Masters Thesis

Hierarchical Network Coding in Real Networks

Riwaj Sapkota

October 2012

Supervisor: Prof. Dr. Werner Henkel
Abstract

Erasures in communication channels play an important role in data transmission in real networks. It has also been well established that linear network coding is enough to achieve the upper bound in multicast problems. In this report, we investigate the problem of minimizing the overall erasure probability in a multicast scenario at multiple levels of hierarchy in a network. Firstly, we use ant colony optimization at the upper level as a technique to obtain an optimal set of nodes for network coding. Based on the result, we also analyze the degree distribution distortion of LT-coded symbols at each of the optimal coding nodes. After that, we focus our investigation at the next lower hierarchy level to identify the nodes inside the logical node where the logical summation of symbols (or simply network coding) takes place. For this, we construct minimum spanning trees on the logical nodes which takes into consideration the erasure probability at each edge. Finally, we discuss the effect of change in topology of the logical node and the edges associated with the nodes inside the logical node, and provide efficient algorithms to update the minimum spanning tree.
Declaration

I, Riwaj Sapkota, hereby declare that the thesis entitled 'Hierarchical Network Coding in Real Networks’ and the work presented in it was composed and originated by myself in cooperation with my supervisor in the School of Engineering and Science, Jacobs University, Germany. Contribution of other authors that I have adopted, whether in sense or literally from other published or non-published works, have been properly referenced as such.

__________________________
Riwaj Sapkota
Matriculation No: 20329472
School of Engineering and Science
Jacobs University, Bremen, Germany.
Date: October 22, 2012.
## Contents

Abstract i

Declaration iii

List of Figures vii

List of Algorithms ix

1 Overview 1

2 Introduction to Graph Theory and Algorithms 3
   2.1 Fundamentals of Graph Theory ........................................ 3
   2.2 Representation of Graphs ............................................. 7
   2.3 Graph Traversal ....................................................... 8
   2.4 Shortest-path Problem ............................................... 11
      2.4.1 Relaxation ....................................................... 13
      2.4.2 Dijkstra’s Algorithm ........................................... 13
   2.5 The Max-flow / Min-Cut Theorem .................................. 15

3 Introduction to Network Coding 21
   3.1 Encoding Vectors .................................................... 23
      3.1.1 Local Encoding Mapping for a Network Code on an Acyclic Network .. 24
      3.1.2 Global Encoding Mapping for a Network Code on an Acyclic Network .. 24
   3.2 Linear Network Code ................................................ 25
   3.3 Erasures in Network Coding ....................................... 26
List of Figures

2.1 A directed graph .................................................. 4
2.2 An undirected graph with a self-loop .......................... 5
2.3 Two representations of an undirected graph. a) An undirected graph with four
vertices and four edges. b) An adjacency-list representation of a graph ........ 7
2.4 Two representations of a directed graph. a) A directed graph with four vertices
and five edges. b) An adjacency-list representation of a graph ............ 7
2.5 (a) An undirected graph G, (b) the depth-first search of G ............ 10
2.6 (a) An undirected graph G, (b) the breadth-first search of G ........... 11
2.7 (a) A graph G with edge capacity, (b) the minimum cut of G, (c) the maximum
flow of G ............................................................. 19
3.1 Butterfly network with unit edge capacity .................... 21
3.2 Butterfly networks with different paths for multicasting two bits ........ 22
3.3 Network coding in the butterfly network ...................... 23
3.4 Binary erasure channel .......................................... 26
4.1 Most ants take shortest path after certain time ............... 32
4.2 Disjoint paths for each destination nodes in an artificial topology .... 34
5.1 Encoding process in LT codes ................................ 38
5.2 Network coding in a butterfly network ...................... 40
5.3 Robust soliton distribution with $k = 1000$, $c = 0.1$, and $\delta = 0.5$ .... 41
5.4 Encoding process in LT codes ................................ 45
5.5 Network decoding in a butterfly network .................... 46
6.1 An undirected graph with weight associated with each edge ........ 48
6.2 Prim’s implementation of Fig. 6.1 . . . . . . . . . . . . . . . . . . . . . . . . . 54
7.1 An erasure-prone network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.2 A network with erasure probability associated with each edge . . . . . . . . . . 58
7.3 (a) A minimum spanning tree, (b) introduction of a new node 4 into the minimum
    spanning tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
7.4 Updated minimum spanning tree after adding a new node . . . . . . . . . . . . . 63
7.5 a) A logical-node network  b) Minimum spanning tree of the network . . . . . . 66
7.6 (a) New logical-node network after deletion of Node 6  (b) Fragments of the old
    minimum spanning tree after deletion of Node 6 . . . . . . . . . . . . . . . . . . 68
7.7 (a) The graph $G''$ obtained after executing Step 2 of Algorithm 7.3  (b) Updated
    minimum spanning tree of the network . . . . . . . . . . . . . . . . . . . . . . 68
7.8 a) Minimum spanning tree on logical node  b) Updated minimum spanning tree
    of the network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.9 a) Disconnected components of minimum spanning tree  b) Updated minimum
    spanning after insertion of an edge with smallest erasure probability . . . . . . . . 71
7.10 a) Addition of edge in MST resulting in a cycle  b) Updated minimum spanning
    after deletion of the largest edge in the cycle . . . . . . . . . . . . . . . . . . . . 72
# List of Algorithms

2.1  **DFS** \((v)\) .................................................. 10  
2.2  **DFS-graph** \((v)\) ............................................. 10  
2.3  **RELAX** \((u, v, w)\) ........................................... 13  
2.4  **DIJKSTRA** \((G, w, s)\) ......................................... 14  
2.5  **Ford-Fulkerson Algorithm** ..................................... 17  
4.1  **FINDING DISJOINT PATHS** ..................................... 34  
6.1  **Generic - MST** \((G, w)\) ......................................... 49  
6.2  **Kruskal - MST** \((G, w)\) ......................................... 51  
6.3  **Prim - MST** \((G, w)\) ............................................. 53  
7.1  **INSERT** \((r)\) .................................................. 61  
7.2  **CONNECTED-COMPONENTS** \((G)\) ................................. 65  
7.3  **MST-NODE DELETION** \((G)\) ...................................... 66
Chapter 1

Overview

In real networks, the transmission of information to end users occurs through different physical transmission mediums ranging from twisted pair cables and coaxial cables to wireless channels. The amount of information that traverses to the end user across physical channels depends on different factors. One such factor is the erasure probability of the channel. The central idea behind this thesis is also related to minimizing the overall erasure probability of a network at the user’s end to ensure maximum transmission of information. In fact, we discuss the problem of minimizing erasure probability at different levels of hierarchy in a network. For this, throughout the report, we model edges in a network as binary erasure channels (BEC).

In 2001, Ahswede et al. [1] published a paper addressing the issue of information flow in networks. This paper marked the beginning of a new theory known as ‘Network Coding’. In the paper, it was stated that by allowing the intermediate nodes in a network to process information in contrast to the traditional ‘store and forward’ approach, a significant increase in the network throughput can be achieved. Li et al. [2] showed that linear network coding, the simplest class of network coding, is sufficient to achieve the the maximum flow of information (max-flow), the theoretical upper bound for network resource utilization, in a multicast scenario. The idea of network coding closely relates to the concepts of coding theory, information theory, and graph theory.

We start by introducing some basic concepts of network coding and graph theory. Chapter 2 discusses some elementary knowledge of graph theory and the algorithms related to it. We also discuss the problem of max-flow and the Ford-Fulkerson algorithm, which we later use to compute the multicast rate in networks. Chapter 3 discusses network coding and the concept of
erasures in network coding.

Although network coding at intermediate nodes of a network ensures maximum flow of information, in a large network, not all intermediate nodes need to participate in network coding operations to achieve the upper bound [3]. This means that, by carefully selecting an optimal set of intermediate nodes, one can still achieve the desired throughput in a network. In Chapter 4, we discuss ant colony optimization (ACO) to obtain a minimal set of intermediate nodes such that the overall erasure probability at the end nodes is minimized. We resort to using ACO as it is applicable in large-scale networks. Once we obtain the minimal set of coding nodes, in Chapter 5, we focus on the issue of degree distribution distortion of LT-coded symbols at each of the coding nodes.

In chapters 6 and 7, we discuss the problem of minimizing the erasure probability at lower levels of the hierarchy in a network. We structure networks according to the concentration of edges and designate the surrounding as logical node which is analogous to the one in asynchronous transfer mode (ATM). The actual summation or network coding of the information takes place at some node inside the logical node. The solution to the problem is looking into the logical node from an end-node perspective. After describing the background of minimum spanning trees in chapter 6, based on this concept, chapter 7 provides a solution using minimum spanning tree which takes into consideration all the inputs and minimizes the overall erasure probability at the node leaving the logical node.

Finally, in Chapter 7, the effect of a change in topology of the logical node is discussed. In addition, algorithms to effectively update the minimum spanning tree inside a logical node when the topology changes is also discussed.
Chapter 2

Introduction to Graph Theory and Algorithms

Since most of our investigation on network coding involves graphs and algorithms revolving around graph theory, we introduce some elementary concepts of graph theory and algorithms related to them in this chapter. We begin with some basic definitions in Section 2.1 followed by representation of graphs in Section 2.2. In Section 2.3, we introduce some fundamental concepts of graph traversal. We also explain in brief the depth-first search. After that, the problem of shortest path in graphs is introduced in Section 2.4. A general formulation of max-flow/min-cut problem and a solution to the problem is introduced in Section 2.5.

2.1 Fundamentals of Graph Theory

In mathematics, a graph is an abstract representation of a problem using a set of objects connected to each other by links. The study of graphs and their configuration is known as graph theory. Graph theory is used in different fields such as natural sciences, computer science, computer and electrical engineering, economics, and social sciences to simplify and optimize problems.

A graph may consist of a set of vertices and edges such that an unordered pair of distinct vertices of the graph represents an edge [4]. The vertices, often known as nodes, are connected to each other by edges. If we consider that a set \( V = \{v_1, v_2, v_3, \ldots, v_n\} \) represents a group of vertices and a set \( E = \{e_1, e_2, e_3, \ldots, e_m\} \) represents a group of edges, the graph can simply be represented as \( G = (V, E) \). In the following section, we will describe some
terminologies relevant to our later discussions.

**Definition 2.1** A directed graph or a digraph is a graph with ordered pairs of vertices and a set of directed edges. In order to represent a digraph, edges of a digraph are drawn as arrows.

Figure 2.1 represents a directed graph with special vertices – the source node S, and the destination nodes D1 and D2. All other vertices are called intermediate vertices. The edges of a directed graph are ordered pairs of distinct vertices.

![Figure 2.1: A directed graph](image)

**Definition 2.2** A graph is called an undirected graph if the edges connected with the vertices are without any implied direction. Undirected graphs are special cases of directed graphs in sense that they are symmetric digraphs.

Figure 2.2 represents an undirected graph with five vertices and six edges.

**Definition 2.3** The degree of a vertex is the number of edges incident on the vertex. The maximum and the minimum degree of a vertex is denoted by $\Delta G$ and $\delta G$, respectively.

In case of a directed graph, the degree of a vertex is further divided into two categories:

1. **In-degree**: It is defined as the number of edges coming into a vertex. A vertex $v$ with in-degree zero is known as source.

2. **Out-degree**: It is defined as the number of edges leaving a vertex. A vertex $v$ with out-degree zero is known as sink.
As an illustration, in Fig. 2.2, the degree of vertex $a$ and vertex $c$ is two while the degree of vertex $b$ and vertex $f$ is three.

**Definition 2.4** A leaf is a vertex with degree 1.

In Fig. 2.2, vertex $a$ and vertex $c$ can be considered as leaves in the network.

**Definition 2.5** A self-loop in a graph is defined as an edge which connects to a vertex at both ends. A self-loop adds two to the degree of a vertex in an undirected graph. In case of a directed graph, self-loop adds one to the in-degree and one to the out-degree.

For example, in Fig. 2.2, the degree of node $d$ is three because of the self-loop $e_4$.

**Definition 2.6** A walk in the graph $G = (V, E)$ is an finite sequence of alternating vertices and edges of $G$ of the form

$v_{i0}, e_{j1}, v_{i1}, e_{j2}, \ldots, e_{jk}, v_{ik}$

such that the sequence always starts at a vertex. The end vertices corresponding to $e_{jt} (t = 1, \ldots, k)$ are $v_{i_{t-1}}$ and $v_{it}$ where $k$ is the length of the walk. The initial vertex and the final vertex of the walk is given by $v_{i0}$ and $v_{ik}$, respectively. A walk is closed if $v_{i0} = v_{ik}$. Otherwise, it is open. A walk is called a trail if all edges in the sequence are traversed at most once [5].

**Definition 2.7** A trail is a path if any vertex is visited at most once except when the initial and the terminal vertices are the same. A closed path is called a circuit [5].

---

1 Also see Definition 2.5
**Definition 2.8** A graph $G$ is connected if all the vertices in the graph are connected to each other, i.e., there exists a path from one vertex to any other vertex in the graph. Likewise, a graph $G$ is said to be disconnected if there exists at least two vertices in the graph such that there is no path between them with the vertices as endpoints.

**Definition 2.9** Let $G = (V, E)$ be an undirected graph. Then $G$ is a tree if it satisfies the following statements $^2$:

1. Any two vertices in $G$ are connected by a unique simple path.
2. $G$ is connected, and $|E| = |V| - 1$.
3. $G$ is acyclic.
4. If any edge $e$ is added to $E$, the resulting graph contains a cycle.
5. The removal of any edge $e \in E$ from $G$ results in a disconnected graph.

However, if $G$ is disconnected, then it is called a *forest*.

