Discrete Mathematics 10/04/2019

Lecture 20: Graph Algorithms

Instructor: Goutam Paul *Scribe: Anubhab Mondal*

Contents

1 Introduction

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two vertices in the graph.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

In this section, we will look at some of the basic graph algorithms.

2 Representation of Graphs

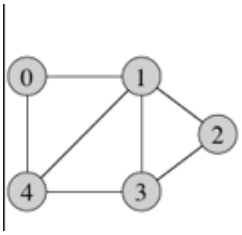
Before going into the main discussion let us make ourselves familiar with some of the basic ideas.

2.1 Adjacency Matrix

For a graph G, with set of vertices V, the adjacency matrix A is a square matrix such that A_{ij} is 1 if there is an edge between vertex i and vertex j, and 0 otherwise.

$$A_{ij} = \begin{cases} 1, & \text{if there is an edge between } v_i \text{ and } v_j \\ 0, & \text{otherwise} \end{cases}$$

The diagonal elements of the matrix are all zero, since edges from a vertex to itself (loops) are not allowed in simple graphs.

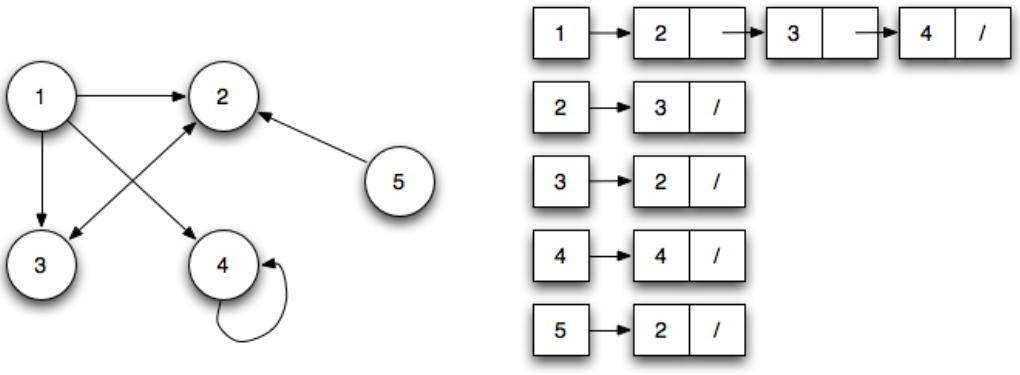


	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Example of adjacency matrix

2.2 Adjacency List

Another way to represent a graph is by adjacency list. An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbours of a vertex in the graph.
In other words, an adjacency list representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges.



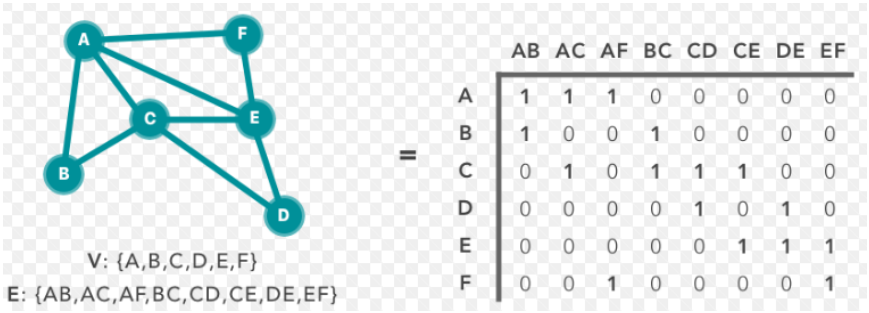
Example of adjacency list

2.3 Incidence Matrix

The incidence matrix of an undirected graph is a $n * m$ matrix B, where n and m are the numbers of vertices and edges respectively, such that

$$B_{ij} = \begin{cases} 1, & \text{if the vertex } v_i \text{ and edge } e_j \text{ are incident} \\ 0, & \text{otherwise} \end{cases}$$

Each column will have exactly two of its entries 1 and the remaining will be 0.



