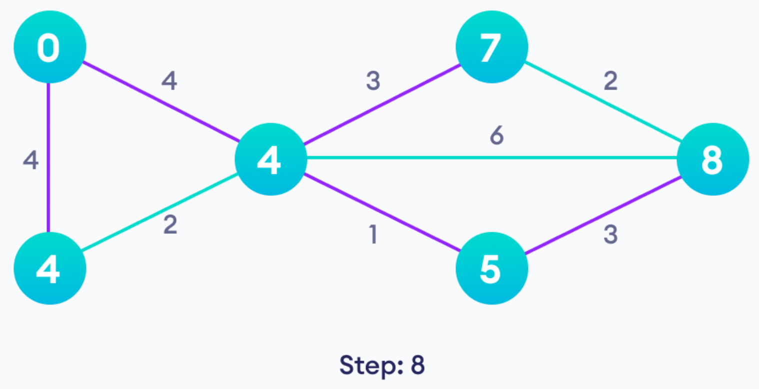


Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



Repeat until all the vertices have been visited

Dijkstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Code for Dijkstra's Algorithm

```java
// Dijkstra's Algorithm in Java

public class Dijkstra {

  public static void dijkstra(int[][] graph, int source) {
    int count = graph.length;
    boolean[] visitedVertex = new boolean[count];
    int[] distance = new int[count];
    for (int i = 0; i < count; i++) {
      visitedVertex[i] = false;
      distance[i] = Integer.MAX_VALUE;
    }

    // Distance of self loop is zero
    distance[source] = 0;
    for (int i = 0; i < count; i++) {

      // Update the distance between neighbouring vertex and source vertex
      int u = findMinDistance(distance, visitedVertex);
      visitedVertex[u] = true;

      // Update all the neighbouring vertex distances
      for (int v = 0; v < count; v++) {
        if (!visitedVertex[v] && graph[u][v] != 0 && (distance[u] + graph[u][v] <
distance[v])) {
          distance[v] = distance[u] + graph[u][v];
        }
      }
    }
    for (int i = 0; i < distance.length; i++) {
      System.out.println(String.format("Distance from %s to %s is %s", source, i,
distance[i]));
```

```
    }

  }

  // Finding the minimum distance
  private static int findMinDistance(int[] distance, boolean[] visitedVertex) {
    int minDistance = Integer.MAX_VALUE;
    int minDistanceVertex = -1;
    for (int i = 0; i < distance.length; i++) {
      if (!visitedVertex[i] && distance[i] < minDistance) {
        minDistance = distance[i];
        minDistanceVertex = i;
      }
    }
    return minDistanceVertex;
  }

  public static void main(String[] args) {
    int graph[][] = new int[][] { { 0, 0, 1, 2, 0, 0, 0 }, { 0, 0, 2, 0, 0, 3, 0 }, {
1, 2, 0, 1, 3, 0, 0 },
        { 2, 0, 1, 0, 0, 0, 1 }, { 0, 0, 3, 0, 0, 2, 0 }, { 0, 3, 0, 0, 2, 0, 1 }, {
0, 0, 0, 1, 0, 1, 0 } };
    Dijkstra T = new Dijkstra();
    T.dijkstra(graph, 0);
  }
}
```

Dijkstra's Algorithm Complexity

Time Complexity: O (E Log V)

where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Bellman Ford’s Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Why would one ever have edges with negative weights in real life?

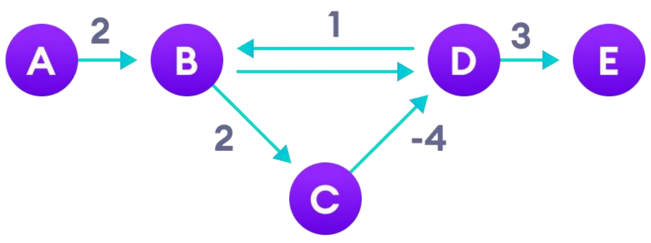
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.



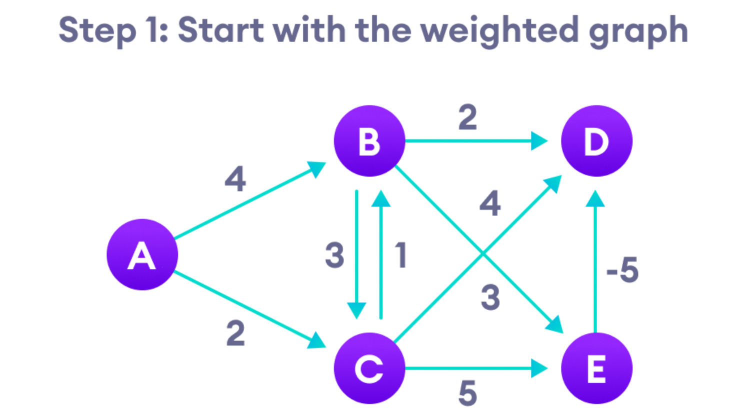
Negative weight cycles can give an incorrect result when trying to find out the shortest path