**Definition 2.10** Consider a connected graph $G = (V, E)$. Then a spanning tree of $G$ is a subgraph of $G$ which is a tree and includes all the vertices of $G$.

**Theorem 2.1** Every connected graph contains at least one spanning tree.

**Proof 1** By definition, a spanning tree of a graph $G$ is a tree $T$ such that $V(T) = V(G)$. Let us assume that $G$ is a connected graph. Then $G$ has either no cycle or contains at least one cycle. If $G$ has no cycle, then $G$ is a spanning tree of itself. Otherwise, let us assume, without loss of generality, that it contains a single cycle, $C$. Then there exists an edge $e \in E$ such that $G - e$ results in a tree $T$ created by the removal of an edge from the cycle. Thus, $T$ is still connected and a spanning tree of $G$ $^7$.

**Theorem 2.2** A graph is a tree if and only if it is connected, and has exactly one spanning tree.

**Proof 2** By definition, $G$ is a tree if and only if it is connected and contains no cycles if and only if $\forall e \in E(G), G - e$ is disconnected. However, this is true if and only if there exists exactly one way in which the vertices can be connected, i.e., $G$ contains exactly one spanning tree.

**Definition 2.11** A forest is an undirected graph such that each connected component represents a tree.

$^2$Taken from [6]
2.2 Representation of Graphs

There are different kinds of data structures that can be used to represent a graph. The two most standard ways to represent the graphs are adjacency lists and adjacency matrix. The preference of one method over the other depends on whether the graphs are sparse or dense. It is preferred to use an adjacency list when the graph is in sparse form. An adjacency matrix is preferred when the graph is dense, i.e., when it is required to know if an edge exists between two vertices.

![Figure 2.3: Two representations of an undirected graph. a) An undirected graph with four vertices and four edges. b) An adjacency-list representation of a graph](image1)

![Figure 2.4: Two representations of a directed graph. a) A directed graph with four vertices and five edges. b) An adjacency-list representation of a graph](image2)

1. **Adjacency-list representation:** Consider a graph \( G = (V, E) \). Then, for each \( u \in V \), the adjacency-list \( Adj[u] \) contains all the vertices \( v \) such that there is an edge \( (u, v) \in E \), i.e., the adjacency-list contains all the adjacent vertices to \( u \) in \( G \) [6]. Such a list is typically maintained in an arbitrary order. Figures 2.3(b) and 2.4(b) illustrate the adjacency-list representation of an undirected and a directed graph, respectively.

2. **Adjacency-matrix representation:** Consider a graph \( G = (V, E) \) such that the elements of \( V \), i.e., \( V = \{v_1, v_2, v_3, ..., v_n\} \), are ordered. The adjacency matrix of \( G \) is \( n \times n \)-matrix

---

The figures for adjacency-list representation have been adapted from [6]
\( M \) with entries \( M_{ij} = 1 \) or \( M_{ij} = 0 \) depending on whether \( v_iv_j \in G \) or \( v_iv_j \notin G \) [8], respectively. For example, the undirected graph in Fig. 2.3 has the following adjacency matrix:

\[
M_{ud} = \begin{bmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0
\end{bmatrix}.
\]

Similarly, the adjacency matrix of Fig. 2.4 can be represented as:

\[
M_d = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}.
\]

### 2.3 Graph Traversal

Graph traversal is the problem of traversing through all the nodes and edges in a graph in a systematic way. It is considered as one of the most fundamental graph problem. Graph traversal is used in counting the number of vertices and edges, identifying disconnected components in a graph, finding paths between vertices in a graph, and validating the contents of edges and vertices— to name a few applications.

In graph traversal algorithms, each ‘visited’ vertex is marked and the unvisited vertices are tracked [9]. Depending on their state, the vertices can be classified into three states:

- **Undiscovered:** The vertices in their initial state.
- **Discovered:** The vertices encountered during graph traversal but with unmarked incident edges.
- **Completely-explored:** The vertices with marked incident edges, i.e., visited vertices.

A definite structure needs to be maintained while traversing through a graph. The first step in graph traversal is to explore an undiscovered vertex and evaluate each edge going out of the vertex. If one of the edges goes to another undiscovered vertex, we mark the current vertex as
discovered and repeat the procedure for the undiscovered vertex. If an edge goes to a completely-explored vertex, we ignore the vertex. This whole process is repeated until all the vertices are completely-explored. The vertices of a directed graph are considered only once when exploring the source edge [9]. However, the vertices of an undirected graph are considered twice, i.e., once at each end-point.

Depending on the containing data structures, we can classify the graph traversal techniques into two important classes:

1. **Breadth-first search**: Breadth-first search (BFS) is one of the simplest graph traversal algorithms based on queueing, i.e., storing the vertices in a first in, first out (FIFO) queue.

2. **Depth-first search**: Depth-first search (DFS) is a recursive graph traversal algorithm which stores the vertices in last in, first out (LIFO) stack. The algorithm continues to traverse the vertices along a path until all the vertices in the path are visited. After this, the algorithm backtracks to the vertex from where it started and then begins the search in another path, if available. The process continues until all the vertices in the graph are visited [9].

In the following section, in relevance to our work, we shall focus more on DFS and provide an implementation of the algorithm.

**Depth-First Search**

A depth-first search explores the undiscovered edges in the forward direction as long as possible. The algorithm begins with selecting any undiscovered vertex \( v \) and visiting it. Then any edge \((v, w)\) incident upon \( v \) is selected. If \( w \) has been visited earlier, then another edge incident on \( v \) is selected. If \( w \) has not been visited earlier, then \( w \) is visited and marked discovered, and a new search started at vertex \( w \). Once the search through all paths beginning at \( w \) is completed, the search returns to \( v \). The process of selecting undiscovered edges incident upon \( v \) continues until the list of all the edges is exhausted [10].

Consider an undirected graph \( G = (V, E) \) for a depth-first search. The edges in \( E \) are partitioned into two sets, \( T \) and \( B \). An edge \((v, w)\) is placed in set \( B \) if vertex \( w \) has been discovered previously when the algorithm is at vertex \( v \) considering edge \((v, w)\). Otherwise, edge \((v, w)\) is
placed at $B$. The algorithm for depth-first search is given below.

**Algorithm 2.1 DFS($v$)**

1: mark $v$ "discovered";
2: for each vertex $w \in Adj[v]$ do
3: if $w$ is marked "undiscovered" then
4: add $(v, w)$ to $T$;
5: DFS($w$)
6: end if
7: end for

With the help of Algorithm 2.1, we can formulate the depth-first search algorithm for an undirected graph.

**Algorithm 2.2 DFS-graph($v$)**

1: $T \leftarrow \emptyset$;
2: for all vertex $v \in V$ do
3: mark $v$ "undiscovered";
4: end for
5: while there exists a vertex $v \in V$ marked "undiscovered" do
6: DFS($v$)
7: end while

![Figure 2.5: (a) An undirected graph $G$, (b) the depth-first search of $G$](image-url)
The time complexity of the algorithm depends upon the Algorithm 2.1. The algorithm $\text{DFS}(v)$ is proportional to the number of edges incident to $v$ and is called only once for a given $v$. Line 2 and Line 5 require $O(n)$ steps if a list of vertices is made. Thus the total time complexity of the algorithm is $O(\text{MAX}(n, e))$ where $n$ is the number of vertices and $e$ is the number of edges in the graph [10].

![Figure 2.6](image)

Figure 2.6: (a) An undirected graph $G$, (b) the breadth-first search of $G$

Figure 2.5 is an example of depth-first search. We can see from (b) that the algorithm searches the depth of a graph and returns recursively only when all the unexplored edges in the graph are discovered. In contrast, Fig. 2.6 starts from a root node and traverses through all the neighboring nodes. As seen in Fig. 2.6(b), the search returns to each of the neighboring node and explores its neighboring nodes. The process continues until all the nodes are explored.

### 2.4 Shortest-path Problem

Consider a graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ mapping each edge $e \in E$ to real-valued weights [6]. The weight of the path is defined to be the sum of the weights of the constituent edges in the path, i.e., if path $p = (v_0, v_1, \ldots, v_k)$, then

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The shortest-path weight from vertex $u$ to vertex $v$ ($u, v \in E$) is given by

$$\delta(u, v) = \begin{cases} 
\min w(p) : u \xrightarrow{p} v, & \text{if there is a path from } u \text{ to } v, \\
\infty, & \text{otherwise} [6].
\end{cases}$$

(2.1)
The shortest path problem is to find the shortest-path weight of any path from \( u \) to \( v \) for each ordered pair of vertices \((u, v)\).

There are several variants of the shortest-paths problem. In [11], the author discusses a number of variations of this problem for different objective functions like bottleneck, algebraic sum, minimum deviation, etc. In summary, the following are the different types of problem variants.

1. **Single-source shortest-path problem**: Find shortest paths from a given source vertex \( s \in V \) to all the other vertices in the graph.

2. **Single-destination shortest-path problem**: Find shortest paths from each vertex \( v \in V \) to a given destination vertex \( t \).

3. **Single-pair shortest-path**: Find a shortest path from \( u \) to \( v \) for given vertices \( u \) and \( v \) [6].

4. **All-pairs shortest-path problem**: Find shortest paths for every pair of vertices \((u, v)\) in the graph.

However, in accordance to our interest, we focus only on the single-source shortest-path problem. We also assume that all edge weights in the graph \( G \) are positive.

Several algorithms have been proposed to solve the problem of single-source shortest paths. Dijkstra [12] proposed a greedy algorithm, asymptotically the fastest algorithm known so far, to solve the arbitrary directed graphs with unbounded non-negative weights with complexity \( O(|V|^2) \). Floyd and Warshall [13] [14] proposed a dynamic programming algorithm to find shortest paths between all pairs of vertices. However, irrespective of the nature of the algorithm, most shortest-path algorithms make use of the property that there exist shortest paths within a shortest path between two vertices. This property of the shortest paths is also known as the optimal substructure property of shortest paths [6].

**Lemma 2.1** *Subpaths of shortest paths are shortest paths*

*Given a weighted, directed graph \( G = (V, E) \) with weight function \( w : E \to \mathbb{R} \). Let \( p = (v_0, v_1, ..., v_k) \) be a shortest path from vertex \( v_1 \) to vertex \( v_k \) and, for any \( i \) and \( j \) such that \( 1 \leq i \leq j \leq k \), let \( p_{ij} = (v_i, v_{i+1}, ..., v_j) \) be the subpath of \( p \) from vertex \( v_i \) to vertex \( v_j \). Then, \( p_{ij} \) is a shortest path from \( v_i \) to \( v_j \).*

Lemma 2.1 is formally proved in [6].
**Definition 2.12** Given a weighted, directed graph \( G = (V, E) \) with weight function \( w : E \rightarrow \mathbb{R} \). If \( p \) is the shortest path from one vertex to another vertex, then \( p \) has no cycles.

**Proof** Let us consider that \( p = (v_0, v_1, \ldots, v_k) \) and \( c = (v_i, v_{i+1}, \ldots, v_j) \) be the shortest path and the positive-weight cycle in this path, respectively. This implies that \( v_i = v_j \) and \( w(c) > 0 \). Then, the path \( p' = (v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_k) \) has weight \( w(p') = w(p) - w(c) \) which is less than \( w(p) \). Hence, \( p \) cannot be a shortest path from \( v_0 \) to \( v_k \) [6].

**Definition 2.13** For a graph \( G = (V, E) \), a predecessor \( \pi[v] \) of a vertex \( v \) is defined as a vertex \( u \) connected to vertex \( v \) by an edge \( e \) such that \( e \) is an outgoing edge from \( u \) to \( v \). The predecessor \( \pi[v] \) in \( G \) is either another vertex or undefined.

### 2.4.1 Relaxation

Relaxation is a technique used in shortest-path algorithms to update the existing value of a short path to a new received minimal value during the process of iteration. For each vertex \( v \in V \), an attribute \( dist[v] \) called shortest-path estimate is maintained which gives the upper bound on the weight of a shortest path from source \( s \) to \( v \). The process of relaxation can be summarized as

<table>
<thead>
<tr>
<th>Algorithm 2.3 RELAX ( (u, v, w) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if ( dist[v] &gt; dist[u] + w(u, v) ) then</td>
</tr>
<tr>
<td>2: ( dist[v] \leftarrow dist[u] + w(u, v) )</td>
</tr>
<tr>
<td>3: ( \pi[v] \leftarrow u )</td>
</tr>
<tr>
<td>4: end if</td>
</tr>
</tbody>
</table>

The process of relaxation decreases the value of shortest-path estimate \( dist[v] \) and updates \( \pi[v] \). In case of shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once [6].

### 2.4.2 Dijkstra’s Algorithm

Dijkstra’s algorithm solves the single-source shortest path problem on a directed graph \( G = (V, E) \) with non-negative weighted edges. One can also use the algorithm to find the shortest path from a source vertex \( s \) to a vertex \( v \in V \) by stopping the algorithm once the vertex \( v \) has
been reached instead of running it for all the vertices in the graph, i.e., the single-pair shortest-path problem.

