Chapter IV. Pathway problems

This chapter is devoted to the study of *pathway problems* in graph theory, which consist of determining specific structures or properties within a graph by following its vertices and edges according to well-defined rules. These problems are fundamental in various applications such as network design, communication systems, transportation, and optimization.

In the first section**,** we focus on the search for connected components, which aims to identify all subsets of vertices where each pair of vertices is connected by a path. The Trémeaux–Tarjan algorithm is introduced as an efficient method for detecting both connected and strongly connected components in undirected and directed graphs, respectively.

The **second section** deals with the problem of **finding the shortest path** between vertices in a weighted graph. After presenting the main conditions of the problem, we describe the **Moore–Dijkstra algorithm**, which provides an optimal and systematic way to compute the minimum distance from a given source vertex to all others.

Finally, the **third section** addresses the construction of a **spanning tree of maximum weight** in a connected graph. The objective is to select a subset of edges that connects all vertices while maximizing the total weight. For this purpose, the **Kruskal algorithm (1956)** is presented as a classical and effective approach.

Through these three parts, the chapter offers a unified understanding of fundamental graph traversal and optimization techniques that form the basis of many modern computational applications.

2. Search for connected components

1.1. Presentation of objectives:

The study of graph connectivity represents one of the most fundamental topics in graph theory. In many applications, it is sufficient to restrict attention to a single strongly connected component, since any graph can be analyzed component by component.

A variety of algorithms have been proposed to determine the set of vertices that belong to the same strongly connected component. In this chapter, we present a simple algorithm designed to identify the strongly connected component containing a specified vertex, following the works of Trémeaux (1882) and Tarjan (1972).

1.2. Trémeaux-Tarjan algorithm:

Principle of the Algorithm:

Let **G** be a directed graph. The main idea of the Trémeaux–Tarjan algorithm is to perform a *depth-first traversal* of **G** starting from a given vertex **x**, exploring both in the *forward direction* (following the orientation of the arcs) and in the *reverse direction* (following the arcs in the opposite orientation. The arcs traversed during the depth-first search form a tree.

The depth-first search (DFS) is implemented by managing the list of vertices to be visited as a stack (LIFO – Last In, First Out) structure. At each step, the algorithm considers the most recently discovered (marked) vertex at the top of the stack. From this vertex, it attempts to visit one of its *unmarked successors* (in the case of forward traversal) or *unmarked predecessors* (in the case of backward traversal). Each time such a vertex is found, it is marked and pushed onto the stack, and the search continues deeper into the graph. If no unmarked successor (or predecessor) exists, the algorithm *backtracks* by removing the top vertex from the stack and returning to the previous vertex. This process continues until no further vertex can be marked.

The first traversal identifies the set of vertices reachable from **x**, while the second traversal determines the set of vertices that can reach **x**. The intersection of these two sets forms the collection of vertices that are both reachable from and can

reach **x**; this set corresponds to the strongly connected component containing the vertex **x**.

Algorithm:

Input: A directed graph $\mathbf{G} = (\mathbf{X}, \mathbf{U})$.

Output: The number $\mathbf{k}$ of strongly connected components of $\mathbf{G}$, and the list $\mathbf{C}_1, \mathbf{C}_2, ..., \mathbf{C}_k$ of these components.

Step 1 – Initialization

1. Initialize $\mathbf{k} = 0$ and create an empty stack $\mathbf{S}$.
2. Choose an arbitrary unmarked vertex $\mathbf{x}$ of $\mathbf{X}$.

Step 2 – Forward Depth-First Search (DFS)

1. Mark $\mathbf{x}$ with a (**+**) sign and push it onto stack $\mathbf{S}$.
2. While $\mathbf{S}$ is not empty, do:
   a) Let $\mathbf{v}$ be the vertex on the top of the stack.
   b) If there exists an unmarked successor $\mathbf{u}$ of $\mathbf{v}$ in $\mathbf{G}$, then mark $\mathbf{u}$ with a (**+**) sign and push $\mathbf{u}$ onto stack $\mathbf{S}$.
   c) Otherwise, pop $\mathbf{v}$ from stack $\mathbf{S}$.
3. Let **ForwardSet** = {all vertices marked with (**+**)}
4. Unmark all vertices.

