

# ON THE OPTIMUM COMMUNICATION COST PROBLEM IN INTERCONNECTION NETWORKS

BY

Khalid Saud Al-Zamil

A Thesis Presented to the  
DEANSHIP OF GRADUATE STUDIES

**KING FAHD UNIVERSITY OF PETROLEUM & MINERALS**  
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the  
Requirements for the Degree of

**MASTER OF SCIENCE**

In

**COMPUTER SCIENCE**

JUNE 2005

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS  
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES


This thesis, written by

**Khalid Saud Abdulaziz Al-Zamil**

under the direction of his thesis advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

Thesis Committee:



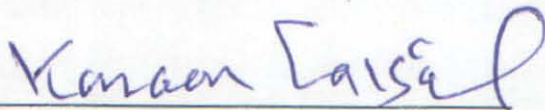
Dr. Mohammad Alsuwaiyel (Advisor)




Dr. Motaz Ahmed (Member)



Dr. Wasfi Al-Khatib (Member)



Dr. Kanaan Faisal  
(Department Chairman)



Dr. Mohammad Al-Ohali  
(Dean of Graduate Studies)

26/9/05  
Date 26-10-2005



## DEDICATION

This thesis is dedicated to my parents and loving wife.

## ACKNOWLEDGEMENT

All praise and thanks be to Almighty Allah, the most Gracious, the most Merciful. Peace and mercy be upon His Prophet.

Acknowledgment is due to King Fahd University of Petroleum & Minerals and Saudi Aramco for supporting this research.

I would like to express my deepest and sincere gratitude to my thesis advisor Dr. Mohammad Alsuwaiyel for his tremendous support, guidance and encouragement. I would also like to thank my thesis committee members: Dr. Motaz Ahmed and Dr. Wasfi Al-Khatib for their invaluable cooperation and support.

Last but not least, I would like to thank my family members for being patient and for their words of encouragement to spur my spirit at moments of depression.

# Contents

Dedication . . . . .	iii
Acknowledgment . . . . .	iv
Contents . . . . .	v
List of Tables . . . . .	vii
List of Figures . . . . .	ix
Abstract (English) . . . . .	x
Abstract (Arabic) . . . . .	xi
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 PRELIMINARIES</b>	<b>4</b>
2.1 Graphs and Multigraphs . . . . .	4
2.2 Trees . . . . .	6
2.3 Interconnection Networks . . . . .	7
2.3.1 Multiple Processor Systems . . . . .	8
2.3.2 The Topological Structure . . . . .	9
2.3.3 Basic Principles of Network Design . . . . .	9
2.4 NP-Complete Problems . . . . .	17
<b>3 ENUMERATING ALL SPANNING TREES OF A GRAPH</b>	<b>19</b>
3.1 Compact Output . . . . .	21

3.2	Basic ideas . . . . .	22
<b>4</b>	<b>THE OPTIMUM COMMUNICATION SPANNING TREE PROBLEM</b>	<b>28</b>
4.1	Problem Description . . . . .	29
4.2	Applications of the OCST problem . . . . .	30
4.3	Existing Approaches . . . . .	32
<b>5</b>	<b>THESIS WORK</b>	<b>37</b>
5.1	Problem Description . . . . .	38
5.2	The Proposed Algorithm . . . . .	40
5.3	Experiment Setup . . . . .	46
5.4	Experiment Results . . . . .	55
<b>6</b>	<b>CONCLUSION</b>	<b>61</b>
6.1	Contribution . . . . .	61
6.2	Future Work . . . . .	62
	APPENDIX . . . . .	64
	Bibliography . . . . .	78
	VITA . . . . .	83

# List of Tables

4.1	Summary of algorithms for solving the Optimum Distance Communication Spanning Tree problem. . . . .	36
5.1	The number of possible spanning trees for the randomly generated graphs $G_1, G_2$ and $G_3$ . . . . .	46
5.2	The number of possible spanning trees for the hypercube (dimension = 1, 2 . . . , 10). . . . .	48
5.3	The number of possible spanning trees for the butterfly (dimension = 1, 2 . . . , 8). . . . .	49
5.4	The congestion costs for the for the randomly generated graphs $G_1, G_2$ and $G_3$ . (The results are produced using the brute-force method). . . . .	51
5.5	The congestion costs for the hypercube (dimension = 1, 2 . . . , 10).(The results are produced using the brute-force method). . . . .	51
5.6	The congestion costs for the butterfly (dimension = 1, 2 . . . , 8). (The results are produced using the brute-force method). . . . .	52
5.7	The congestion costs for a shortest-path spanning tree (SPT) and hamiltonian spanning tree (HT) for the hypercube (dimension = 1, 2 . . . , 10), where ratio = $\frac{HT_{cost}}{SPT_{cost}}$ . . . . .	53

5.8	The congestion costs for a shortest-path spanning tree (SPT) for the butterfly (dimension = 1, 2 ..., 8). . . . .	54
5.9	The congestion costs for the randomly generated graphs $G_1$ , $G_2$ and $G_3$ . (The results are produced by the ContractedRandomSpan algorithm) . . . . .	56
5.10	The stretch costs for the randomly generated graphs $G_1$ , $G_2$ and $G_3$ . (The results are produced by the ContractedRandomSpan algorithm) . . . . .	56
5.11	The congestion costs for the hypercube (dimension = 1, 2 ..., 10) generated by the ContractedRandomSpan algorithm. . . . .	57
5.12	The stretch costs for the hypercube (dimension = 1, 2 ..., 10) generated by the ContractedRandomSpan algorithm. . . . .	58
5.13	The congestion costs for the butterfly (dimension = 1, 2 ..., 8) generated by the ContractedRandomSpan algorithm. . . . .	58
5.14	The stretch costs for the butterfly (dimension = 1, 2 ..., 8) generated by the ContractedRandomSpan algorithm. . . . .	59



# List of Figures

2.1	The linear array network topology $LA_5$ . . . . .	11
2.2	The mesh network topology $G(8, 4)$ . . . . .	12
2.3	The hypercube network topology for dimensions 1, 2 and 3. . .	12
2.4	An $H_4$ created by merging two $H_3$ . . . . .	13
2.5	A butterfly network topology $BF_3$ . . . . .	15
2.6	Two $BF_2$ are obtained by eliminating level 0 nodes from $BF_3$ . .	17
3.1	Graph $G_1$ and two spanning trees $T^0$ and $T^c$ . . . . .	23
3.2	Spanning tree <i>child-parent</i> relationships in $S(G_2)$ . . . . .	24
3.3	Enumeration of all spanning trees for $G_2$ . . . . .	27
5.1	A comparison between the RandomSpan algorithm and the ContractedRandomSpan algorithm when finding a spanning tree for the hypercube and butterfly topologies up to dimension = 10. . . . .	47
5.2	The graphs $G_1, G_2$ and $G_3$ that were randomly generated for the experiments. . . . .	48
5.3	Two spanning trees for $H_4$ . . . . .	50

## THESIS ABSTRACT

**NAME :** Khalid Saud Al-Zamil.

**TITLE :** On the Optimum Communication Cost Problem in Interconnection Networks.

**MAJOR FIELD :** Computer Science.

**DATE OF DEGREE :** June 2005.

In the *optimum communication spanning tree* (OCST) problem, a tree that connects all vertices for a complete graph has to be found. The spanning tree found must satisfy the communication requirements needed by the vertices with a minimum total cost. A special case of the OCST problem is the *optimum distance spanning tree* (ODST) problem, where the requirements are restricted to be constant. Both problems are known to be NP-hard. In this thesis, we propose a randomized algorithm to efficiently solve two special cases of the ODST problem. This can be achieved by randomly generating spanning trees with certain properties. We conjecture that such an approach can yield near-optimum solutions. An experiment has been conducted to evaluate the proposed algorithm. The experiments involve testing the proposed algorithm to solve these special cases using several randomly generated graphs, in addition to the hypercube and butterfly network topologies to some specified dimension.

## خلاصة الرسالة

الاسم: خالد سعود عبدالعزيز الزامل

العنوان: على مسألة إيجاد تكلفة الاتصال المثلى في الشبكات المترابطة

الدرجة: ماجستير في العلوم

تاريخ الدرجة: حزيران 2005

في مسألة إيجاد شجرة الاتصال الممتدة المثلى، لا بد من إيجاد شجرة تربط جميع الرؤوس في الرسم البياني الكامل، هذه الشجرة الممتدة لا بد أن توفر جميع متطلبات الاتصال اللازمة للرؤوس وبأصغر تكلفة إجمالية. مسألة الشجرة الممتدة ذات المسافة المثلى (ODST)، والتي تكون متطلباتها ثابتة، هي حالة خاصة من مسألة الشجرة الممتدة ذات الاتصال الأمثل (OCST)، كلا المسألتين تعرفان بأنهما (NP-hard).

نقوم في هذه الرسالة، باقتراح خوارزميات عشوائية لحل حالتين خاصتين من مسألة (ODST) وبفعالية عالية، وذلك بتوليد أشجار ممتدة عشوائياً مع خصائص معينة. لدينا حدس بأن استخدام هذه الطريقة يؤدي إلى الحصول على حلول قريبة جداً من الحل الأمثل، قمنا بعمل تجارب لتقييم الخوارزميات المقترحة، هذه التجارب تضمنت اختبار الخوارزميات المقترحة لحل الحالات الخاصة باستخدام عدة رسومات بيانية مولدة عشوائياً، بالإضافة إلى شبكات الـ (hypercube) و (butterfly) إلى أبعاد محددة.

# Chapter 1

## INTRODUCTION

Many researchers are currently interested in the design of optimum communication and transportation networks. These type of problems are formally known as topology design problems. In topology design problems, it is required to effectively design a network such that the constraints are met and the objectives are optimized. Topology design problems have many real world applications, an example of such are telecommunications, computer networking, and oil & gas pipelines.

Different types of topology design problems have been studied, which resulted in either exact solutions or heuristics [8, 20]. Some of the most popular topology design problems are known as constrained minimum spanning tree problems. Some examples are the *optimum communication spanning tree* (OCST) problem, the degree-constrained minimum spanning tree problem, the minimum steiner tree problem, or the capacitated minimum spanning tree problem [27].

The OCST problem was first introduced by Hu [17]. The problem states that a tree that connects all vertices for a complete graph has to be found. The spanning tree found must satisfy the communication requirements needed by the vertices with a minimum total cost. Let  $T$  be a spanning tree for a given graph  $G(V, E)$ . The communication cost for  $T$  is defined as follows. Define a pair of vertices  $x$  and  $y \in V(G)$  where  $x \neq y$ . The distance  $d_{x,y}$  is the distance between vertices  $x$  and  $y$  restricted on  $T$ . The communication requirement  $r_{x,y}$  for the pair  $x$  and  $y$  is provided. There is a unique path in  $T$  between  $x$  and  $y$ . The *distance* of the path is the sum of distances of edges in the path. The *communication cost* for the pair  $x$  and  $y$  is  $r_{x,y}$  multiplied by the distance of the path. Summing over all  $\binom{n}{2}$  pairs of vertices, we have the communication cost for  $T$ .

Hu also formulated a special case for the OCST problem and called it the *optimum distance spanning tree* problem. In the ODST problem, the communication requirement between each pair of vertices is constant, while the distances between each pair of vertices is arbitrary. Both the OCST problem and the ODST problem have been shown to be NP-hard in [18].