The algorithm works by constructing a set of vertices $S$ whose distances from the source are known [10]. At each iteration, the vertex $u \in V - S$ with the least shortest-path estimate is selected and added to $S$, and the values of the weights leaving the edge $u$ are relaxed. The algorithm is presented below [6]:

**Algorithm 2.4 DIJKSTRA** $(G, w, s)$

1: for each vertex $v \in V(G)$ do
2: \[ dist[v] \leftarrow \infty \]
3: \[ \pi[v] \leftarrow \text{undefined} \]
4: end for
5: $S \leftarrow \emptyset$
6: $Q \leftarrow V[G]$
7: while $Q \neq \emptyset$ do
8: \[ u \leftarrow \text{vertex in } Q \text{ with smallest } dist[ ] \]
9: \[ \text{if } dist[u] = \infty \text{ then} \]
10: \[ \text{break} \]
11: end if
12: \[ S \leftarrow S \cup \{u\} \]
13: for each neighbour $v$ of $u$ do
14: \[ \text{RELAX}(u, v, w) \]
15: end for
16: end while

The algorithm works as follows: After the execution of Line 4, $\pi[v]$ remains undefined for all $v \in V$, $dist[s] = 0$, and $dist[v] = \infty$ for $v \in V - \{s\}$. Line 6 initializes $Q$ to contain all the vertices of $G$. With each iteration through lines 7 – 16, a vertex $u$, which has the shortest-path estimate, is extracted and added to the set $S$ [6]. Lines 13 and 14 relax each edge $(u, v)$ and updates the estimate $dist[v]$ and the predecessor $\pi[v]$ provided there is the shortest path to $v$ through $u$. The algorithm terminates with $dist[u] = \delta(s, u)$ for all vertices $u \in V$ [6].

Dijkstra’s original algorithm runs in $O(|V|^2)$ [12]. We have implemented min-priority queue $Q$ by calling priority-queue operations, i.e., Line 6 taking $O(1)$ time, line 8 taking $O(V)$ and line 14 taking $O(1)$, in the algorithm. Each neighboring edge of vertices $v \in V$ is examined
in the lines 13 – 14 only once during the algorithm. The overall operation takes a total time of \( O(V^2 + E) = O(V^2) \). One can also achieve a lesser running time of \( O(V \log V + E) \) by implementing the min-priority queue with a Fibonacci heap [15].

### 2.5 The Max-flow / Min-Cut Theorem

The problem of maximum-flow (max-flow) is significant in many practical contexts and also as a sub-problem in the context of algorithms [16]. The max-flow problem concerns with maximizing the amount of information from source to sink without violating any capacity constraints [6]. Among all the max-flow theorems, the one relevant to our discussion is the max-flow min-cut theorem. In this section, we focus on the max-flow problem in communication networks. Later in the section, we shall introduce the Ford-Fulkerson method for solving the problem.

Let us consider a directed graph \( G = (V, E) \) which consists of a node set \( V(G) \) and an edge set \( E(G) \) where an ordered pair of nodes \( (i, j) \) with \( i, j \in V(G) \) consists an edge. We determine a variable to measure the information flowing through each edge. We refer to the variable as the flow of an edge or simply flow [16]. Mathematically, the flow of an edge \( (i, j) \) is a scalar, which we denote by \( f_{ij} \). We assume that a positive \( f_{ij} \) indicates that the flow moves along the direction of the edge in a graph, whereas a negative \( f_{ij} \) suggests that the flow moves in a direction opposite to that of the edge. We refer to the set of flows \( \{f_{ij} | (i, j) \in E\} \) as flow vector [16].

**Definition 2.14** The divergence of the node \( i \), also called as divergence vector \( y \), is the total flow departing from the node \( i \) minus the total flow arriving at it.

\[
y_i = \sum_{(j | (i, j) \in E(G) \}} f_{ij} - \sum_{(j | (j, i) \in E(G) \}} f_{ji} \quad \forall i \in V(G) \quad [16].
\]  

**Definition 2.15** If the amount of flow is constrained to lie between given lower and upper bounds, i.e.,

\[
b_{ij} \leq f_{ij} \leq c_{ij}, \quad \forall (i, j) \in E
\]

the flow is considered to be feasible.

**Definition 2.16** A path \( P \) in a network is said to be unblocked with respect to \( f \) if the flow along \( P \) is positive without violating the bound constraints [16].
**Definition 2.17** A cut in a graph $G = (V, E)$ is a partition of the node set $V$ into two nonempty subsets, a set $X$ and its complement $V(G) - X$. We use the notation

$$Q = [V, V(G) - X].$$  

(2.3)

Note that the partition is ordered in the sense that the cut $[X, V(G) - X]$ is distinct from the cut $[V(G) - X, X]$. For a cut $Q$, we use the notation

$$Q^+ = \{(i, j) \in E(G) | i \in X, j \notin X\},$$  

(2.4)

$$Q^- = \{(i, j) \in E(G) | i \notin X, j \in X\},$$  

(2.5)

and we say that $Q^+$ and $Q^-$ are the sets of forward and backward arcs of the cut, respectively [16].

**Definition 2.18** The flux across a non-empty cut $Q = [X, V(G) - X]$ is defined to be the total net flow coming out of $X$, i.e.,

$$F(Q) = \sum_{(i, j) \in Q^+} x_{ij} - \sum_{(i, j) \in Q^-} x_{ij}.$$  

(2.6)

**Definition 2.19** Given lower and upper bounds $b_{ij}$ and $c_{ij}$ for each edge $(i, j)$, the capacity of $Q = [X, V(G) - X]$ is

$$C(Q) = \sum_{(i, j) \in Q^+} c_{ij} - \sum_{(i, j) \in Q^-} b_{ij}.$$  

(2.7)

For any capacity-feasible flow vector $x$, the flux $F(Q)$ across $Q$ is smaller or equal to the cut capacity. When the flux is equal to the cut capacity, then $Q$ is called a saturated cut with respect to $x$.

**Definition 2.20** Max-flow problem: To compute a feasible flow between the source node $S$ and destination node $D$, given the lower and upper bounds such that the value of the flow $\sum_{(i, j) \in E(N)} f_{ij}$ is maximum.

Before discussing the solution of the max-flow problem, we discuss the max-flow min-cut theorem which relates the minimum cut of the graph with the maximum flow of the network.
Theorem 2.3 Max-Flow / Min-Cut Theorem:

1. Let \( x^* \) be a feasible flow between the source node \( S \) and destination node \( D \) such that the flow is maximum. Then the divergence out of \( S \) corresponding to \( x^* \) is equal to the minimum cut capacity over all cuts separating \( S \) from \( D \).

2. If all lower edge flow bounds are zero, the max-flow problem has an optimal solution, and the maximal divergence out of \( S \) is equal to the minimum cut capacity over all cuts separating \( S \) from \( D \).

The max-flow problem can be solved using a flow augmenting algorithm called Ford-Fulkerson algorithm. The algorithm computes the maximum flow in a network within the capacity constraints. Basically, it is assumed that given a feasible flow vector and a flow-augmenting path, it is possible to increase the flow along all the forward edges and decrease the flow along all the backward edges of the flow-augmenting path. We call the increment in flow change as \( \delta_{ij} \). The algorithm starts with a feasible flow vector. An iterative search is then made for a feasible path that is unblocked with respect to \( x \) from the source to the sink [16]. This path is called flow-augmenting path. The algorithm stops if no augmenting path is found. This unblocked path search method is used iteratively to either generate a new feasible flow vector with larger divergence out of \( S \) or finding a saturated cut separating \( S \) from \( D \). At each iteration, the increment in flow change is added to the feasible flow vector. Let us represent the resulting flow vector by \( \bar{x} \). The operation of replacing \( x \) by \( \bar{x} \) is known as flow augmentation. With this, the entire algorithm can be summarized as below:

**Algorithm 2.5 Ford-Fulkerson Algorithm**

1. Search for an augmenting path \( P \) from source node \( S \) to destination node \( D \)
2. if no augmenting path found then
3. current flow is maximum
4. terminate algorithm
5. else
6. Perform flow augmentation along path \( P \)
7. goto STEP 1
8. end if
The formal proof of the algorithm is given in [16].

Figure 2.7 illustrates the max-flow/min-cut theorem. The capacity of the minimum cut for (a) is 12 as shown in (b). This is equal to the maximum flow to the destination node $D$ as shown in (c).
Figure 2.7: (a) A graph $G$ with edge capacity, (b) the minimum cut of $G$, (c) the maximum flow of $G$
Chapter 3

Introduction to Network Coding

In July 2000, Ahlswede et al., inspired by computer network applications, introduced a new class of problems related to the network information flow [1]. It was argued that there was no reason to limit the function of a node to merely relaying the information from one input link to another and replicate information received from an input link. In contrast to the general intuition, the authors’ work revealed that by using nodes as an encoder instead of a simple ‘store-and-forward’ approach, a significant amount of bandwidth can be saved [1]. This was termed as network coding by the authors. Since then, the topic has interested quite a number of researchers and has been investigated in wireless communications, networking, information, and coding theory.

![Butterfly network with unit edge capacity](image)

Figure 3.1: Butterfly network with unit edge capacity

While there have been many schemes of applying network coding in a network, linear network coding is the simplest of all network coding scheme. In linear network coding, any block
of information is regarded as a vector over a certain base field. Nodes are allowed to apply linear transformations to a vector before passing it on to the next node [2]. The basic idea, like in all other schemes, is to achieve the maximum possible flow (max flow)\(^1\) from the source to all receiving nodes.

One example to illustrate how network coding works is demonstrated here. Figure 3.1 shows a butterfly network where the circles denote the nodes. The source node is denoted by S while the nodes where the information needs to be multicasted to are denoted by D1 and D2, respectively. The other 4 nodes are neither source nor sink nodes. The line connecting the two nodes is called an *edge or link* which is often considered to be a noiseless communication link for the transmission of data per unit time [17]. Let us assume, in this case, that the links have unit capacity.

![Butterfly network diagram](image)

**Figure 3.2: Butterfly networks with different paths for multicasting two bits**

Now, consider Fig. 3.2 where the bits to be multicasted to D1 and D2 are \(x\) and \(y\). From a traditional point of view, it would mean that every link in the graph has to carry either \(x\) or \(y\). The subsequent bit-receiving node will replicate the bit received and forward it until the bits reach the sink nodes. This means that the sink node D1 receives \(x\) via path S-N1-D1 and \(y\) via path S-N2-I1-I2-D1. Likewise, sink D2 receives \(x\) via path S-N1-I1-I2-D2 and \(y\) via path S-N2-D2. Either way, in both cases, the link between intermediate nodes, I1 and I2, being common to the transmission of both bits reduces the capacity by half. Thus, the total multipath rate achieved is 1.5 bits per unit time for both the sink nodes.

\(^1\text{Maximum amount of flow from a source node to a sink node}\)
However, if the intermediate node I1 is used for coding to generate $x \oplus y$ upon reception of $x$ and $y$ bits, then the link between nodes I1 and I2 can transmit $x \oplus y$ to the sink nodes D1 and D2. The node D1 receives $x$ from N1 and $x \oplus y$ from I2 from which $y$ can be retrieved. In a similar fashion, the other sink node D2 can retrieve $x$. This is shown in Fig. 3.3. In this way, the multicast rate of 2 bits per unit time is possible using the same network which is a significant improvement. Not only does the coding help to reduce both latency and energy consumption by minimal use of link, but also it maximizes the bit rate [17].

![Network coding in the butterfly network](image)

Figure 3.3: Network coding in the butterfly network

### 3.1 Encoding Vectors

In this section, the network discussed will be acyclic with a single source node. The symbols $\mathcal{I}(N)$ and $\mathcal{O}(N)$ represent the incoming and outgoing edges, respectively, at a node N. In order to account for the source node, $\mathcal{I}(S)$ denotes the set of imaginary edges terminating at the source node S without any originating node. The number of the imaginary edges is equal to $\omega$ in a graph.

We assume that the capacity of each edge is one per unit time. An element of a certain base field $\mathbb{F}$ represents one data unit [17]. For example, $\mathbb{F} = GF(2)$ represents the binary field. The message consists of $\omega$ data units and hence, is represented by an $\omega$-dimensional row vector such that $x \in \mathbb{F}^\omega$, i.e., $x = (x_1, x_2, \ldots, x_\omega)$ in $\mathbb{F}^\omega$. The source node S generates the message $x$ and subsequently, is sent out by transmitting a symbol $\tilde{f}_e(x) \in \mathbb{F}$ over every channel in the network. Any node 'N' is able to receive the information vector only if $\text{maxflow}(N) \geq \omega$, i.e., all nodes
fulfilling the max-flow bound can receive the information vector.

Each non-source node, with the help of its encoding function, maps the received symbols from incoming edges into a symbol for each outgoing edge. The encoding mechanism in the non-source node specifies the network coding to be implemented in the node. The encoding function of the cumulative code is formed by the combination of network coding operations at non-source (intermediate) node and is called "Global encoding function". The network code of the node itself is known as "Local encoding function".

### 3.1.1 Local Encoding Mapping for a Network Code on an Acyclic Network

**Definition 3.1** Let $\mathbb{F}$ be a finite field and $\omega$ a positive integer. An $\omega$-dimensional $\mathbb{F}$-valued network code on an acyclic communication network consists of a local encoding mapping

$$\tilde{k}_e : \mathbb{F}^{[\mathcal{I}(N)]} \rightarrow \mathbb{F}$$

for each node $N$ in the network and each edge $e \in \mathcal{O}(N)$ [17].

However, the following definition is not enough to describe the development along edges in a channel. This asks for an equivalent definition for a network code which takes into account the local encoding mechanism as well as the cumulative effect of the previous local network codes.