Step 3 – Backward Depth-First Search (DFS)

1. Mark $\mathbf{x}$ with a (**–**) sign and push it onto stack $\mathbf{S}$.
2. While $\mathbf{S}$ is not empty, do:
   a. Let $\mathbf{v}$ be the vertex on the top of the stack.
   b. If there exists an unmarked predecessor $\mathbf{u}$ of $\mathbf{v}$ in $\mathbf{G}$, then mark $\mathbf{u}$ with a (**–**) sign and push $\mathbf{u}$ onto stack $\mathbf{S}$.
   c. Otherwise, pop $\mathbf{v}$ from stack $\mathbf{S}$.
3. Let **BackwardSet** = {all vertices marked with (**–**)}.
4. Unmark all vertices.

Step 4 – Identification of the Strongly Connected Component

1. Increment $\mathbf{k} = \mathbf{k} + 1$.
2. Define $\mathbf{C}_k$ **= ForwardSet • BackwardSet.**
3. Remove the vertices of $\mathbf{C}_k$ from $\mathbf{X}$, and let $\mathbf{X} = \mathbf{X} \setminus \mathbf{C}_k$.

4. If $\mathbf{X}$ is not empty, let $\mathbf{G}'$ be the subgraph of $\mathbf{G}$ induced by the remaining vertices. Set $\mathbf{G} = \mathbf{G}'$ and repeat from Step 2.
5. Otherwise, terminate.

Step 5 – Output

The number of strongly connected components of $\mathbf{G}$ is $\mathbf{k}$.
Each set $\mathbf{C_i}$, $\mathbf{i} = 1,\ldots, \mathbf{k}$, corresponds to the set of vertices forming a strongly connected component of $\mathbf{G}$.

Example:

We apply the previous marking algorithm to determine the strongly connected components of the following graph:

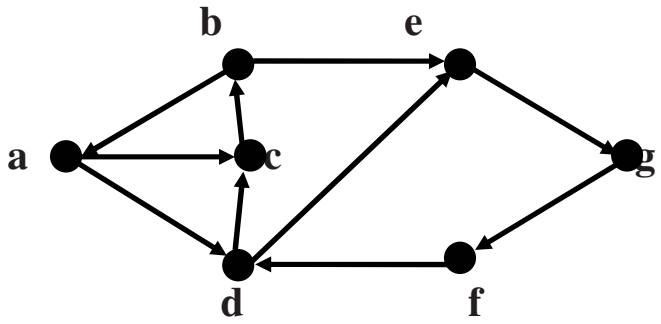


Figure 1: Search for strongly connected components.

We begin by setting $\mathbf{k} = 0$ and $\mathbf{S} = <>$.
From the set $\mathbf{X}$, we select vertex $\mathbf{a}$, mark it with the symbol (+), and push it onto the stack $\mathbf{S}$. Thus, $\mathbf{S} = <\mathbf{a}>$.
Since $\mathbf{S}$ is not empty, the vertex on the top of the stack is $\mathbf{a}$. Vertex $\mathbf{a}$ has two unmarked successors, $\mathbf{c}$ and $\mathbf{d}$. During the forward depth-first traversal, we arbitrarily choose vertex $\mathbf{d}$, mark it with (+), and push it onto the stack. We then obtain $\mathbf{S} = <\mathbf{a},\mathbf{d}>$. Note that there is no particular preference when choosing between $\mathbf{c}$ and $\mathbf{d}$.
Now, $\mathbf{d}$ is at the top of the stack and has one unmarked successor. We mark this successor with (+) and push it onto the stack, giving $\mathbf{S} = <\mathbf{a},\mathbf{d},\mathbf{e}>$. Similarly, for vertex $\mathbf{e}$, we obtain $\mathbf{S} = <\mathbf{a},\mathbf{d},\mathbf{e},\mathbf{f}>$.

At this stage, vertex **f** has no unmarked successors, so we pop **f** from the stack, yielding $\mathbf{S} =< \mathbf{a},\mathbf{d},\mathbf{e} >$.

For the same reason, we pop vertex **e**, obtaining $\mathbf{S} =< \mathbf{a},\mathbf{d} >$. Returning to vertex **d**, we find that it has one remaining unmarked successor, **c**. We then push **c** onto the stack, resulting in $\mathbf{S} =< \mathbf{a},\mathbf{d},\mathbf{c} >$.

The forward depth-first traversal continues as follows: $\mathbf{S} = \langle \mathbf{a},\mathbf{d},\mathbf{c},\mathbf{b} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{d},\mathbf{c} \rangle$, then $\mathbf{S} = \langle \mathbf{a},\mathbf{d} \rangle$, $\mathbf{S} = \langle \mathbf{a} \rangle$, and finally $\mathbf{S} = \langle\ \rangle$.