The objective of this thesis is to attempt to find an efficient randomized algorithm to solve a special case of the ODST problem. This special case is different than the original ODST problem in two ways. First, rather than limiting the ODST problem to finding solutions for complete graphs only. We propose to solve the ODST problem for general graphs. Second, this special case restricts the distance of each edge to be constant and equal to one. This is in contrast to the original ODST problem where the distances of the edges are arbitrary. We will refer to this special case as the *optimum congestion cost problem*. We also solve the case where only the distances between vertices that share an edge in the graph are included in the computation. We will refer to this special

case as the *optimum stretch cost* problem, which up to our knowledge has no research history.

In our approach, we attempt to generate the spanning trees for a given graph randomly. The proposed algorithm generates spanning trees that are biased towards shortest-path spanning trees with higher probability. A shortest-path spanning tree can be obtained by performing a breadth-first search on the given graph. By generating spanning trees with such properties, we are capable of scoping the search space of possible solutions efficiently. We conjecture that we can find near-optimum solutions by following this approach.

An experimental study has been performed to evaluate the performance of the proposed algorithm. The experiments were conducted using several types of special graphs. The special graphs selected for these experiments include three randomly generated graphs, in addition to the hypercube and butterfly network topologies to some specified dimension. The experiments involve finding an optimum solution for the previously mentioned special graphs.

The thesis is organized as follows. Chapter 2 is dedicated to the explanation of some preliminaries. This is followed by chapter 3, which is a literature survey on the different deterministic algorithms that exist for enumerating all spanning trees of a graph. In chapter 4, a literature survey is given about the OCST problem in addition to some of its applications. The literature survey will mainly focus on the different approaches that currently exist, which try to find an optimum solution for the ODST problem. The proposed algorithm is discussed in chapter 5, along with the experiment setup and results. The conclusion is in chapter 6, where the contribution of this thesis will be listed, as well as to highlighting some potential areas for future work.

# Chapter 2

## PRELIMINARIES

### 2.1 Graphs and Multigraphs

A graph  $G$  can be defined as a triple  $(V(G), E(G), \varphi(G))$ . Where  $V(G)$  is a nonempty set of vertices,  $E(G)$  is a set of edges, and  $\varphi(G)$  is a mapping  $E(G) \rightarrow V(G) \times V(G)$  which maps an edge into a pair of vertices called end-vertices of the edge.

A graph  $G$  is called to be empty if  $E(G) = 0$ . It is also called trivial if  $V(G) = 1$  and nontrivial otherwise. The graph  $G$  is finite if both  $V(G)$  and  $E(G)$  are finite. It is possible under  $\varphi(G)$ , to have more than one edge mapped into a single element  $V(G) \times V(G)$ . Edges that undergo such a mapping are referred to as parallel edges. An edge with identical end-vertices is called a loop. A simple graph is a graph with no parallel edges or loops.

In the case of a simple graph  $G$ , the mapping is  $\varphi(G)$  injective. This means that for each edge  $e$ , there exists a unique pair of vertices that connect the edge. Therefore, it is convenient to only use a subset  $V(G) \times V(G)$  instead of the edge set  $E(G)$ . The graph  $G$  can be written as  $G = (V(G), E(G))$  instead of  $(V(G), E(G), \varphi(G))$ .

A graph  $G$  is called *directed* (also known as a digraph) if  $V(G) \times V(G)$  is considered as a set of *ordered* pairs. For a directed edge  $e$  in the directed graph, if  $\varphi_G(e) = x, y$ , then vertex  $x$  is called the tail and vertex  $y$  is called the head. The edge is called an outgoing edge of  $x$  and an incoming edge of  $y$ .

A graph  $G$  is called *undirected* if  $V(G) \times V(G)$  is considered as a set of *unordered* pairs. The edges of the graph  $G$  are called undirected edges. An unordered pair of vertices can be denoted as  $xy$  or  $yx$ , instead of using the notation  $\{x, y\}$ .

A graph  $G$  can be drawn on the plane by representing each vertex of  $G$  as a small circle. In the case of the graph  $G$  being a digraph, then each directed edge connecting two vertices is represented as a directed line (or curve) segment pointing from the tail  $x$  to the head  $y$ . However, if the graph  $G$  was an undirected graph, then each undirected edge is represented as a line (or curve) segment joining the two vertices  $x$  and  $y$ . Such drawings help show the incidence relations that hold between the vertices and edges of a graph  $G$ .

In this paper, a graph  $G$  denotes a nontrivial, nonempty, finite, undirected and simple graph. This definition is assumed throughout the rest of this paper.

Suppose a graph  $G$  (termed to the conditions specified earlier) has two vertices  $x$  and  $y$ . An  $xy$ -walk of length  $k$  in  $G$  is a sequence  $W = (x, x_1, x_2, \dots, x_{k-1}, y)$ . The vertices  $x$  and  $y$  are called the *origin* and *terminus* respectively. The other vertices are called *intermediate* vertices.

If the edges in  $W$  are all distinct, then it is called a *trail*. If  $W$  is a trail and the vertices are also distinct then it is called a *path*. If the origin and terminus are identical for a walk, then it is known as a *closed walk*. A closed trail is a *circuit*, and a closed path is a *cycle*. A path that contains every vertex of  $G$  is called a *Hamiltonian path*. A cycle that contains every vertex of  $G$  is called a *Hamiltonian cycle*. A graph is Hamiltonian if it contains a Hamiltonian cycle.



A graph  $H$  is called a *subgraph* of a graph  $G$  if  $H \subseteq G$ , i.e. if  $V(H) \subseteq V(G)$  and  $E(H) \subseteq E(G)$ . A graph  $H$  is called a *spanning subgraph* if  $V(H) = V(G)$ . Two vertices  $x$  and  $y$  of the graph  $G$  are said to be connected if there is an  $xy$ -path in  $G$ . It should be noted that the “is connected” is an equivalence relation on  $V(G)$ . Thus, there exists an equivalence partition  $\{V_1, \dots, V_w\}$  of  $V(G)$  for  $w = |V(G)|$ . The subgraph  $G[V_i]$  is known as a connected subgraph or as a *connected component*. In other words, if  $C$  is a connected component of the graph  $G$ , then all vertices of the graph  $G$  are reachable using the edges of  $C$ .

## 2.2 Trees

Trees are considered to be very fundamental graph-theoretic models. They have particular interest to the fields of: computer graphics and visualization, artificial intelligence, information theory, data structure and analysis, and combinatorial optimization. A connected graph that contains no cycles is known as a *tree*. A spanning subgraph of  $G$  is called a *spanning tree* if it is a tree. Any connected graph must contain a spanning tree. A spanning tree with  $n$  vertices has exactly  $n - 1$  edges, that is, the total number of edges in a spanning tree is less than the total number of vertices by one.

**Theorem 1** : *Let  $G$  be a connected graph, then*

1.  $G$  is a tree.
2.  $G$  has exactly  $n - 1$  edges.
3. If one edge is added to  $G$  then a cycle is created.

The tree, as defined, is sometimes referred to as a *free* tree. It is a mathematical object. This is to be contrasted with a *rooted* tree, in which there is one distinguished vertex called the *root* and implicit directions from and to the root. A tree that has a path that contains all the vertices for the Graph  $G$  is known as a *hamiltonian* spanning tree.

## 2.3 Interconnection Networks

A system can be defined informally as a collection of components, which are connected to form a coherent entity with a well defined function or purpose. The function performed by the system is based on the function of its components in addition to how these components are interconnected [16].

Examples of interconnection networks are computer systems, computer networks, communication systems and transportation systems. In the case of a computer system, the components might be the processors, storage units and I/O equipment. The function of the computer system is to basically transform a set of input information into a set of output results.

A Multiple Processor System is a system that contains two or more autonomous processors as its components. It is quite natural to model an interconnection network as a graph  $G$ . Where the vertices represent the components of the interconnection network, and the edges represent the communication links between them. A graph of this type is known as a topological structure. In the next sections, a discussion about Multi Processor Systems is going to be given followed by a discussion on topological structures and the basic principles of network design.

### 2.3.1 Multiple Processor Systems

A Multiple Processor System (MPS) is a system that contains two or more autonomous processors as its components. These processors cooperate to achieve a particular task. MPS's that consist of thousands of processors can execute parallel algorithms, resulting in solving large problems in real time.

According to [29], MPS architectures can be classified into two broad classes. The first class of MPS's are systems consisting of  $n$  number of identical processors that are interconnected through a switch to  $n$  number of memories. The MPS's in this class are also known as *tightly coupled* systems. All the processors share the same global memory and have the same address space. The shared memory is used for intercommunication and synchronization amongst the processors.

The main benefit of these types of architectures is that the data access is transparent to the user. The data is held in a large memory accessible by all the processors. However, there are two main drawbacks to this class of MPS architectures. First, the memory sharing architecture, adopted by these MPS's architectures, cannot take advantage of some inherent properties in some problems. Second, the switching network becomes more complicated as the number of processors increase in the system.

The second class of MPS's are systems where each processor has its own local memory, and the processors are interconnected according to some pattern. The MPS's in this class are also known as *loosely coupled* systems. There is no global synchronization or shared memory. The computation is data driven, and the intercommunication and synchronization amongst the processors is achieved through message passing. The main advantage of this type of architectures is the simplicity of their design. The processors are identical or are of

a few different kinds.

### 2.3.2 The Topological Structure

An *interconnection network* (or network for short) can be defined as the connection pattern of the components in a system. The interconnection network of a system provides logically a specific mechanism to connect all the components of the system. The interconnection network can be modeled as a graph  $G$ . The vertices of the graph represent the components of the system and the edges represent the physical communication links between them. Such a graph is called the topological structure for the interconnection network (or network topology for short). The terms network topology and graph  $G$  are going to be used interchangeably in this paper.

Network topologies can be classified into two main groups: *static* and *dynamic*. The first class of MPS architectures are dynamic systems. The communication links can be reconfigured by setting the networks active switching elements. The second class of MPS architectures are static systems. In this case, the communication links are passive and can not be altered. Several examples of static systems are given in the following sections.

### 2.3.3 Basic Principles of Network Design

There are a number of fundamental principles that should be conformed to when designing an interconnection network. A network topology, as discussed earlier, is a graph. That is why graph theory is used to explaining some of these principles, as far as the topological structure is concerned. These principles as stated by [7] can be summarized as follows:

1. **Small and fixed degree:** The *degree* of a network topology can be defined as the maximum number of connections to a component. A larger degree will compromise the overall scalability of the system, it also means more wiring. Thus a small or fixed maximum degree is desirable.
2. **Small transmission delay:** The *diameter* of a network topology can be defined as the maximum distance between any two components. A small diameter is desirable since it is proportional to sending a message from one component to another.
3. **Maximum fault tolerance:** The network should function properly regardless of an edge or vertex failure. The maximum connectivity is desirable because it is the maximum fault tolerance of the network.
4. **Easy routing algorithm:** Routing is considered to be an important function in communication networks. It is responsible for specifying a fixed route between two components for communication.
5. **Embeddability of other topologies:** This is a crucial issue that deals with the ability of an architecture to take advantage of an algorithm developed for a different type of architecture.
6. **Large bisection width:** The *bisection width* is defined as the minimum number of edges, whose removal will result in two connected components of approximately the same size. A large bisection width is desirable, because it will result in more data traveling in parallel between the two connected components. In other words, a larger bisection width will mean faster communication and higher fault tolerance.

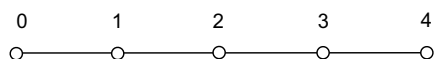


Figure 2.1: The linear array network topology  $LA_5$ .

7. **Extendibility:** It should be possible to concatenate two or more networks into a single network. When extending a network, some desirable properties should be remained while other useful parameters should be calculated easily.

### Examples of Popular Network Topologies

Several network topologies have emerged with respect to the previously listed basic principles of network design. In the following sections, some of the popular network topologies are going to be discussed. The linear array and the mesh network topologies will be briefly explained. This will be followed by a discussion on the hypercube and butterfly network topologies, which are of particular interest to the research conducted in this paper.