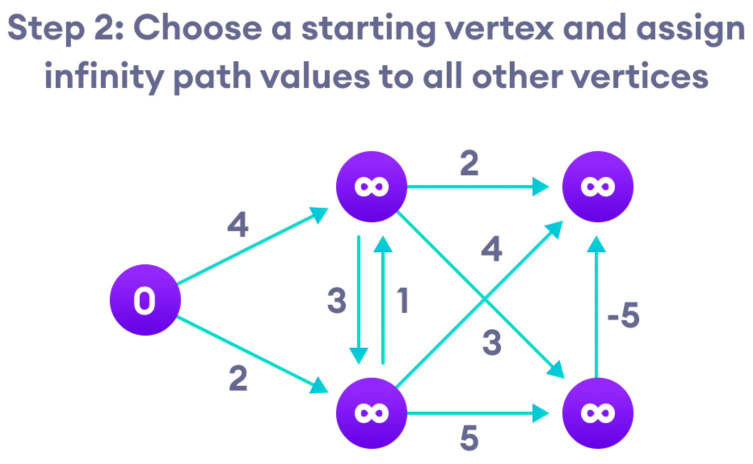
Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

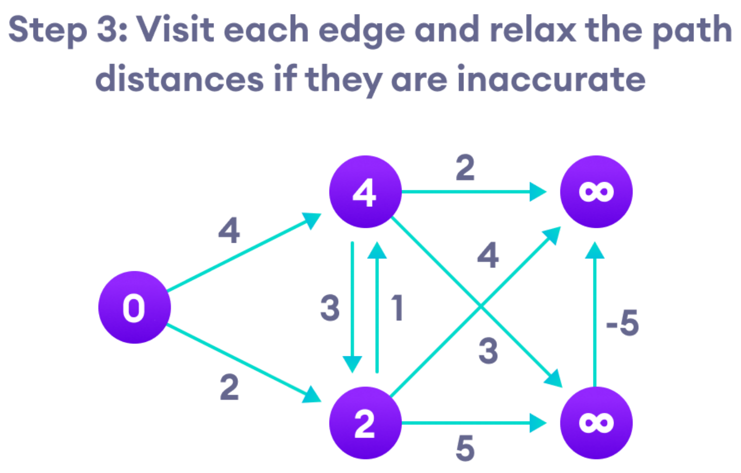
How Bellman Ford's algorithm works

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

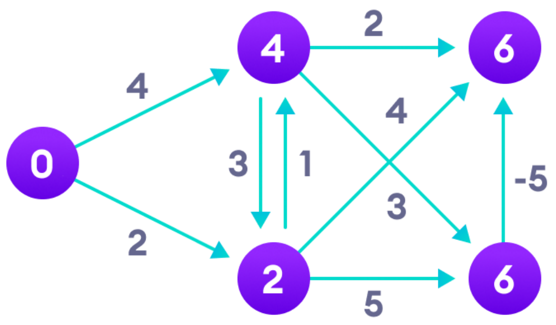
By doing this repeatedly for all vertices, we can guarantee that the result is optimized.



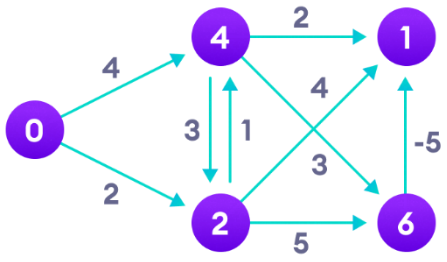

Step 2: Choose a starting vertex and assign infinity path values to all other vertices
0
∞
∞
∞
∞
4
2
3
1
2
4
3
-5
5


Step 3: Visit each edge and relax the path distances if they are inaccurate
0
4
2
∞
∞
4
2
3
1
2
4
3
-5
5

Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



Step 5: Notice how the vertex at the top right corner had its path length adjusted



Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists

  return distance[], previous[]
```

Bellman Ford vs Dijkstra

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbours of a vertex, Bellman goes through each edge in every iteration.

Implementation in Java

```java
// Bellman Ford Algorithm in Java

class CreateGraph {

  // CreateGraph - it consists of edges
  class CreateEdge {
    int s, d, w;

    CreateEdge() {
      s = d = w = 0;
    }
  };

  int V, E;
  CreateEdge edge[];