The process terminates, and we obtain the **ForwardSet = X**.

During the backward depth-first traversal, the stack evolves as follows:

$\mathbf{S} = \langle\ \rangle$, $\mathbf{S} = \langle \mathbf{a} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{f} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{f},\mathbf{e} \rangle$, then $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{f} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b},\mathbf{c} \rangle$, $\mathbf{S} = \langle \mathbf{a},\mathbf{b} \rangle$, $\mathbf{S} = \langle \mathbf{a} \rangle$, and finally $\mathbf{S} = \langle\ \rangle$.

Thus, the **BackwardSet = X**.

We obtain the first strongly connected component:

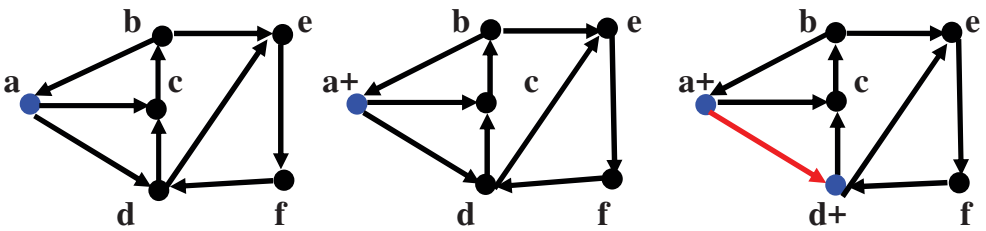
$\mathbf{C}_1 =$ ForwardSet $\cap$ BackwardSet = X.

After removing from **X** the vertices belonging to C_1, we obtain: $X = X \backslash C_1 = \varnothing$.

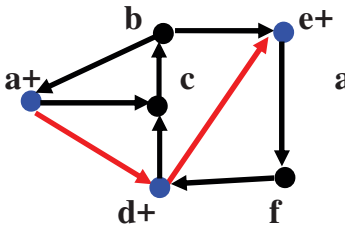
The process is thus completed. The graph **G** contains one strongly connected components. It follows that the graph is strongly connected.

The following figures present, in a stepwise manner, the marking procedure and the associated stack configurations observed during the forward and backward depth-first traversals:

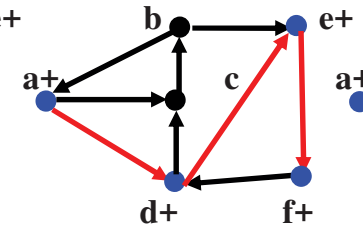
1) **<u>Forward depth-first traversals:</u>**



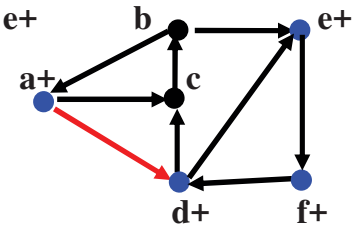
$\mathbf{S} = \langle\ \rangle$ $\mathbf{S} = \langle \mathbf{a} \rangle$ $\mathbf{S} = \langle \mathbf{a},\mathbf{d} \rangle$