**The Linear Array** The *linear array* network topology, denoted as  $LA_n$ , is basically a set of  $n$  components which are connected in a linear fashion [22]. Figure 2.1 illustrates an example of  $LA_5$ . The  $LA_n$  has a *degree* = 2, since each component is connected to at most two components. It is obvious that the *diameter* =  $n - 1$ , and that the *bisection width* = 1.

**The Mesh** The *mesh* network topology (also known as the grid network topology) is defined as the Cartesian product  $P_l \times P_m$  of undirected paths  $P_l$  and  $P_m$  denoted by  $G(l, m)$  [22]. In the literature,  $G(l, m)$  is usually called an  $l \times m$  mesh. Figure 2.2 shows an  $8 \times 4$  mesh. For a mesh with  $n$  components, the *degree* = 4, the *diameter* =  $2\sqrt{n}$ , and the *bisection width* =  $\text{sqrt}(n)$ .

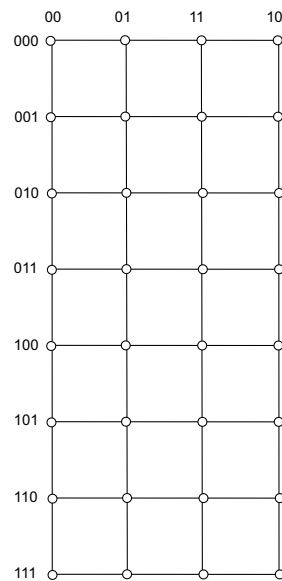


Figure 2.2: The mesh network topology  $G(8, 4)$ .

**The Hypercube** The *hypercube* network topology, as suggested by [32], is one of the most popular and efficient network topologies. It is considered to be the first choice for a network topology for parallel processing and computing systems [21].

Let us define a  $d$ -dimensional hypercube denoted as  $H_d$ , where  $d \geq 1$ . In graph theory,  $H_d$  can be represented as a graph with the vertex set  $V(G)$  that consists of all the binary sequence of length  $d$  on the set  $\{0, 1\}$ , i.e.:

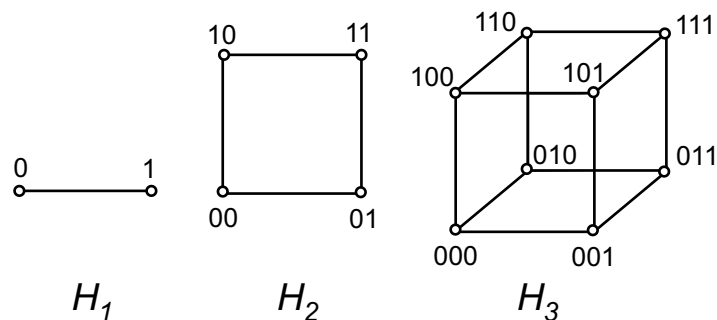


Figure 2.3: The hypercube network topology for dimensions 1, 2 and 3.

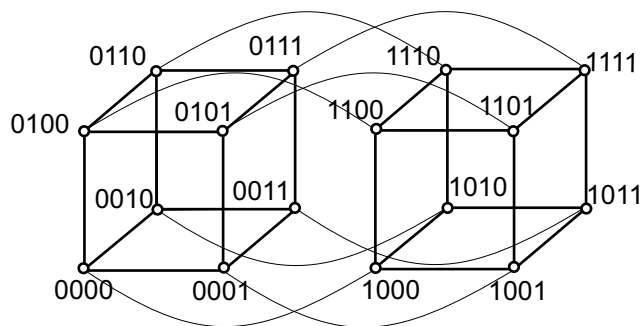


Figure 2.4: An  $H_4$  created by merging two  $H_3$ .

$$V = \{x_1x_2 \dots x_n | x_i \in \{0, 1\}, i = 1, 2, \dots, d\}.$$

Two vertices  $x = x_1x_2 \dots x_d$  and  $y = y_1y_2 \dots y_d$  are linked by an edge if and only if the two vertices differ in exactly one position, i.e.  $|x_i - y_i| = 1$ . Examples for the first three dimensions of the hypercube are shown in Figure 2.3.

**Properties of the Hypercube** The hypercube has many attractive properties; some of these properties are summarized in the following:

1. The number of nodes in  $H_d$  is  $n = 2^d$ , and has  $d2^{d-1}$  edges.  $H_d$  has a *diameter* =  $d$ , *degree* =  $\log d$ , and *bisection width* =  $\frac{n}{2}$ .
2. The hypercube can be constructed recursively from lower dimensional cubes. Suppose that a hypercube  $H_d$  is required. To construct such a cube, we start with two cubes  $H_{d-1}$ . Then, an extra bit is added to the binary sequences of each cube. The bit added is set to one for all binary sequences of one cube, and reset to zero for the other. The final step is to link the nodes that match in the lower  $d - 1$  bits together. The hypercube  $H_4$ , displayed in Figure 2.4, was generated by merging two hypercubes of *dimension* = 3.



3. The distance between two vertices  $x = x_1x_2 \dots x_d$  and  $y = y_1y_2 \dots y_d$  can be obtained by starting with  $x$  and changing its bits continuously until  $y$  is reached. For example, suppose  $x = 01000$  and  $y = 10101$ , then by starting with  $x$ , the following can be made:

$x = 01000 \rightarrow 11000 \rightarrow 10000 \rightarrow 10100 \rightarrow 10101 = y$ . This is a shortest path between  $x$  and  $y$ , and the distance between them is equal to the number of bits changed (*distance* = 4 in our example). The distance between the two vertices  $x$  and  $y$  is often referred to as the *Hamming distance* of  $x$  and  $y$ , denoted by  $H(H_d, x, y)$ .

4. A shortest path between any two nodes in the hypercube can be achieved by changing the bits of one of the nodes continuously. However, the shortest path obtained using this method is normally not unique. This is because changing the bits can occur from left-to-right and vice versa. Hence, a routing algorithm can be found that finds the distance between two nodes of the hypercube efficiently.
5. A Hamiltonian cycle  $C_d$  of  $H_d$  represents a ring sequence on  $n$ -bit binary numbers such that any two successive binary numbers differ in one bit only. A sequence of binary numbers that adhere to this property are called *n-bit Gray codes* denoted as  $G_n$ . For example,  $G_4 = \{000, 001, 011, 010, 110, 111, 101, 100\}$ .

From the properties mentioned previously, the hypercube satisfies most of the requirements of the basic principles of network design. The hypercube is considered to be a good tradeoff between the different objectives of an interconnection network. One of these tradeoffs is that the *diameter* of a hypercube is large. Several variations to the hypercube have been proposed in the literature to overcome this limitation. An example of such are the *crossed* cube

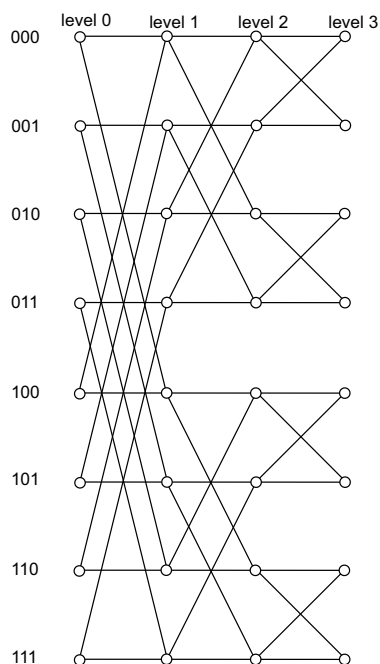


Figure 2.5: A butterfly network topology  $BF_3$ .

[10, 12] and the *folded* cube [11]. Another disadvantage to the hypercube is that the node degree increases with its size. This lead to devising interconnection networks that have similar computational properties as the hypercube but with bounded degree. The *butterfly* is a variation to the hypercube with  $degree = 4$ .

**The Butterfly** The *butterfly* network topology is a bounded-degree derivative of the hypercube [22]. Let  $BF_d$  denote a  $d$ -dimensional butterfly, where  $d \geq 1$ . In graph theory,  $BF_d$  can be represented as a graph having the vertex set  $V = \{(x; i) | x \in V(H_d), 0 \leq i \leq d\}$ . Two vertices  $(x; i)$  and  $(y; j)$  are linked by an edge in  $BF_d$  if and only if  $j = i + 1$  and either:

1.  $x = y$ , or
2.  $x$  differs from  $y$  in precisely the  $j$ th bit.

An example of  $BF_3$  is shown in Figure 2.5

An edge is said to be *straight* if  $x = y$ , otherwise it is called a *cross* edge. For a fixed  $i$ , the vertex  $(x; i)$  is a vertex on level  $i$ . The total number of edges is  $d2^{d+1}$ . The  $BF_d$  has  $d + 1$  levels with  $2^d$  vertices in each level which results in a total number of nodes  $n = (d + 1)2^d$ .

**Properties of the Butterfly** The butterfly has several interesting properties; some of these properties are listed below:

1. Each vertex has a *degree* 2 or 4. This makes the butterfly a bounded-degree network of *degree* 2 or 4.
2. As in the hypercube, the butterfly has a large bisection width. The  $n$ -node butterfly has a *bisection width*  $= \frac{n}{\log n}$ .
3. One of the useful properties of the  $BF_d$  is that the level 0 node in any row is linked to the level  $d$  node in any other row by a unique path of length  $d$ . The path traverses each level exactly once, using the cross edge from level  $i$  to  $i + 1$  if and only if  $x$  and  $y$  differ in the  $(i + 1)$  bit. As a consequence, it is easy to see that the  $n$ -node butterfly has a *diameter*  $= \log n$ .
4. The butterfly and hypercube are very similar to each other in structure. In fact,  $H_d$  can be obtained from  $BF_d$  by merging all the butterfly nodes that are in the same row, and then removing the extra copy of each edge. In effect, the hypercube is just a folded up butterfly.

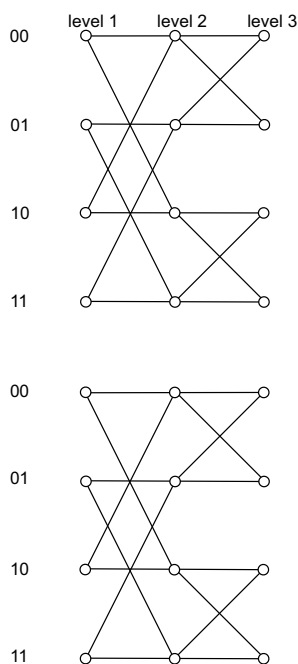


Figure 2.6: Two  $BF_2$  are obtained by eliminating level 0 nodes from  $BF_3$ .

5. The butterfly has a simple recursive structure. This is due to the fact that it is quite similar to the hypercube. Removal of the level 0 nodes from  $BF_d$  will result in two  $BF_{d-1}$ . An example of obtaining two  $BF_2$  by eliminating the level 0 nodes of  $BF_3$  is demonstrated in Figure 2.6. Alternatively, the level  $d$  nodes can be removed which will give a similar result.

## 2.4 NP-Complete Problems

Computational complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem).

The class P consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; the class NP consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine.

A problem is NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem" although it might, in fact, be harder. Examples of NP-hard problems include the *Optimal Communication Spanning Tree* problem, the Hamiltonian cycle problem, and the traveling salesman problem.

A problem which is both NP (verifiable in nondeterministic polynomial time) and NP-hard (any other NP-problem can be translated into this problem) is known as NP-complete. The NP-complete problems are the hardest problems in NP, in the sense that they are the ones most likely not to be in P. The reason is that if you could find a way to solve an NP-complete problem quickly, then you could use that algorithm to solve all NP problems quickly.

