# Table of Contents

**BASICS OF GRAPH THEORY** ........ **1**

Graph theory, vertex (node), edge, directed and undirected graph, weighted and unweighted graph ........ 1
Representation of graphs ........ 1
Adjacency List ........ 1
Adjacency Matrix ........ 1
Taking a graph as input from keyboard and storing it into memory ........ 2
A simple C/C++ program to input and store a *directed unweighted* graph ........ 2

**BFS (BREADTH-FIRST SEARCH)** ........ **3**

Algorithm (informal) ........ 3
Applications ........ 3
Graph traversing using BFS (in C++) ........ 3
Finding the shortest path between the source and another node ........ 5
Finding the most distant node from the source node ........ 6
Printing the (shortest) path from source to destination ........ 6
Finding a cycle from *source* and printing its path ........ 6
Detecting *any* cycle within an undirected graph ........ 6
Testing a graph for bipartiteness ........ 7
Finding all connected components in a graph ........ 7
Finding all nodes *within* one connected component ........ 7

**DFS (DEPTH-FIRST SEARCH)** ........ **8**

Algorithm (Pseudocode) ........ 8
Graph traversing using DFS (in C++) ........ 8
Checking whether a graph is connected ........ 9
Edge detection ........ 9
Use of time in DFS ........ 10
Detecting cycles in a directed graph ........ 11
Theorem: An undirected graph is *acyclic iff* a DFS yields no back edges ........ 11
Topological Sort ........ 11
Unique Topological Sort ........ 12
Articulation Point / Cut-Vertex and Biconnected Components ........ 12
Bridge ........ 13
SCC (Strongly Connected Components) ........ 14

**DIJKSTRA'S ALGORITHM** ........ **15**

**BELLMAN-FORD ALGORITHM** ........ **17**

**SINGLE-SOURCE SHORTEST PATHS IN DAGS** ........ **18**

**FLOYD-WARSHALL ALGORITHM** ........ **19**

**PRIM'S ALGORITHM** ........ **20**

**KRUSKAL'S ALGORITHM** ........ **22**

# BASICS OF GRAPH THEORY

**Graph theory, vertex (node), edge, directed and undirected graph, weighted and unweighted graph**

In mathematics and computer science, graph theory is the study of *graphs*: mathematical structures used to model pair-wise relations between objects from a certain collection. A *graph* in this context refers to a collection of *vertices* or *nodes* and a collection of *edges* that connect pairs of vertices. A graph may be *undirected*, meaning that there is no distinction between the two vertices associated with each edge; or its edges may be *directed* from one vertex to another. If some value is assigned to the edges of a graph, then that graph is called a *weighted* graph. If the weights of all the edges are the same, then the graph is called an *unweighted* graph.

**Representation of graphs**

Graphs are commonly represented in two ways:

1. Adjacency List
2. Adjacency Matrix

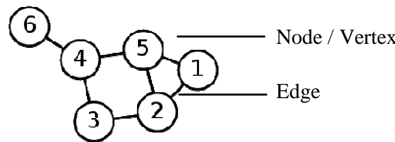



**Figure 1:** An *undirected unweighted* graph.

**Adjacency List**

In this representation, each vertex has a list of which vertices it is adjacent to. This causes redundancy in an undirected graph: for example, if vertices *A* and *B* are adjacent, *A*'s adjacency list contains *B*, while *B*'s list contains *A*. Adjacency queries are faster, at the cost of extra storage space.

For example, the undirected graph in *Figure 1* can be represented using adjacency list as follows:

| Node | Adjacency List |
|---|---|
| 1 | 2, 5 |
| 2 | 1, 3, 5 |
| 3 | 2, 4 |
| 4 | 3, 5, 6 |
| 5 | 1, 2, 4 |
| 6 | 4 |

**Adjacency Matrix**

This is the *n* by *n* matrix, where *n* is the number of vertices in the graph. If there is an edge from some vertex *x* to some vertex *y*, then the element $a_{x,\,y}$ is 1 (or, in case of weighted graph, the weight of the edge connecting *x* and *y*), otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

For example, the undirected graph in *Figure 1* can be represented using adjacency matrix as follows:

| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| **1** | 0 | 1 | 0 | 0 | 1 | 0 |
| **2** | 1 | 0 | 1 | 0 | 1 | 0 |
| **3** | 0 | 1 | 0 | 1 | 0 | 0 |
| **4** | 0 | 0 | 1 | 0 | 1 | 1 |
| **5** | 1 | 1 | 0 | 1 | 0 | 0 |
| **6** | 0 | 0 | 0 | 1 | 0 | 0 |

### Taking a graph as input from keyboard and storing it into memory

It is easier to store a graph into memory and perform operations on it using adjacency matrix rather than adjacency list. First, we need to declare a global 2D array. Then, we should prompt the user to input the edges. The user should provide 2 numbers ($x$, $y$) representing the edge between those two nodes (nodes will be denoted as numbers rather than letters). For each pair of ($x$, $y$), the corresponding location $a_{x,\ y}$ at the matrix will be filled with a '1'. This is the case for an *unweighted*, but *directed* graph. In case of unweighted, but *directed* graph, the location $a_{y,\ x}$ should also be filled with a '1'. In case of *weighted* graph, the user should input another number indicating the weight, and *that* number should be stored at the location instead of '1'. After the input of the edges, all other locations should be filled with '0'. However, in case of C, C++ or Java, this is automatically done as global variables are automatically zero-filled when initialized.

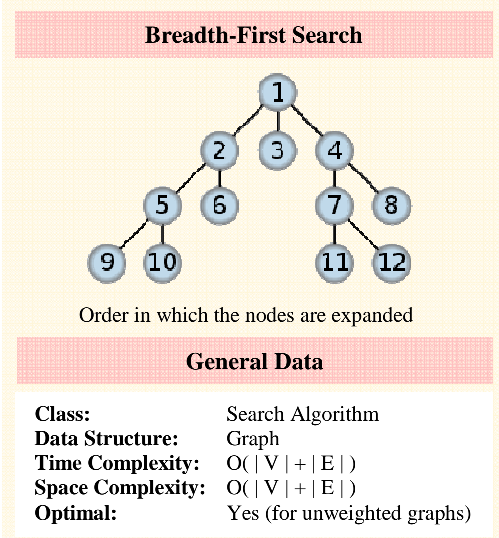
### A simple C/C++ program to input and store a *directed unweighted* graph

```
1  #include <stdio.h>
2  
3  int graph[100][100];     //A matrix for storing a graph containig 100 nodes maximum
4  
5  int main(int argc, char** argv) {
6      printf("Enter number of edges: ");
7      int edges;
8      scanf("%d", &edges);
9      for (int i = 1; i <= edges; i++) {
10         printf("Enter edge %d: ", i);
11         int x, y;                 //Or, int x, y, weight; - for storing weight of edge
12         scanf("%d %d", &x, &y); //Or, scanf("%d %d %d", &x, &y, &weight); - for weighted graph
13         graph[x][y] = 1;          //Or, graph[x][y] = weight; - for weighted graph
14         //graph[y][x] = 1;        //This line should be added for undirected graph
15     }
16 
17     return 0;
18 }
19 
```

# BFS (BREADTH-FIRST SEARCH)

### *Finds single-source shortest path in unweighted graph*

In graph theory, *breadth-first search* (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.




### Algorithm (informal)

1. Put the root node on the queue.
2. Pull a node from the beginning of the queue and examine it.
3. If the searched element is found in this node, quit the search and return a result.
4. Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
5. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
6. Repeat from Step 2.

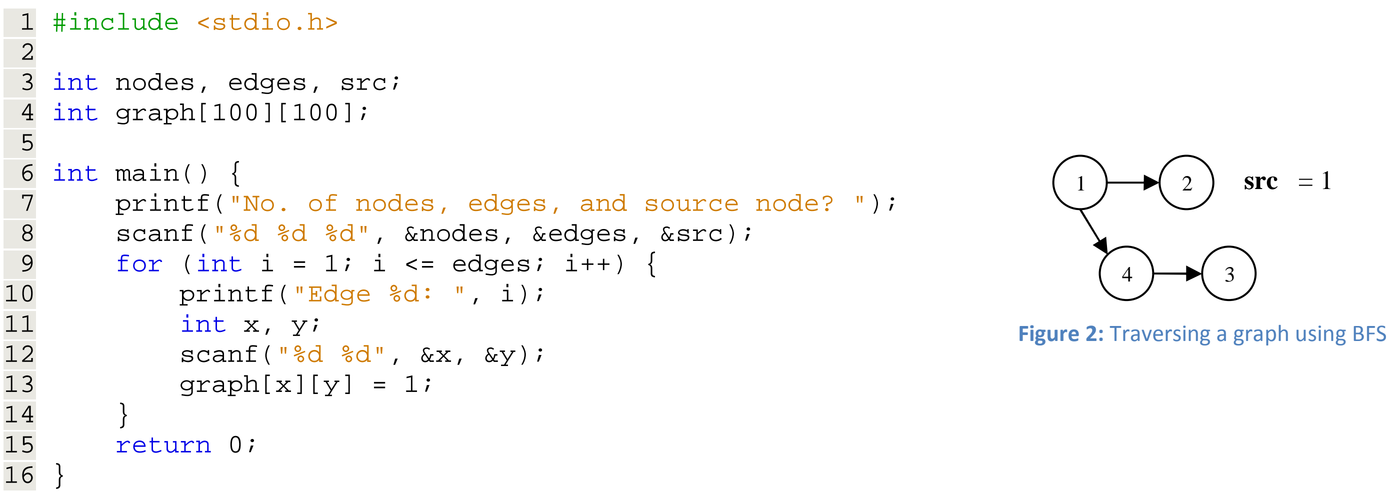
### Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all connected components in a graph
- Finding all nodes within one connected component
- Finding the shortest path between two nodes *u* and *v*
- Testing a graph for bipartiteness