	AB	AC	AF	BC	CD	CE	DE	EF
A	1	1	1	0	0	0	0	0
B	1	0	0	1	0	0	0	0
C	0	1	0	1	1	1	0	0
D	0	0	0	0	1	0	1	0
E	0	0	0	0	0	1	1	1
F	0	0	1	0	0	0	0	1

Note that the matrix can be defined in a different way for directed graphs.

Now we are in a position to discuss the various algorithms.

3 Graph search from a fixed vertex

Definition 3.1 (Graph Traversal)**.** *Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, one must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm .*

Problem: *We have to traverse through a (di)graph to find some kind of vertices or edges.*

We assume that the (di)graph is connected and loopless. For disconnected graphs, we have to go through the components separately. We ignore loops, if the (di)graph has any.

3.1 Depth First Search

We choose a vertex r (we call this vertex "root") to start the search. Then we traverse an edge $e = (r, v)$ to go to the vertex v. At the same time , we direct an edge, e from r to another vertex v. Now we say that the edge is examined and we call it tree edge.The vertex r is called the $father$ of v and we denote it $r = FATHER(v)$.

Depth first search for undirected graph:
We continue the search.At a vertex x, there are two cases:

1. If every edge incident to x has been examined, return to the father of x and continue the process from $FATHER(x)$.Then the vertex x is said to be completely scanned.

2. If there exists some unexamined edges incident to x, then we choose one such edge $e = (x, y)$ and direct it from x to that unexamined vertex y. Now this edge is said to be examined. We have two subcases now:

   (a) If y has not been visited before , then we traverse the edge (x, y), visit y and continue the search from y. In this case e is a $tree-edge$ and $FATHER(y) = x$.

   (b) If y has been visited before , then we select some other unexamined edge incident to x. In this case the edge e is called a $back-edge$.

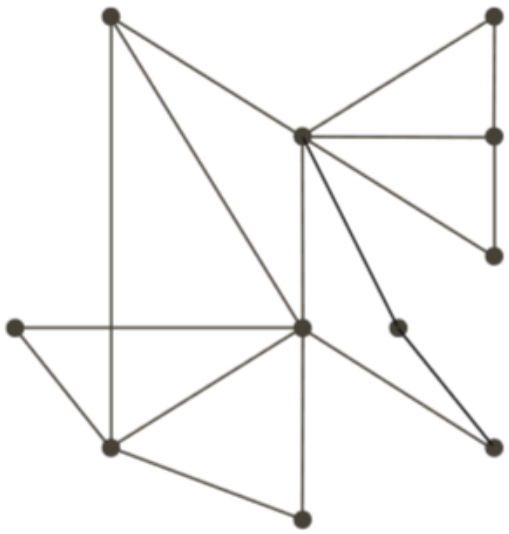
Every time we come to a new vertex which has never been visited before, we give it a distinct number.The number of the root is 1.We write :

$$\text{DFN}(x) = \text{running number of vertex } x$$

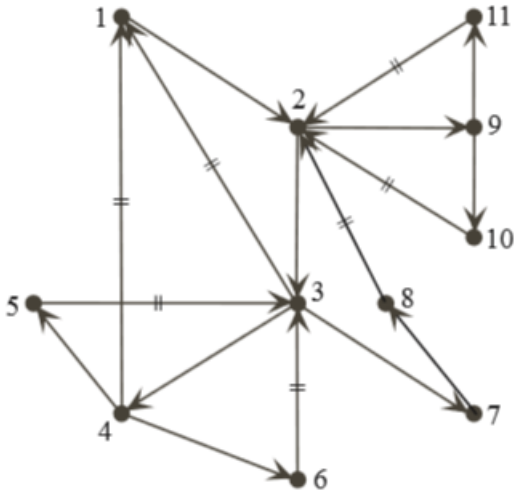
A complete DFS ends when we traverse back to the root and we have visited every vertex or when we have found the desired edge or vertex.