## Chapter 3

# ENUMERATING ALL SPANNING TREES OF A GRAPH

Several algorithms for enumerating all possible spanning trees of a given graph currently exist. Many algorithms for solving this problem have been developed and can be classified into two main categories.

The first category of algorithms uses a technique called *backtracking*. This technique is useful for enumerating all the spanning trees of a graph in addition to listing all the kinds of subgraphs (e.g. cycles, and paths). The algorithms introduced by [26, 33] were later redefined by [14]. The redefined algorithm uses  $O(N|V| + |V| + |E|)$  time and  $O(|V| + |E|)$  space, where  $N$  is the number of all spanning trees,  $|V|$  is the number of vertices and  $|E|$  is the number of edges for the graph  $G(V, E)$ . If the spanning trees of a graph are enumerated by outputting all edges of each spanning tree, then this algorithm is optimal in terms of time and space complexities.

The second category of algorithms depends on another technique instead of using backtracking. They depend on finding a new spanning tree by exchanging a pair of edges. Moreover, if all spanning trees of a graph are enu-

merated by outputting only relative changes of edges between spanning trees, then the size of the output can be compressed to  $O(N + |V|)$ . Hence, the total time complexity may be reduced.

An algorithm proposed by Kapoor and Ramesh [19] adopts such a *compact output* and uses  $O(N + |V| + |E|)$  time and  $O(|V| \cdot |E|)$  space. This algorithm is optimal in the sense of time complexity. Another algorithm was proposed by Matsui [23] that uses  $O(N|V| + |V| + |E|)$  time and  $O(|V| + |E|)$  space. This algorithm enumerates all spanning trees explicitly, by applying the *reverse-search* scheme [6]. The reverse search is a scheme for general enumeration problems (see [4, 5]). Shioura and Tamura [30] also proposed an algorithm that uses a compact output and depends on the reverse-search technique. This algorithm has the same time and space complexities as the Kapoor-Ramesh algorithm. However, the Kapoor-Ramesh algorithm and the Shioura-Tamura algorithm, are not efficient in terms of space complexity. That's because both algorithms require  $O(|V| \cdot |E|)$  space.

Later, Uno [34] introduced a new approach for speeding up general enumeration algorithms. This approach relied on manipulating the data structures needed by the different enumeration algorithms. As a result, the space complexity for the Shioura-Tamura algorithm was enhanced and became known as the *Shioura-Tamura-Uno* (STU) algorithm [31]. The STU algorithm has a space complexity  $O(|V|)$ . In the following, a deeper look at the STU algorithm will be given. The focus will be on the mechanism of enumerating all the spanning trees of a given graph. Not much attention will be paid towards manipulating the data structures.

### 3.1 Compact Output

To discuss the STU algorithm we need to first define a compact output. Consider a graph  $G$  (not necessary simple) with the vertex set  $V(G) = \{v_1, v_2, \dots, v_n\}$  and the edge set  $E(G) = \{e_1, e_2, \dots, e_m\}$ , where  $n = |V|$  and  $m = |E|$ . The STU algorithm relies on two types of edge sets, which are known as *fundamental cuts* and *fundamental cycles*. Let  $T$  be a spanning tree of  $G$ , then we can represent  $T$  by its edge set  $E(G)$  of size  $m = |V| - 1$ .

For any edge  $f \in T$ ; the deletion of  $f$  from  $T$  yields two connected components. The fundamental cut associated with  $T$  and  $f$  is defined as the set of edges connecting these components and is denoted by  $Cut(T/f)$ . The fundamental cycle associated with  $T$  and  $g \notin T$  as the set of edges contained in the unique cycle of  $T \cup g$ , this will be denoted as  $Cyc(T \cup g)$ .

By definition,  $T \setminus f \cup g$  is a spanning tree for any  $f \in T$  and any  $g \in Cut(T \setminus f)$ . Similarly, for any  $g \notin T$  and any  $f \in Cyc(T \cup g)$ ;  $T \cup g \setminus f$  is also a spanning tree. These properties are used by the STU algorithm because by using fundamental cuts or cycles, different spanning trees can be created from a given one by only exchanging one edge.

Let  $S(G) = (\tau, A)$ , where  $\tau$  is the set of all spanning trees of  $G$ , and  $A$  consists of all pairs of spanning trees that are obtained from each other by exchanging exactly one edge using some fundamental cut or cycle. The STU algorithm finds all spanning trees of  $G$  by implicitly traversing some spanning tree  $T$  of  $S(G)$ . To output all the  $|V| - 1$  edges for each spanning tree, then  $\theta(\tau \cdot |V|) = \theta(N \cdot |V|)$  time is required. On the other hand, if all the edges for the first spanning tree are output and then only the sequence of exchanged edge pairs of  $G$  (that are obtained by traversing  $T$ ), then this will reduce to  $\theta(\tau + |V|) = \theta(N + |V|)$  time.



From the previous discussion we can see that all the spanning trees for a given graph can be constructed by scanning such a compact output. Therefore, the objective is to find a spanning tree from a current one efficiently in constant time.

## 3.2 Basic ideas

Let us define the total orders over the vertex set  $V(G) = \{v_1, v_2, \dots, v_n\}$  and the edge set  $E(G) = \{e_1, e_2, \dots, e_m\}$  of the graph  $G$  by their indices as  $v_1 < v_2 < \dots < v_n$  and  $e_1 < e_2 < \dots < e_m$ . The smallest vertex  $v_1$  is called the *root*.

For each edge  $e$ , let  $\partial^+$  denote the smaller incident vertex (called the *tail*) and let  $\partial^-$  denote the larger incident vertex (called the *head*). Relative to a spanning tree  $T$  of  $G$ ; if the unique path in  $T$  from the vertex  $v$  to the root  $v_1$  contains a vertex  $u$ , then  $u$  is called an ancestor of  $v$  and  $v$  is a descendant of  $u$ . Similarly, for two edges  $e$  and  $f$  in  $T$ ; we call  $e$  an ancestor of  $f$  and  $f$  a descendant of  $e$  if the unique path in  $T$  from  $f$  to the root  $v_1$  contains  $e$ . A *depth-first-search* spanning tree of  $G$  is a spanning tree which is found by some depth-first search of  $G$ . A depth-first spanning tree is defined as a spanning tree such that for each edge of  $G$ ; its one incidence vertex is an ancestor of the other. There are five main assumptions used by the STU algorithm, which are:

1. Assumption 1:  $T^0$  is a depth-first spanning tree of the graph  $G$ .
2. Assumption 2:  $T^0 = \{e_1, e_2, \dots, e_m\}$ .
3. Assumption 3: Any edge in  $T^0$  is smaller than its proper descendants.
4. Assumption 4: Each vertex  $v$  is smaller than its proper descendants relative to  $T^0$ .

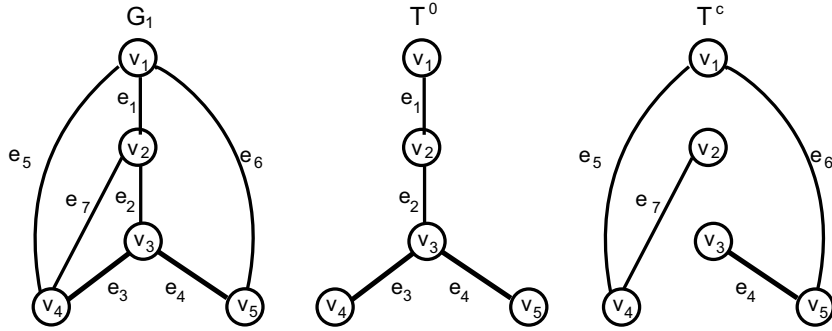


Figure 3.1: Graph  $G_1$  and two spanning trees  $T^0$  and  $T^c$ .

5. Assumption 5: For any two edges  $e$  and  $f \notin T^0$ ; if  $e < f$ , then  $\partial^+ e \leq \partial^+ f$ .

By applying Tarjan's depth first search [33], it is easy to generate a depth-first-search spanning tree  $T^0$  of  $G$  with its vertices and edges sorted to satisfy the above assumptions. This can be performed in  $O(|V| + |E|)$  time. It is important to note that Assumptions 1, 2 and 3 are sufficient for the correctness of the algorithm. However, Assumptions 4 and 5 are required for an efficient implementation.

For any nonempty subset  $S$  of the edge-set  $E(G)$ , let  $Min(S)$  denotes the smallest edge in  $S$ . For convenience, assume that  $Min(0) = e_m$ . The STU algorithm depends on the following lemmas for enumerating all the spanning trees for a given graph:

**Lemma 1** Under Assumptions 1 and 3, for any spanning tree  $T^c \neq T^0$ ; if  $f = Min(T^0 \setminus T^c)$ , then  $Cyc(T^c \cup f) \cap Cut(T^0 \setminus f) \setminus f$  contains exactly one edge.

Given a spanning tree  $T^c \neq T^0$  and the edge  $f = Min(T^0 \setminus T^c)$ ; let  $g$  be the unique edge in  $Cyc(T^c \cup f) \cap Cut(T^0 \setminus f)$ . Clearly,  $T^p = T^c \cup f \setminus g$  is a spanning tree. The tree  $T^p$  is called the parent of  $T^c$ , and  $T^c$  is a child of  $T^p$ .

For example, consider the graph  $G_1$  displayed in Figure 3.1. Two spanning

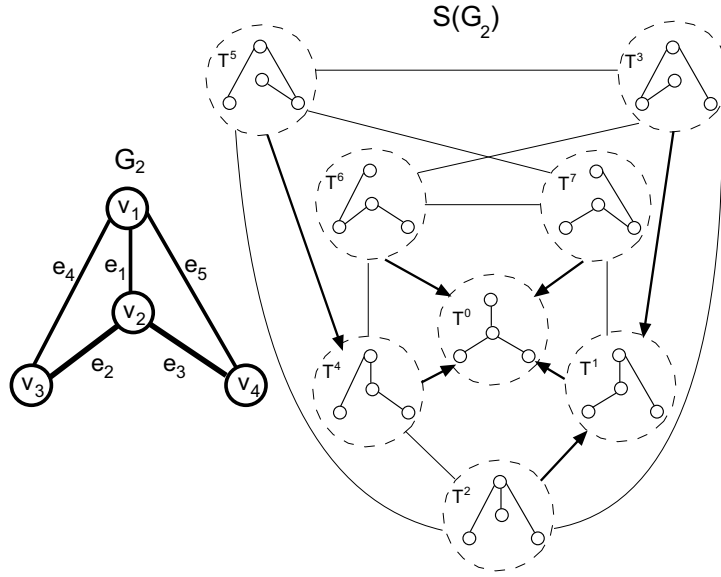


Figure 3.2: Spanning tree *child-parent* relationships in  $S(G_2)$ .

trees are displayed which are  $T^0 = \{e_1, e_2, e_3, e_4\}$  and  $T^c = \{e_4, e_5, e_6, e_7\}$ . Now,

$$\begin{aligned} f &= \text{Min}\{e_1, e_2, e_3\} = e_1 \\ \text{Cyc}(T^c \cup f) &= \{e_1, e_5, e_7\} \\ \text{Cut}(T^0 \setminus f) &= \{e_1, e_5, e_6\} \end{aligned}$$

Therefore,  $\text{Cyc}(T^c \cup f) \cap \text{Cut}(T^0 \setminus f) = \{e_5\}$ .

Lemma 1 guarantees that each spanning tree other than  $T^0$  has a unique parent. Since  $|T^p \cap T^0| = |T^c \cap T^0| + 1$  holds, then  $T^0$  is the ancestor of all spanning trees. Figure 3.2 shows a graph  $G_2$  and all the spanning tree *child-parent* relationships. The arrows point from a child spanning tree to its parent.