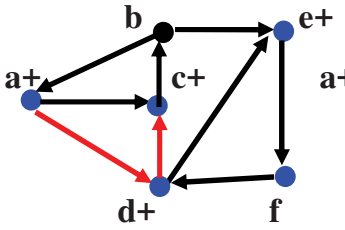
$S = \langle a, d, e \rangle$



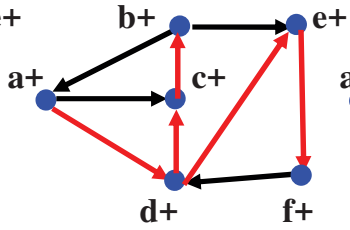
$S = \langle a, d, e, f \rangle$



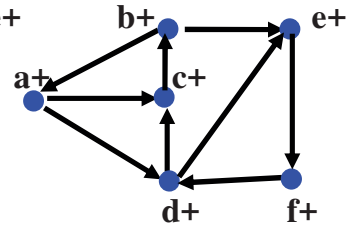
$S = \langle a, d \rangle$



$S = \langle a, d, c \rangle$

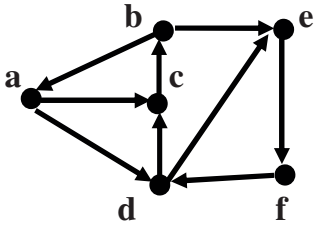


$S = \langle a, d, c, b \rangle$

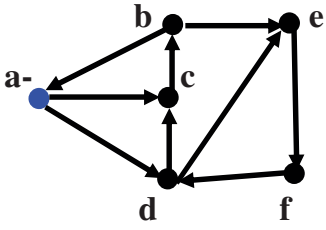


$S = \langle \ \rangle$

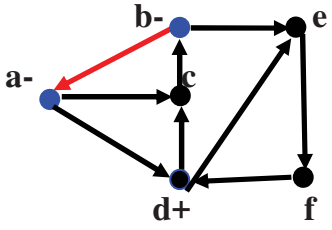
2) **Backward depth-first traversals:**



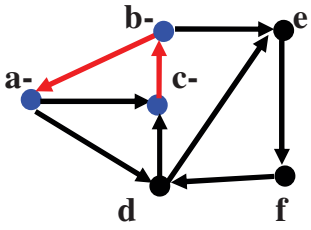
$S = \langle \ \rangle$



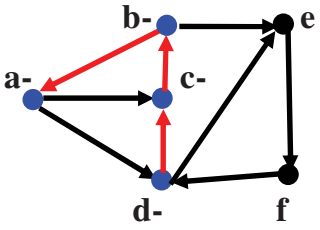
$S = \langle a \rangle$



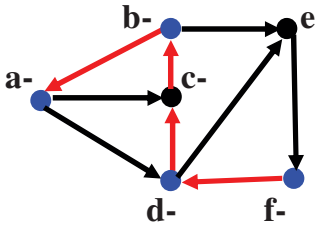
$S = \langle a, b \rangle$



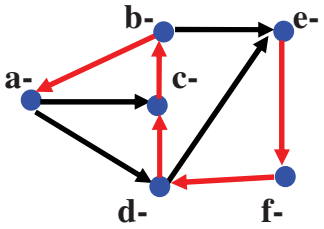
$S = \langle a, b, c \rangle$



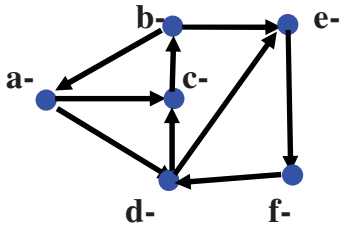
$S = \langle a, b, c, d \rangle$



$S = \langle a, b, c, d, f \rangle$



$S = \langle a, b, c, d, f, e \rangle$



$S = \langle \ \rangle$

2. Finding the shortest path

2.1. Presentation of the conditions

Let $\mathbf{G}$ be a graph in which each arc $\mathbf{u}$ is assigned a length $\mathbf{C(u)} \geq 0$.

Given two vertices $\mathbf{s}$ and $\mathbf{k}$ of $\mathbf{G}$, the goal is to find a path μ from $\mathbf{s}$ to $\mathbf{k}$ such that the sum of the lengths of its arcs is as small as possible.

The path μ is then called the *shortest path* from $\mathbf{s}$ to $\mathbf{k}$.

2.2 Moore-Dijkstra algorithm

This algorithm is applied to determine a *shortest-distance spanning tree* on a transport network $\mathbf{R = (X, U, C)}$, where the arc lengths are non-negative, that is, $\mathbf{C(u)} \geq 0$ for every $\mathbf{u} \in \mathbf{U}$.

Principle of the algorithm:

The idea of the Moore (1957) and Dijkstra (1959) algorithm is to progressively compute the shortest-distance spanning tree from a given starting vertex $\mathbf{s}$ to any other vertex $\mathbf{k}$. A distinctive feature of this algorithm is that the distances are introduced in increasing order.

Algorithm:

Input:

A a transport network $\mathbf{R = (X, U, C)}$, where $\mathbf{C(u)} \geq 0, \quad \forall u \in U$.

A source vertex $\mathbf{s} \in \mathbf{X}$.

Output:

The shortest paths from $\mathbf{s}$ to every other vertex of the graph.