### 3.1.2 Global Encoding Mapping for a Network Code on an Acyclic Network

**Definition 3.2** Let $\mathbb{F}$ be a finite field and $\omega$ a positive integer. An $\omega$-dimensional $\mathbb{F}$-valued network code on an acyclic communication network consists of a local encoding mapping $\tilde{k}_e : \mathbb{F}^{[\mathcal{I}(N)]} \rightarrow \mathbb{F}$ and a global encoding mapping $\tilde{f}_e : \mathbb{F}^\omega \rightarrow \mathbb{F}$ for each edge $e$ in the network such that:

1. For every node $N$ and every edge $e \in \mathcal{O}(N)$, $\tilde{f}_e(x)$ is uniquely determined by the recursively derived $\tilde{f}_e(x)$, $d \in \mathcal{I}(N)$, and $\tilde{k}_e$ is the mapping via

$$\tilde{f}_d(x), d \in \mathcal{I}(N) \rightarrow \tilde{f}_e(x)$$
2. For the ω imaginary edges e, the mappings \( f_e \) are the projections from the space \( \mathbb{F}^\omega \) different coordinates, respectively [17].

### 3.2 Linear Network Code

Apart from transmission delay over the edges, another significant problem in network coding is the processing delay at the nodes. Hence, it makes more sense to use a code with low complexity to encode information in order to reduce processing delay. Linear network coding is one such alternative since it only uses linear transformations. Besides, by using linear network coding, the multicast capacity can reach the max-flow bound [2].

In linear network coding, any output vector, local or global, at a node is a linear combination of the incoming input vectors and the mapping vectors derived from coefficients of a finite field.

#### Encoding Kernels for Linear Network Codes

**Definition 3.3** Let \( \mathbb{F} \) be a finite field and \( \omega \) a positive integer. An \( \omega \)-dimensional \( \mathbb{F} \)-valued linear network code on an acyclic communication network consists of a scalar \( k_{d,e} \) called the local encoding kernel, for every adjacent pair \((d, e)\). The local encoding kernel at the node \( N \) means the \( |\mathcal{I}(N)| \times |\mathcal{O}(N)| \) matrix \( K_N = [k_{d,e}]_{d \in \mathcal{I}(N), e \in \mathcal{O}(N)} \) [17].

**Definition 3.4** Let \( \mathbb{F} \) be a finite field and \( \omega \) a positive integer. An \( \omega \)-dimensional \( \mathbb{F} \)-valued linear network code on an acyclic communication network consists of a scalar \( k_{d,e} \) called the local encoding kernel, for every adjacent pair \((d,e)\) in the network as well as the \( \omega \)-dimensional column vector \( f_e \) for every edge \( e \) such that

1. \( f_e = \sum_{d \in \mathcal{I}(N)} k_{d,e} \cdot f_d \) where \( e \in \mathcal{O}(N) \)

2. The vectors \( f_e \) for the \( \omega \) imaginary edges \( e \in \mathcal{I}(N) \) for the natural basis of the vector space \( \mathbb{F}^\omega \) [17].

The vector \( f_e \) is called the global encoding kernel for the edge \( e \).

As an illustration, if the source generates a message \( x \) in the form of an \( \omega \)-dimensional row vector. A node \( N \) will then receive \( x \cdot f_d \), \( d \in \mathcal{I}(N) \). This value is used to calculate the symbol
$x \cdot f_e$ which is sent onto each edge $e \in \mathcal{O}(N)$ [17].

\[ x \cdot f_e = x \cdot \sum_{d \in \mathcal{I}(N)} k_{d,e} f_d = \sum_{d \in \mathcal{I}(N)} k_{d,e} (x \cdot f_e). \]

### 3.3 Erasures in Network Coding

Erasure channels are one way of modeling communication channels such that the information is either transmitted without distortion or erased. Hence, the binary erasure channel (BEC) is a erasure channel in which single bits are transmitted and either received correctly or lost. The erasure of the data, usually indicated by ‘e’, is typically caused by buffer overflows at intermediate routers or excessive delays and packet loss or by shadowing in wireless links. The model was first proposed by Elias in 1955 as a toy example [18]. The BEC has been used to model data networks especially after the advent of the Internet.

![Binary erasure channel](image)

**Figure 3.4: Binary erasure channel**

Figure 3.3 illustrates the BEC model. The channel input $X$ at any time $t$ is binary, i.e., $X_t \in \{0, 1\}$. The channel output takes the values from the alphabet \{0, 1, e\}. If we assume that the erasure probability is given by $p_e$, then the capacity of the BEC($p_e$) can be expressed as [19]:

\[ C_{\text{BEC}}(p_e) = 1 - p_e . \]

The BEC is assumed to be memoryless meaning that erasures occur independently.
In our investigation, we use the erasure probabilities obtained from loss rates associated with edges in a real Internet network, i.e., PlanetLab Europe– a testbed for computer networking and distributed systems research\(^2\).

\(^2\)The erasure probabilities associated with each link is based on our experiment conducted in February 2012
Chapter 4

Ant Colony Optimization

As discussed in the previous chapter, network coding helps to achieve the maximum multicast rate in a network. This involves the use of intermediate nodes in a network to combine incoming data using algebraic operations, and pass it along the network. Li et al. [2] proved that linear network coding achieves the upper bound for a multicast network with a single source and multiple destinations. The upper bound, in this case, is given by the max-flow min-cut theorem.

However, in a large network, there is no guarantee that the selected intermediate nodes for network coding are the optimal ones. In fact, a large number of network coding operations in intermediate nodes could be futile, i.e., the upper bound of the multicast capacity can still be achieved even if we ignore some of the ‘redundant’ network coding in the intermediate nodes [3]. By carefully selecting optimal intermediate nodes for network coding, one can still achieve the desired multicast capacity within the upper bound.

The problem of finding an optimal set of intermediate nodes for network coding involves confirming the bound of the selected set of intermediate nodes and the edges associated with them. Most of the current research focuses on genetic algorithm to solve the problem. While genetic algorithms such as the one proposed in [20] aim at solving the problem by providing a solution set, such sets have a large solution space leading to a very inefficient convergence speed [3]. Moreover, most genetic algorithms do not take into consideration the edges associated with the intermediate nodes. This makes them practically useless in our case since we need to consider each edge and its erasure probability. In our case, the optimization problem involves finding an optimal set of intermediate nodes for coding such that it minimizes the erasure probability of the path as information is transmitted from source node to destination node. In [3], the authors
combine the Ford-Fulkerson algorithm and ant colony optimization to solve the problem. In this approach, firstly, all possible paths with non-overlapping edges are found. After this, the head nodes of overlapping edges in the disjoint paths are identified, which will be the set of intermediate nodes for coding. We use this method in our experiment.

Consider a connected, undirected network $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ mapping each edge $(u, v) \in E$ to real-valued weights. The weight function associated with each edge represents the success probability of the edge. Let us assume a single source node $S \in V$ and a set of destination nodes $D \subseteq V$ in the network such that the maximum multicast rate is $h$, i.e., every $D \subseteq V$ receives all the information sent out from $S$.

**Definition 4.1** Coding edge: The immediate edge which transmits the information after network coding operations in the intermediate node take place is known as coding edge.

**Definition 4.2** Coding node: The node in which the network coding operations take place is known as coding node.

**Definition 4.3** Disjoint paths: We define disjoint paths as the paths with non-overlapping edges in a network.

**Definition 4.4** Edge cost: It is the cost of transmitting information in terms of occupied bandwidth and edge length. In our case, instead of edge length, we assume the success probability associated with the edges as edge cost.

**Definition 4.5** Optimal path: Out of all possible paths between a source node $S$ and a destination node $v \in D$, we define an optimal path to be the path in which the sum of the logarithm of the success probabilities associated with each edge is maximum.

We aim at optimizing the edge cost, i.e., to maximize the total success probability of the paths when transmitting information from a single source to all sinks on the premise of achieving the maximum multicast rate $h$ [3]. We proceed with computing the maximum multicast rate $h$ for the network using Ford-Fulkerson algorithm. Once we know the maximum multicast rate for the network, we find the disjoint paths in the network. Given a network such that each edge has a unit capacity, if a single source $S \in V$ transmits information at rate of $h$ to a set of destination nodes $D \subseteq V$, then we can expect to find $h$ disjoint paths [17]. Using this theory, we expect to
find \( h \) disjoint paths for each destination node such that the total success probability for each disjoint path is maximum. We use ant colony optimization to find the disjoint paths.

### 4.1 Introduction to Ant Colony Optimization

Ant colony optimization (ACO) is a general-purpose heuristic algorithm which is used to find an optimal solution from a set of finite possible solutions. The algorithm is one of many others developed around ant colony algorithms, the first of which was proposed in 1992 by Marco Dorigo in his PhD thesis [21]. Ant colony optimization is a population-based approach which exploits the positive feedback loop to enforce an attractive choice. In this technique, the behavior of the ants, particularly the search activity, is mimicked to an large extent and modeled to solve problems.

The basic idea in ACO is derived from the behavior of ants which enables them to find a short route from their colony to the source of food and vice-versa despite having a relatively poor eye-sight. When ants search for food, they wander randomly in all directions. Upon finding the food, the ants return to their nests but in doing so, deposit pheromone on their trail. In the mean time, if any other ants from the colony find the path, they are more likely to follow the trail with pheromone than any other random path. This results in positive reinforcement as eventually, further deposition of pheromone in the selected path attracts more ants as they search for food.

However, the level of pheromone in the trail evaporates with time, thus reducing the ‘attractiveness’ of a path. The longer it takes an ant to return to its nest, the higher is the rate of evaporation of the pheromone. Thus, a shorter path becomes more attractive for other ants while a longer path having less concentration of the pheromone is followed by a decreasing number of ants.

In order to illustrate the idea, let us consider the following example. In Fig. 4.1, there are two routes, A and B, from the nest (C) to the food source (F). Initially, the ants randomly follow both paths towards the source of food. Since the route B is shorter than the route A, the ants following route B reach the source of food faster. The ants which reach the food source return back to the nest but in doing so, deposit pheromone along the path. The pheromone concentration, however, decreases with time due to evaporation. Once the ants return to their nest, a fresh set of ants begin their journey to the source of food. The fresh ants are attracted more towards the
path with more pheromone concentration, and hence more ants opt for path B. After a while, the pheromone concentration in the shorter path increases more than the longer path, because more ants deposit their pheromone along the path. Eventually, almost all ants choose the shortest path.

4.2 Finding Disjoint Path using ACO

The artificial ants we use in our network differ from the natural ants in the following aspects:

- The artificial ants used as optimization tool are not completely blind.
- The ants remember the set of nodes they visit.
- The artificial ants operate in a discrete time environment [21].

We consider a connected, undirected network \( G = (V, E) \). Let us denote the success probability of edge \((u, v)\) by \(w(u, v)\), which is \(1 - e(u, v)\), the erasure probability on the edge. We assume that the total number of nodes and ants in the network is \(n\) and \(m\), respectively. For each node \(u, v \in V\), let \(\tau_{uv}\) be the amount of pheromone deposited by ants in the edge connecting nodes.
Let us define \( \text{tabu}_k \), the dynamically growing vector which contains the set of all the nodes visited by each ant \( k \in m \). Let \( \eta_{uv} = \left( \frac{1}{w(u,v)} \right) \) be the visibility of the edge \( (u, v) \). Each ant in the network travels from node \( u \) to one of the nodes from a set of unvisited nodes according to a probability \( p_{uv} \). This probability is defined as:

\[
p_{uv} = \begin{cases} \frac{\tau^\alpha_{uv}(t)^\beta}{\sum_{s \in \text{allowed}_k} \tau^\alpha_{us}(t)^\beta \eta_{uv}^\beta}, & v \in \text{allowed}_k, \\ 0, & \text{otherwise} \end{cases}
\]  

(4.1)

where \( \alpha \) and \( \beta \) are two positive parameters which control the relative importance of trail versus visibility. The \( \text{allowed}_k \) is a set of unvisited nodes of ant \( m \) such that \( \text{allowed}_k = V(G) - \text{tabu}_k \). This ensures that ants do not go back to the previously visited nodes on their way to the destination nodes. When an ant reaches the destination node, it moves back to the source node but in doing so, it deposits pheromones in the links

\[
\tau_{uv}(t + 1) = \rho \cdot \tau_{uv}(t) + \sum_{k=1}^{m} \Delta \tau^k_{uv}
\]

where \( \rho \) is the evaporation rate of the pheromone, and

\[
\Delta \tau^k_{uv} = \begin{cases} Q \cdot L_k, & \text{if } (u, v) \in \text{path described by } \text{tabu}_k, \text{tabu}_k(\text{end}) \in \text{destination nodes}, \\ 0, & \text{otherwise} \end{cases}
\]  

(4.2)

where \( Q \) is a constant to represent the quantity of pheromone deposited by ants. We define \( L_k \) as

\[
L_k = \sum_{(u, v) \in \text{path of } k^{th} \text{ant}} \log w(u, v)
\]

Updating the pheromone level increases the pheromone trail on edges. As the number of iterations increase, edges of the path with low \( L_k \) value have the stronger pheromone trail.

We apply ant colony optimization to find an optimal path from source node to a destination node. Then, we block all the edges of the optimal path and continue to find another optimal path from the source to the destination node. We continue this process until having found all the \( h \) disjoint paths from source node to the particular destination node. Once all the disjoint paths for a destination node has been found, we select another destination node and repeat the procedure.

We summarize the whole algorithm as follows [3]:

\[
\]
Algorithm 4.1 FINDING DISJOINT PATHS

1: Compute the maximum multicast rate \( h \) for the given network using Ford-Fulkerson algorithm.

2: From the set of destination nodes \( D \), select a destination node which has not been selected, yet.

3: Find \( h \) disjoint paths from source to the destination node.

4: Check if all the destination nodes have been selected. If yes, go to Step 5, else go to Step 3

5: Put all the \( h \) disjoint paths into the set of paths.