**Lemma 2** Let  $T^p$  be an arbitrary spanning tree of  $G$ ; and let  $f$  and  $g$  be two distinct edges. Under Assumptions 1, 2, and 3,  $T^c = T^p \setminus f \cup g$  is a child of  $T^p$ , if and only if  $f$  and  $g$  satisfy the following conditions:  $f < \text{Min}(T^0 \setminus T^p)$  and  $g \in \text{Cut}(T^p \setminus f) \cap \text{Cut}(T^0 \setminus f) \setminus f$ .

Suppose that  $e_k$  is the largest edge less than  $\text{Min}(T^0 \setminus T^p)$ . Then, from Lemma 2, all the children of  $T^p$  can be found if the edge sets  $\text{Cut}(T^p \setminus e_j) \cap \text{Cut}(T^0 \setminus e_j)$  for  $j = 1, 2, \dots, k$  are known.

For example, consider the graph  $G_1$  displayed in Figure 3.1. Let  $T^p = T^1$ , then  $e_1$  and  $e_2$  are the only edges smaller than  $\text{Min}(T^0 \setminus T^1) = e_3$ . This leads to the following:

$$\begin{aligned} \text{Cut}(T^1 \setminus e_2) \cap \text{Cut}(T^0 \setminus e_2) \setminus e_2 &= \{e_2, e_4\} \cap \{e_2, e_4\} \setminus e_2 = \{e_4\} \\ \text{Cut}(T^1 \setminus e_1) \cap \text{Cut}(T^0 \setminus e_1) \setminus e_1 &= \{e_1, e_3, e_4\} \cap \{e_1, e_4, e_5\} \setminus e_1 = \{e_4\} \end{aligned}$$

Therefore,  $T^1$  has two children:  $T^1 \setminus e_2 \cup e_4$  and  $T^1 \setminus e_1 \cup e_4$ . The STU-algorithm is shown in the next page. The algorithm uses  $O(N + |V| + |E|)$  time and  $O(|V|)$  space.

In the STU-algorithm, the *findchildren* procedure is used to find all the children for each spanning tree. The *findchildren* procedure is called with the two arguments  $T^p$  and  $k$ . It then finds all the spanning trees for  $T^p$  not containing the edge  $e_k$ . Whenever a child spanning tree  $T^c$  is found, it recursively calls itself again to find all the children for  $T^c$ . The arguments at this stage are set to  $T^c$  and  $k - 1$ . That is because if  $k > 1$ , then  $e_{k-1}$  becomes the largest edge less than  $\text{Min}(T^0 \setminus T^c)$ .

**Algorithm STU**

**input:** a graph  $G(V, E)$  with a vertex set  $V(G) = \{v_1, v_2, \dots, v_n\}$  and an edge set  $E(G) = \{e_1, e_2, \dots, e_m\}$ , where  $n = |V|$  and  $m = |E|$ .

**output:** The set of edges for each spanning tree of  $G$ .

1. find a depth-first spanning tree  $T^0$  for  $G$ .
2. sort vertices and edges to satisfy Assumptions 2, 3, 4, and 5.
3. **output**  $T^0$
4. findchildren( $T^0, |V| - 1$ )

**Procedure** findchildren( $T^p, k$ )

1. **if**  $k \leq 0$  **then return**
2. **for** each  $g \in \text{Cut}(T^p \setminus e_k) \cap \text{Cut}(T^0 \setminus e_k)$
3.      $T^c = T^p \setminus e_k \cup g$
4.     **output**  $T^c$
5.     findchildren( $T^c, k - 1$ )
6. **end for**
7. findchildren( $T^p, k - 1$ )

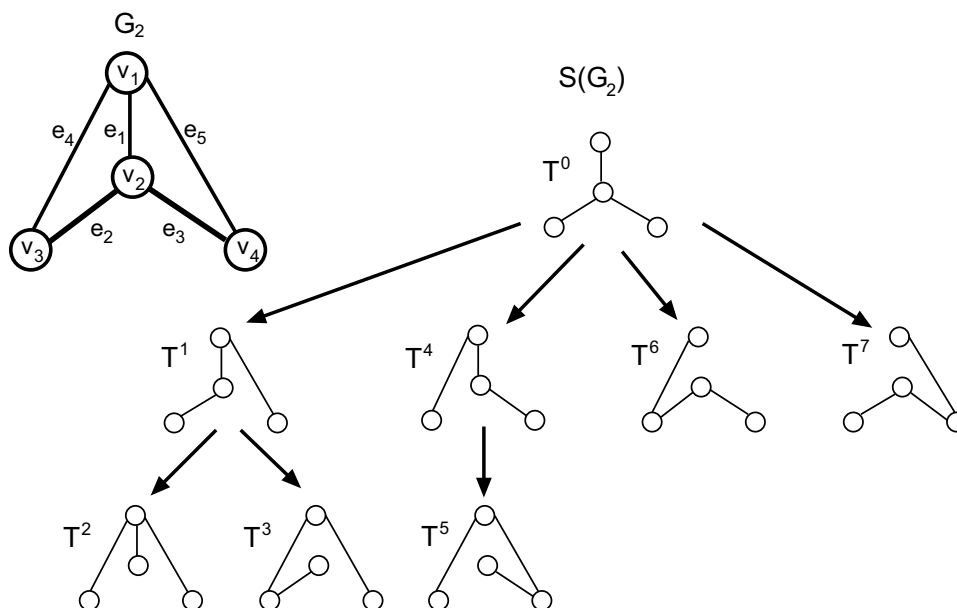


Figure 3.3: Enumeration of all spanning trees for  $G_2$ .

If all the children for  $T^p$  not containing  $e_k$  have been found, then it recursively calls itself again to find all the children of  $T^p$  not containing  $e_{k-1}$ . The arguments in this case are  $T^p$  and  $k - 1$ . The *findchildren* procedure is initially invoked with arguments  $T^0$  and  $V - 1$  which results in enumerating all the spanning trees for the input graph. An example of the output generated by the STU-algorithm is displayed in Figure 3.3.

**Lemma 3** *Algorithm STU outputs each spanning tree exactly once.*

## Chapter 4

### THE OPTIMUM

### COMMUNICATION SPANNING

### TREE PROBLEM

There exists a significant amount of literature interested in the design of optimum communication and transportation networks that satisfy a given set of requirements. Many different variants, with or without additional constraints have been studied, which resulted in either exact solutions or heuristics [8, 20]. Several constrained minimum spanning tree problems currently exist. An example of some are the *optimum communication spanning tree* (OCST) problem, the degree-constrained minimum spanning tree problem, the minimum steiner tree problem, or the capacitated minimum spanning tree problem [27].

The OCST problem (also known as the minimum communication cost problem) was first introduced by Hu [17]. The problem states that a tree that connects all vertices for a complete graph has to be found. The spanning tree found must satisfy the communication requirements needed by the vertices with a minimum total cost.

## 4.1 Problem Description

The communication cost of a spanning tree  $T$  for a given graph  $G(V, E)$  is defined as follows. Define a pair of vertices  $x$  and  $y \in V(G)$  where  $x \neq y$ . The distance  $d_{x,y}$  is the distance between vertices  $x$  and  $y$  restricted on  $T$ . The communication requirement  $r_{x,y}$  for the pair  $x$  and  $y$  is provided. There is a unique path in  $T$  between  $x$  and  $y$ . The *distance* of the path is the sum of distances of edges in the path. The *communication cost* for the pair  $x$  and  $y$  is  $r_{x,y}$  multiplied by the distance of the path. Summing over all  $\binom{n}{2}$  pairs of vertices, we have the communication cost for  $T$ .

In order to calculate the optimum communication cost for a graph  $G$  the following objective is required:

$$\min_{T \in \mathfrak{S}} \left[ \sum_{(x,y) \in V} r_{x,y} d_{x,y} \right] \quad (4.1)$$

Where  $\mathfrak{S}$  is the set of all possible spanning trees for the graph  $V(G)$ . In the case that  $V(G)$  is a complete graph, the number of possible spanning trees is  $|\mathfrak{S}| = n^{n-2}$  as shown in [9].

The goal is to find a spanning tree  $T$  such that the average delay of communicating between any pair using  $T$  is minimized. The delay between a pair of vertices is the sum of the delays of the edges in the path between them in  $T$ . Minimizing the average delay is equivalent to minimizing the total delay between all pairs of vertices using  $T$ .



## 4.2 Applications of the OCST problem

Topology design problems for broad band communication networks has attracted the interest of many researchers in this area. The problem is to effectively design a network such that the constraints are met and the objectives are optimized. This problem is extremely important for many real world applications. An example of such are telecommunications, computer networking, and oil & gas pipelines.

In some cases, the communication network systems are designed with fiber optic cable. Fiber optic cable has many advantages such as having a huge bandwidth, low signal attenuation, low signal distortion, low power requirement, and small space requirement. However, the disadvantage to fiber optic cable is that it is expensive. Therefore, it is desirable to design the network architecture as a spanning tree.

It is important to find the best layout of components when designing the topology of a communication network system. A best layout can be reached by optimizing a performance criteria, such as cost, message delay, traffic and reliability. The performance criteria of these systems are important and rely heavily on the network topology. The OCST problem can also be considered as a performance criteria with the objective of optimizing the total communication cost of the network topology.

Besides the obvious application of the OCST problem to topology design problems. The OCST problem has applications to other disciplines as well, such as in the field of computational biology. Computational biology relies heavily on the methodologies of *multiple sequence alignments* (MSA). MSA is important for detecting patterns common to a set of genetic sequences.

Let  $S = \{s_1, s_2, \dots, s_k\}$  be a set of strings, where  $k > 1$ . The number of

characters for each string  $s_i \in S$  is variable for  $1 \leq i \leq k$ . A multiple alignment of  $S$  is performed by inserting gaps in to the strings  $s_i \in S$  and then arranging the strings into a  $(k \times l)$  matrix  $M$ . Each  $s_i$  is input into row  $i$  of  $M$  with some blanks possibly inserted between the characters. The following is an example of an alignment of the three strings ATTCGAC, TTCCGTC, and ATCGTC:

```

A T T _ C G A _ C
_ T T C C G _ T C
A _ T _ C G _ T C

```

To identify common patterns amongst several sequences, it is required to arrange the strings in  $M$  such that all the characters for every column match, as much as possible. Therefore, the MSA problem has typically been formalized as an optimization problem in which some explicit objective function is minimized or maximized. One of the most popular objective functions for the MSA problem is a generalization of the *pairwise-alignment* problem [35].

In the pairwise-alignment problem, two sequences need to be aligned optimally. Therefore, a minimum mutation path between the two sequences needs to be found. In this problem, the costs for inserting, deleting and substituting one character of the alphabet for another are given. It is required to find a minimum-cost mutation path from one sequence to the other. The cost of this path is the edit distance between them.

The generalization of the pairwise-alignment problem to MSA leads to the *sum-of-pairs* objective. The sum-of-pairs objective for MSA is to minimize the sum, over all pairs of sequences, of the pairwise distance between them in the alignment. The distance of two sequences in an alignment with  $l$  columns is obtained by adding up the costs of the pairs of characters appearing at positions  $1, 2, \dots, l$ .

### 4.3 Existing Approaches

Hu did not tackle the original OCST problem but chose to solve two special cases of the problem instead. The first case is when the distances for all the edges are equal to one ( $d_{x,y} = 1$ ), while the requirements are arbitrary. This is known as the *optimum requirement spanning tree* (ORST) problem. Hu showed that the ORST problem is solvable in polynomial time using the Gomory-Hu spanning tree algorithm [15, 17].