### Graph traversing using BFS (in C++)

Suppose, we have to traverse the directed graph of *Figure 2*. We'll start from the node 'src'. Let's assume we have completed the preliminary task of taking input of the graph[1]:



```
1  #include <stdio.h>
2
3  int nodes, edges, src;
4  int graph[100][100];
5
6  int main() {
7      printf("No. of nodes, edges, and source node? ");
8      scanf("%d %d %d", &nodes, &edges, &src);
9      for (int i = 1; i <= edges; i++) {
10         printf("Edge %d: ", i);
11         int x, y;
12         scanf("%d %d", &x, &y);
13         graph[x][y] = 1;
14     }
15     return 0;
16 }
```



Figure 2: Traversing a graph using BFS

[1] Notice that we're taking the number of nodes as input. Why? Initially, we're taking a large matrix for *100* nodes. But we need the actual number of nodes in the graph when we try to traverse it. We'll be using a loop, and it should stop running when we've completed examining the last node.

Now, let’s try to implement the BFS traverse algorithm:

```
19      //run BFS
20      queue<int> q;            //create a queue
21      q.push(src);             //1. put root node on the queue
22      do {
23          int u = q.front();   //2. pull a node from the beginning of the queue
24          q.pop();
25          printf("%d ", u);    //print the node
26          for (int i = 1; i <= nodes; i++) {  //4. get all the adjacent nodes
27              if (graph[u][i] == 1) {   //if an edge exists between these two nodes
28                  q.push(i);   //4. push this node into the queue
29              }
30          }
31      } while (!q.empty());    //5. if the queue is empty, then all the nodes have been visited
```

**Analysis of the above program (i.e. how it works)**

| Step | Line | Queue | u | `graph[u][i]` | Output |
|---|---|---|---|---|---|
| **1** | 21 | 1 | - | - | - |
| **2** | 23, 24, 25 | - | 1 | - | 1 |
| **3** | 26, 27 | - | | `graph[1][2]` | |
| **4** | 28 | 2 | | | |
| **5** | 26, 27 | 2 | | `graph[1][4]` | |
| **6** | 28 | 4 2 | | | |
| **7** | 31 | 4 2 | - | - | |
| **8** | 23, 24, 25 | 4 | 2 | - | 1 2 |
| **9** | 31 | 4 | - | - | |
| **10** | 23, 24, 25 | - | 4 | - | |
| **11** | 26, 27 | - | | `graph[4][3]` | |
| **12** | 28 | 3 | | | |
| **13** | 31 | 3 | - | - | |
| **14** | 23, 24, 25 | - | 3 | - | 1 2 3 |
| **15** | 31 | - | | | **1 2 3** |

i →

| | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **1** | 0 | 1 | 0 | 1 |
| **2** | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 1 | 0 |

The `graph` array

**A problem with the above program**

However, our program is correct for the graph in *Figure 2*, but it won’t work for *any* graph. Consider the graph in *Figure 3*. When our program finds that node 1 is adjacent to node 4, it will enqueue it. Then, when it will see that node 4 is adjacent to node 1, it will enqueue it again. This will continue forever. So, we must have a mechanism to detect whether a node has *already been* visited.

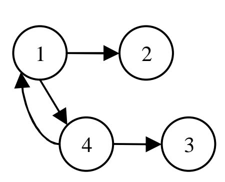


Figure 3

We can use an array of nodes named `visited`, which will record a 1 if a node has been used (i.e., all the adjacent of it have been discovered), and a 0 otherwise. But there lies another problem. Again, this solution will work for the graph in *Figure 3*, but won’t work for *any* graph. Consider the graph of *Figure 4*. At the point of finding the adjacents of node 4, we enqueue nodes 2 and 3 and record a 1 to node 4. But when we’ll search for adjacents of 2, we’ll again queue node 3.

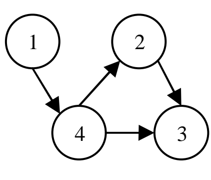


Figure 4

To solve this problem, we can use two numbers instead of one – we’ll assign 0 to nodes which haven’t yet been touched, 1 to nodes which we’ve just reached out, and 2 to a node when we’re finished with searching its adjacent nodes. We’ll try to search for adjacents of *only* those nodes whose assigned value is *1*. Let’s call this process ‘graph coloring’. First, all the nodes are colored white (0). We’ll color each

*discovered* node with gray (1) and then when we're finished finding all the adjacent nodes of a particular node, we'll color it with black (2)[2].

The modified version of the program is as follows:

```
1 #include <stdio.h>
2 #include <queue>
3 using namespace std;
4
5 int nodes, edges, src;
6 int graph[100][100], color[100];
7 const int WHITE = 0;
8 const int GRAY = 1;
9 const int BLACK = 2;
10
11 int main() {
12     printf("Nodes, edges, source? ");
13     scanf("%d %d %d ", &nodes, &edges, &src);
14     for (int i = 1; i <= edges; i++) {
15         printf("Edge %d: ", i);
16         int x, y;
17         scanf("%d %d", &x, &y);
18         graph[x][y] = 1;
19     }
20
21     //run BFS
22     queue<int> q;            //create a queue
23     q.push(src);             //1. put root node on the queue
24     do {
25         int u = q.front();   //2. pull a node from the beginning of the queue
26         q.pop();
27         printf("%d ", u);    //print the node
28         for (int i = 1; i <= nodes; i++) {  //4. get all the adjacent nodes
29             if ((graph[u][i] == 1)  //if an edge exists between these two nodes,
30                 && (color[i] == WHITE)) {    //and this adjacent node is still WHITE,
31                 q.push(i);                   //4. push this node into the queue
32                 color[i] = GRAY;             //color this adjacent node with GRAY
33             }
34         }
35         color[u] = BLACK;    //color the current node black to mark it as dequeued
36     } while (!q.empty());    //5. if the queue is empty, then all the nodes have been visited
37
38     return 0;
39 }
```

### Finding the shortest path between the source and another node

Suppose for a given directed unweighted graph, we have to find the shortest path from 'src' to 'dest', where 'src' is the source node and 'dest' is the destination node (*Figure 5*).

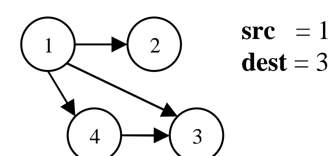


Figure 5: Finding the shortest path.

So, while taking input of the graph, we have to take another input – `dest`.

Now, first, how do we find the *distance* between two nodes (e.g. nodes 1 and 3) using BFS? (Forget about finding the *shortest* distance for the time being.) Let's take the route 1 → 4 → 3. The distance between 1 and 4 is 1, and 4 and 3 is also 1. So, the total distance between 1 and 3 is 1 + 1 = 2. Therefore, starting from the root, the most adjacent nodes of the source are of distance 1. The distance of the next level of nodes from the source is of distance 1 *plus* the distance of their just previous adjacent nodes. So, we can use an array – something named `distance` – to store the distance of each node (from the source node).

[2] We need to color the dequeued node with BLACK if we want to detect a cycle within a graph. We'll be discussing it in a while.

Now, because in BFS a node is enqueued only once, we can be assured that its distance from the source is the shortest distance.

Then, what changes should we bring to our program?

1. Declare an `int` array `distance[100]` on line 6.
2. Add the statement `distance[i] = distance[u] + 1;` after line 32.
3. Finally, the shortest path between 'src' and 'dest' is the value of `distance[dest]`.

**Finding the most distant node from the source node**

The lastly queued node is the most distant node from the source node. So, the value of the variable *u* after finishing the BFS is our most distant node from the source node. However, in our program, *u* is declared as a local variable *inside* the `do-while` loop. To get its value *after* the loop, we need to declare it *outside* the loop.

**Printing the (shortest) path from source to destination**

To print the path from a source to a destination, we need to keep track of the intermediate nodes between those two nodes. We can use an array to store the previous node of the current node. Thus we can get a chain of nodes between the source and destination nodes from that array.

To store the previous node in an array, we can simply declare a global array `prev[100]`, and add the statement `prev[i] = u;` after line 32.

Now, how do we print the path from that array? The `prev` array for the graph of *Figure 4* would somewhat look like the array in *Figure 6*. Here, we have to *backtrack* from destination to source. So, to print *forward* from source to destination, we can use a recursion like the following:

```
13 void print(int node) {
14     if (node == 0)
15         return;
16     print(prev[node]);
17     printf("%d -> ", node);
18 }
```

| | |
|---|---|
| **1** | 0 |
| **2** | 4 |
| **3** | 4 |
| **4** | 1 |

Figure 6

**Finding a cycle from *source* and printing its path**

For any node, if we can find out that there is an edge *from* it *to* the source, we can easily say that a cycle exists from source to that node. Now, to print the path, simply print the path from source to that node as described above, and then print the source node again. The code is somewhat like below:

```
57     //find and print cycle from source
58     for (int i = 1; i <= nodes; i++) {
59         if (graph[i][src] == 1) {
60             print(i);
61             printf("%d\n\n", src);
62         }
63     }
```

**Detecting *any* cycle within an undirected graph**

If we can detect that there exists an edge from a WHITE or GRAY node to a BLACK node, we can easily say that a cycle exists from that BLACK node to the WHITE / GRAY node.