DFS divides the edges of G into tree edges and back edges.Obviously,the tree edges form a spanning tree of G, also known as the DFS tree.

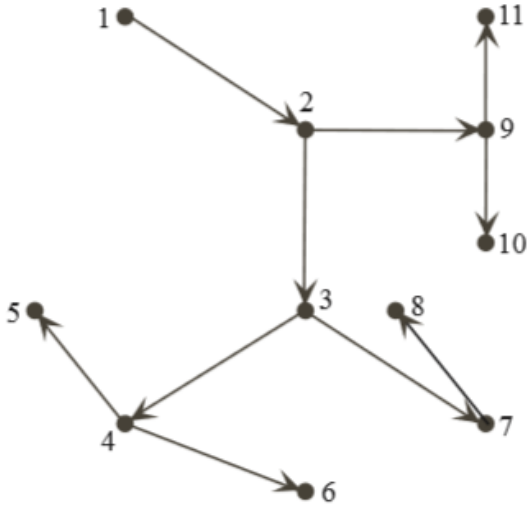
Example *For the graph*



we start the DFS from a root in the upper left corner. The back edges are marked with two lines



The corresponding DFS tree is



In the following we denote,

$$K(x) = \begin{cases} 0, & \text{if vertex x has not been visited} \\ 1, & \text{if vertex x has not been visited} \end{cases}$$

and TREE and BACK are set variables containing the direct tree edges and the back tree edges.

Algorithm:

1. Set TREE $\leftarrow \Phi$, BACK $\leftarrow \Phi$ and $i \leftarrow 1$. For every x of G, set FATHER$(x) \leftarrow 0$ and $K(x) \leftarrow 0$.

2. Choose a vertex r for which $K(r) = 0$ (this condition is needed only for disconnected graphs , see step 6). Set $DFN(r) \leftarrow$ i, $K(r) \leftarrow 1$ and $u \leftarrow r$.

3. If every edge incident to u has been examined , go to step 5. Otherwise, choose an edge $e = (u, v)$ that has not been examined.

4. We direct edge e from u to v and label it examined.

   (a) If $K(v) = 0$, then set $i \leftarrow i + 1$, $DFN(v) \leftarrow i$, $TREE \leftarrow TREE \cup \{e\}$, $FATHER(v) \leftarrow u$ and $u \leftarrow v$. go back to step 3.

   (b) If $K(v) = 1$, then set $BACK \leftarrow BACK \cup \{e\}$ and go to step 3.

5. If FATHER$(u) \neq 0$, then set $u \leftarrow FATHER(u)$ and go back to step 3.

6. (Only for disconnected graphs so that we can jump from one component to another)If there is a vertex r such that $K(r) = 0$, then set $i \leftarrow i + 1$ and go back to step 2.

7. Stop.

Algorithm using stack:
Each of our algorithm uses a stack S of vertices. A vertex can be pushed onto the stack more than once, in fact the same vertex can be in the stack in several places simultaneously. being pushed onto the stack does not necessarily imply visitation; we must include the visitation of a vertex v by explicitly writing $visited[v] := TRUE$ and the assignment may not occur more than once.

Here is a stack algorithm for DFS, which we call DFS-A(G, s). DFS A(G, s)

for all v in $V(G)$ do
visited[v] := FALSE
end for.
S := Empty Stack
Push(S, s)
while not Empty(S) do
u := pop(S)
if not visited[u] then
visited[u] := TRUE
for all w in $Adj[u]$ do
if not visited[w] then
Push(S, w)
end if
end for
end if
end while

3.2 Breadth First Search

Breadth-first search (BFS) for undirected graph:
Let us consider a connected graph G.