The second case is when all the requirements are equal to one ( $r_{x,y} = 1$ ) while the distances are arbitrary. This is known as the *optimum distance spanning tree* (ODST) problem. Hu derived a weak condition under which the optimum solution is a star if the distances satisfy the triangular inequality. It was proven in [18] that only cases of the ODST problem, where the distances satisfy the triangular inequality, can be solved in polynomial time. Hence, all other versions are NP-hard.

Wong [38] developed a 2-approximation algorithm for the ODST problem. This approximation algorithm considers the shortest path tree rooted at every vertex in turn, and picks the one with the optimum communication cost. For graphs with metric distances obeying the triangle inequality, every shortest path tree is isomorphic to a star. Wong proved that spanning trees generated by this algorithm have a cost at most twice the total cost of the graph itself.

Later, [40, 39] introduced another polynomial-time approximation scheme for the ODST problem. This approach was influenced by the observation, noted earlier by Hu, regarding the optimum solution being a star. The authors demonstrate how simple generalizations of stars are sufficient to guarantee any desirable accuracy in approximating optimum spanning trees.

The trees generated by [40, 39] approximate an optimum cost spanning tree by a restricted class of trees called  $k$ -stars. For any fixed size  $k \geq 1$ , a  $k$ -star is a tree in which at most  $k$  vertices have degree greater than one. The authors also rely on special subtrees called *separators*. The separators are used to break a tree into sufficiently small components.

For a given accuracy parameter  $\epsilon$ , all  $\lceil \frac{2}{\epsilon} - 1 \rceil$ -stars are considered and the minimum cost spanning tree is output. As a result, this algorithm is a  $(1 + \epsilon)$ -approximate solution. The running time for this algorithm is  $O(n^{2\lceil \frac{2}{\epsilon} \rceil - 2})$ . It is important to note that this approximation algorithm is bound to the optimal spanning tree cost. This is opposed to Wong's proof that shows that his approximation algorithm is bound to twice the communication cost of the input graph.

Peleg [25] showed that the the ODST problem is reducible to a problem called the *minimum average stretch spanning tree* (MAST) problem. Since both problems are equivalent to each other, then the approximation algorithms for the MAST problem can be used for the ODST problem. In the MAST problem, which was introduced in [2], a graph  $G$  and a distance weights matrix  $W$  are given and a spanning tree  $T$  has to be found that minimizes the average stretch of the edges.

Formally, let  $G(V, E)$  be a connected graph on  $V$ , with edge set  $E$ , distance weights  $w_{u,v} \geq 1$  and multiplicities  $m_{u,v} \geq 0$ , for every edge  $(u, v) \in E$ . Let  $M = \sum_{(u,v) \in E} m_{u,v}$ . For a spanning tree  $T$  of  $G$ , the stretch over the vertex pair  $u$  and  $v$  is defined as  $\frac{d_{u,v}}{w_{u,v}}$ , where  $d_{u,v}$  is the distance between the vertices  $u$  and  $v$  restricted on  $T$ . The average stretch  $\bar{S}(T)$  of  $T$  is:

$$\bar{S}(T) = \frac{1}{M} \sum_{(u,v) \in E} m_{u,v} \cdot \frac{d_{u,v}}{w_{u,v}}$$

The randomized approximation algorithm given by [2] constructs a spanning tree with average stretch  $O(2^{\sqrt{\log n}})$ . This implies an approximation algorithm with the same ratio for the ODST problem.

Several *genetic algorithms* (GA) have also been proposed to solve the ODST problem. Such methods do not construct a tree according to an algorithmic method, but search through the search space consisting of all possible trees. It is well known that the proper choice of a representation is crucial for the performance of metaheuristics. A representation determines how trees are encoded such that search operators can be applied. Commonly, trees are encoded as vectors or lists of strings where different strings encode different trees.

One of the first GA approaches for the ODST problem was presented by Palmer [24]. He recognized that the design of a proper tree representation is crucial for the performance of the GA. Palmer compared different types of problem representations for trees and developed a new representation, the *link and node biased* (LNB) encoding.

It is stated in [20] that the cost of a spanning tree strongly depends on the distances of the edges used from the graph. In other words, spanning trees that are composed of low distance edges tend to have on average lower overall costs. When focusing on 2-dimensional grids (resulting in Euclidean distance weights), the weights of edges near the gravity center of a graph are lower than the weights of edges that are far away from the gravity center. Therefore, it is useful to run more traffic over the vertices near the gravity center of a tree rather than over vertices at the edge of the tree. Consequently, vertices are characterized as either interior (some traffic only transits) or leaves (all traffic terminates). The more important an edge is and the more transit traffic that crosses one of the two vertices, the higher is on average the degree of the vertex. Vertices near the gravity center tend to have higher degrees than leaf

vertices.

This observation inspired Palmer in designing the LNB encoding. The LNB encoding considers the relative importance of vertices in a tree and the more important a node is, the more traffic transits over it. The LNB encoding is an illustrative example of how properties of good solutions for the ODST problem can be used for the design of a high-quality representation. The GA using the LNB encoding showed good results in comparison to a greedy star search heuristic ([24], chapter 5).

A more recent GA is the work of [28, 27]. A statistical analysis was performed on the properties of optimal solutions for randomly generated ODST problems. They compared the average distances of randomly created trees towards the *minimum spanning tree* (MST), to the average distances of the optimal solutions towards the MST. The results showed that the average distance between the optimal solution spanning tree and the MST is significantly smaller than the average distance between a randomly created tree and the MST. Therefore, optimal solutions for the ODST problem are biased towards the MST. The authors proposed a new representation called the *link biased* (LB) encoding. The LB encoding makes use of the problem specific property of the ODST problem and encodes solutions similar to the MST with higher probability.

Another special cases for the OCST is the work of [1]. The authors proposed polynomial time algorithms for two constrained cases for the general OCST problem. The first constrained case is when a spanning tree must contain specified vertices as leaves. The second constrained case is when the spanning tree has to contain a specified set of edges from the input graph. It is shown that there exists at most  $(n - 1)$  optimum spanning trees that can be constructed in  $O(n^4)$ .

Table 4.1: Summary of algorithms for solving the Optimum Distance Communication Spanning Tree problem.

Author	Resource	Algorithm Type	Complexity / Ratio	Description of Algorithm
Hu (1974)	[17]	Poly.	Poly.	Solution is a star only if costs obey the triangle inequality
Wong (1980)	[38]	Deter. Approx.	2	Considered the shortest path tree rooted at every vertex
Wu et al. (1998)	[40, 39]	Rand. Approx.	$(1 + \epsilon)$	Restricted the solution to $k$ -stars
Peleg (1997)	[25]	Rand. Approx.	$O(2^{\sqrt{\log n}})$	Reduced the MAST to the ODST
Palmer (1994)	[24]	GA	N/A	Designed the LNB-encoding (biased towards Gravity-Center)
Rothlauf et al. (2004)	[28, 27]	GA	N/A	Designed the LB-encoding (biased towards MST)

Table 4.1 gives a summary of the approaches that are used for solving the ODST problem.

# Chapter 5

## THESIS WORK

In this thesis work, we attempt to find an efficient randomized algorithm to solve a special case of the ODST problem. This special case is different than the original ODST problem in two ways. First, rather than limiting the ODST problem to finding solutions for complete graphs only. We propose to solve the ODST problem for general graphs. Second, this special case restricts the costs of the edges to be constant and equal to one. This is in contrast to the original ODST problem where the costs of the edges are arbitrary.

An experimental study has been conducted to measure the performance of the proposed algorithm using several special graphs. The graphs selected for the experimental study are: three randomly generated graphs, the hypercube and butterfly network topologies. The experiments involve finding an optimum solution for the previously mentioned special graphs.



## 5.1 Problem Description

Let  $T$  be a spanning tree for a given graph  $G(V, E)$ . Then, the special case of the ODST problem, that we consider, can be formally defined as the following:

- Minimum Congestion Cost in  $G(V, E)$ , denoted by  $\psi(G)$  is the sum of all distances over all pairs of vertices in  $G$ . That is,

$$\psi(G) = \sum_{u,v \in V(G)} d_{u,v} \quad (5.1)$$

where  $d_{u,v}$  is the distance between vertices  $u$  and  $v$  restricted on  $T$ .

- Minimum Stretch Cost in  $G(V, E)$ , denoted by  $\phi(G)$  is the same as  $\psi(G)$ , except that the sum of edges is taken over those pairs of vertices that are connected by an edge  $e \in E(G)$ . That is,

$$\phi(G) = \sum_{(u,v) \in E(G)} d_{u,v} \quad (5.2)$$

where, again,  $d_{u,v}$  is the distance between vertices  $u$  and  $v$  restricted on  $T$ .

The maximum number of routes that go through an edge is formally known as the *congestion*. This definition can be further extended to spanning trees. The congestion cost is referred to in biochemistry as the Wiener index [36], which has been extensively studied for more than half a century. The congestion cost of  $T$  can be computed by including all the pairs of vertices for  $G$  in the computation; or by including only a subset of the pairs of vertices for  $G$  in the computation. When all pairs of vertices are included in the computation then this is referred to as the *congestion cost* for  $T$ . However, if only those pairs of vertices that share an edge in  $G$  are included in the computation, then we will refer to this as the *stretch cost* for  $T$ .

The minimum stretch cost problem has been formulated based on a problem called the *minimum average stretch spanning tree* (MAST) problem [2]. In the MAST problem, a graph  $G$  and a distance weights matrix  $W$  are provided. The objective is to find a spanning tree  $T$  that minimizes the average stretch of the edges.

Formally, let  $G(V, E)$  be a connected graph on  $V$ , with edge set  $E$ , distance weights  $w_{u,v} \geq 1$  and multiplicities  $m_{u,v} \geq 0$ , for every edge  $(u, v) \in E$ . Let  $M = \sum_{(u,v) \in E} m_{u,v}$ . For a spanning tree  $T$  of  $G$ , the stretch over the vertex pair  $u$  and  $v$  is defined as  $\frac{d_{u,v}}{w_{u,v}}$ , where  $d_{u,v}$  is the distance between the vertices  $u$  and  $v$  restricted on  $T$ . The average stretch  $\bar{S}(T)$  of  $T$  is:

$$\bar{S}(T) = \frac{1}{M} \sum_{(u,v) \in E} m_{u,v} \cdot \frac{d_{u,v}}{w_{u,v}}$$

(Note that the term *optimum congestion cost* problem will be used to refer to both the minimum congestion cost problem and the minimum stretch cost problem unless otherwise specified).

In this thesis, we propose an algorithm that can find a spanning tree for a given graph randomly. The spanning trees found by this algorithm have properties that may minimize their congestion costs. We believe that the proposed algorithm can be used to find the optimum solutions for the ODST problem for general graphs. To justify our claim, an experimental study has been conducted on three randomly generated graphs in addition to the hypercube and butterfly topologies to a certain dimension. In the next section, the proposed algorithm will be explained. The experiment setup will be discussed and the results of the experiments will be listed.

## 5.2 The Proposed Algorithm

The efficiency of a specific optimization method decreases with the number of problems that should be solved. In [37], the authors state that if an optimization method is designed to solve all possible optimization problems that can be designed on a specific search space, then this method does not perform on average better than random search. However, optimization methods can perform better than random search if they focus only on a subset of all possible optimization problems.

In the the optimum congestion cost problem, a spanning tree  $T$  that minimizes the total congestion cost for a given graph  $G$  is required. Let  $T^r$  be a *rooted shortest-path* spanning tree for the graph  $G$  that is rooted at the vertex  $r \in V(G)$ .  $T^r$  can be obtained by performing a *breadth-first search* on the graph  $G$ . A well-known property for  $T^r$  is that the path from  $r$  to any other vertex  $u \in V(G)$  has the least number of edges. Therefore,  $T^r$  minimizes the congestion cost between  $r$  and any other vertex  $u \in V(G)$ .