### Testing a graph for bipartiteness

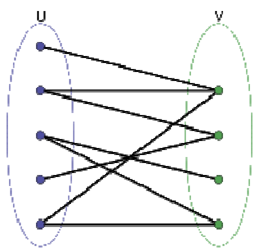


**Figure 7:** Bipartite Graph

In graph theory, a *bipartite graph* (or *bigraph*) is a graph whose vertices can be divided into two disjoint sets *U* and *V* such that every edge connects a vertex in *U* to one in *V*.

First, we should color the source node with a particular color – say BLUE. Then, we have to color all the adjacent nodes with another color – say RED. Similarly, we'll color all the adjacent nodes of RED nodes as BLUE and all the adjacent nodes of BLUE nodes as RED. While performing this, if we encounter any node which already has a similar color to his adjacent node, then we can conclude that the graph is *not* bipartite. Else, after finishing running the BFS, we can conclude that the graph *is* bipartite.

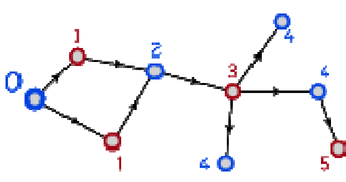


**Figure 8:** Testing a graph for bipartiteness

To apply this logic, we need to define *another* array of nodes containing another three colors – say GREEN, RED and BLUE. If an adjacent node's color is GREEN, we'll color it with a different color than the current node (i.e., BLUE if the color of current node is RED and vice-versa). If we find that the adjacent's color is already different from the color of the current node, then no problem; we'll simply skip. However, if we find that the adjacent's color is already the same as the color of the current node, then we'll stop and notify that the graph is not bipartite. The code for this might look like the following:

```
31      if (parity[i] == GREEN) {
32          if (parity[u] == BLUE) {
33              parity[i] = RED;
34          } else {
35              parity[i] = BLUE;
36          }
37      } else if (parity[i] == parity[u]) {
38          //Break and notify that the graph is not bipartite...
39      }
```

**Remember:** Every *tree* is bipartite. Also, cycle graphs with an even number of edges are bipartite.

### Finding all connected components in a graph

In an undirected graph, a *connected component*, or simply, *component* is a maximal connected subgraph. There are three connected components in *Figure 9*. Two vertices are defined to be in the same connected component if there exists a *path* between them. A graph is called *connected* when there is *exactly one* connected component.

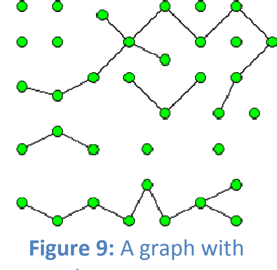


**Figure 9:** A graph with three components

To find all the connected components in a graph, first, run a BFS from any node. After the BFS ends, detect which nodes aren't traversed (using color). From any of those untraversed nodes, run another BFS. Repeat the process until all the nodes have been traversed. The number of times the BFS is run, the number of components the graph has.

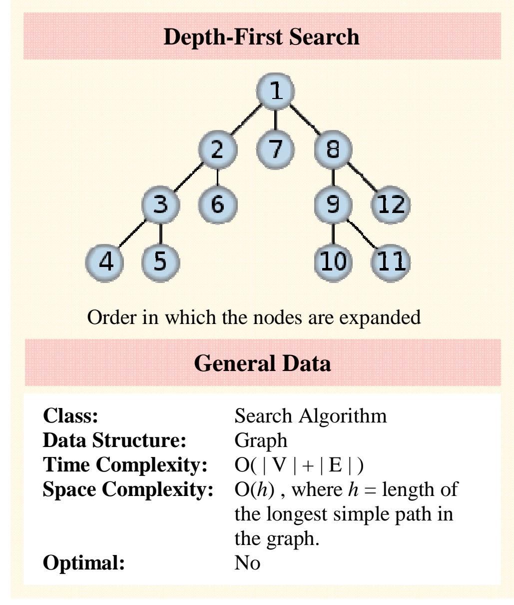
### Finding all nodes *within* one connected component

Simple – just run the BFS once!

# DFS (Depth-First Search)

*Generates spanning tree from a graph*

In graph theory, *depth-first search* (DFS) is an uninformed search[3] that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring.




### Algorithm (Pseudocode)

```
function dfs(vertice v) {
     mark v as visited;
     preorder-process(v);
     for all vertices i adjacent to v such that i is
       not visited {
         dfs(i);
      }
     postorder-process(v);
}
```

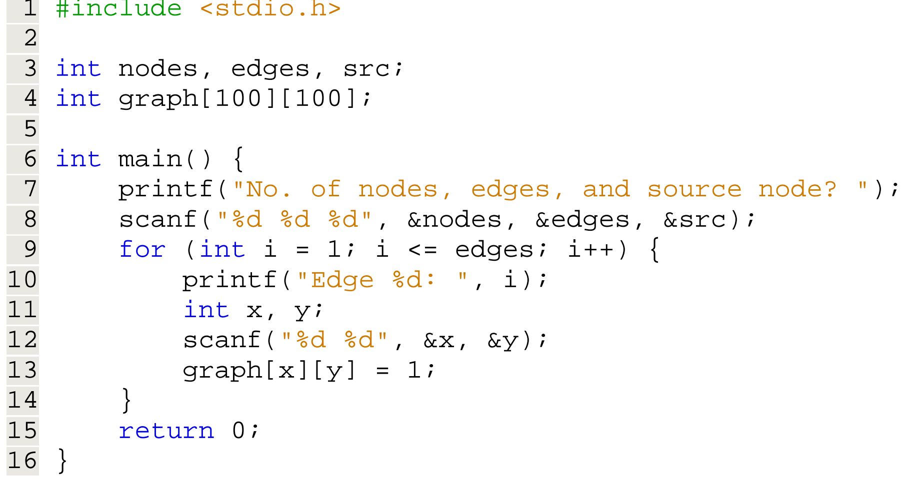
### Applications

Here are some algorithms where DFS is used:

- Finding connected and strongly connected components
- Detecting biconnectivity (articulation points / cut-vertex) and bridges
- Topological sorting
- Edge detection
- Finding all-pair paths between source and destination nodes
- Solving puzzles with only one solution, such as mazes

### Graph traversing using DFS (in C++)

Suppose, we have to traverse the directed graph of *figure 2*. We'll start from the node 'src'. Let's assume we have completed the preliminary task of taking input of the graph[4]:



```
 1 #include <stdio.h>
 2
 3 int nodes, edges, src;
 4 int graph[100][100];
 5
 6 int main() {
 7     printf("No. of nodes, edges, and source node? ");
 8     scanf("%d %d %d", &nodes, &edges, &src);
 9     for (int i = 1; i <= edges; i++) {
10         printf("Edge %d: ", i);
11         int x, y;
12         scanf("%d %d", &x, &y);
13         graph[x][y] = 1;
14     }
15     return 0;
16 }
```

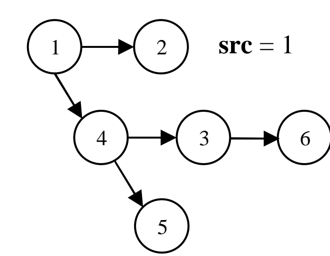



**Figure 2:** Traversing a graph using DFS

[3] An *uninformed search algorithm* is one that does not take into account the specific nature of the problem. As such, they can be implemented in general, and then the same implementation can be used in a wide range of problems. The drawback is that most search spaces (i.e., the set of all possible solutions) are extremely large, and an uninformed search (especially of a tree) will take a reasonable amount of time only for small examples.

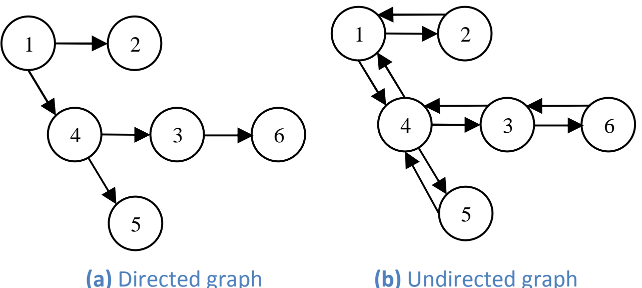
[4] Notice that we're taking the number of nodes as input. Why? Initially, we're taking a large matrix for *100* nodes. But we need the actual number of nodes in the graph when we try to traverse it. We'll be using a loop, and it should stop running when we've completed examining the last node.

Now, let’s try to implement the DFS traverse algorithm:

```
18 void dfs(int node) {
19     color[node] = GREY;      //mark node as visited
20     printf("%d ", node);     //preorder-process(node)
21     for (int i = 1; i <= nodes; i++) {
22         if ((graph[node][i] == 1) && (color[i] == WHITE))  {
23             dfs(i);
24         }
25     }
26                              //postorder-process(node) [none in this case]
27 }
```

### Checking whether a graph is connected

An *undirected* graph can be easily checked for connectivity. A *directed* graph, however, poses a problem. So, if we convert a directed graph into an undirected graph, then we can easily find whether the graph is connected. To convert a directed graph into an undirected graph, just add reversed edges to all the edges (*figure 3(b)*). Now, run DFS (or BFS) from any node. After the traversing finishes, check whether there is any node marked as WHITE. If no such node can be found, then the graph is connected.




