# Chapter 5 *Topology design and analysis*

**<u>Topics covered:</u>**
Topology design. Network design algorithms. Terminal assignment. Concentrator location. Traffic flow analysis and performance evaluation. Network reliability. Network simulation.

## *5.1* Topology design

### 5.1.1 Centralized Network design

- **<u>Centralized network:</u>** is where all communication is to and from a single central site.

- The "central site" is capable of making routing decisions.
  → Tree topology provides only one path through the center (For reliability, lines between other sites can be included)

- Three different problems:

  - **<u>Multipoint line topology:</u>** selection of links connecting terminals to concentrators or directly to the center.

  - **<u>Terminal assignment:</u>** association of terminals with specific concentrators.

  - **<u>Concentrator location:</u>** deciding where to place concentrators, and whether or not to use them at all.

### 5.1.2 Finding Trees in Graphs

- Used to design and analyze networks.

- Connect a number of nodes to a central node:

  - **<u>Node:</u>** Hub, Switch, Router, etc.

  - **<u>Central node:</u>** backbone

- A tree is a graph with no loops, with only one path between any pair of nodes.
- Trees are minimal networks: provide connectivity without any unnecessary additional links:
  - Minimally reliable and robust
  - Networks are more highly connected (but design starts with a tree)

#### 5.1.2.1 Tree Traversals

- Visit all nodes in a tree: edges are traversed twice.
- First, identify a node as the root
- Assume the tree is directed (outward from the root)
- Two algorithms:
  - BFS (Breadth First Search):
    - Nodes closest to root are visited first
    - Implemented using a queue (FIFO)
  - DFS (Depth First Search):
    - Visits an unvisited neighbor of the node just visited.
    - Implemented using a stack (LIFO)
- Both traversals (BFS and DFS) can be preorder traversals (i.e., visit nodes then successors) or post-order traversals (i.e., successors visited first).

- Traversal is generalized to undirected graphs by keeping track of which nodes were visited, and not visiting them again.

- In a BFS or DFS traversal, edges visited form a tree (if the graph is connected) or a forest (if the graph is not connected).

### 5.1.2.2 Minimum Spanning Trees (MSTs)

- Use DFS to find a spanning tree in a graph, if one exists
  → Arbitrary tree

- Useful to find the "best" tree
  → Minimum Spanning Tree (e.g., minimum total length. Where length is: distance, cost, function(delay), function(reliability), etc.)

- If the graph is not connected → minimum spanning forest
  - For $n$ nodes, $c$ components, and $e$ edges, we have: $n = c + e$
  - For a tree, $c = 1$.

- DFS will not, in general, find the spanning tree with minimum total cost.

#### *5.1.2.2.1 The Greedy Algorithm*

- At each stage, select the shortest edge possible.

- May not find a feasible solution when one exists.

- Efficient and simple to implement → widely used.

- Basis of other more complex and effective algorithms.

- In the case of MST, the greedy algorithm guarantees both optimality and reasonable computational complexity.

  - Start with empty solution $s$
  - While elements exist
    - Find $e$, the best element not yet considered
    - If adding $e$ to $s$ is feasible, add it; if not, discard it.

### *5.1.2.2.2 Kruskal's Algorithm*

- A greedy algorithm for finding MSTs.
- Sort the edges, shortest first and then include all edges which do not form cycles with the edges previously selected.
- n: number of nodes
- **Algorithm:**

  1. Sort all edges in ascending order (least cost first)
  2. Select among edges not yet selected, the one with the least cost.
  3. Add it if it does not create a cycle.
  4. If the number of edges selected < ***n-1***, go to step (2), otherwise exit (tree completed)

- **Complexity:**

- **Example:**

Given a network with five nodes, labelled A to E, and characterized by the following cost matrix:

| | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| **A** | - | 3 | 3 | 5 | 10 |
| **B** | 3 | - | 6 | 4 | 8 |
| **C** | 3 | 6 | - | 3 | 5 |
| **D** | 5 | 4 | 3 | - | 7 |
| **E** | 10 | 8 | 5 | 7 | - |

#### *5.1.2.2.3 Prim's Algorithm*

- A greedy algorithm for finding MSTs.
- Advantageous if the network is dense.
- Well suited to parallel implementation.
- **Algorithm:**

    1. Start with one node (root node) in the tree
    2. Find node ***i***, not in the tree, which is the nearest to the tree.
    3. Add node ***i*** to the tree and edge ***e*** connecting ***i*** to the tree.

- **Complexity:**

- **Example:**

## 5.1.3 Constrained/Capacitated MST (CMST)

- The algorithms presented in the previous subsections are called "unconstrained MST algorithms"
  - No constraint on flow of information
  - No constraint on the number of ports at each node.

- For the unconstrained spanning tree problem, all these algorithms produce a minimum cost spanning tree.

- **CMST Problem:** Given a central node $N_0$ and a set of other nodes ($N_1$, $N_2$, …, $N_n$), as et of weights ($W_1$, $W_2$, …, $W_n$) for each node, the capacity of a link, $W_{max}$, and a cost matrix $C_{ij}$ = Cost(i,j), find a set of trees $T_1$, $T_2$, …, $T_k$ such that each $N_i$ belongs to exactly one $T_j$ and each $T_j$ contains $N_0$.

- **Objective:** Find a tree of minimum cost and which satisfies a number of constraints such as:
  - Flow over a link
  - Number of ports

- **Example:**
  - Assume we are allowed to use one type of links only that can accommodate a maximum of 5 units of flow per unit time.