  // Creates a graph with V vertices and E edges
  CreateGraph(int v, int e) {
    V = v;
    E = e;
    edge = new CreateEdge[e];
    for (int i = 0; i < e; ++i)
      edge[i] = new CreateEdge();
  }

  void BellmanFord(CreateGraph graph, int s) {
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];

    // Step 1: fill the distance array and predecessor array
    for (int i = 0; i < V; ++i)
      dist[i] = Integer.MAX_VALUE;

    // Mark the source vertex
    dist[s] = 0;

    // Step 2: relax edges |V| - 1 times
    for (int i = 1; i < V; ++i) {
      for (int j = 0; j < E; ++j) {
        // Get the edge data
        int u = graph.edge[j].s;
        int v = graph.edge[j].d;
        int w = graph.edge[j].w;
        if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v])
          dist[v] = dist[u] + w;
      }
    }

    // Step 3: detect negative cycle
    // if value changes then we have a negative cycle in the graph
    // and we cannot find the shortest distances
    for (int j = 0; j < E; ++j) {
      int u = graph.edge[j].s;
      int v = graph.edge[j].d;
      int w = graph.edge[j].w;
      if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
        System.out.println("CreateGraph contains negative w cycle");
        return;
      }
    }

    // No negative w cycle found!
    // Print the distance and predecessor array
    printSolution(dist, V);
  }

  // Print the solution
  void printSolution(int dist[], int V) {
    System.out.println("Vertex Distance from Source");
    for (int i = 0; i < V; ++i)
      System.out.println(i + "\t\t" + dist[i]);
  }

  public static void main(String[] args) {
    int V = 5; // Total vertices
    int E = 8; // Total Edges

    CreateGraph graph = new CreateGraph(V, E);

    // edge 0 --> 1
    graph.edge[0].s = 0;
    graph.edge[0].d = 1;
    graph.edge[0].w = 5;

    // edge 0 --> 2
    graph.edge[1].s = 0;
    graph.edge[1].d = 2;
    graph.edge[1].w = 4;

    // edge 1 --> 3
    graph.edge[2].s = 1;
    graph.edge[2].d = 3;
    graph.edge[2].w = 3;

    // edge 2 --> 1
    graph.edge[3].s = 2;
    graph.edge[3].d = 1;
    graph.edge[3].w = 6;

    // edge 3 --> 2
    graph.edge[4].s = 3;
    graph.edge[4].d = 2;
    graph.edge[4].w = 2;