**(a)** Directed graph **(b)** Undirected graph

**Figure 3:** Converting a directed graph into an undirected graph.

### Edge detection

The most natural result of a depth first search of a graph (if it is considered as a function rather than a procedure) is a *spanning tree* of the vertices reached during the search.[5] Based on this spanning tree, the edges of the original graph can be divided into four classes:

1. **Tree edge**, edges which belong to the spanning tree itself.
2. **Back edge**, which point from a node to one of its ancestors.
3. **Forward edge**, which point from a node of the tree to one of its descendants (except the tree edge).
4. **Cross edge**, which point from a node to another node which is *not* its descendant.

**Figure 4:** The four types of edges defined by a spanning tree.

It can be shown that if the graph is *undirected*, then all of its edges are either tree edges or back edges.

### 1. Tree edge detection

Just run the DFS. Whenever we traverse an adjacent white node (and in turn, its adjacent node) from a source node, that’s a tree edge.

[5] **Spanning tree:** a spanning tree of a *connected*, *undirected* graph G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are formed.

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

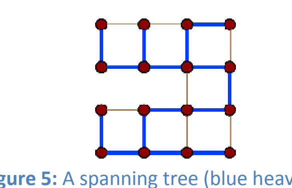


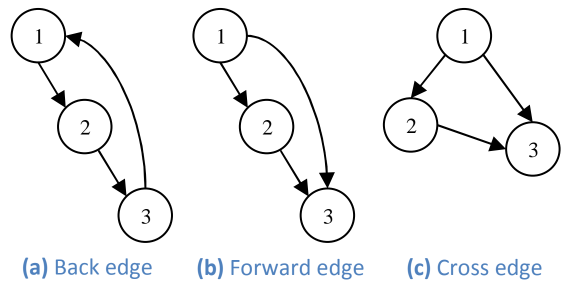
**Figure 5:** A spanning tree (blue heavy edges) of a grid graph.

### 2. Back edge detection

Consider the graph in *figure 6(a)*. If we try to traverse from 3 to 1, we find that 1 is already GREY. So, if we can find a GREY node while traversing, then there's a back edge.

### 3. Forward and cross edge detection

Consider the graph in *figure 6(b)*. After getting back to 1, we find that another path exists from 1 to 3. Here, 3 is already GREY. According to our just-taken decision, this edge should be a back edge. But is it in fact?




**(a)** Back edge **(b)** Forward edge **(c)** Cross edge

**Figure 6:** Back, forward and cross edge detection.

So, we might introduce another color – BLACK. Whenever we finish searching all the adjacent nodes of a particular node, we mark it as BLACK. Thus, in *figure 6(b)*, when we try to traverse from 1 to 3, we find it as BLACK. So, if we can find a BLACK node while traversing, then there's a forward edge.

However, there lies a problem with this decision. Let's change the *appearance* of the graph in *figure 6(b)* so that it looks like the graph in *figure 6(c)*. Now, while in *figure 6(b)* 2 was 3's ancestor, in *figure 6(c)*, 2 is not 3's ancestor. So, in that case, the edge (2, 3) is a cross edge rather than a forward edge. But note, however, that the graphs are the same.

Therefore, finding a BLACK means there might be either a forward edge, or a cross edge. Then how would we differentiate between the two?

#### Use of time in DFS

While traversing a graph using DFS, we can set a starting time (or discovery time) and a finishing time for each node. This technique will help us solve the edge detection problems as well as all the other problems following those.

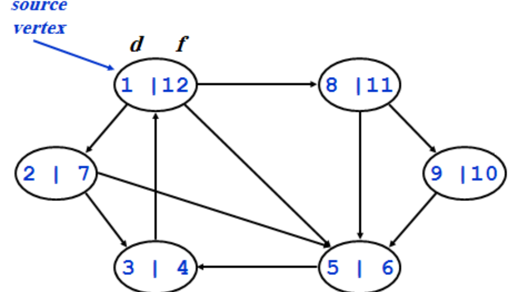
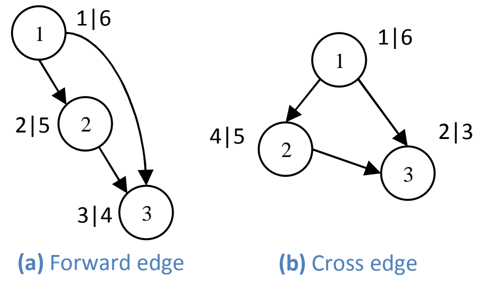



**Figure 7:** Using time in DFS

The technique is fairly simple – while running DFS, just set the starting time of a node whenever we discover it, and set the finishing time when we are sure that all its adjacent nodes have been discovered. An example is given in *figure 7*.




**(a)** Forward edge **(b)** Cross edge

**Figure 8:** Forward and cross edge detection.

So, how would you program it? Simple – take a global variable – say, `time` – and initialize it with `1`. Take two global arrays – e.g. `start_time` and `finish_time` – where the starting and finishing time of all the nodes will be stored. Whenever a node is discovered, we set the current time as its start time and increment the value of time. After the `for` loop in the code for DFS, i.e., when working with that node is done, we set the current time as its finishing time and again increment the value of time.

Now, let's try to solve the problem of detecting forward and cross edges. Consider *figure 8 (a)* and *(b)*. The graphs are the same as those in *figure 6 (b)* and *(c)*. In case of *figure 8(a)*, we find that the discovery time of 1 is *less* than the discovery time of 3. So, 3 is 1's descendant. Hence, the edge (1, 3) is a forward

edge. On the other hand, in case of *figure 8(b)*, we find that the discovery time of 2 is *greater* than the discovery time of 3. So, 2 is not an ancestor of 3, and hence the edge (2, 3) is a cross edge.[6]

Summing up the edge detection techniques, we can conclude the following:

Let, ($u$, $v$) is an edge.

- If (`color[v] == WHITE`), then ($u$, $v$) is a *tree* edge.
- If (`color[v] == GREY`), then ($u$, $v$) is a *back* edge.
- If (`color[v] == BLACK`), then ($u$, $v$) is a *forward* or *cross* edge.
    - If (`start_time[u] < start_time[v]`), then ($u$, $v$) is a *forward* edge.
    - If (`start_time[u] > start_time[v]`), then ($u$, $v$) is a *cross* edge.

### Detecting cycles in a directed graph

If we can detect a back edge while running DFS, then we can say that the graph has a cycle. If a graph does not contain any cycle, then it is called an *acyclic* graph.

### Theorem: An undirected graph is *acyclic iff*[7] a DFS yields no back edges

After running DFS, if no back edges can be found, then the graph has only tree edges. (As the graph is *undirected*, therefore, there will be no forward/cross edges but only tree and back edges.) Only tree edges imply we have a tree or a forest, which, by definition, is acyclic.

### Topological Sort

*Topological sort* of a directed acyclic graph (DAG) G = (V, E): a linear order of vertices such that if there exists an edge ($u$, $v$), then $u$ appears before $v$ in the ordering.

The main application of topological sort is in scheduling a sequence of jobs. The jobs are represented by vertices and there is an edge from $x$ to $y$ if job $x$ must be completed before job $y$ can be done. (For example, washing machine must finish before we put the clothes to dry.) Then, a topological sort gives an order in which to perform the jobs.

Another application of topological sort is in open credit system, where courses are to be taken (in order) such that, pre-requisites of courses will not create any problem.

The algorithm for applying topological sort is quite simple – just sort the nodes in descending (or non-increasing) order according to their finishing time. Why? Because, in DFS, the deepest node is finished processing first. Then its parent is finished processing. Thus, we can say that the deepest node must come *after* its parent.

So, how do we write the code for sorting the nodes? Easy! Just create an array – e.g. `top_sort` – and whenever you blacken out a node (or, in other words, assign its finishing time), insert the node into it. Now, after the DFS finishes, you get an array with nodes inserted according to the increasing order of their finishing time. How? The first time you assign a node its finishing time, the finishing time is of the lowest value. Eventually, you end up assigning the highest finishing time to the last blacked out node. Thus, you get an array of nodes inserted according to the increasing order of their finishing time. Now what? Just print the array in reverse. That's the topologically sorted list.

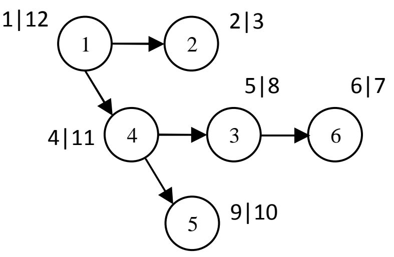


**Figure 9:** Graph for topological sort.

[6] Note that the result of DFS might vary according to traversal algorithm. If the graph in *figure 8(a)* is traversed as 1-2-3, then (1, 3) would be its forward edge. On the contrary, if the graph is traversed as 1-3-2, then (2, 3) would be its cross edge. For this reason, in most applications of DFS, we don't need to distinguish between a forward edge and a cross edge.

[7] i.e., *if and only if*.