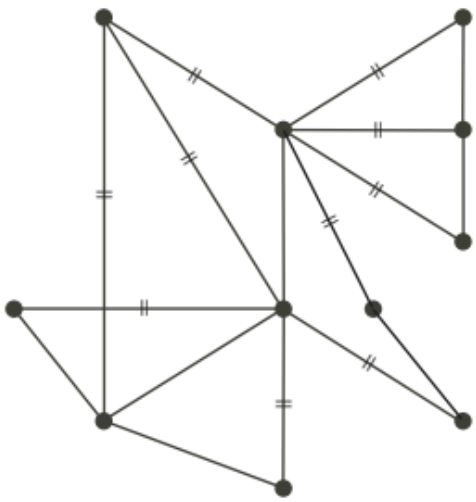
1. In the beginning, no vertex is labeled. Set $i \leftarrow 0$
2. Choose a(unlabeled) starting vertex r(root) and label it with i.
3. Search the set J of vertices that are not labeled and are adjacent to some vertex labeled with i.
4. If $J \neq \phi$, then set $i \leftarrow i + 1$. label the vertices in J with i and go to step 3.
5. (Only for disconnected graphs so that we can jump from one component to another) If a vertex is unlabeled, then set, $i \leftarrow 0$ and go to step 2.
6. Stop.

BFS also produces a spanning tree called the BFS tree, when we take the edges

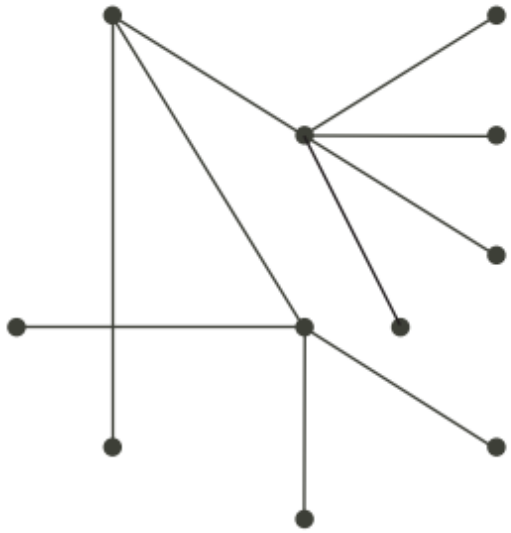
(vertex labeled with i, unlabeled vertex)

while forming J. One such $tree - edge$ exists for each vertex J. We obtain the *directed BFS tree* by orienting the edges away from the labeled vertex to the unlabeled vertex. BFS as presented above does not however orient every edge in graph. Obviously, the label if the vertex is the length of the shortest path from the root to it, in other words, the *distance* from the root.

Example *BFS in the graph we had in the previous example starts at a root in the upper left corner and proceeds as follows.(Tree edges are marked with two crossed lines.)*



The corresponding BFS tree is



We obtain the directed BFS tree by orienting the branches away from the tree.

Note: For both of the algorithms, in case of directed graph we will start from a vertex and move to only the $out - degree$ vertices and avoid the $in - degree$ vertices.

Algorithm using queue:
Here we will use a queue of vertices. Here goes the algorithm: BFS A(G, s)

for all v in $V[G]$ do
visited[v] := FALSE
end for
Q := Empty Queue
Enqueue(Q, s)
while not Empty(Q) do
u := Dequeue(Q)
if not visited[u] then
visited[u] := TRUE
for all w in $Adj[u]$ do
if not visited[w] then
Enqueue(Q, w)
end if
end for
end if
end while

4 Shortest Path in arbitrary path

4.1 Dijkstra's Algorithm

Problem: *The edges of a (di)graph are given non-negative weights. The weight of a path is the sum of the weight of the path traversed.We are to find the shortest path in the graph from vertex u to vertex v if the path exists. We should clearly state if such path exists or not.*

Obviously, we can assume that we do not have any loops or parallel edges. Otherwise we simply remove the loops and choose the edge with the lowest weight out of the parallel edges. From now on, we only consider directed graphs . Undirected graphs can be treated in the same way by replacing an edge with two arcs in opposite directions with the same weight.