Figure 4.2: Disjoint paths for each destination nodes in an artificial topology

We evaluate the performance of the algorithm by carrying out simulations on various network topologies. Figure 4.2 illustrates a network topology with each edge having no erasure probability. The maximum multicast rate for the network is 2. We consider Node 1 as the source node and nodes 16, 17, 18, and 19 as destination nodes. Table 4.1 summarizes the disjoint paths for each of the destination node illustrated in Fig. 4.2.
<table>
<thead>
<tr>
<th>Destination Node</th>
<th>First Disjoint Path</th>
<th>Second Disjoint Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 16</td>
<td>{1, 2, 6, 8, 16}</td>
<td>{1, 3, 4, 5, 6, 9, 12, 14, 16}</td>
</tr>
<tr>
<td>Node 17</td>
<td>{1, 2, 6, 8, 12, 14, 17}</td>
<td>{1, 3, 4, 5, 6, 9, 17}</td>
</tr>
<tr>
<td>Node 18</td>
<td>{1, 2, 4, 5, 7, 10, 18}</td>
<td>{1, 3, 7, 11, 13, 15, 18}</td>
</tr>
<tr>
<td>Node 19</td>
<td>{1, 2, 4, 5, 7, 10, 13, 15, 19}</td>
<td>{1, 3, 7, 11, 19}</td>
</tr>
</tbody>
</table>

By carefully analyzing the set of disjoint paths obtained by ACO, it can be observed that nodes 1, 4, 12, and 13 are the head nodes of overlapping edges in the paths, i.e., the coding nodes. However, we exclude Node 1 from the coding node set as we do not include source nodes for network coding operations [3].

### 4.3 Conclusion

In our experiments, we resort to using ant colony optimization for finding an optimal set of coding nodes in a network with erasures. The optimization takes into consideration the erasure probability of each edge. However, during the implementation of the algorithm, to ensure that paths found earlier do not block the essential edges for other paths, backtracking should be used. In the next chapter, we shall discuss the distortion of degree distribution of LT-encoded symbols at the coding nodes obtained by applying ACO in a network.
Chapter 5

Network Coding with LT Codes

In this chapter, we investigate the problem of applying LT-encoding at the source node to provide reliable data distribution at end nodes. We assume that the source node passes LT-coded symbols to the network. However, due to the erasure in channels, there is a distortion of the degree distribution in received LT-coded symbols at each coding node\(^1\).

We start by reviewing some important preliminaries of LT codes in Section 5.1. In Section 5.2, we discuss the effect of erasure channels regarding the distortion of the degree distribution at each coding node and take first steps for computing degree distributions of LT-coded symbols at each edge in a network.

5.1 LT Codes

LT codes are a class of erasure correcting codes that can potentially generate a limitless sequence of encoded symbols from a finite set of input symbols such that the receiver can recover the original input symbols by using any subset of encoding symbols which is equal to or slightly larger than the number of original input symbols [22]. LT codes have a special property that the transmitter does not require any acknowledgment from the receiver which makes it extremely desirable in multicast applications. This property significantly reduces the overhead delay incurred due to processing of acknowledgments from multiple receivers [23]. LT codes are also known to have encoding and decoding processes which are simple and have low time complexity. Such rateless codes have, of course, the drawback of flooding a network with possibly unnecessary

\(^1\)The identification of network coding nodes is discussed in the previous chapter.
parity symbols, which would counteract the goal of Ahlswede’s network coding [1], namely to reduce network load.

### 5.1.1 Encoding

Let us assume that there is a number of $k$ information symbols. The following steps are essential to generate an LT-coded symbol.

- Determine degree $d$ of encoding symbol chosen randomly from a node degree distribution.
- Choose $d$ distinct input symbols uniformly at random as the neighbors of the encoding symbol.
- XOR the chosen $d$ neighbors which will result in an encoding symbol.

The encoding process corresponds to a bipartite graph, such as Fig. 5.1, which connects the code symbols with the information symbols.

![Figure 5.1: Encoding process in LT codes](image)

*Figure 5.1: Encoding process in LT codes*
5.1.2 Decoding

Once the encoding is completed, the encoded symbols are transmitted into the network. We assume each edge of the network as a BEC with success probability $p_e$, i.e., if an edge in the network has a success probability of 0.6, then on average, for every ten symbols sent through the edge, the receiving node of the edge receives six of those symbols while the remaining four are lost. Apart from the code symbols, the decoder also needs to know the neighbors of each encoding symbol. This information can be transferred in several ways. One way to communicate this information between the transmitter and the receiver is by making them share the same random-number generator seed. Once the decoder gathers the necessary information, the decoding procedure, also called as LT process starts:

1. **Release**: All code symbols connected to only one input symbol are released.

2. **Cover**: The released input symbols are sent a set of unprocessed input symbols gathered through iterations. Let us call the set as 'ripple'. The size of the ripple is the number of elements in the set.

3. **Process**: From the ripple, one information symbol is selected to be processed. All edges connecting the information symbol, present as neighbors to the rest of the encoding symbols are removed. The processed information symbol is released from the ripple [23].

The above described procedure is repeated until all the information symbols are decoded. If not all the information symbols can be decoded, the procedure is termed as *decoding failure*.

5.2 Degree Distribution in LT codes

The degree distribution plays a major role in LT codes for determining the successful decoding of information symbols. The selection of the degree distribution of encoding symbols is a tricky taste. On the one hand, the release rate of encoding symbols should be kept low enough so as not to waste encoding symbols. On the other hand, the release rate should not be so low that the ripple in the LT process dies out [23].

In our investigations, we use the robust soliton distribution which strikes a balance between
the above mentioned criteria. The ideal soliton distribution maintains a constant expected ripple size and stabilizes the decoding procedure when the ripple is of large expected size.

**Definition 5.1** Let $R$ denote the expected ripple size and $\delta$ denote the allowable failure probability. The robust soliton distribution (RSD) $\mu(i)$ is given by the two distributions $\rho(i)$, the ideal soliton distribution, and $\tau(i)$:

$$
\mu(i) = \frac{\rho(i) + \tau(i)}{\beta}, \quad 1 \leq i \leq k,
$$

where

$$
\rho(i) = \begin{cases} 
1/k, & i = 1, \\
1/(i(i - 1)), & 2 \leq i \leq k,
\end{cases}
$$

$$
\tau(i) = \begin{cases} 
R/(i \cdot k), & 1 \leq i \leq (\text{round}(k/R)) - 1, \\
R \ln(R/\delta)/k, & i = \text{round}(k/R), \\
0, & \text{otherwise},
\end{cases}
$$

$$
R = c \cdot \sqrt{k} \cdot \ln(k/\delta),
$$

$$
\beta = \sum_{i=0}^{k} (\rho(i) + \tau(i))
$$

The term $\beta$ is a normalization factor, while parameter $c > 0$ ensures that the probability of successful decoding is greater than $1 - \delta$. In the robust soliton distribution, we assume that the number of code symbols is given by a number $K = k + O(\sqrt{k} \cdot \ln^2(k/\delta))$ [22].

![Network coding in a butterfly network](image)

Figure 5.2: Network coding in a butterfly network
Figure 5.3 illustrates the robust soliton distribution with parameters $k = 1000$, $c = 0.1$, and $\delta = 0.5$. As we can observe, the figure has two peaks at 2 and 42.

Now, consider the network in Fig. 5.2. Let us assume that source node $S$ would like to multicast LT-coded outputs $x$ and $y$ to the destination nodes $D1$ and $D2$. The intermediate node $I1$ receives $x$ and $y$ and the process of network coding takes place. The node $D1$ receives $x$ and $x + y$, and obtains $y$ by deducting $x$ from $x + y$. This process is called ”network decoding”. After this, the encoded symbols are LT-decoded to obtain the original information.

However, in real networks, the channels have certain erasure probability, i.e., the network coding and network decoding processes are influenced by the erasure probability of the channels, which means that the degree distribution of the encoded symbols can be distorted before reaching the coding nodes. Furthermore, the optimal coding node can act as either network-coding node or network-decoding node depending on the combination of distribution it receives. In the next section, we discuss the effect of erasures in channels and sum up the conditions in which a coding node can act as a network-coding node or a network-decoding node.

![Figure 5.3: Robust soliton distribution with $k = 1000$, $c = 0.1$, and $\delta = 0.5$](image)
5.3 Degree Distribution Distortion in Binary Erasure Channels

Consider the network in Fig. 5.4. As illustrated there, each edge is associated with a certain success probability. The destination node in the network are nodes 16, 17, 18, and 19. Using the Ford-Fulkerson algorithm, we find that the multicast rate for the network is two, i.e., there are two disjoint paths from the source node to each of the destination nodes. Table 4.1 summarizes the disjoint paths for each of the destination node.

The disjoint paths, although different for each destination node, may share some edges. The success probability of a disjoint path can be computed from the success probabilities of the edges constituting the path from the source to a destination node. In a similar way, one can also compute the success probability of a section of a disjoint path from the source to a coding node using success probabilities of the edges constituting the path from source to the coding node.

Let \( p_{i,j,k} \) represent the success probability of the \( k \)th edge constituting the \( j \)th path of the \( i \)th destination node. The overall success probability of information symbols from the \( j \)th path constituting the \( i \)th destination node, denoted by \( P_{e,i,j} \), is,

\[
P_{e,i,j} = \prod_{k=1}^{\left| E(i,j) \right|} p_{i,j,k},
\]

where \( \left| E(i,j) \right| \) denotes the number of edges in the \( j \)th path of the \( i \)th destination node [24].

Before we discuss degree distribution distortion in binary erasure channels due to erasure, let us introduce a helpful property of random variables.

**Lemma 5.1** The degree distribution of the sum of two independent random variables is the convolution of each of their distribution.

**Proof 3** Let \( X \) and \( Y \) are two independent random variables with distribution function \( \Phi_1(i) \) and \( \Phi_2(i) \), respectively. Let us assume that \( Z \) be the result of addition of the random variables and \( \Psi(i) \) be the degree distribution of the resultant. Suppose that \( Z \) and \( X \) take on values \( z \) and \( k \), respectively where \( z \) and \( k \) are arbitrary integers. Then \( Z = z \) if and only if \( Y = z - k \). In this case, the probability that \( Z = z \) is the union of two events, i.e., probability that \( X = k \) and...
probability that $Y = z - k$ where $k$ runs over the integers. Thus, we have

$$P(Z = z) = P(X = k) \text{ and } P(Y = z - k),$$

$$P(Z = z) = \sum_k P(X = k) \cdot P(Y = z - k).$$

With this, we can safely say that the distribution function $\Psi(i)$ is a convolution of $\Phi_1(i)$ and $\Phi_2(i)$, i.e., $\Psi(i) = \Phi_1(i) \ast \Phi_2(i)$ which is given by

$$\Psi(j) = \sum_k \Phi_1(k) \cdot \Phi_2(j - k),$$

where $j = \ldots -2, -1, 0, 1, 2, \ldots$ [25].

The overall degree distribution of a symbol resulting from a network-coding procedure depends on the success probabilities of the paths traversed by the symbols before the network coding. This process is a union of several mutually exclusive events such as when path traversed by one of the symbol is erased or when paths traversed by all symbols are erased / not erased. In order to illustrate this, let us consider the network in Fig. 5.4. With reference to the network, let us assume that Node 1 likes to multicast two symbols $x$ and $y$ to destination nodes. Let $\Phi_1$ and $\Phi_2$ be the degree distribution of $x$ and $y$, respectively. The network coding nodes which consists of the coding nodes $\{4, 12, 13\}$ and the destination nodes, encode or decode the symbols depending on the information they receive in order to achieve the multicast rate of 2 bits per unit time. We take into consideration the success probability of each edge the symbols traverse while computing the degree distribution at coding nodes. We also assume that each edge represents the capacity of one bit per unit time.

Now, the first coding node, Node 4, receives both symbols, $x$ and $y$, only when the path traversed by both the symbols are erasure-free. In this situation, Node 4 does the simplest form of coding by XOR addition of $x$ to $y^2$. By Lemma 5.1, this is equal to the convolution of each of their distribution. However, due to the erasures associated with each edge, the degree distribution is distorted by an factor equal to the product of success probabilities of the paths traversed by each symbol. By (5.3), the success probability of the path traversed by symbol $x$ from the source node to Node 4 is $e_1 \cdot e_2$ or simply $e_1e_2$. Likewise, the success probability of the path traversed by symbol $y$ from the source node to Node 4 is $e_3e_4$. Therefore, the contribution of this event to the overall degree distribution of the coded symbol is $e_1e_2e_3e_4(\Phi_1 \ast \Phi_2)$. 

\footnote{We approximate XOR addition by simple addition}
However, Node 4 receives only one symbol if the path traversed by the other symbol is erased. This event comprises of two potentially inclusive events, i.e., the erasure of one and the non-erasure of other symbol. Now, if the symbol $x$ is erased, the path traversed by which consists of edges $\{1, 2, 4\}$, then the erasure probability of the path is the complement of its success probability, i.e., $(1 - e_1e_2)$. Thus, the contribution of this event to the overall degree distribution of the coded symbol is $(1 - e_1e_2) e_3e_4\Phi_2$. Similarly, in the event of only $y$ being erased, the degree distribution thus obtained is $(1 - e_3e_4) e_1e_2\Phi_1$. In the event that both symbols are erased, Node 4 does not receive any symbol, and hence the event does not contribute anything to the overall degree distribution.

Thus, the overall degree distribution at Node 4 is obtained by union of all the above-mentioned events. We summarize the process in Table 5.1.