In our approach, we attempt to randomly generate the spanning trees for a given graph. However, unlike other methods that randomly generate spanning trees with equal probability. The proposed algorithm generates spanning trees with properties similar to the shortest-path spanning trees with higher probability. The spanning trees generated are grown starting from a given vertex in a fashion similar to Prim's algorithm for finding a minimum spanning tree.

Initially, the proposed algorithm starts by randomly selecting a vertex  $v \in V(G)$  and adds it to the set  $SelV$ , which is initially empty. Next, a vertex  $u \in V(G)$  that is adjacent to  $v$  is randomly selected and added to  $SelV$ . The edge  $(u, v)$  is selected as an edge for the spanning tree  $T$  that is being

constructed. The algorithm then proceeds to grow  $T$  by randomly picking a vertex  $x \in SelV$  and then selecting a vertex  $w \in V(G)$  that is adjacent to  $x$ . If  $w \notin SelV$ , then it is added to  $SelV$  and the edge  $(x, w)$  is selected as an edge for  $T$ . However, if  $w \in SelV$ , then this means that the edge  $(x, w)$  will introduce a cycle in  $T$ , and hence will not be considered. The algorithm continues to grow  $T$  by randomly selecting the vertices that are adjacent to  $SelV$  until  $T$  is fully constructed. We will refer to this method as the **RandomSpan** algorithm. The algorithm is shown more formally in the next page.

The RandomSpan algorithm randomly constructs a spanning tree in a fashion similar to breadth-first search. In breadth-first search, a vertex is first visited then the vertices that are adjacent to it are visited next. In the RandomSpan algorithm, whenever a vertex is visited, it is added to the set  $SelV$ . The next vertex to be visited will be any vertex that is adjacent to one of the vertices in  $SelV$ . The RandomSpan algorithm is similar to breadth-first search in the sense that it randomly chooses a vertex that is *adjacent* to a set of vertices. Therefore, the spanning trees generated by the RandomSpan algorithm are biased towards shortest-path spanning trees with higher probability.

**Algorithm RandomSpan**

**input:** a graph  $G(V, E)$  with a vertex set  $V(G) = \{1, 2, \dots, n\}$  and an edge set  $E(G) = \{e_1, e_2, \dots, e_m\}$ , where  $n = |V|$  and  $m = |E|$ . Let  $\gamma(v)$  be all the vertices that are adjacent to  $v$ .

**output:** The set of edges  $T$  for a spanning tree of  $G$ .

1.  $T \leftarrow \{\}; SelV \leftarrow \{\};$
2. randomly pick  $v \in V(G)$  {pick a vertex to grow the spanning tree}
3.  $SelV \leftarrow SelV \cup \{v\}$  {add  $v$  to the set of selected vertices}
4. **while** ( $|T| < (n - 1)$ )
5.     randomly pick  $v \in SelV$
6.     randomly pick  $w \in \gamma(v)$
7.     **if** ( $w \notin SelV$ )
8.          $T \leftarrow T \cup \{(w, v)\}$
9.          $SelV \leftarrow SelV \cup \{w\}$
10.     **end if**
11. **end while**
12. **output** the set  $T$

One advantage of the RandomSpan algorithm is that it is a simple, yet powerful, algorithm. However, a quick analysis of the RandomSpan algorithm reveals that lots of time might be spent in step 7. That is because it might be the case that  $w \in SelV$  for a large number of consecutive iterations. The probability of finding a new vertex is inversely proportional to the size of the current size of  $T$ . When  $|T|$  is large, the probability of finding a new vertex approaches  $\frac{1}{n}$ . This means, the expected number of iterations before a successful hit is  $O(n)$ . As a result, the running time of the algorithm is  $n \cdot O(n) = O(n^2)$ .

However, this anomaly can be overcome if all the edges for a vertex that has been randomly selected are stored. This is essentially what we did to improve the algorithm's performance. Initially, the **ContractedRandomSpan** algorithm has two empty sets  $ConV$  and  $ConE$ . The algorithm starts by randomly selecting a vertex  $v \in V(G)$  and adds (contracts) it to the set  $ConV$ . Next, all the edges that are connected to  $v$  are added to  $ConE$ . The algorithm then proceeds to grow  $T$  by randomly picking an edge  $(x, y) \in ConE$  for  $x, y \in V(G)$ . If one of the vertices  $x$  or  $y \notin ConV$ , then this vertex is added to  $ConV$  and the edge  $(x, y)$  will be added as an edge for  $T$ . Otherwise, the edge is discarded and the algorithm proceeds to select another edge from  $ConE$ . It is important to note that it is never the case that both  $x$  and  $y \notin ConV$ . That is because an edge is always added to  $ConE$  when at least one of the vertices connected to it  $\in ConV$ . However, it is possible to have both  $x$  and  $y \in ConV$ . In this case, the edge  $(x, y)$  should be discarded from  $ConE$  because it will introduce a cycle in  $T$ . The ContractedRandomSpan algorithm is shown more formally in the next page.

Let us analyze the ContractedRandomSpan algorithm by focusing on the for loop from step 7 to 14. In the worst case, step 8 will have to pick all the edges, and is either added to  $T$  or is discarded. This step will take at most  $O(|E|)$  time. Step 9 & 13 can be performed in constant time. The for loop in step 10-12, takes at most  $O(2 \cdot |E|)$  time. That is because an edge is examined at most twice, once for each time one of the vertices it is connected to is contracted. Therefore, the ContractedRandomSpan algorithm takes at most  $O(|E|)$  time.

The ContractedRandomSpan algorithm is much more efficient than the RandomSpan algorithm in terms of the number of iterations required to find a new edge for  $T$ . As we have seen, the RandomSpan algorithm may spend a long time in step 7. On the other hand, the ContractedRandomSpan algorithm almost always finds a new edge for  $T$  in each iteration.

Figure 5.2 shows a comparison between the performance of the RandomSpan algorithm and the ContractedRandomSpan algorithm when trying to find a spanning tree for the hypercube and butterfly topologies up to dimension = 10. The results have been obtained as an average of 10 runs for each dimension. It is evident that the ContractedRandomSpan algorithm is much more efficient than the RandomSpan algorithm since it requires a fewer calls to the random function.

### Algorithm ContractedRandomSpan

**input:** a graph  $G(V, E)$  with a vertex set  $V(G) = \{1, 2, \dots, n\}$  and an edge set  $E(G) = \{e_1, e_2, \dots, e_m\}$ , where  $n = |V|$  and  $m = |E|$ .  $\gamma(v)$  means the set of vertices that are adjacent to  $v$  and  $\notin ConV$ , where  $ConV$  is the set of contracted vertices.

**output:** The set of edges  $T$  for a spanning tree of  $G$ .

1.  $T \leftarrow \{\}; ConV \leftarrow \{\}; ConE \leftarrow \{\};$
2. randomly pick  $v \in V(G)$  {pick a vertex to grow the spanning tree}
3.  $ConV \leftarrow ConV \cup \{v\}$  {add  $v$  to the set of contracted vertices }
4. **for each vertex**  $w \in \gamma(v)$
5.      $ConE \leftarrow ConE \cup \{(w, v)\}$
6. **end for**
7. **for** ( $i \leftarrow 1$  to  $(n - 1)$ )
8.     randomly pick  $(w, v) \in ConE$  and  $w \notin ConV, v \in ConV$
9.      $T \leftarrow T \cup \{(w, v)\}$
10.    **for each vertex**  $u \in \gamma(w)$
11.        $ConE \leftarrow ConE \cup \{(w, u)\}$
12.    **end for**
13.      $ConV \leftarrow ConV \cup \{w\}$
14. **end for**
15. **output** the set  $T$



### 5.3 Experiment Setup

In this section, we investigate the congestion cost of spanning trees for several special graphs. A spanning tree can be generated randomly by using the ContractedRandomSpan algorithm. Therefore, we can use this algorithm to generate several spanning trees and then compute their corresponding congestion costs. We conjecture that the spanning trees found with the lowest congestion cost are near-optimum solutions. Clearly, the quality of these results will improve as the number of spanning trees generated increases. By adapting such a technique then the search space of available solutions can be scoped efficiently.

Three graphs  $G_1$ ,  $G_2$  and  $G_3$ , with  $V(G_i) = 10$  for  $i = 1, 2$  and  $3$ , have been randomly generated to test our proposed algorithm. See Figure 5.2 for an illustration of these graphs. The experiments also include finding optimum solutions for the hypercube ( $dimension = 1, 2 \dots, 10$ ) and the butterfly ( $dimension = 1, 2 \dots, 8$ ). Table 5.1 shows the number of possible spanning trees for  $G_1$ ,  $G_2$  and  $G_3$ . The number of possible spanning trees for  $H_1, H_2, \dots, H_{10}$  and  $BF_1, BF_2, \dots, BF_8$  are shown in Table 5.2 and Table 5.3, respectively.

In order to measure the performance of the ContractedRandomSpan al-

Table 5.1: The number of possible spanning trees for the randomly generated graphs  $G_1$ ,  $G_2$  and  $G_3$ .