We denote $\alpha(r, s)$ as the weight of the edge (r,s). Dijkstra's algorithm marks the vertices as *permanent* or *temporary* vertex. The label of a vertex r is denoted by $\beta(r)$ and we define

$$\gamma(r) = \begin{cases} 0, & \text{if the label is temporary} \\ 1, & \text{if the label is permanent} \end{cases}$$

A permanent label $\beta(r)$ expresses the weight of the shortest directed $u - r$ path. A temporary label $\beta(r)$ gives an upper limit to this weight(can be ∞). Furthermore, we denote

$$\pi(r) = \begin{cases} \text{the predecessor vertex } r \text{ on the shortest directed } u - r \text{ path , if such a path exists} \\ 0, \quad \text{otherwise} \end{cases}$$

so we can construct the directed path with the lowest weight.

Algorithm:

1. Set $\beta(u) \leftarrow 0$ and $\gamma(u) \leftarrow 1$. For all other vertices r, set $\beta(r) \leftarrow \infty$ and $\gamma(r) \leftarrow 0$. Furthermore, set $w \leftarrow u$.

2. For every edge (w, r), where $\gamma(r) = 0$ and $\beta(r) > \beta(w) + \alpha(w, r)$, set $\beta(r) \leftarrow \beta(w) + \alpha(w, r)$ and $\pi(r) \leftarrow (w)$.

3. Find a vertex r^* for which $\gamma(r^*) = 0, \beta(r^*) < \infty$ and

   $$\beta(r^*) = \min_{\gamma(r)=0} \{\beta(r)\}$$

   Set

   $$\gamma(r^*) \leftarrow 1 \; and \; w \leftarrow r^*$$

   . If there is no such vertex r^*, a directed $u - v$ path does not exist and we stop.

4. If $w \neq v$ then go to step 2.

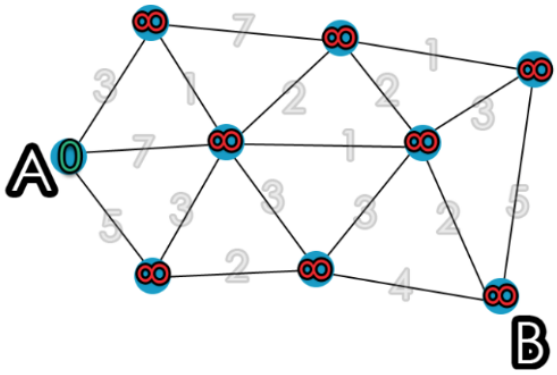
5. Stop.

We see that the algorithm is correct as follows.We denote (for every step):

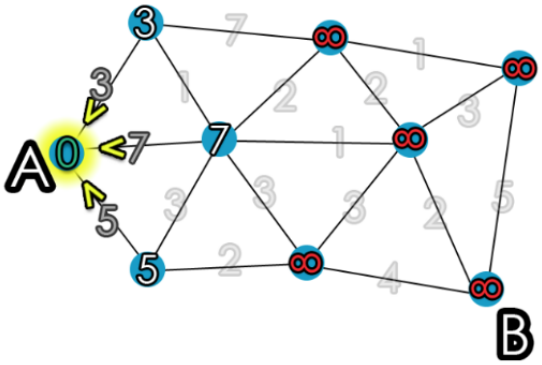
$$V_1 = \{\text{permanently labeled vertices}\}$$

$$V_2 = \{\text{temporarily labeled vertices}\}$$

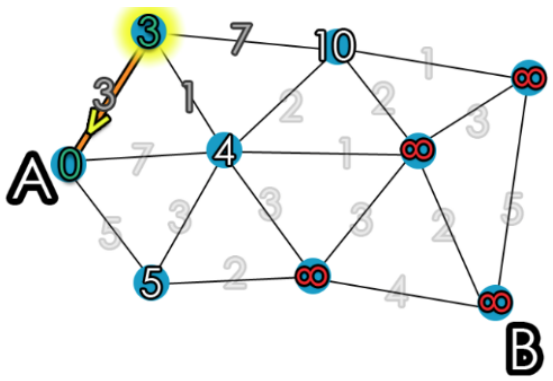
Example: Let us look at the following example.We want to move from A to B.
We will initialize the distance according to the algorithm.