### Unique Topological Sort

The above trick for applying topological sort is fine. But how can we find whether the topological sort of a particular graph is unique or not? For example, topological sort of the graph in *Figure 9* is not unique. The sort might be any of the followings:

a. 1 – 4 – 5 – 3 – 6 – 2
b. 1 – 2 – 4 – 3 – 6 – 5
c. 1 – 2 – 4 – 5 – 3 – 6

One thing we may do is to run the DFS algorithm multiple times while choosing different paths each time. If we get multiple topological sorts, then we can conclude that the sort for the graph is not unique. However, coding this is complex and inefficient. There is another algorithm for finding topological sort using *in-degree of edges* which is efficient to find out whether the sort is unique.

In this algorithm, we first determine the in-degree of all the nodes. Now, the node with the lowest in-degree must come *before* a node with a greater in-degree than that. We put that node as the first node in the topological sort order. Then, we have to find its adjacent nodes. At least one of them must come *immediately after* the first node. To find the next node in the sort order, we simply decrement the in-degree of the first node's adjacent nodes by 1 and then repeat the previous steps again. We repeat these steps for *n* times where *n* is the number of nodes in the graph.

A formal approach of describing the algorithm is as follows:

1. Take an array – named, for example, `indegree` – and put the in-degree of all the nodes in it.
2. Take another array – named, for example, `sorted` – where the nodes will be kept in topological sort order. The values in this array after the algorithm finishes are the final result.
3. From the `indegree` array, determine the node with the *lowest* in-degree. Put the node in the `sorted` array.
4. Set the in-degree of this node to ∞ in the `indegree` array.
5. Decrement the in-degree of the adjacent nodes of this node by 1 in the `indegree` array.
6. Repeat steps 3 to 5 for *n* times, where *n* is the number of nodes in the graph.

Now let's return to our concerned problem – how we can determine whether a topological sort is unique. Easy! While running the above algorithm, at step 3, if we detect that there are more than one lowest in-degree values, then we can easily conclude that the topological sort is *not* unique.

### Articulation Point / Cut-Vertex and Biconnected Components

The *articulation points* or *cut-vertices* of a graph are those nodes, deleting which causes the graph to become disconnected. If a graph contains no articulation points, then it is *biconnected*. If a graph does contain articulation points, then it is useful to split the graph into the pieces where each piece is a maximal biconnected subgraph called a *biconnected component*.

So, how do we find the articulation points? One way is to delete a node and then run BFS/DFS to see whether the graph is connected. If the graph is not connected, then that deleted node is an articulation point. But this brute-force method is highly inefficient.

Here is an efficient algorithm. We'll observe how far above the parent a child can go. If we find that no child of a particular parent can go higher than the parent node, then we can conclude that the parent node is an articulation point.

So far, so good. But how would we code this? From the algorithm, we find that we need to find out whether any child of a parent can go above it. Therefore, we need to keep track of how far above a node can travel. Let's take an array of nodes where the value of a particular node will represent the maximum ancestor node that can be reached by it. Let's call this array `low`. Why? We'll see later.

**Figure 10:** Articulation Point.

Now, while running DFS, we'll first set the *discover time* of a node as its *low* value, because the node can go from itself to itself. Now, while backtracking, when we've visited all the adjacents of a parent, we have to update the low value of it. How? We'll take the low values of all its children. We'll also take the discovery time of all the nodes it has a back edge with. And we already have the discovery time of that node. Then, the updated low value of the parent would be the *lowest* low value among all these low values. In other words, if the parent node is *v* and its child or back-edged node is *w*, then:

$$low[v] = min. \left\{ \begin{array}{c} d[v] \\ lowest\ d[w] among\ all\ back\ edges\ (v, w) \\ lowest\ low[w]\ among\ all\ tree\ edges\ (v, w) \end{array} \right\}$$

Now, if the low value of *any* of the parent's children is greater than or equal to the discovery time of the parent, then that parent is an articulation point. In other words, a parent node, *v* will be an articulation point if, for any node *w* connected with *v* as a *back edge* or *tree edge*,

$$low(w) >= d[v]$$

Note that to update the low value, we're taking the *lowest* of all the other values. That's why we're calling the array `low`.

Note further that, when doing the actual coding, we need to perform the checking *before* updating the low value of the parent.

Now, there lies a problem. When we compare the low value of *B* with the discovery time of *A* (*figure 10*), we will get that *A* is an articulation point. But in fact, it isn't. So, as *A* is the root of this tree, it needs to be handled separately. A root might or might not be an articulation point. For example, the root in *figure 10* is not an articulation point, whereas the root in *figure 9* is. Actually, a root is an articulation point *iff* it has two or more children. To determine whether a root is an articulation point, we should run a DFS from the root, and after traversing the first path from the root, if there is any WHITE node left, we can conclude that the root is an articulation point.

**Bridge**

A bridge is an edge deleting which causes the graph to become disconnected.

Any edge in a graph that does not lie on a cycle is a bridge. Bridges either end in articulation points or in dead-ends. As an example, in the graph of *figure 11*, (*C*, *D*) and (*D*, *E*) are bridges, *C* and *D* are articulation points, while *E* is a dead-end. Note that (*A*, *B*), (*A*, *C*) and (*B*, *C*) are not bridges as they lie on the cycle *ABC*.

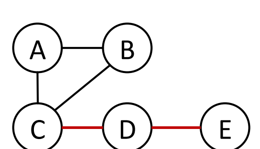



**Figure 11:** Graph with bridges marked.

**SCC (Strongly Connected Components)**

A directed graph is called strongly connected if there is a path from *each* vertex in the graph to *every* other vertex.

The strongly connected components (SCC) of a directed graph are its maximal strongly connected subgraphs.

পাশের গ্রাফে a, b আর e নোডগুলা নিয়ে একটা SCC; কারণ a থেকে কোন না কোনভাবে b, e দুটোতেই যাওয়া যায়, b থেকে e, a দুটোতেই যাওয়া যায়, আর e থেকে a, b দুটোতেই যাওয়া যায়। একইভাবে, (f, g) আর (c, d, h)–ও দুটো SCC।

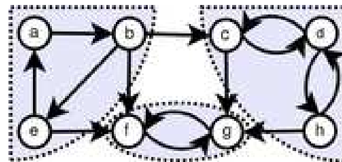



**Figure 22:** Graph with strongly connected components marked.

So, how can we find the SCCs? Follow the steps below:

1. Run DFS from any node (for example ***a***) to compute the finishing time of all the nodes.
2. Reverse the direction of the edges.
3. Now run DFS again from the node **whose finishing time is the *lengthiest***. (You can find it from the topologically sorted array populated in step 2).
4. While running the second DFS, output the nodes in the DFS tree. These nodes comprise an SCC.
5. If there are still unvisited (i.e., white colored) nodes, then repeat steps 3 and 4 from the WHITE node whose finishing time is the lengthiest; until there are no more WHITE nodes.

Note that a node itself is a strongly connected component if nowhere cannot be gone from that node. For example, in *Figure 12*, nodes 1, 2 and 4 comprise an SCC. Again, only node 3 is also an SCC.

**Figure 12**

# DIJKSTRA'S ALGORITHM

### *Finds single-source shortest path in weighted graph*

The problem with BFS is that it won't produce correct result when applied to a *weighted* graph. Consider the graph in *figure 1*. BFS would say that (1, 3) is the shortest path. But considering weight, we find that (1, 2, 3) is the shortest path. Therefore, how would we calculate shortest path from a weighted graph?

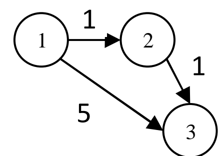



**Figure 1:** Limitation of BFS.

Dijkstra comes to the rescue! Dijkstra's algorithm is a greedy algorithm where each time we try to take the route with the lowest cost (or weight). So, whenever we find more than one adjacents of a node, we'll take the adjacent node whose cost is the lowest and proceed.

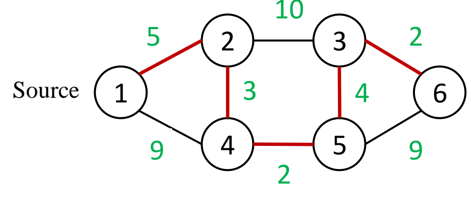
First, let's take an array `cost` where the cost of the nodes *from the source node* would be placed. Now, let's set the cost of the source node as 0, because the cost to visit the source node from itself is 0. Well, let's set the cost of all the other nodes as ∞. Why? Because, initially, we're considering that we can go *nowhere* from the source node. When we'll discover that we're able to go somewhere, then we'll update the cost with some other values.

Now, follow the steps below:

1. Put all the nodes into a priority queue and sort them according to the *non-decreasing* order of their cost.
2. Pop the front node $v$ from queue.
3. For all adjacent nodes $i$ of $v$, update the cost of $i$ if $cost[v] + graph[v][i] < cost[i]$; i.e., the cost from the source to node $i$ (via node $v$) is *less* than the cost of node $i$ (via some other route).
4. Repeat steps 2 and 3 until the queue is empty.

If we save the previous node while updating into another array, then we'll be able to find the shortest path of all the nodes from the source node using that array.