**Table 5.1: Degree distribution at node 4**

<table>
<thead>
<tr>
<th>Erasure-prone paths</th>
<th>Degree distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>When path through ${1, 2, 4}$ is erased</td>
<td>$(1 - e_1e_2) e_3e_4\Phi_2$</td>
</tr>
<tr>
<td>When path through ${1, 3, 4}$ is erased</td>
<td>$(1 - e_3e_4) e_1e_2\Phi_1$</td>
</tr>
<tr>
<td>When no paths are erased</td>
<td>$e_1e_2e_3e_4(\Phi_1 \ast \Phi_2)$</td>
</tr>
<tr>
<td>When both paths are erased</td>
<td>0</td>
</tr>
<tr>
<td>Overall degree distribution</td>
<td>$(1 - e_1e_2) e_3e_4\Phi_2 + (1 - e_3e_4) e_1e_2\Phi_1 + e_1e_2e_3e_4(\Phi_1 \ast \Phi_2)$</td>
</tr>
</tbody>
</table>

At each coding node, the probability of obtaining a certain degree distribution depends on the erasure of paths converging to the coding node. In fact, the overall degree distribution of the symbols after network coding depends on the success probability / erasure probability of the edges that constitute the disjoint paths. The degree distribution of the LT-coded symbols can be distorted due to erasures at any of the edges traversed by the symbols to reach the destination.
However, if a coding node receives an already network-coded symbol from the an edge, i.e., one of the edges converging on a coding node transmits a symbol which is an XOR addition of both LT-coded symbols, $x + y$, by virtue of passing through another coding node along the path, then the coding node needs to network-decode the symbols in order to separate $x$ and $y$.

In order to illustrate this, let us consider the network in Fig. 5.4 and assume that edges $e_1, e_2, e_3,$ and $e_4$ are erasure-free, i.e., the success probability of the edges is 1. Assume the product of success probabilities $e_1, e_6, e_{10},$ and $e_{13}$ is $A$. Similarly, the product of success probabilities $e_5, e_8,$ and $e_{11}$ is $B$. Now, as shown in Fig. 5.5, node 12, which is a coding node, receives $x$ and $x + y$. At this instance, the degree distributions of $x$ and $x + y$ are $A\Phi_1$ and $B(\Phi_1 \ast \Phi_2)$, respectively. Upon this, the node 12 first deduces $x$ from $x + y$ to obtain $y$, i.e., the process of network decoding takes place in Node 12. Subsequently, the degree distribution of the forwarded symbol is then $AB\Phi_2$ as shown in Fig. 5.5.
5.4 Conclusion

With a more detailed study on the effect of erasures in network coding, one can apply the above stated concepts to compute the distorted degree distribution of symbols in the context of generalized networks. With the results, the degraded receiver performance due to distortion of the degree distribution in received LT-coded symbols can be compared with the one with no distortion of the degree distribution in LT-coded symbols.

Figure 5.5: Network decoding in a butterfly network
Chapter 6

Introduction to Minimum Spanning Tree

Consider a connected, undirected graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$ mapping each edge $(u, v) \in E$ to real-valued weights. Let us consider an acyclic subset $T = (V, E_T)$ such that $T \subseteq E$ connecting all the vertices in $G$ and whose total weight is

$$w(T) = \sum_{(u, v) \in T} w(u, v).$$

The problem of determining the tree $T$ such that it is minimized is known as minimum-weight spanning tree or simply minimum spanning tree (MST) [6]. When $e \in (E - E_T)$ is the largest edge in the cycle created by adding $e$ to $T$, then such a condition is the necessary and sufficient for $T$ to be MST [26].

A MST minimizes the sum of the edges over all possible spanning trees. However, unless the weights in each edge are distinct, a graph can have more than one MST. For example, if a graph $G = (V, E)$ has unit weight in all edges, then every spanning tree of the graph is a MST with weight $|V| - 1$. Only distinct weights on each edge guarantees that the MST of a graph is unique.
6.1 Basic Properties of Minimum Spanning Trees

Let us consider a graph $G$ and any spanning tree $T$ of $G$. We define the following sets for $G$ in the context of minimum spanning trees:

- For any $e \not\in T$, let us denote $c_T(e)$ as the set of edges in the unique cycle formed in $T \cup e$.

- For any $e \in T$, its removal creates the node set components $U, V \backslash U$, we denote the cut $(U, V \backslash U)$ in $G$ by $\text{cut}_T(e)$.

**Theorem 6.1** Cycle property The maximum-cost edge in any cycle cannot be in the minimum spanning tree.

**Proof 4** Consider $e$ is the maximum-cost edge in any cycle $C$. Let us assume that $T$ is a MST and $e \in T$. Let us delete $e$ from $T$. Let $U$ and $V \backslash U$ be the two connected components in $T \setminus \{e\}$. Let $x \in C$ be another edge such that $x \neq e$. However, since $e$ is the maximum-cost edge, we can safely say that $w(e) > w(x)$. Thus, the tree $T' = T \cup x \setminus \{e\}$ is the spanning tree such that $w(T') < w(T)$. This contradicts the minimality of $T$.

**Theorem 6.2** Cut property The minimum-cost edge in any cut must be in the minimal spanning tree.

**Proof 5** Let $e = (u, v)$ be the minimum-cost edge a cut $C$, i.e., one end point in $U$ and another in $U, V \backslash U$. Assume $T$ is a MST and $e \not\in T$. Let us create $T' = T \cup \{e\}$ creating a cycle $C$ containing the edge $e$ in $T'$. The cycle $C$ must have another edge $x$ with exactly one end-point
in $U$. However, from our assumption, $w(e) < w(x)$. Then the spanning tree $T' = T \cup e \setminus \{x\}$ is a spanning tree with $w(T') < w(T)$. This contradicts the minimality of $T$.

The two most used algorithms for solving the minimum-spanning-tree problem for undirected networks are Kruskal’s [27] and Prim’s algorithms [28]. Both algorithms are greedy algorithms, i.e., they make optimal local choices at each step. Generally, such a strategy does not guarantee to find a globally optimal solution to the problem [6]. However, in case of finding a minimum spanning tree, the algorithms we shall describe result in finding a spanning tree with minimum weight. In the following section, we introduce a generic greedy strategy which grows a spanning tree one edge at a time.

### 6.2 Growing Minimum Spanning Trees

Consider a connected, undirected graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$ mapping each edge $(u, v) \in E$ to real-valued weights. The algorithm works by adding edges to a set $A$ such that prior to each iteration, $A$ is a subset of a minimum spanning tree. We call this condition as loop invariant. At each step of the algorithm, an edge $(u, v)$ is added to $A$ only if $A \cup \{(u, v)\}$ is also a subset of some minimum spanning tree. Let us define such edges which can be added to $A$ as safe edges. With this, we present the generic algorithm for growing a minimum spanning tree.

**Algorithm 6.1 Generic - MST($G, w$)**

1: $A \leftarrow \emptyset$

2: while $A$ is not a spanning tree do

3: find an edge $(u, v)$ that is safe for $A$

4: $A \leftarrow A \cup \{(u, v)\}$

5: end while

6: return $A$

In the algorithm 6.1 [6], lines 2 – 4, add only the safe edges to $A$ and hence fulfill the loop invariant. Once all the edges are added to $A$, the set is returned in Line 6. The returned set is a minimum spanning tree.

One can expand this generic algorithm with certain variations to obtain the Kruskal and Prim

---

1Taken from [6]
6.3 Kruskal’s Algorithm

Kruskal’s algorithm is based on the generic minimum-spanning-tree algorithm. The minimum spanning tree is built in a forest. At each step, it adds an edge of least possible weight to the forest. Initially, each vertex is a part of its own tree in a forest. The algorithm searches for an edge \((u, v)\) that connects two different trees and adds the edge to the set of edges of the minimum spanning tree. The corresponding two trees connected by the edge \((u, v)\) are merged into a single tree. However, the edge \((u, v)\) is discarded if it connects two vertices in the same tree. The algorithm uses disjoint-set data structure to determine the connection of vertices in disjoint trees.

Before we formally present the algorithm, we define few disjoint-set data structures.

- **MAKE-SET\((v)\)**: Creates a new set whose only member is \(v\). For this operation, \(v\) must not already be in some other set.
- **FIND-SET\((v)\)**: Returns the pointer to the set containing \(v\).
- **UNION\((u, v)\)**: Unites the dynamic sets that contain \(u\) and \(v\) into a new set that is the union of these two sets.

These algorithms work only for undirected graphs. One needs to use Edmond’s algorithm to find a minimum spanning tree for a directed graph. 

\(^2\)Adapted from [6]
Algorithm 6.2 Kruskal - MST\((G, w)\)

1: \(MST \leftarrow \emptyset\)

2: for each vertex \(v \in V[G]\) do

3: \(\text{MAKE-SET}(v)\)

4: end for

5: Sort all edges of \(E\) into nondecreasing order by weight \(w\)

6: for each edge \((u, v) \in E\) taken from the sorted list do

7: if \(\text{MAKE-SET}(u) \neq \text{MAKE-SET}(v)\) then

8: \(MST \leftarrow MST \cup \{(u, v)\}\)

9: \(\text{UNION-SET}(u, v)\)

10: end if

11: end for

12: return \(MST\)

The algorithm creates a forest such that each vertex in the graph is treated as a separate tree. Then, all the edges in the graph are sorted in a set by weight in nondecreasing order. The algorithm then picks each edge from the set, and if that edge connects two different trees, it adds the edge to the forest, thereby, combining two trees into a single tree. Otherwise, the edge is discarded. This process is continued until the forest is not a spanning and all the edges in the set have been visited.

In Algorithm 6.2, lines 1 – 3 initialize the set MST. In Line 5, the edges in \(E\) are sorted by weight into nondecreasing order. Lines 6 – 9 checks if the edge \((u, v)\) belongs to the same tree or not. If they do not, the edge is added to MST in Line 8, and the vertices from the two different trees are merged in the same set in Line 9. The initialization process in line 1 takes a running time of \(O(1)\). Line 5 has a running time of \(O(E \log E)\). Lines 6 – 9 take \(O(E)\) running time for operations on the disjoint-set forest. When combined with the \(|V|\) MAKE-SET operation in Line 3, these take a total of \(O((V + E) \log E)\).\(^4\) Now, in a connected graph, \(|E| \geq |V| - 1\), the total running time is \(O(E \log E) + O(E \log E)\) which is equal to \(O(E \log E)\) [6].

\(^4\)The total running time is \(O((V + E)\alpha(V))\) where \(\alpha\) is a slowly decreasing function. \(\alpha(|V|) = O(\log V) = O(\log E)\)
6.4 Prim’s Algorithm

Prim’s algorithm is another special case of the generic minimum-spanning-tree algorithm. The algorithm chooses an arbitrary ‘root’ vertex $u$ and the spanning tree grows until all the vertices of the graph are added to the tree. At each step, only those edges are added to the tree which contribute the minimum amount to the weight of the spanning tree.

Like Kruskal’s algorithm, Prim’s algorithm is a greedy algorithm. The decision to add a new edge to the minimum spanning tree is made at each iterative stage. Thus, the graph under construction is a connected subgraph of the minimum spanning tree and spans a subset of vertices $V' \subset V$. A label $L[v]$ is maintained, for each vertex $v$, to record the edge of least weight from a vertex in $V'$ to a vertex $v \in (V \setminus V')$, i.e., each $L[v]$ is initialized to weight $w(u, v)$ of the edge $(u, v)$ given $(u, v) \in E$. Else, the weight is maintained as either infinity, i.e., $w(u, v) = \infty$, if $u$ and $v$ are distinct or zero, i.e., $w(u, v) = 0$ if they are not distinct. Finally, the algorithm updates the label $L[v]$ whenever a new vertex $x$ is added to $V'$. With this, we formally present the algorithm [29].
Algorithm 6.3 Prim - MST\((G, w)\)

1: \(V' \leftarrow \{u\}\)
2: \(L[u] \leftarrow 0\)
3: for all \(v \in (V \setminus V')\) do
4: \(\text{if edge } (u, v) \text{ exists then}\)
5: \(\text{set } L[v] \leftarrow w(u, v)\)
6: \(\text{else}\)
7: \(\text{set } L[v] \leftarrow \infty\)
8: \(\text{end if}\)
9: \(\text{end for}\)
10: while \(V' \neq V\) do
11: find a vertex \(x\) such that \(L[x] = \min\{L[v] | v \in (V \setminus V')\}\)
12: \(V' \leftarrow V' \cup \{x\}\)
13: for all \(v \in (V \setminus V')\) do
14: \(L[v] = \min\{L[v], w(x, v)\}\)
15: \(\text{end for}\)
16: \(\text{end while}\)

The while loop in the Algorithm 6.3 is executed \(n - 1\) times. Within each execution of the loop, lines 11 – 14 are executed within \(O(n)\) steps. Thus, the overall complexity of the algorithm is \(O(n^2)\). A more efficient implementation of the algorithm would involve disjoint data-set data structure and use a Fibonacci heap [6].

Figure 6.2 illustrates the execution of Prim’s algorithm at each iteration for the graph in Fig. 6.1. The nodes selected by the algorithm are represented by a red color. At each iteration, a new node is selected greedily based on the weight associated with the edge connecting it with its predecessor, as shown in figure. The iteration terminates when all the nodes have been selected.
Figure 6.2: Prim’s implementation of Fig. 6.1
Chapter 7

Hierarchical Network Coding

In a large network, there are a number of nodes with higher concentration of edges connected to them in comparison to other nodes in the network. Let us define a term concentration measure which gives the measure of concentration of edges in the neighborhood of the node \( u \), i.e., the edges connected to the node \( u \) combined with the ones associated with the nodes connected directly to \( u \). We assume the surrounding specifies a super-node, which would be comparable to a "peer group", a set of nodes grouped together within a level, in an asynchronous transfer mode (ATM) network, a telecommunications concept for video, voice or data networks. This is also analogous to the concept of "Area", a collection of routers grouped together, in open shortest path first (OSPF), an adaptive routing protocol for Internet Protocol (IP). We call the super-nodes as logical nodes.