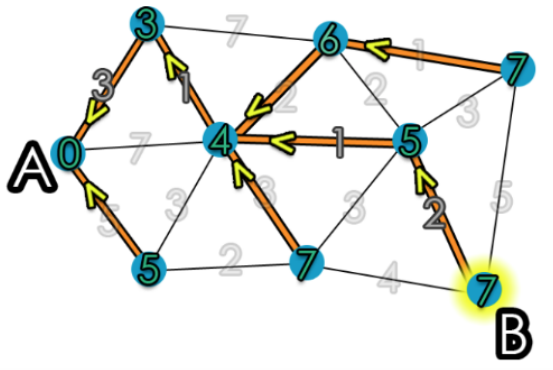
We will pick the first node and the distances to the adjacent nodes.



Now, we will pick next node with minimal distance; and repeat adjacent node distances calculation.



Finally, we will find the result of the shortest path.



Thus, we get the required path.

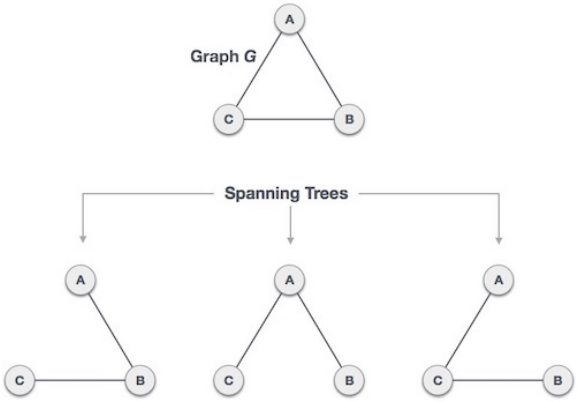
5 Minimum Spanning Tree

In order to understand the next algorithm, we should know certain terms.

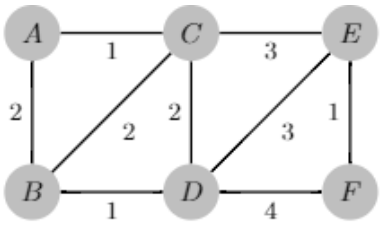
Definition 5.1 (Spanning Subgraph)**.** *A spanning subgraph is a subgraph that contains all the vertices of the actual graph.*

Definition 5.2 (Spanning tree:)**.** *A spanning tree is a subset of a graph G, which has all the vertices covered with minimum possible number of edges.*

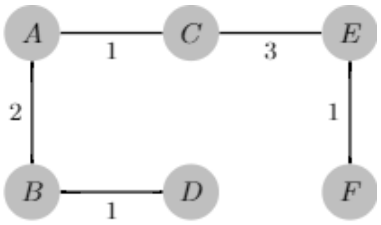
Hence, a spanning tree does not have cycles and it cannot be disconnected. By this definition, we can say that every connected and undirected graph has at least one spanning tree.



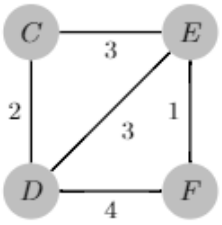
Definition 5.3 (Minimum spanning tree)**.** *A minimum spanning tree or a minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the possible total edge weight.*



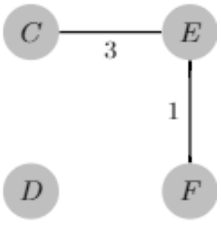
(1) Graph G



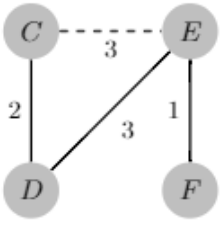
(2) An MST T_G of graph G



(3) A connected subgraph H of graph G



(4) $T_G \cap H$



(5) Some MST T_H of H

The above diagram illustrates the three definitions.