  - Assume that the flow generated from each node to the central node ($a_1$) is as follows: $a_1$=0, $a_2$=2, $a_3$=3, $a_4$=2, $a_5$=2 (in units/time_unit).

  - Effect of constraint violation:
    - As a result, a queue will build up since node 3 can service only 5 units/time_unit. If node 3 does not have a large queue to accommodate all coming units, some units will be lost. So, these units are retransmitted, which may cause the network to collapse.

- The CSMT problem is NP-hard (i.e., cannot be solved in polynomial time)
  → Resort to heuristics (approximate algorithms)

- These heuristics will attempt to find a good feasible solution, not necessarily the best, that:
  - Minimizes the cost
  - Satisfies all the constraints

- Well-known heuristics:
  - Kruskal
  - Prim
  - Esau-Williams

### 5.1.3.1 Kruskal's Algorithm for CMST

- Sort all edges in ascending order, ***e* ← 0**.

1. Select edge with minimum cost (from edges not yet selected)

2. If it satisfies constraints (i.e., <u>no cycles</u> and <u>no violation of flows on links</u>)

   - Then: add it to the tree, ***e* ← *e* + 1**
     - Else: go to step (2)

- If (**e = n – 1**) then exit, else go to step (2)

**<u>Example:</u>**

Given a network with five nodes, labelled A to E, and characterized by the following cost matrix:

| | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| **A** | - | 3 | 3 | 5 | 10 |
| **B** | 3 | - | 6 | 4 | 8 |
| **C** | 3 | 6 | - | 3 | 5 |
| **D** | 5 | 4 | 3 | - | 7 |
| **E** | 10 | 8 | 5 | 7 | - |

### 5.1.3.2 Prim's Algorithm for CMST

1. Start with one node (root node) in the tree.

2. Find node ***i***, not in the tree, which is the nearest to the tree

3. Add node ***i*** to the tree and edge ***e*** connecting ***i*** to the tree if it satisfies constraints (i.e., no cycles and no violation of flows on links)

**Example:**

### 5.1.3.3 Esau-Williams Algorithm for CMST

Node 1 is the central node.

1. Compute $t_{ij} = c_{ij} - c_{i1}$ for all i, j.
   $t_{ij}$: is the tradeoff of connecting i to j or i directly to the root.
   - If ($t_{ij} < 0$) → connect i to j
   - If ($t_{ij} \geq 0$) → connect i directly to the root

2. Select the link (m,n) such that: $t_{mn} = \min(t_{ij})$

3. If $t_{mn} < 0$, then go to step (4)
   Else, connect all the nodes not connected yet to the node 1, and exit.

4. Verify constraints (e.g., exceeds the maximum weight)
   - If satisfied go to step (5)
   - Else: $t_{mn} = \infty$, go to step (2)

5. Add link (m,n) and update $t_{ij}$ to indicate that **m** is now connected to **n**.
   → $t_{mn} = \infty$,
   → $t_{ij} = c_{ij} - \min(c_{k1})$ [$k \in C_i$] if $t_{ij} \neq \infty$.

6. If tree has (n-1) links then exit,
   Else, go to step (2)

**Example:**

Given a network with five nodes, labelled A to E, and characterized by the following cost matrix:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **A** | - | 3 | 3 | 5 | 10 |
| **B** | 3 | - | 6 | 4 | 8 |
| **C** | 3 | 6 | - | 3 | 5 |
| **D** | 5 | 4 | 3 | - | 7 |
| **E** | 10 | 8 | 5 | 7 | - |

## 5.1.4 Terminal Assignment

### 5.1.4.1 Problem Statement

- **Terminal Assignment:** Association of terminals with specific concentrators.

**Given:**

- **T** terminals (stations) $i = 1, 2, \ldots, T$
- **C** Concentrators (hubs/switches) $j = 1, 2, \ldots, C$
- $\mathbf{C_{ij}}$: cost of connecting terminal i to concentrator j
- $\mathbf{W_j}$: capacity of concentrator j

Assume that terminal i requires $W_i$ units of a concentrator capacity.

Assume that the cost of all concentrators is the same.

- $x_{ij} = 1$; if terminal i is assigned to concentrator j.
- $x_{ij} = 0$; otherwise.

**Objective:**

### 5.1.4.2 Augmenting path algorithm

**Based on the following observations:**

1. Ideally, every terminal is assigned to the nearest concentrator.

2. Terminals on concentrators that are full are moved only to make room for another terminal that would cause a higher overall cost if assigned to another concentrator.

3. An optimal partial solution with ***k+1*** terminals can be found by finding the least expensive way of adding the **$(k+1)^{th}$** terminal to the ***k*** terminal solution.

**Assignment problem:**

Given a cost matrix:
- One column per concentrator
- One row per terminal

Assume that:
- Weight of each terminal is 1 (i.e., each terminal consumes exactly one unit of concentrator capacity)
- A concentrator has a capacity of W terminals (e.g., number of ports)

A feasible solution exists iff $T \leq W * C$

**Algorithm:**

1. Initially, try to associate each terminal to its nearest concentrator

2. If successful in assigning all terminals without violating capacity constraints, then stop (i.e., an optimal solution is found)

3. Else,
    - **Repeat**
        1. Build a compressed auxiliary graph
        2. Find an optimal augmentation
    - **Until** all terminals are assigned

**Building a compressed auxiliary graph:**

**Example:**