Step 1:

1. Assign to each vertex $\mathbf{x} \in \mathbf{X}$ a label $\mathbf{d(x)}$ representing the current shortest known distance from from the starting vertex $\mathbf{s}$ to $\mathbf{x}$.
2. Set $\mathbf{d(s)} = 0$ and $\mathbf{d(x)} = +\infty$ for all other vertices.
3. Let $\mathbf{S}$ be the set of *unvisited* vertices, initially $\mathbf{S} = \{\mathbf{s}\}$.
4. Let $\mathbf{A}$ be the list of vertices included in the shortest-path spanning tree, initially $\mathbf{A} = \varnothing$.
5. Let α the the latest introduced vertex in $\mathbf{A}$. Set $\alpha = \mathbf{s}$.

Step 2:

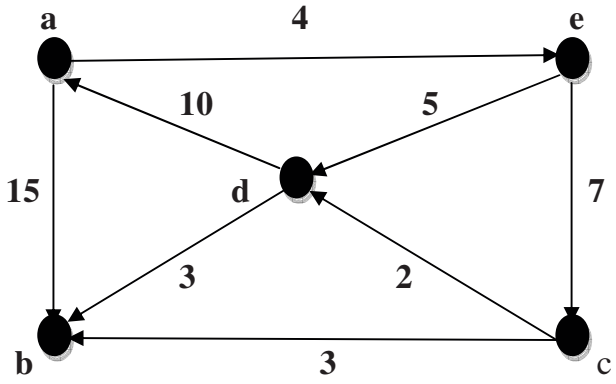
Examine all arcs $\mathbf{u}=(\alpha,\mathbf{y})$ such that the initial vertex is α and the terminal vertex $\mathbf{y}$ is not yet included in the set $\mathbf{A}$. If the condition $\mathbf{d}(\alpha)+\mathbf{C}(\mathbf{u})<\mathbf{d}(\mathbf{y})$ is satisfied, update the label of vertex $\mathbf{y}$ by setting $\mathbf{d}(\mathbf{y})=\mathbf{d}(\alpha)+\mathbf{C}(\mathbf{u})$.

Step 3:

Choose a vertex $\mathbf{z}\notin\mathbf{A}$ such that $\mathbf{d}(\mathbf{z})=\min\{\mathbf{d}(\mathbf{y}):\ \mathrm{y}\notin\mathrm{A}\}$.

1. If $\mathbf{d}(\mathbf{z})=\infty$, stop. The vertex $\mathbf{s}$ is not a root in $\mathbf{R}$.
2. If $\mathbf{d}(\mathbf{z})<\infty$, set $\alpha=\mathbf{z}$ and $\mathbf{A}:=\mathbf{A}\cup\{\alpha\}$.
3. If $\mathbf{A}=\mathbf{X}$, stop. $\mathbf{A}$ defines the shortest-path tree rooted at $\mathbf{s}$.
4. If $\mathbf{A}\neq\mathbf{X}$, return to Step 2.

Example:



Initialization:

Vertex **x**	**a**	**b**	**c**	**d**	**e**
d(x)	0	∞	∞	∞	∞

$\mathbf{A}=\{\mathbf{a}\}$, $\mathbf{T}=\varnothing$, and $\alpha=\mathbf{a}$.

1) Update of distances

We examine the two arcs **(a,b)** and **(a,e)**:

d(a) + C(a,b) = 15 < d(b) = ∞ ⇒ d(b) = 15.

d(a) + C(a,e) = 4 < d(e) = ∞ ⇒ d(e) = 4.

Vertex **x**	**a**	**b**	**c**	**d**	**e**
d(x)	0	15	∞	∞	4

Selection

d(e) = min { d(b), d(c), d(d), d(e)}

Update

α = e, A = {a, e}, $\mathbf{T} = \{(\mathbf{a},\mathbf{e})\}$.

2) Update of distances

We examine the two arcs **(e,c)** and **(e,d)**:

d(e) + C(e,c) = 11 < d(c) = ∞ ⇒ d(c) = 11.

d(e) + C(e,d) = 9 < d(d) = ∞ ⇒ d(d) = 9.

Vertex **x**	**a**	**b**	**c**	**d**	**e**
d(x)	0	15	11	9	4

Selection

d(d) = min{d(b), d(c), d(d)}.

Update

α = 4, A={a, e, d}, $\mathbf{T} = \{(\mathbf{a},\mathbf{e}),(\mathbf{e},\mathbf{d})\}$.

3) Update of distances

We examine arc **(d,b)**:

d(d) + C(d,b) = 12 < d(b) = 15 ⇒ d(b) = 12.