5.1 Kruskal's algorithm

Problem: *We have to find the spanning tree with the lowest weight of a connected graph if the edges of the graph has been weighted arbitrarily and the weight of a tree is the sum of all the weights of the branches.*

Obviously, we can assume that the graph $G =(V, E)$ is non-trivial and simple. Otherwise, we simply remove the loops and choose the edge with the lowest weight out of the parallel edges. We denote the weight of a edge e as $\alpha(e)$ and the weight of the spanning tree T as $\gamma(T)$. As usual, we write the number of vertices as n, number of edges as m, $V = \{v_1, ..., v_n\}$ and $E = \{e_1, ...e_m\}$.

The *distance* between two spanning trees T_1 and T_2 of G is

$$n - 1 - (cardinality\ of\ (T_1 \cap T_2)) = d(T_1, T_2)$$

where cardinality of ($T_1 \cap T_2$) is the number of edges in the intersection of T_1 and T_2. Obviously, $d(T_1, T_2) = 0$ if and only if $T_1 = T_2$. If $d(T_1, T_2)$ is 1 then they are neighboring trees.

In $Kruskal'sAlgorithm$, the edges of the graph G (and their weights) are listed as $e_1, ..., e_m$.
The algorithm constructs a minimal spanning tree by going through the list to take some edges to form the tree. This is specially effective if the edges are sorted in ascending order by weight.

(The are various forms of this algorithm, but only a few have been given.)

Algorithm1:

Here we assume that the edges are given in ascending order by weight.

1. Set $k \leftarrow 1$ and $A \leftarrow \phi$.
2. If e_k does not form a closed path with the edges in A, then set $A \leftarrow A \cup \{e_k\}$ as well as $k \leftarrow k + 1$ and go to step 4.
3. If e_k forms a closed path with the edges in A, then set $k \leftarrow k + 1$ and go to step 4.
4. If (V, A) is not a tree, then o to step 2. Otherwise stop and output the spanning tree $T = (A)$.

Whenever we leave out an edge from A (step 3), its end vertices are already connected in A. Thus, the vertices of G are connected in T as they are in G. Since T is not a closed path (step3), it is also a spanning tree of G.At each stage, the branches of the fundamental closed path defined by the link belonging to T^* (step 3) are predecessors of that link in the list. Hence, T is minimal.

Remark: *In each step, the branches and links are permanent. We have to know the edges beforehand as long as we process them one by one in ascending order. The rank of the graph(number of branches in a spanning tree) is then required beforehand so we know when to stop.*

Algorithm 2:

Here we assume that the edges are in arbitrary order.

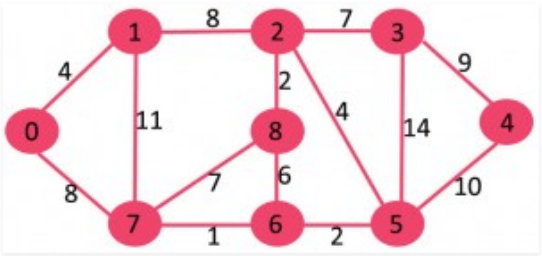
1. Set $k \leftarrow 1$ and $A \leftarrow \phi$.
2. If $(A \cup \{e_k\})$ contains no closed path, then set $A \leftarrow A \cup \{e_k\}$ as well as $k \leftarrow k+1$ and go to step 4.
3. If $(A \cup \{e_k\})$ contains a closed path C, then choose the edge with the largest weight e in C (if there are more than one, take away), set $A \leftarrow A \cup \{e_k\} - \{e\}$ as well as $k \leftarrow k+1$ and go to step 4.
4. If $k \leq m$, then go to step 2. Otherwise, stop and output the spanning tree $T = (A)$.

Whenever we leave out on an edge from A (step 3), its end vertices are already connected in A. Thus, the edges of G are connected in T as they are in G. Since T is obviously not closed (step 3) it is a spanning tree of G.