**Note:** Dijkstra's algorithm *might* fail if negative cost is used.




| | | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Initial** | cost | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| **Step 1** | cost | 0 | 0 + 5 = 5 | ∞ | 0 + 9 = 9 | ∞ | ∞ |
| **Step 2** | cost | 0 | 5 | 5 + 10 = 15 | 5 + 3 = 8 | ∞ | ∞ |
| **Step 3** | cost | 0 | 5 | 15 | 8 | 8 + 2 = 10 | ∞ |
| **Step 4** | cost | 0 | 5 | 10 + 4 = 14 | 8 | 10 | 10 + 9 = 19 |
| **Step 5** | cost | 0 | 5 | 14 | 8 | 10 | 14 + 2 = 16 |
| **Step 6** | cost | 0 | 5 | 14 | 8 | 10 | 16 |

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| queue | 1 | 2 | 3 | 4 | 5 | 6 |
| queue | 2 | 4 | 3 | 5 | 6 | |
| queue | 4 | 3 | 5 | 6 | | |
| queue | 5 | 3 | 6 | | | |
| queue | 3 | 6 | | | | |
| queue | 6 | | | | | |
| queue | | | | | | |

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| prev | 0 | | | | | |
| prev | 0 | 1 | | 1 | | |
| prev | 0 | 1 | 2 | 2 | | |
| prev | 0 | 1 | 2 | 2 | 4 | |
| prev | 0 | 1 | 5 | 2 | 4 | 5 |
| prev | 0 | 1 | 5 | 2 | 4 | 3 |
| prev | 0 | 1 | 5 | 2 | 4 | 3 |

**Figure 2:** Dijkstra's Algorithm.

The run-time complexity of Dijkstra's algorithm is:

- $O(n^2)$ – when we use *array* instead of heap.
- $O(E \log V)$ – when *min-heap* is used to find the minimum element from *KEY*. (KEY[$v$] is the minimum cost of any edge connecting a vertex $v$ to a vertex in the tree being generated from the source node.)
- $O(E + V \log V)$ – when *fibonacci-heap* is used to find the minimum element from *KEY*.

# BELLMAN-FORD ALGORITHM

### *Finds single-source shortest path in weighted graph and detects negative cycles*

If negative cost is used, Dijkstra's algorithm might fail. As an example, for the graph of *figure 1(a)*, Dijkstra would produce correct result, whereas for the graph of *figure 1(b)*, if would get into an infinite loop when printing shortest path.

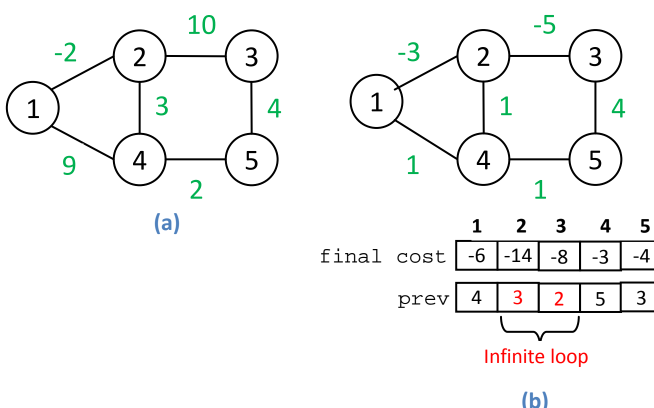



**Figure 1:** Dijkstra's Algorithm with negative cost.

If the graph does contain a cycle of negative weights, Bellman-Ford, however, can only detect this; Bellman-Ford cannot find the shortest path that does not repeat any vertex in such a graph.

Bellman-Ford is in its basic structure very similar to Dijkstra's algorithm, but instead of greedily selecting the minimum-weight node not yet processed to relax[8], it simply relaxes all the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each node at most once.

The algorithm is as follows:

```
// Step 1: Initialize graph
for each vertex v in vertices:
    if v is source then v.distance := 0
    else v.distance := infinity
    v.predecessor := null

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge uv in edges: // uv is the edge from u to v
        u := uv.source
        v := uv.destination
        if v.distance > u.distance + uv.weight:
            v.distance := u.distance + uv.weight
            v.predecessor := u

// Step 3: check for negative-weight cycles
for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if v.distance > u.distance + uv.weight:
        error "Graph contains a negative-weight cycle"
```

Initial

Pass 1

Pass 2

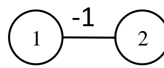
**d[v] > d[u] + w(u, v)**

**Fig. 2:** Negative Cycle

In words, we have to update all the nodes' cost for $(|V| - 1)$ times. Then, we have to perform the checking for negative cycles for each edge.

Bellman–Ford runs in $O(|V| \cdot |E|)$ time.

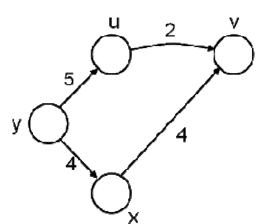
**Note:** Negative cycle is guaranteed if *negative weight undirected* edge can be found. For example:



[8] *Relaxing* an edge $(u, v)$ means testing whether we can improve the shortest path to $v$ found so far by going through $u$.

# SINGLE-SOURCE SHORTEST PATHS IN DAGS

So far we've seen algorithms to solve single-source shortest path problem for *undirected* graphs. What about *directed* graphs, or, more precisely, directed *acyclic* graphs (as shortest path cannot be calculated for a directed graph containing cycles)?



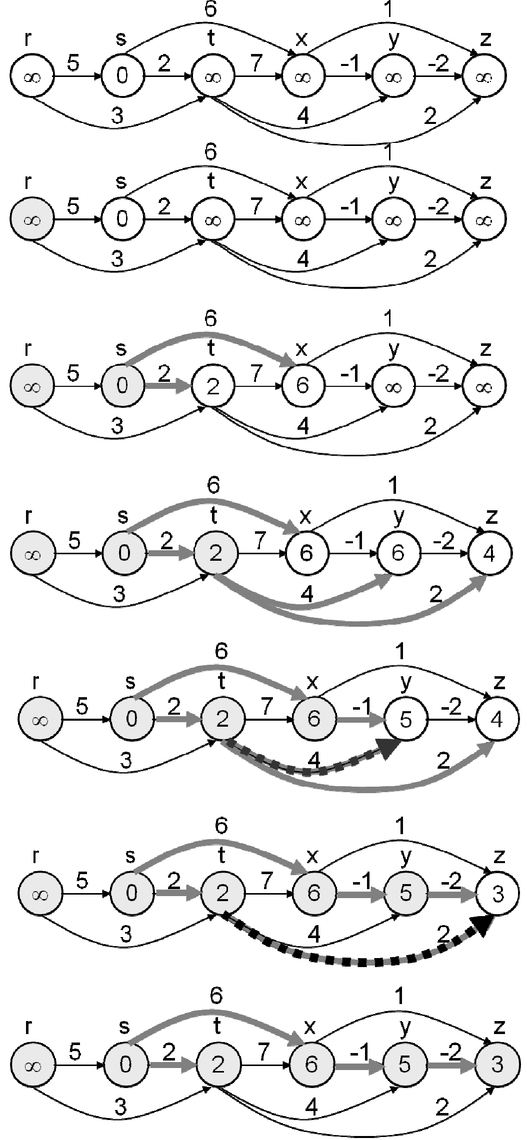

The idea is to topologically sort the vertices of the graph and relax the edges according to the order given by the topological sort. For each vertex, we relax each edge that starts from that vertex.

The running time of this algorithm is O($V + E$).

Note that shortest paths are well defined in a DAG as (negative weight) cycles cannot exist.

**Example**

In the following graph, we have to find the shortest paths from node *s*. Let, the topologically sorted order of the nodes is r – s – t – x – y – z. We initialize the costs of all nodes as ∞ – except the source node, *s*, which we initialize to 0.



# Floyd-Warshall Algorithm

***Finds all-pair shortest paths in weighted graph***

The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. Here, all we do is to find out the shortest path between two nodes *i* and *j* via all the nodes (1, 2, …, *n*).

The Floyd-Warshall algorithm is an example of dynamic programming.

### Algorithm

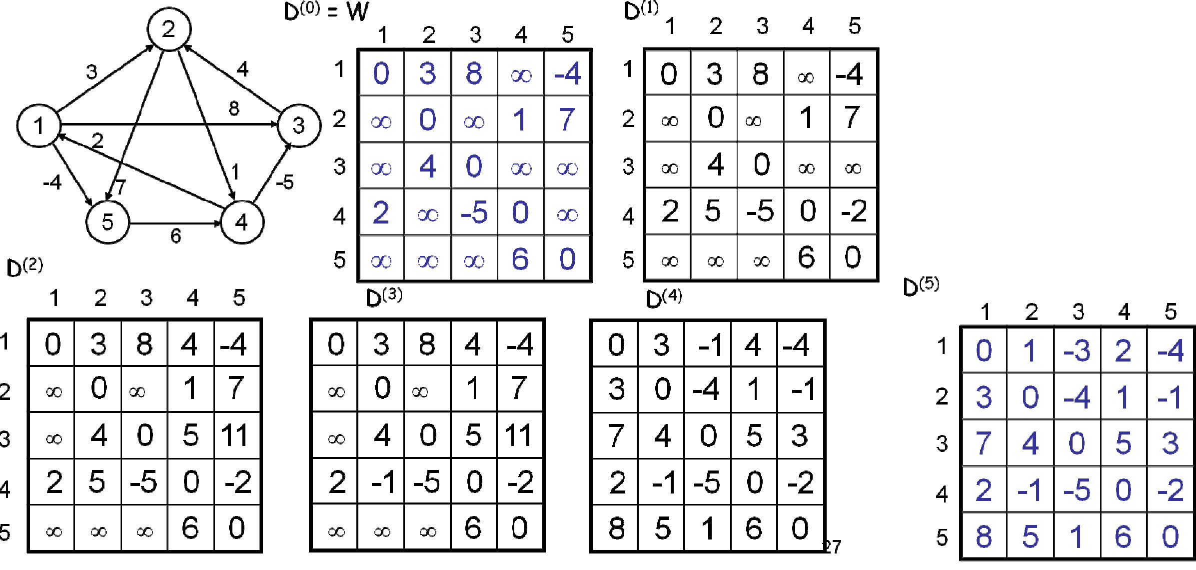
```
/* Assume a function edgeCost(i,j) which returns the cost of the edge from i to j
   (infinity if there is none).
   Also assume that n is the number of vertices and edgeCost(i,i)=0
*/

int path[][];
/* A 2-dimensional matrix. At each step in the algorithm, path[i][j] is the shortest
   path from i to j using intermediate values in (1...via). Each path[i][j] is
   initialized to edgeCost(i,j).
*/

procedure FloydWarshall()
   for via = 1 to n
      for each (i, j) in (1...n)
         path[i][j] = min ( path[i][j], path[i][via] + path[via][j] );
```

The time complexity of this algorithm is $O(|V|^3)$.