A logical node is simply a combination of nodes seen as one entity. Hence, a logical node can also consist of lower-level logical nodes in a hierarchical fashion.

Inside the logical node, we intend to apply network coding. However, for this procedure, one needs to identify the nodes performing linear network-coding operation inside the logical node or simply the coding nodes where the summation of information takes place. The logical node also has its own effect on the overall erasure probability of the network. Other nodes in the network are, of course, also described by internal erasure probabilities for the actually used edge. As an analogy, this information is typically available in ATM or IP networks due to status information transfer between nodes inside a peer group.

The motivation to construct a minimum spanning tree is that it results in a tree that optimizes the lowest overall erasure probability at each leaf. While some of the edges in a network may
have larger erasure probability than the rest, a minimum spanning tree results in a graph that connects each node in a network without a cycle such that the total cost in terms of erasure probability in the network is the least.

In this chapter, we investigate the problem of identifying coding nodes inside a logical node where the summation of information takes place. The solution of the problem is found from the end-node perspective. Seen the problem from there, we find the minimum spanning tree (MST) of the network that reaches all the inputs. This minimum takes into account the erasure probabilities on all the edges. We analyse the structure of the minimum spanning tree, and identify the nodes in the minimum spanning tree which have more than one predecessor. We call such nodes as coding nodes.

7.1 Hierarchical Coding Mechanism

We compare a logical node to the autonomous system (AS), a collection of routers with a common routing protocol. Within an AS, Interior Gateway Protocol is used to distribute routing information. OSPF is one such protocol. OSPF collects constitutes of the erasure probability of the edges as the link state information. It also collects the topology information and computes the shortest path tree which is later used to route packets inside the AS. In OSPF, based on certain leadership priorities, a router is ’elected’ as a designated router among all the routers. The designated router is responsible for maintaining a topology table of the network, updating routing information, and sending the updates to the other routers in the network. Based on this analogy, we define the following hierarchical coding protocol.

- The logical node consists of at least one node whose duties are analogous that of a designated router in OSPF. We call this node as smart node.
- The smart node is aware of the ’outlet’ node of the logical node. The outlet node is analogous to the gateway in a network.
- The smart node is aware of the ’head’ node of the logical node. The head node is the first node in the logical node which receives information which will later be summed in the coding node inside the logical node.
• The smart node has access to information on the topology of the logical node. With this information, the smart node is able to compute a minimum spanning tree and inform the other not-so smart nodes how to forward/route the information they receive.

• With the information received from the smart node, every node in the logical node builds a 'routing table'. The routing table helps each node to identify the adjacent nodes for passing information or to combine the information received from incoming edges.

![Diagram of an erasure-prone network](image)

**Figure 7.1: An erasure-prone network**

### 7.2 Constructing Minimum Spanning Tree

We take a logical node and consider it as a network $G = (V, E)$ with each edge $(u, v) \in E$ associated with an erasure probability $e$. We consider at least two head nodes in the network. We construct the minimum spanning tree of the graph $G$ starting from the outlet node. Once the minimum spanning tree is constructed, we trace back the predecessor of each node using depth-first search. Based on the predecessors of each node, we find the path from each head node to
the outlet node. By analyzing the set of paths, we identify the coding node or the set of coding nodes, if any, inside the logical node.

Figure 7.1 illustrates a logical node with 12 nodes and 14 edges. Let us consider Node 1 as the outlet node and nodes 5, 10, and 12 as head nodes for the network. Figure 7.2 is the minimum spanning tree constructed for the graph in Fig. 7.1. The path from 5 to 1 is \(5, 6, 1\). The path from 10 to 1 is \(10, 9, 12, 6, 1\). Likewise, the path from 12 to 1 is \(12, 6, 1\). By analyzing the set of paths, we can see coding nodes are the nodes of overlapping edges including \(1, 6\).

![Figure 7.2: A network with erasure probability associated with each edge](image)

### 7.3 Node Insertion

In this section, we discuss the change in logical-node minimum spanning tree when a new node is added to the logical node. Addition of a new node results in a change in topology. One could easily reconstruct the minimum spanning tree for the new network topology. However, such procedures are inefficient in terms of time complexity. This calls for an update in the minimum spanning tree constructed earlier in the network taking into consideration the newly
added node(s). In Section 7.3.1, we will update the hierarchical coding mechanism when a new node is added to the network. We also discuss an efficient algorithm for updating the minimum spanning tree constructed on the logical-node network when a new node is added to it in Section 7.3.2.

7.3.1 Updating the Hierarchical Coding Mechanism

Let us consider that a new node $z$ is added to the network $G$. The following updates need to be incorporated in the mechanism described earlier.

- The new node, like every other node, informs its neighbors of its presence by flooding a Hello message with time to live (TTL) equal to some fixed value. The neighbors, in turn, respond by acknowledging the Hello packet.

- If the Hello packet is not heard, the neighbor is considered 'dead' or the adjacency is considered to be broken. This implies that the node is lost.

- The new node will then flood the network with its identity (ID) and the information which reveals its one-hop neighbors.

- At this time, the smart node will update its adjacency matrix which is necessary to update the spanning tree.

- If the new node introduced in the network is a leaf in the previous spanning tree, the smart node only updates the parent of the leaf, i.e., the overall solution of the spanning tree remains unchanged.

- If the new node introduced in the network is not a leaf in the previous spanning tree, then the spanning tree needs to be updated.

7.3.2 Updating the Minimum Spanning Tree

Let the number of nodes in the network be $n$. Let $T = (V, E_T)$ be the minimum spanning tree of $G$. Let $T$ be represented by adjacency lists, i.e., a label $L[v]$ is maintained, for each vertex $v \in V$, to record the edges in $T$. The erasure probability of any edge $(u, v) \notin E$ is maintained as infinity.

\(^1\)TTL gives a bound on the time that a packet can exist in a network.
i.e., $w(u, v) = \infty$. We suppose that $z$ is the new node that needs to be added to the network with weight on the edges $(z, v)$ for $v \in V$. Thus, the algorithm will construct a new minimum spanning tree $T = (V', E_{T'})$ on the network such that $V' = V \cup \{z\}$. Before we proceed to the algorithm, we present a lemma which is helpful to understand the algorithm [26].

**Lemma 7.1** Let $T = (V, E_T)$ be the minimum spanning tree of $G = (V, E)$. Let us denote the new network created by adding a new edge $x$ to $G$ by $G' = (V, E \cup \{x\})$. The updated minimum spanning tree $T'$ on $G'$ is obtained by adding $x$ to $T$ and deleting the largest edge in the cycle created.

**Proof 6** Let us assume that $T$ is not the minimum spanning tree of $G'$. Then, there must exist an edge $x'$ which when added to $T'$ can create a cycle with a larger edge. This contradicts the minimality of $T$ on $G$.

We apply the Algorithm **INSERT** adapted from [26] to insert a new node in the network. With this, new edges with certain erasure probabilties are also introduced in the network as the new node is connected to other nodes in the network. The algorithm maintains a set undiscovered which contains nodes from the original minimum spanning tree. The set $E_T$ is initialized as null, i.e., $E_T = \emptyset$. The algorithm chooses any node $r \in V$ as a root node. The edge with the largest erasure probability, i.e., the largest edge, in the path between any adjacent node $x \in L[r]$ and $z$, and is denoted by a global variable $t$, whereas $m$ is the largest edge between $r$ and $z$. The algorithm ends when all the nodes are marked as ‘completely-explored’. We call such nodes as old.
**Algorithm 7.1 INSERT**

1. mark \( r \) 'old'
2. \( \text{undiscovered} \leftarrow (\text{undiscovered}\setminus r) \)
3. \( m \leftarrow (r, z) \)
4. for each node \( x \) on \( L[r] \) do
5.   if \( x \in \text{undiscovered} \) then
6.     \( \text{INSERT}(x) \)
7.     if \( w(t) \geq w(x, r) \) then
8.       \( k \leftarrow t \)
9.       \( h \leftarrow (x, r) \)
10.  else
11.    \( k \leftarrow (x, r) \)
12.    \( h \leftarrow t \)
13.  end if
14.  \( E_T' \leftarrow E_T' \cup \{h\} \)
15.  if \( w(k) < w(m) \) then
16.    \( m \leftarrow k \)
17.  end if
18. end if
19. end for
20. \( t \leftarrow m \)

The Algorithm 7.1 performs a recursive depth first search. At each iteration, the root, local to the iteration, is removed from the label \( \text{undiscovered} \). Basically, the algorithm keeps on adding edges from the old minimum spanning tree until it finds a smaller edge to add between new node \( z \) and another node in the old minimum spanning tree. The value \( m \) represents the largest edge in a path from \( r \) to \( z \), local to each run of \( \text{INSERT} \). This value is lowered at each iteration of the loop, if possible, thereby removing the larger edges stored in \( m \) from the spanning tree, which is later used to update \( t \). The first vertex, say \( x \), which completes the \( \text{INSERT}(x) \) is a tip of \( T' \). Thus, lines 7 to 19 will be skipped and \( t \) will be assigned \((x, z)\), which would be the only edge joining \((x, z)\). In the end, \( t \) will be the lowest edge that is inserted to complete the construction of new minimum spanning tree, i.e., \( E_T = E_T \cup t \). Lines 7 to 16 deletes the largest edge in the
cycle, i.e., the largest edge among \( m, (w, r) \) and \( t \), and updates the minimum spanning tree and \( m \). By Lemma, the graph thus obtained will be a minimum spanning tree. The formal proof of the algorithm is given in [26].

In the algorithm, \textbf{INSERT}(r) is called \( n \) times (\( n - 1 \) recursively in line 6 and once at the start of the algorithm). Lines 1, 2, 3, 7 to 16, and 20 are executed \( n \) times. Lines 4 and 5 are executed \( \sum_{v \in V} |L[v]| \) times with each tree edge counted twice, i.e., \( \sum_{v \in V} |L[v]| = 2(n - 1) \). Therefore, the overall complexity of the algorithm is \( O(n) \) [26].

![Figure 7.3: (a) A minimum spanning tree, (b) introduction of a new node 4 into the minimum spanning tree](image)

Let us illustrate the algorithm with an example. Figure 7.3(a) denotes a minimum spanning tree. We introduce an additional node, Node 4, to the network as shown in Fig. 7.3(b). The new node is connected to Node 2 and Node 3 with erasure probabilities 3 and 4, respectively. This changes the topology of the minimum spanning tree as illustrated in Fig. 7.3(b). Initially, the label \textit{undiscovered} contains nodes 1, 2, and 3. Let us choose Node 1 as the root node. Thus, \( m \) is initialized to \((1, 4)\) in Line 3. The label \textit{undiscovered} is updated and contains only nodes 2 and 3. At this stage, the label \( L[1] \) contains Node 3, i.e., \( x = 3 \). In Line 6, \textbf{INSERT}(3) is executed. With this, the label \textit{undiscovered} contains only Node 2, and the value of \( m \) changes to \((3, 4)\) with Node 3 as the root node. The label \( L[r] \) is updated to \( L[3] \) and contains Node 2, i.e., \( x = 2 \). Once again, in Line 6, the recursive function \textbf{INSERT}(2) is executed. This changes the value of \( m \) to \((2, 4)\). The algorithm now moves to Line 5 and deducts that all the nodes in the
graph have already been visited. Hence, \textbf{INSERT}(x) is not executed. Instead, lines 7 to 19 are skipped and in Line 20, the value in \( m \) is copied to \( t \), i.e., \( t \) is updated to \((2, 4)\). After this, the algorithm returns to Line 7 where the erasure probability of the edge \((t)\) is compared with that of \((x, r)\). The erasure probability of \( t = (2, 4) \), which is 3, is less than erasure probability of edge \((x, r) = (2, 3)\), which is 5. Therefore, \( k \) is assigned \((2, 3)\) while \( h \) is assigned \((2, 4)\). In Line 14, \( E_T \) stores \( h \), i.e., \( E_T = \{(2, 4)\} \), and \( t \) is updated to \((3, 4)\) in Line 20. This completes the first recursive call. Now, the algorithm returns to Line 7 to complete the second recursive call, which means that the root node \( r \) is Node 1. At this stage, the erasure probability of \( t = (3, 4) \), which is 4, is greater than that of edge \((x, r) = (3, 1)\), which is 2. Therefore, \( k \) and \( h \) are updated to \((3, 4) \) and \((3, 1)\), respectively. In Line 14, \( E_T \) is updated to \( \{(2, 4) (3, 1)\} \). However, at this stage, the erasure probability of edge represented by \( k \), i.e., \((3, 4)\), is smaller than that represented by \( m \), i.e., \((1, 4)\). Therefore, in line 15 and 16, the value of \( m \) is updated to that of \( k \), which is \((3, 4)\). Finally, in Line 20, \( t \) is updated to \((3, 4)\). With this, the algorithm comes to an end. The edges of the new updated minimum spanning tree is thus obtained by combining \( t \) with \( E_T \), i.e., \( E_T \cup \{t\} \). In this case, the minimum spanning tree consists of edges \( \{(2, 4) (3, 1) (3, 4)\} \). Figure 7.4 represents the new updated minimum spanning tree after the addition of the new node.

![Updated minimum spanning tree after adding a new node](image)

**Figure 7.4:** Updated minimum spanning tree after adding a new node
7.4 Node Deletion

In this section, we discuss the change in network topology when one of the nodes in the logical node is deleted / lost. Under such circumstances, the minimum spanning tree constructed on the logical-node network has to be updated. Deleting a node from a network results in loss of certain fragments of the minimum spanning tree built on the network. The fragments of the old minimum spanning tree that remain after the deletion can be used to construct a new minimum spanning tree on the network. In fact, in case the deleted node is a leaf in the minimum spanning tree, then the fragment of the minimum spanning tree that remains after deleting the node is a minimum spanning tree of the network, which would mean that no updating or reconstruction of the minimum spanning tree is necessary. The only instance when a total reconstruction of a new spanning tree on a network after the deletion of a node is desirable is when the original minimum spanning tree is a star graph and the deleted node is the one in the center [30].