We see T is minimal by the following logic: During the whole process, (A) is a forest(A forest is an undirected graph, all of whose connected components are trees) by step 4. In addition, if u and w are connected in (A) at some point, then they are also connected afterwards. The $u-w$ path in (A) is unique but it can change to another path later in step 3. Nevertheless, whenever this changes occur, the maximum value of the weights of the edges of the path cannot increase anymore. Every link c of T^* has been removed from A in step 3. Then, the weight of c is at least as large as the weights of the other edges in C. After we have gone through step 3, the only connected end vertices of c in (A) have to go through the remaining edges of C. The final connection between the end vertices of c in T goes through the edge of fundamental closed path defined by c. Therefore, the weights of the edges of this fundamental closed path $\leq \alpha(c)$.

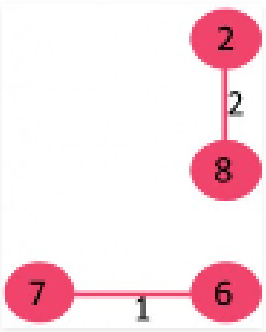
Remark: *In each step, the links(e in step 3) are permanent and the branches are not. We do not have to know the edges beforehand as long as we process them one by one. However, we need to know the nullity of the graph (nullity of the adjacency matrix) so that we know when to stop. The algorithm can also be used to update a minimal spanning tree if we add edges to the graph or decrease their weights.*

Example: Let us look at following example.

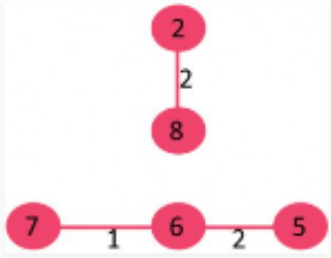


The graph contains 9 vertices and 14 edges. So, the minimum spanning tree will have 8 edges. We will choose according to the weights of its adjacent edges.
At first,pick edge 7-6. No cycle is formed, so include it.

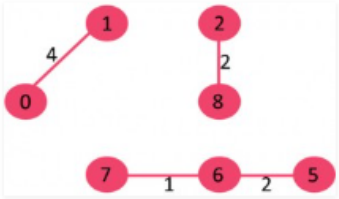
Then, pick edge 8-2, no cycle is formed so include it.



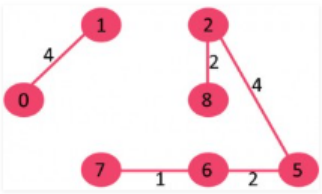
Now, pick edge 6-5 and include it.



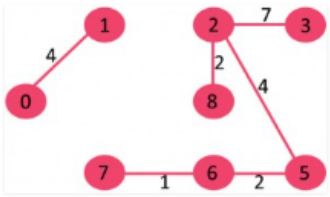
Pick edge 0-1 and include it.



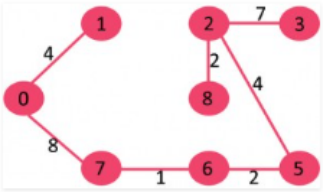
Pick edge 2-5 and include it.



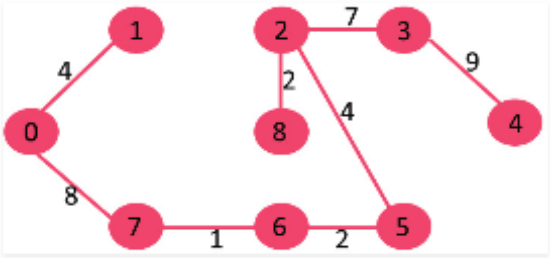
Pick edge 8-6. Since this results in a cycle so discard this edge. Pick edge 2-3 and include it.



Pick edge 7-8 but this also forms a cycle, so discard it. Pick edge 0-7.



Pick edge 3-4 and include it.



Since the number of edges is 8, the algorithm stops here.