Vertex **x**	**a**	**b**	**c**	**d**	**e**
d(x)	0	12	11	9	4

Selection

$d(c) = \min\{d(b), \pi(c)\}$.

Update

$\alpha = c$, $A=\{a,e,d,c\}$, $T = \{(a,e),(e,d),(e,c)\}$.

4) Update of distances

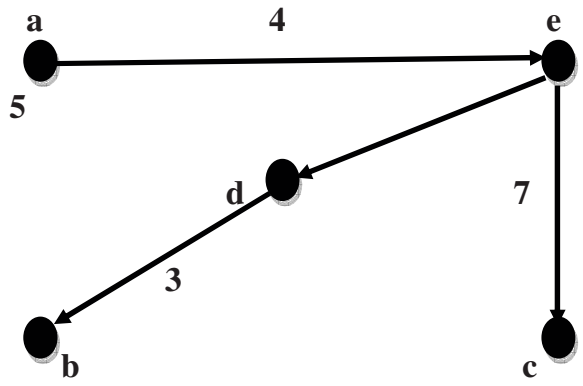
We examine arc **(c,b)**:

$d(c)+C(c,b) = 14 > d(b) = 12$.

Update

$\alpha = b$, $A = \{a,e,d,c,b\}$, $T = \{(a,e),(e,d),(e,c),(d,b)\}$.

The shortest-path tree originating from vertex **a** is



3. Search for a spanning tree of maximum weight in a graph

3.1. Presentation of objectives

Each edge of the graph $G = (X, U)$ is assigned a value (or weight). The minimum spanning tree problem consists in finding a subgraph that is a tree, such that the sum of the weights of its edges is minimal.

3.2. Kruskal's Algorithm (1956):

Principle of the algorithm:

The idea of Kruskal's algorithm is first to sort the edges in increasing order of their weights. Then, the algorithm gradually constructs the tree $\mathbf{A}$ by adding edges one by one in that order.

An edge is added only if its inclusion does not create a cycle in $\mathbf{A}$. Otherwise, the algorithm skips that edge and continues with the next one in the sorted list.

Algorithm

Input:

Weighted multigraph $\mathbf{G}=(\mathbf{X},\mathbf{U},\mathbf{C})$.

Output:

A tree or forest $\mathbf{A}=(\mathbf{V},\mathbf{W})$ of minimum total weight.

Step 1

Sort and number the edges of $\mathbf{G}$ in non-decreasing order of their weights: $\mathbf{C}(\mathbf{u}_1)\le\mathbf{C}(\mathbf{u}_2)\le\ldots\le\mathbf{C}\le(\mathbf{u}_\mathbf{m})$

Step 2

$\mathbf{W}=\varnothing$, $\mathbf{k}=0$.

Step 3

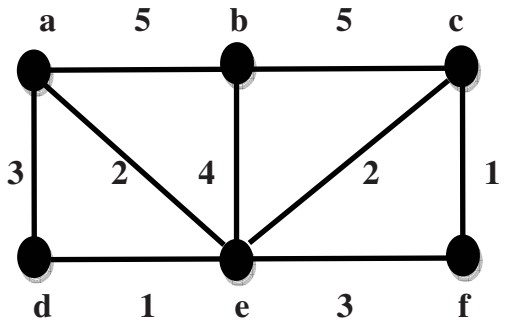
While $\mathbf{k}<\mathbf{m}$ and $|\mathbf{W}|<\mathbf{m}-1$ do:

If $\mathbf{u}_{\mathbf{k}+1}$ does not create a cycle with the edges in $\mathbf{W}$, then:

- $\mathbf{W}=\mathbf{W}\cup\{\mathbf{u}_{\mathbf{k}+1}\}$.
- $\mathbf{k}=\mathbf{k}+1$.

Example:

Let the following Weighted multigraph $\mathbf{G}=(\mathbf{X},\mathbf{U},\mathbf{C})$:



Sorted edges by increasing weight:

C(d,e)≤C(c,f)≤C(a,e)≤C(c,e)≤C(a,d)≤C(e,f)≤C(b,e)≤C(a,b)≤C(b,c)

Construction process:

W = ∅

W = {(d,e)}

W = {(d,e), (c,f)}

W = {(d,e), (c,f), (a,e)}

W = {(d,e), (c,f), (a,e), (c,e)}

W = {(d,e), (c,f), (a,e), (c,e), (b,c)}

The final set W represents the minimum spanning tree of (G):