Let the number of nodes in the network be \( n \). Let \( \mathbf{T} = (V, E_T) \) be the minimum spanning tree of \( G \). Let us suppose that a node \( z \) is deleted from \( T \). Now, we discuss the algorithm for updating the minimum spanning tree when a node is deleted from the network.

The algorithm we present in this section for handling node deletion in updating the minimum spanning tree is based on the following lemma reported in [26] and [30].

**Lemma 7.2** Let us consider that a node \( z \in V \) and all the nodes incident with \( z \) are deleted from \( G \) and \( G' = (V', E') \) is the new network. Then, each of the resulting components of \( \mathbf{T} \) is a minimum spanning tree on the graph induced by its vertices.

**Proof 7** Let us suppose that some component \( T_0 = (V_0, E_0) \) is not a minimum spanning tree. Then there exists an edge in the subgraph of \( V_0 \) which is not the largest edge in the cycle it creates in \( T_0 \). This contradicts the minimality of \( T \).

**Lemma 7.3** Let \( T_z = (V', E_z) \) and \( G' = (V', E') \) be the graphs obtained from \( T \) and \( G \), respectively, after deleting a node \( z \) and all the nodes incident with \( z \) from the network. Let \( G'' = (V'', E'') \) be an undirected graph such that \( V'' \) is the set of connected components of \( T_z \) and every edge \( e = (u, v) \in E'' \) corresponds to an edge \( e' \) in \( G' \) such that \( e' \) has the smallest weight among all the edges connecting the components \( u \) and \( v \) of \( T_z \) in \( G' \). If \( T'' = (V'', E''_T) \) is a minimum spanning tree for \( G'' \) then \( T' = (V', E'_T) \) is a minimum spanning tree for \( G' \) where \( E'_T = E_z \cup E''_T \).
Proof 8 Assume that $T'$ is not a minimum spanning tree. Then there exists an edge $e^* \in (E' \setminus E_T')$ which is not the largest edge in the cycle that it creates in $T'$. By Lemma 7.2, this edge must join vertices in different components of $T$. However, by minimality of $T''$, only the largest edge in the cycle should join nodes in the same component of $T$. This contradicts the minimality of $T$ in $G$ [30].

Based on Lemma 7.3, one can say that the edge set of the minimum spanning tree $T'$ of $G'$ and the edge set of the minimum spanning tree $T$ of $G$ coincide only at the edge set of $E_z$ of $T_z$. Also, the edges present in $T''$ are the edges $e \in (T' \setminus T)$. This means that the algorithm to update the minimum spanning tree has to be constructed in two steps. In the first step, we determine the connected components of $T_z$. By Lemma 7.3, this enables us to determine the set of vertices $V''$. We also determine the set of edges $E''$ and construct the graph $G'' = (V'', E'')$. In the second step, we determine $T''$ of the graph $G''$. Then, we combine $T_z$ and $T''$ to obtain $T'$, i.e., $E_{T'} = E_z \cup E_{T''}$ [30].

We now present an algorithm to determine the connected components of a graph $G$.

Algorithm 7.2 CONNECTED-COMPONENTS($G$)

1: for each node $v \in V[G]$ do  
2: MAKE-SET($v$)  
3: end for  
4: for each edge $(u, v) \in E[G]$ do  
5: if FIND-SET($u$) $\neq$ FIND-SET($v$) then  
6: UNION($u$, $v$)  
7: end if  
8: end for

With this, we present the algorithm for updating the minimum spanning tree when a node is deleted from the network [30].

\[ \text{\textsuperscript{2}Taken from [6].} \]
Algorithm 7.3 MST-NODE DELETION

1: Determine the connected-components of $T_z$ using CONNECTED-COMPONENTS$(G)$
2: Construct graph $G'' = (V'', E'')$
3: Determine minimum spanning tree $T'' = (V'', E''_T)$ of $G''$
4: Find $T' = (V', E''_{T'})$ such that $E''_{T'} = E_z \cup E''$

The time complexity for updating minimum spanning tree when a node is deleted from the network is related to the number of connected components or left-over fragments of the old minimum spanning tree after a node has been deleted. This, in turn, is equal to the degree of the deleted node in the old minimum spanning tree. As discussed earlier, the problem is easiest to solve when the degree of the node deleted is one. The complexity of the algorithm is maximum when the degree of the deleted node is $n - 1$, i.e., when the deleted node is the center node in a star-shaped minimum spanning tree. In this case, no fragment of $T$ is left behind for updating the spanning tree. The complexity of the algorithm, in this case, is $O((n - 1)^2)$ [30]. In any case, this method is still more effective than reconstructing the minimum spanning tree from scratch once a node has been deleted from the network.

Figure 7.5: a) A logical-node network  b) Minimum spanning tree of the network
In order to illustrate the algorithm, let us consider the following example. Figure 7.5 (a) and Fig. 7.5(b) represents the logical-node network and its corresponding minimum spanning tree, respectively. Let us assume that Node 6 is deleted. Subsequently, the edges incident with Node 6 in the network are also deleted. Thus, we obtain a new logical-node network $G'$ after deletion of Node 6 as shown in Fig. 7.6(a). The deletion of Node 6 from the old minimum spanning tree results in fragmentation of the tree. Let us denote this graph by $T_z = (V', E_z)$. As such, in Fig. 7.6(b), one can see that there are three components of this tree, i.e., nodes 1 to 4 form the first component, Node 5 forms the second component and Node 7 forms the third component. By Lemma 7.2, each resulting component is a minimum spanning tree on the graph induced by its vertices. The Algorithm 7.3 seeks an edge $e \in G'$ for each component $u, v$ such that $e$ has the smallest erasure probability among all the edges connecting $u$ and $v$. In this case, edge $(4, 5)$ with erasure probability 7 connects the first component with vertices $\{1, 2, 3, 4\}$ to the second component, i.e., the one with vertex 5. Similarly, edge $(4, 7)$ with erasure probability 8 connects the first component to the third component, i.e., the one with vertex 7. Likewise, edge $(5, 7)$ with erasure probability 6 connects the second component to the third component. Figure 7.7 illustrates the resultant network. This graph contains the same set of nodes as the one illustrated in Fig. 7.6(b). The algorithm then computes the minimum spanning tree of this network. The edge set of the minimum spanning tree thus computed is $\{(1, 2), (1, 3), (3, 4), (4, 5), (5, 7)\}$. We call this graph as $T'' = (V'', E_{T''})$. By Lemma 7.3, the edge set of the updated minimum spanning tree should be a union of edge set of $T''$ and $T_z$. The algorithm constructs this set and the graph thus obtained is the updated minimum spanning tree. The updated minimum spanning tree is shown in Fig. 7.7(b).
Figure 7.6: (a) New logical-node network after deletion of Node 6  (b) Fragments of the old minimum spanning tree after deletion of Node 6

Figure 7.7: (a) The graph $G''$ obtained after executing Step 2 of Algorithm 7.3  (b) Updated minimum spanning tree of the network
7.5 Change in Erasure Probability Associated with Edge

Let $G = (V, E)$ and $T = (V, E_T)$ be a logical node and its corresponding minimum spanning tree respectively. Consider a scenario in which the erasure probability $p_e$ associated with edge $e \in E$ inside a logical node changes to $p_e^*$. Under such circumstances, one might find it necessary to update the minimum spanning tree constructed for the logical node. Updating the minimum spanning tree, however, depends on whether the edge (the erasure probability of which has changed) lies in the minimum spanning tree or outside the minimum spanning tree, or whether the erasure probability associated with the edge has increased or decreased. In this section, we discuss algorithms for updating the minimum spanning tree for a logical node when the erasure probability associated with an edge has changed.

7.5.1 When the Edge is a part of the Minimum Spanning Tree

Let us consider that the erasure probability of $e \in T$ changes to $p_e^*$. Under such circumstances, we determine the value of erasure probability of the edge and consider the following:

1. If $p_e^* < p_e$

   In this case, we simply update $p_e$ to $p_e^*$. To illustrate this, consider a logical node and its minimum spanning tree as shown in Fig. 7.5 which shows a logical node and its minimum spanning tree. As shown in figure, the erasure probability of the edge connecting nodes 1 and 3 is 8. Consider that the erasure probability of the edge changes to 7. Figure 7.8(b) shows the updated minimum spanning tree which is obtained by simply updating the erasure probability.
2. If $p_e^* > p_e$

In this case, we delete $e$ from $T$ and insert an edge with smallest erasure probability between the resulting two components. By Lemma 7.3, the resulting spanning tree will be an updated minimum spanning tree.

Once again, let us consider Fig. 7.5 which shows a logical node and its minimum spanning tree. Now, consider that the erasure probability of the edge connecting nodes 3 and 6 changes from 1 to 10. Figure 7.9(a) illustrates two components of the spanning tree, i.e., nodes 1, 2, 3, and 4 form the first component while nodes 5, 6, and 7 form the second component of the spanning tree. The algorithm seeks an edge with the smallest erasure probability between the components, i.e., edge (4, 5) with erasure probability 7, and inserts it between the two components as shown in Figure 7.9(b).
Figure 7.9: a) Disconnected components of minimum spanning tree  
    b) Updated minimum spanning after insertion of an edge with smallest erasure probability

### 7.5.2 When the Edge is not a part of the Minimum Spanning Tree

Before we proceed to the algorithm, let us introduce a helpful lemma.

**Lemma 7.4** Let $T = (V, E_T)$ be a minimum spanning tree on $G = (V, E)$. Let us consider that the erasure probability associated with an edge $e$ outside $T$ changes from $p_e$ to $p_e^*$. As such, let us denote the edge with the new erasure probability by $e'$ and construct an updated logical node $G' = (V, E \cup \{e'\})$. If $p_e^*$ is less than $p_e$, then an updated minimum spanning tree $T'$ on $G'$ of the logical node is obtained by adding $e'$ to $T$ and deleting the largest edge in the cycle created.

**Proof 9** Let us assume that $T'$ is not the minimum spanning tree on $G'$. Then there exists an edge $e^*$ which when added to $T'$ can create a cycle in which there is a larger edge, and this contradicts the minimality of $T$ on $G$ [26].

Now, we proceed to present the algorithm on updating the minimum spanning tree.
1. **If** $p_e^* < p_e$

   In this case, we follow the given procedure
   
   - Add $e$ to $T$ creating a cycle
   - Remove the edge with largest erasure probability on the cycle.
   - By Lemma 7.4, the updated graph will be the minimum spanning tree.

   In order to illustrate this, let us assume the erasure probability of the edge connecting nodes 2 and 5 in Fig. 7.5(a) to change from 10 to 2. The edge, however, is not part of minimum spanning as shown in Fig. 7.5(b). Figure 7.10(a) illustrates the cycle in the minimum spanning tree upon addition of this edge to the minimum spanning tree. The updated minimum spanning tree obtained by removing the largest erasure probability in the cycle is shown in Fig. 7.10(b).

![Figure 7.10](image)

Figure 7.10: a) Addition of edge in MST resulting in a cycle b) Updated minimum spanning after deletion of the largest edge in the cycle

2. **If** $p_e^* > p_e$

   Under such circumstance, no update in the minimum spanning tree is required.
7.6 Conclusion

Due to the dynamic nature of real networks, the network requires efficient updating especially when the parameters related to the network change. The addition of new nodes and edges in a network might also change other parameters in a network. As such, the performance of network coding considering the updating of network parameters should be studied further. One can also study the performance of the network coding when delay in propagation of the symbols is introduced.
Chapter 8

Summary and Conclusion

In an effort to improve reliability, efficiency, and throughput in a network, two major coding schemes have been proposed so far. While network coding is used to encode information at intermediate nodes to increase throughput for multicasting information in a network, LT codes allows source node to encode in such a way that the recovery of some subset of message will allow for successful decoding.

This thesis is an attempt to discuss the problem of erasures at different hierarchies in a network. It has been proposed that, in a large network, not all intermediate nodes need to participate in network coding operations to achieve the upper bound. In the upper level, one can still achieve the desired throughput by carefully selecting an optimal set of coding nodes. Ant colony algorithm is one possible solution to find such a set. The fundamental idea behind this algorithm is to chose an option such that the option results into a more desirable one by virtue of positive feedback in the system.

However, this type of encoding when applied along with LT-encoding sequentially in binary erasure channels leads to distortion of degree distribution in received symbols as a result of dependency of LT-encoding process on the degree of distribution of the codes. This issue has been discussed in Chapter 5 and an attempt has been made to build concepts on computing the distorted degree of distribution of symbols in generalized networks.

In a lower hierarchy, the network topology has to be taken into consideration while addressing the issue of minimizing erasures in networks. One solution to look into problem is by structuring networks according to the concentration of edges, and building a minimum spanning tree which takes into consideration the nodes through which information enters the so-called logical node.
This gives a definite advantage in identifying the nodes where the actual logical summation or network coding takes place in a logical node, as discussed in Chapter 7.

The network topology however is subjected to change owing to addition or deletion of nodes and edges. A problem occurs since new links added in the network may have different erasure probabilities. Moreover, any change in the network topology will change the overall erasure probability of the network. Thus, the minimum spanning tree on the logical node needs to be updated. This problem is discussed in Chapter 7 and some algorithms to update the spanning is proposed as a solution.
Bibliography