### Behavior with negative cycles

For numerically meaningful output, Floyd-Warshall assumes that there are no negative cycles (in fact, between any pair of vertices which form part of a negative cycle, the shortest path is not well-defined, because the path can be arbitrarily negative). Nevertheless, if there are negative cycles, Floyd–Warshall can be used to detect them. A negative cycle can be detected if the path matrix contains a negative number along the diagonal. If `path[i][i]` is negative for some vertex `i`, then this vertex belongs to *at least* one negative cycle.

### Example




$D^{(0)} = W$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | ∞ | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | ∞ | ∞ |
| 4 | 2 | ∞ | -5 | 0 | ∞ |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(1)}$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | ∞ | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | ∞ | ∞ |
| 4 | 2 | 5 | -5 | 0 | -2 |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(2)}$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 8 | 4 | -4 |
| 2 | ∞ | 0 | ∞ | 1 | 7 |
| 3 | ∞ | 4 | 0 | 5 | 11 |
| 4 | 2 | 5 | -5 | 0 | -2 |
| 5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(3)}$

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 8 | 4 | -4 |
| ∞ | 0 | ∞ | 1 | 7 |
| ∞ | 4 | 0 | 5 | 11 |
| 2 | -1 | -5 | 0 | -2 |
| ∞ | ∞ | ∞ | 6 | 0 |

$D^{(4)}$

| | | | | |
|---|---|---|---|---|
| 0 | 3 | -1 | 4 | -4 |
| 3 | 0 | -4 | 1 | -1 |
| 7 | 4 | 0 | 5 | 3 |
| 2 | -1 | -5 | 0 | -2 |
| 8 | 5 | 1 | 6 | 0 |

$D^{(5)}$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | -3 | 2 | -4 |
| 2 | 3 | 0 | -4 | 1 | -1 |
| 3 | 7 | 4 | 0 | 5 | 3 |
| 4 | 2 | -1 | -5 | 0 | -2 |
| 5 | 8 | 5 | 1 | 6 | 0 |

# PRIM'S ALGORITHM

### *Generates minimum spanning tree (MST) (using node-based approach)*

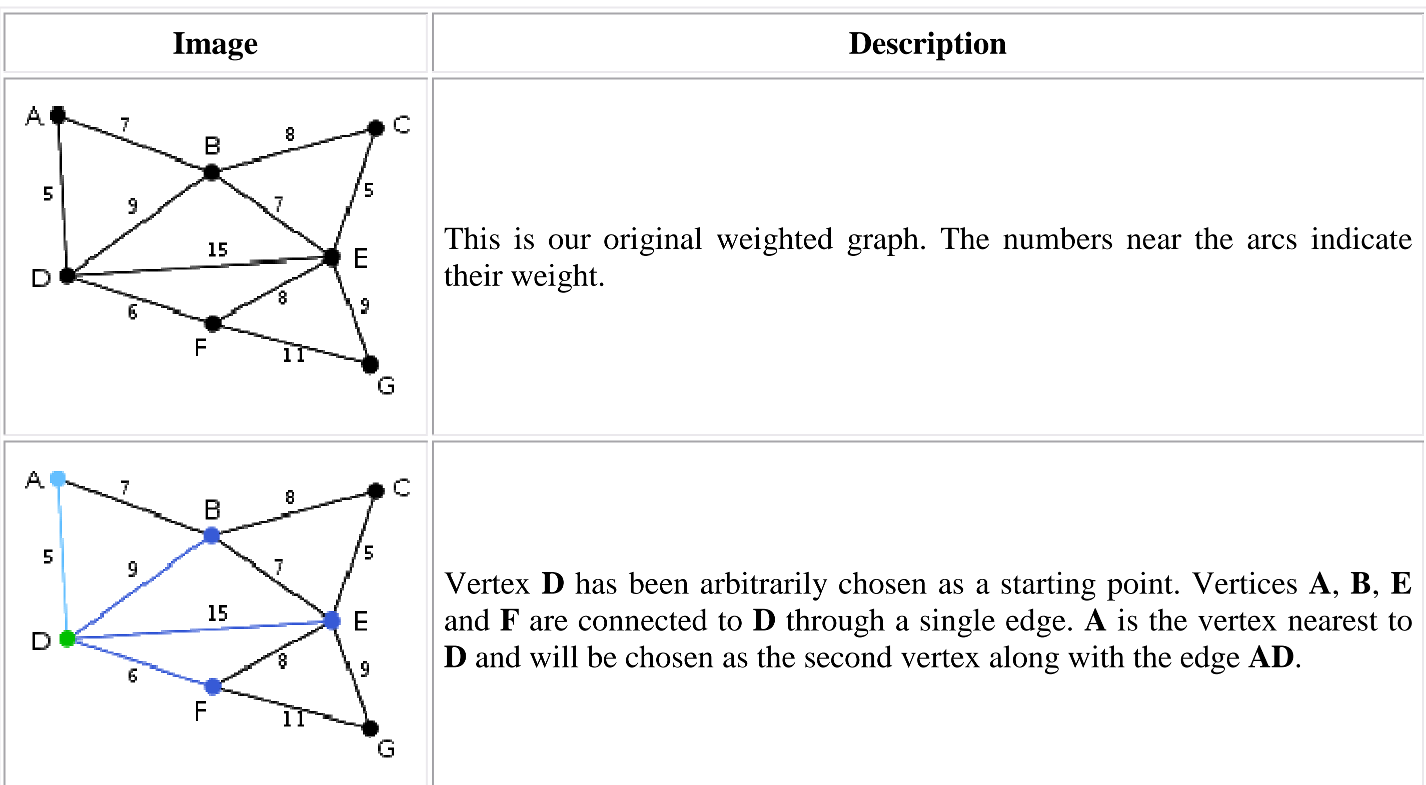
In case of a weighted graph, there might be several spanning trees. But if we need to find the minimum spanning tree, i.e., the spanning tree which has the minimum total cost, we have to use Prim's or Kruskal's algorithm.

The main idea of Prim's algorithm is to grow an MST from the current spanning tree by adding the nearest (lowest weight) vertex and the edge connecting the nearest (lowest weight) vertex to the MST. The algorithm may be presented informally as follows:

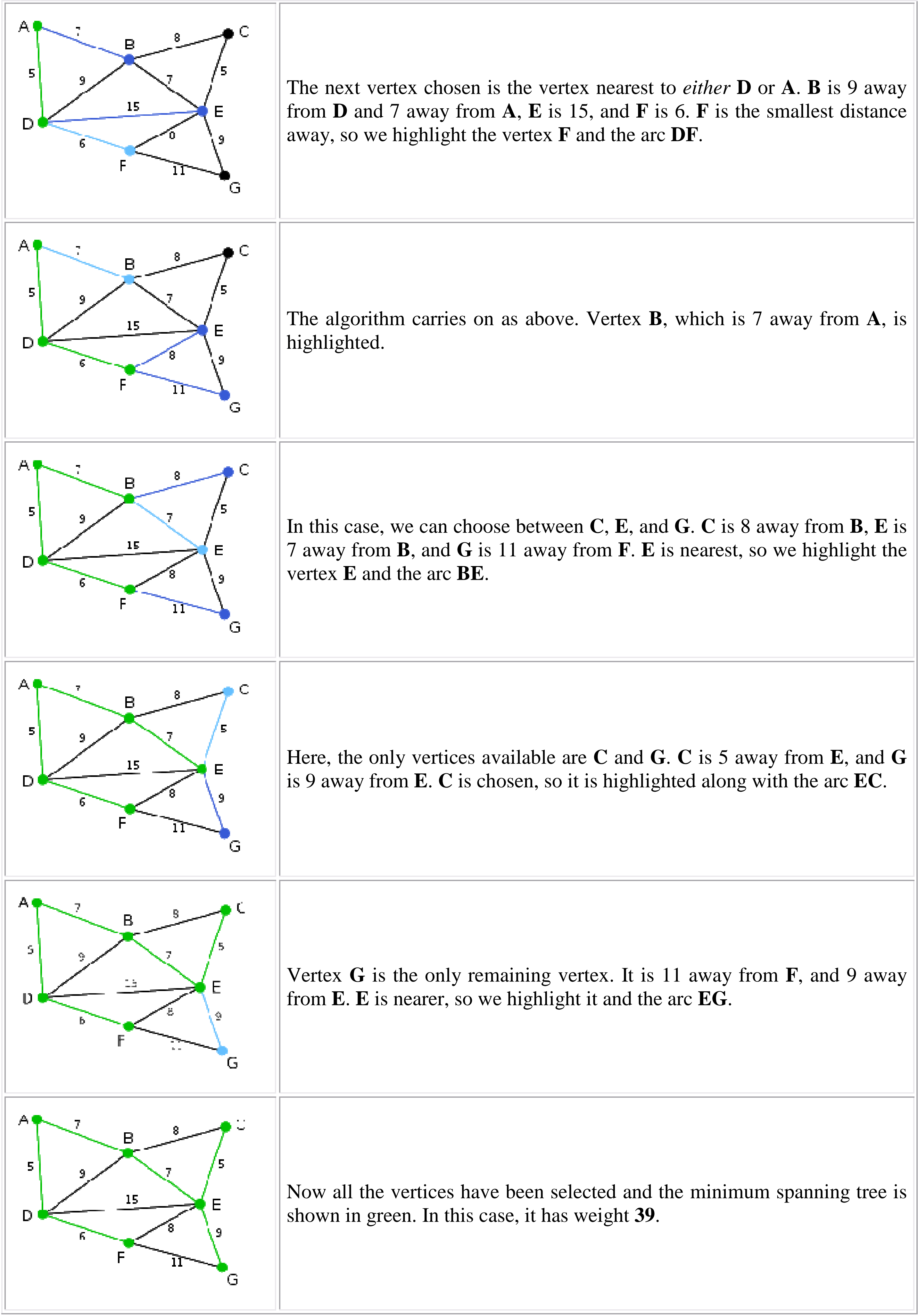
```
Select a vertex to be a tree-node.
while (there are non-tree vertices) {
    if there is no edge connecting a tree node with a non-tree node
        return "no spanning tree".
    Select an edge of minimum weight between a tree node and a non-tree node.
    Add the selected edge and its new vertex to the tree.
}
return tree.
```

The run-time complexity of Prim's algorithm is the same as that of Dijkstra's algorithm. In fact, Prim's algorithm is similar to Dijkstra's algorithm – the only difference is that Dijkstra's algorithm may or may not produce a spanning tree, whereas Prim's algorithm always produces a spanning tree.

**Example**



| Image | Description |
|---|---|
| (graph: A, B, C, D, E, F, G) | This is our original weighted graph. The numbers near the arcs indicate their weight. |
| (graph: A, B, C, D, E, F, G) | Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**. |



The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.

The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.

In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **BE**.

Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.

Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.

Now all the vertices have been selected and the minimum spanning tree is shown in green. In this case, it has weight **39**.

# KRUSKAL'S ALGORITHM

### *Generates minimum spanning tree (MST) (using edge-based approach)*

The main idea of Kruskal's algorithm is to grow an MST from a forest of spanning trees by adding the smallest edge connecting two spanning trees. A formal algorithm along with complexity analysis is as follows:

```
                MST-Kruskal(G,w)
             01 A ← ∅
    O(V) {   02 for each vertex v ∈ V[G] do
             03    Make-Set(v)
 O(ElogE)    04 sort the edges of E by non-decreasing weight w
    O(E)     05 for each edge (u,v) ∈ E, in order by non-decreasing weight do
         (   06   if Find-Set(u) ≠ Find-Set(v) then
    O(V) {   07      A ← A ∪ {(u,v)}
         (   08      Union(u,v)
             09 return A
```

The overall complexity of the algorithm is: O($VE$).

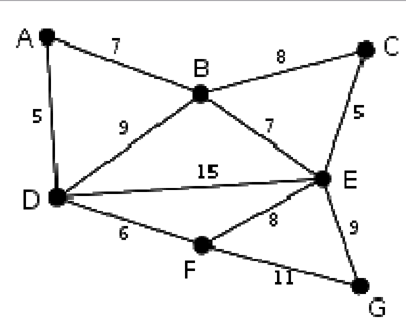
The operations used in the above algorithm are described as follows:

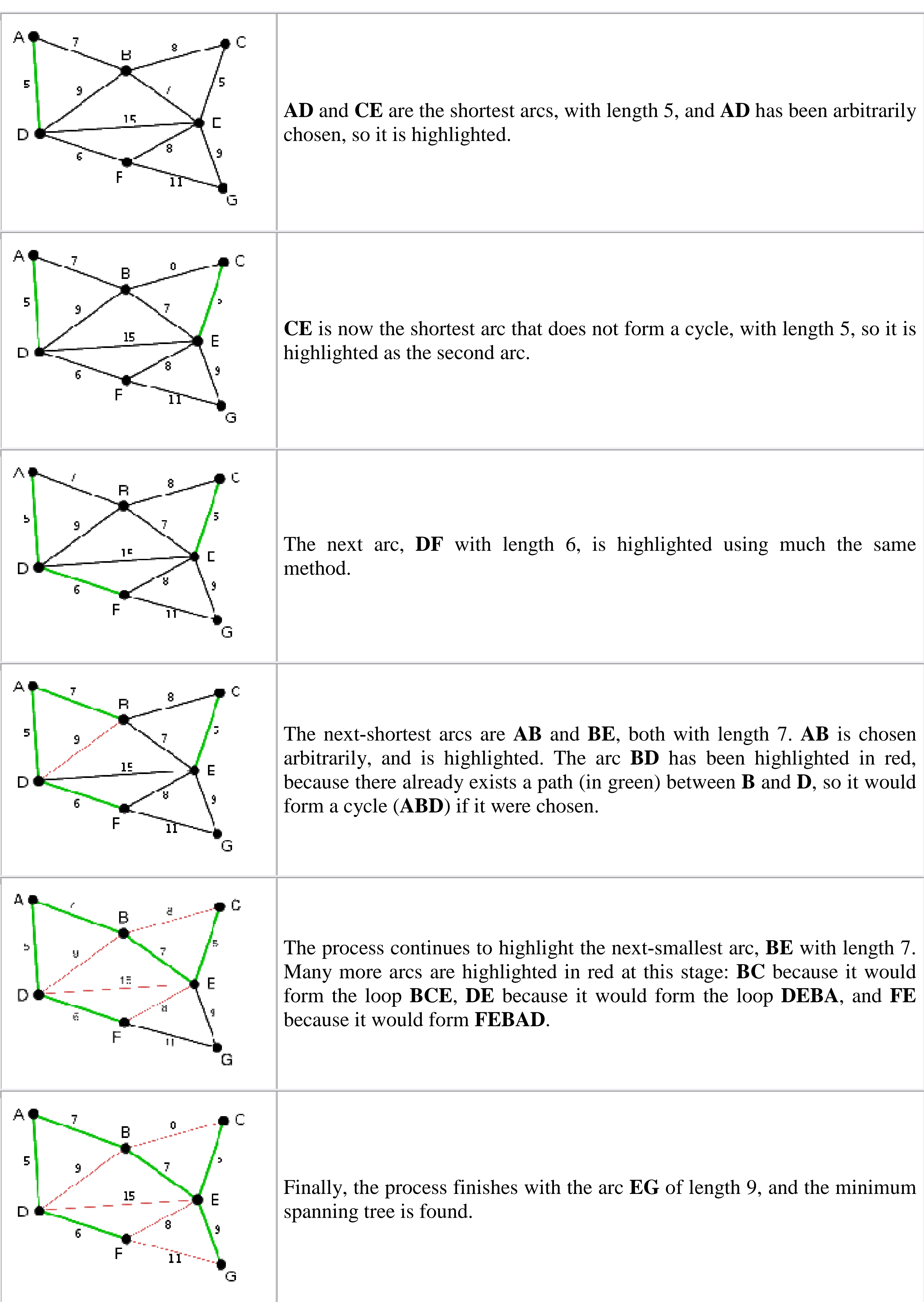
- **Make-Set($x$)** – creates a new set whose only member is $x$.
- **Union($x$, $y$)** – unites the sets that contain $x$ and $y$, say, $S_x$ and $S_y$, into a new set that is the union of the two sets.
- **Find-Set($x$)** – returns a pointer to the representative of the set containing $x$.

The algorithm (from coding point of view) is as follows:

1. Sort the edges in ascending order.
2. Now begin choosing the lowest cost edges.
3. However, while choosing edges, we should pay attention so that a cycle is not formed. To prevent forming cycles, we need to keep track of whose parent is who. Initially, all the nodes' parents are their values, themselves. Now, while choosing edges, we'll check whether the parents of the two nodes belonging to the edge are the same. If so, then we'll discard that edge. But if they're not the same, then we'll replace the higher parent value with the lower parent value. However, when we update the parent value of a particular node, we need to update its all other adjacent nodes' parent values with its parent value.
4. The process continues until all the edges have been visited.

**Example**

| | |
|---|---|
|   | This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted. |




| | |
|---|---|
| | **AD** and **CE** are the shortest arcs, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted. |
| | **CE** is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc. |
| | The next arc, **DF** with length 6, is highlighted using much the same method. |
| | The next-shortest arcs are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The arc **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen. |
| | The process continues to highlight the next-smallest arc, **BE** with length 7. Many more arcs are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**. |
| | Finally, the process finishes with the arc **EG** of length 9, and the minimum spanning tree is found. |