    graph.BellmanFord(graph, 0); // 0 is the source vertex
  }
}
```

Bellman Ford's Complexity

Time Complexity

Best Case Complexity	O(E)
Average Case Complexity	O(VE)
Worst Case Complexity	O(VE)

Space Complexity

And, the space complexity is O(V).

Bellman Ford's Algorithm Applications

1. For calculating shortest paths in routing algorithms
2. For finding the shortest path

Problem Statements

Iterative DFS

Explanation:

The non-recursive implementation of DFS is similar to the non-recursive implementation of BFS but differs from it in the following ways:

- It uses a stack instead of a queue.
- The DFS should mark discovered only after popping the vertex, not before pushing it.
- It uses a reverse iterator instead of an iterator to produce the same results as recursive DFS.

Water Jug problem using BFS

Explanation:

You are given an m litre jug and a n litre jug. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2.

Determine the path from the initial state (x_i, y_i) to the final state (x_f, y_f), where (x_i, y_i) is (0, 0) which indicates both Jugs are initially empty and (x_f, y_f) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

1. Empty a Jug, (X, Y) --> (0, Y) Empty Jug 1
2. Fill a Jug, (0, 0) --> (X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) --> (X-d, Y+d)

Example:

Input: 4 3 2

Output: {(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)}

Approach:

Here, we keep exploring **all the different valid cases of the states of water in the jug simultaneously** until and unless we reach the required target water.

As provided in the problem statement, at any given state we can do either of the following operations:

1. Fill a jug
2. Empty a jug
3. Transfer water from one jug to another **until either of them gets completely filled or empty.**

Algorithm:

We start at an initial state in the queue where both the jugs are empty. We then continue to explore all the possible intermediate states derived from the current jug state using the operations provided.

We also, maintain a visited matrix of states so that we avoid revisiting the same state of jugs again and again.

Cases	Jug 1	Jug 2	Is Valid
Case 1	Fill it	Empty it	✓
Case 2	Empty it	Fill it	✓
Case 3	Fill it	Fill it	Redundant case
Case 4	Empty it	Empty it	Already visited (Initial State)
Case 5	Unchanged	Fill it	✓
Case 6	Fill it	Unchanged	✓
Case 7	Unchanged	Empty	✓
Case 8	Empty	Unchanged	✓
Case 9	Transfer from	Transfer into	✓
Case 10	Transfer into	Transfer from	✓

From the table above, we can observe that the state **where both the jugs are filled is redundant** as we won't be able to continue ahead / do anything with this state in any possible way.

So, we proceed, **keeping in mind all the valid state cases** (as shown in the table above) and we do a BFS on them.

In the BFS, we firstly skip the states which was already visited or if the amount of water in either of the jugs exceeded the jug quantity.

If we continue further, then we firstly mark the current state as visited and check if in this state, if we have obtained the target quantity of water in either of the jugs, we can empty the other jug and return the current state's entire path.

But, if we have not yet found the target quantity, we then derive the intermediate states from the current state of jugs i.e. we derive the valid cases, mentioned in the table above (go through the code once if you have some confusion).

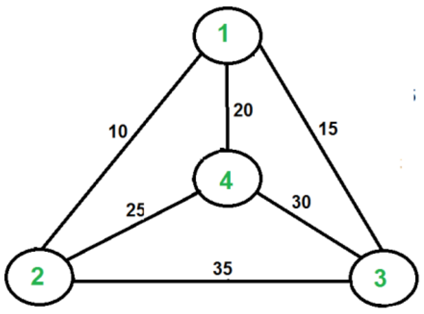
We keep repeating all the above steps until we have found our target or there are no more states left to proceed with.

Travelling Salesman Problem

Explanation:

Given a set of cities and distance between every pair of cities as an adjacency matrix, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note - The difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Simple Approach:

- Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
- Now, we will generate all possible permutations of cities which are (n-1)!
- Find the cost of each permutation and keep track of the minimum cost permutation.
- Return the permutation with minimum cost.

Dynamic Programming Approach:

In this algorithm, we take a subset N of the required cities that need to be visited, the distance among the cities dist, and starting city s as inputs. Each city is identified by a unique city id which we say like 1,2,3,4,5………n

Here we use a dynamic approach to calculate the cost function Cost(). Using recursive calls, we calculate the cost function for each subset of the original problem.

To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on.

There are at most O(n2^n) subproblems, and each one takes linear time to solve. The total running time is, therefore, O(n^22^n). The time complexity is much less than O(n!) but still exponential. The space required is also exponential.

Greedy Approach:

- First of all, we have to create two primary data holders.
    - First of them is a list that can hold the indices of the cities in terms of the input matrix of distances between cities

    - And the Second one is the array which is our result
- Perform traversal on the given adjacency matrix tsp[][] for all the city and if the cost of reaching any city from the current city is less than the current cost the update the cost.
- Generate the minimum path cycle using the above step and return their minimum cost.

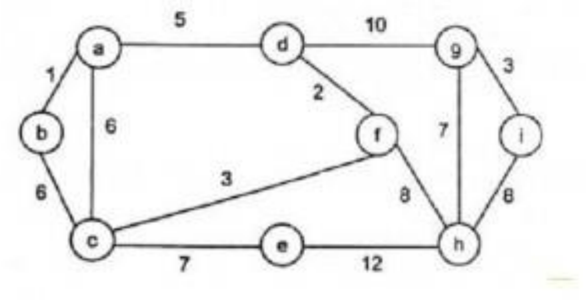
Cycles of length n in an undirected and connected graph (For the learners to try)

Assessment

1) Which of the following is an advantage of adjacency list representation over adjacency matrix representation of a graph?
    a. In adjacency list representation, space is saved for sparse graphs.
    b. DFS and BSF can be done in O(V + E) time for adjacency list representation. These operations take O(V^2) time in adjacency matrix representation. Here is V and E are number of vertices and edges respectively.
    c. Adding a vertex in adjacency list representation is easier than adjacency matrix representation.
    d. All of the above

2) Given an undirected graph G with V vertices and E edges, the sum of the degrees of all vertices is
    a. E
    b. 2E
    c. V
    d. 2V

3) For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Spanning Tree?



    a. (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)
    b. (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)
    c. (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)
    d. (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

4) Consider a directed graph with n vertices and m edges such that all edges have same edge weights. Find the complexity of the best known algorithm to compute the minimum spanning tree of the graph?
    a. O (m+n)

b. O (m logn)
c. O (mn)
d. O (n logm)

5) Which of the following data structure is useful in traversing a given graph by breadth first search?
   a. Stack
   b. List
   c. Queue
   d. None of the above

6) If we want to search any name in the phone directory , which is the most efficient data structure?
   a. Binary Search Tree
   b. Balanced BST
   c. Trie
   d. Linked List