Graph	# of spanning trees
$G_1$	379185
$G_2$	1212519
$G_3$	823415

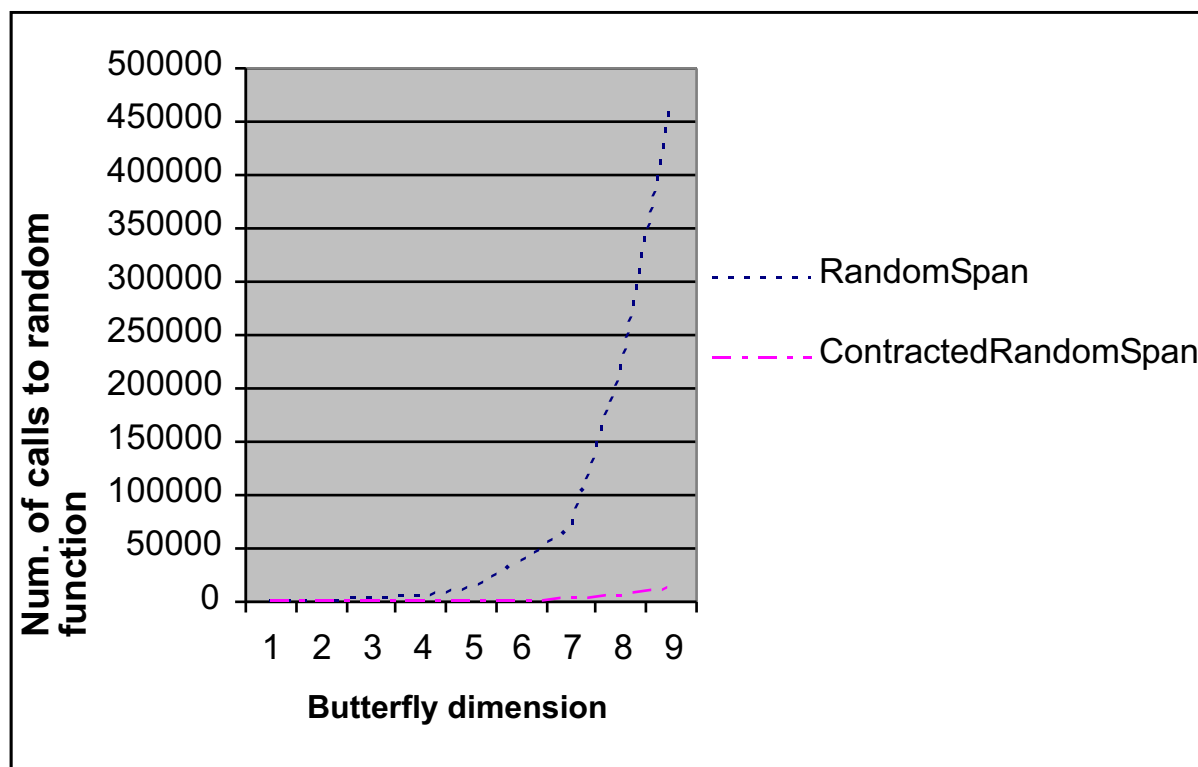
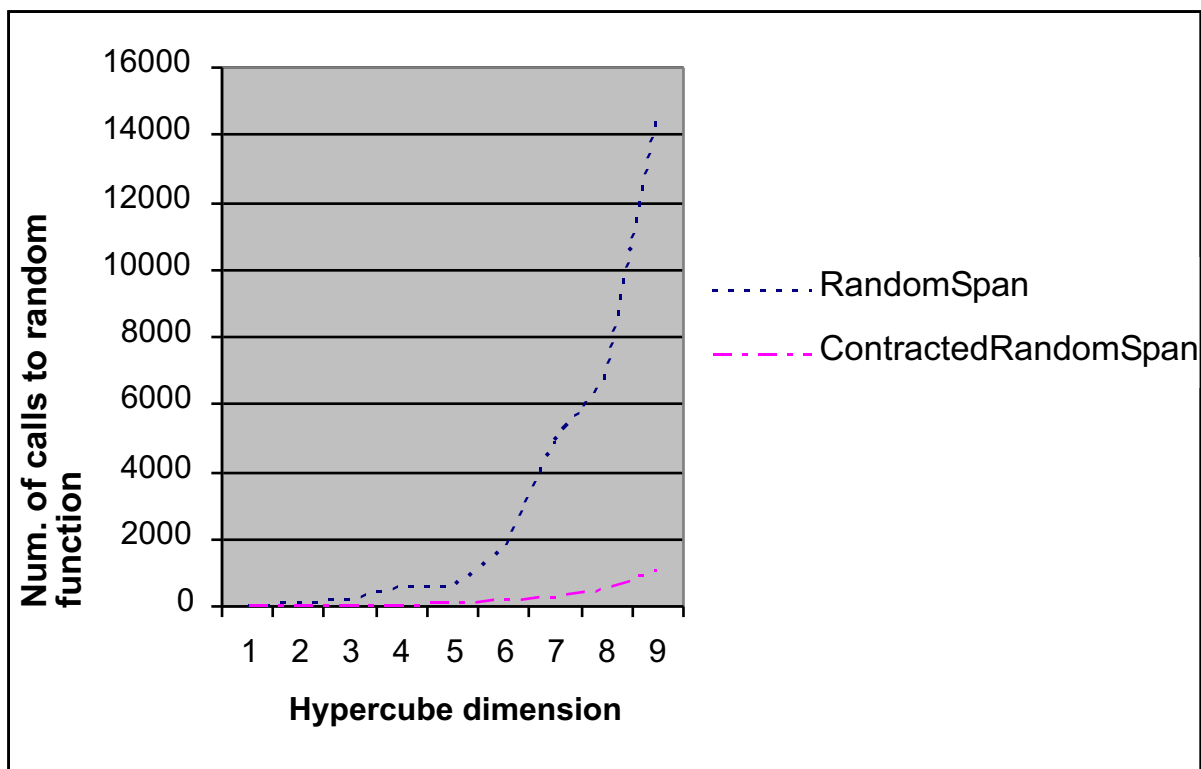


Figure 5.1: A comparison between the RandomSpan algorithm and the ContractedRandomSpan algorithm when finding a spanning tree for the hypercube and butterfly topologies up to dimension = 10.

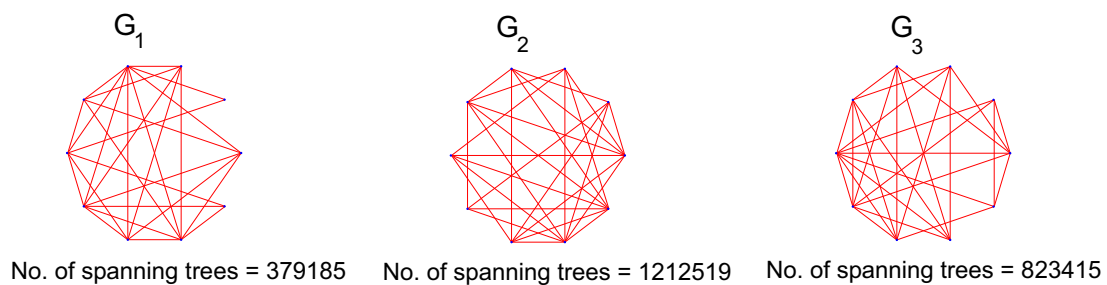


Figure 5.2: The graphs  $G_1$ ,  $G_2$  and  $G_3$  that were randomly generated for the experiments.

Table 5.2: The number of possible spanning trees for the hypercube (dimension = 1, 2 ..., 10).

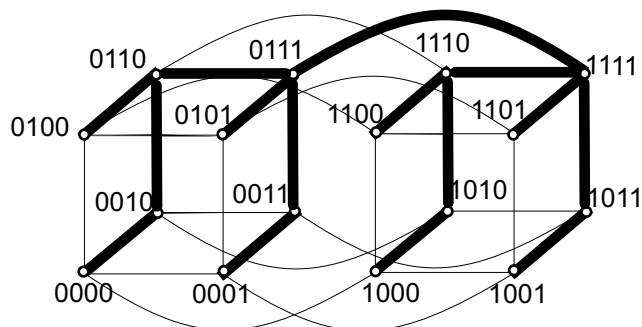
Dimension	# of possible spanning trees
1	1
2	4
3	384
4	$4.25 \times 10^7$
5	$2.08 \times 10^{19}$
6	$1.66 \times 10^{45}$
7	$1.54 \times 10^{101}$
8	$1.74 \times 10^{220}$
9	$6.8 \times 10^{470}$
10	$2.1 \times 10^{994}$

Table 5.3: The number of possible spanning trees for the butterfly (dimension = 1, 2 . . . , 8).

Dimension	# of possible spanning trees
1	4
2	1024
3	$1.2910^{10}$
4	$5.01 \times 10^{28}$
5	$2.87 \times 10^{74}$
6	$8 \times 10^{182}$
7	$2.66 \times 10^{433}$
8	$3.06 \times 10^{1001}$

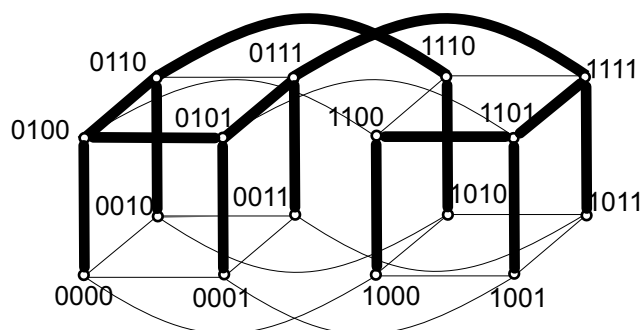
gorithm, an optimum solution for the graph being experimented has to be known in advance. This is required so that the results generated by the ContractedRandomSpan algorithm can be compared against the optimum solution found. A naive approach to finding a solution to the congestion cost problem is to use the brute-force method. This can be achieved by enumerating all the spanning trees for a given graph and then computing their corresponding congestion costs. Any spanning tree with the minimum congestion cost is an optimum solution. The STU-algorithm (see chapter 3) can be used to enumerate all the spanning trees, and the *all-pair shortest path* algorithm [13] can be used to compute their communication costs.

Therefore, the brute-force method was developed to allow us determine the optimum solutions for the graphs we were experimenting on. The STU-algorithm was implemented using the **Mathematica** package and was used to enumerate all the spanning trees for the graphs. It was chosen because it is an



Congestion Cost = 392

Restricted Congestion Cost = 80



Congestion Cost = 456

Restricted Congestion Cost = 92

Figure 5.3: Two spanning trees for  $H_4$ .

intelligent algorithm that efficiently enumerates all the spanning trees by exchanging edges. By using the brute-force method, we managed to enumerate all the spanning trees for the randomly generated graphs  $G_1$ ,  $G_2$  and  $G_3$ . The minimum and maximum values for the congestion costs and stretch costs for the spanning trees generated were recorded. The congestion costs for  $G_1$ ,  $G_2$  and  $G_3$  are given in Table 5.4.

Unfortunately, we had little success with the brute-force method when using it to find the congestion cost values for the hypercube and butterfly topologies. The brute-force method is not practical even when trying to solve this problem for low dimensions of these topologies. The maximum dimensions

Table 5.4: The congestion costs for the for the randomly generated graphs  $G_1$ ,  $G_2$  and  $G_3$ . (The results are produced using the brute-force method).

Graph	Stretch Cost		Congestion Cost	
	min. cost	max. cost	min. cost	max. cost
$G_1$	53	118	88	165
$G_2$	47	102	88	165
$G_3$	51	114	88	165

Table 5.5: The congestion costs for the hypercube (dimension = 1, 2 . . . , 10).(The results are produced using the brute-force method).

Dimension	Stretch Cost		Congestion Cost	
	min. cost	max. cost	min. cost	max. cost
1	1	1	1	1
2	6	6	4	10
3	24	28	68	84

that the brute-force method managed to solve 3 for the hypercube and 2 for the butterfly. Moreover, it took around 10 days to generate %25 of the total number of spanning trees for  $H_4$ . Therefore, a more efficient method was required to determine the optimum solutions. The congestion costs for the hypercube and butterfly that were successfully obtained by the brute-force method are listed in Table 5.5 and Table 5.6.

The reason for selecting the hypercube is because it enjoys several desirable properties. The most important properties are its recursive structure and symmetry. These properties can be used to help determine the optimum congestion cost for the hypercube. In [3], it has been shown that, in the case of

Table 5.6: The congestion costs for the butterfly (dimension = 1, 2 . . . , 8). (The results are produced using the brute-force method).

Dimension	Stretch Cost		Congestion Cost	
	min. cost	max. cost	min. cost	max. cost
1	6	6	10	10
2	34	34	212	244

the hypercube, a shortest-path spanning tree is always an optimum solution for the optimum congestion cost problem. Since the hypercube is vertex transitive, doing a breadth-first search starting at one vertex is equivalent to performing a breadth-first search at any vertex, as all resulting spanning trees will be isomorphic.

Since the optimum solutions for the hypercube can be easily obtained, which means that the ContractedRandomSpan algorithm can be evaluated for higher dimensions of the hypercube. Hence, the hypercube is used as a benchmark, in our experimental study, to accurately measure the performance of the ContractedRandomSpan algorithm. Figure 5.3 shows two spanning trees for  $H_4$ . The spanning tree displayed on the top is a shortest-path spanning tree.

On the other hand, in order to represent a possibly maximum solution, we resort to finding a spanning tree that has all of its edges highly congested. It is well-known that the hamiltonian path  $P_n$  exhibits the spanning tree of order  $n$  with maximum congestion.

The congestion costs collected for both the shortest-path spanning trees and the hamiltonian spanning trees for the hypercube can be used to indicate the range of possible solutions. This will help in giving an indication of how far the results obtained from the ContractedRandomSpan algorithm are from the

Table 5.7: The congestion costs for a shortest-path spanning tree (SPT) and hamiltonian spanning tree (HT) for the hypercube (dimension = 1, 2 ..., 10), where  $ratio = \frac{HT_{cost}}{SPT_{cost}}$ .

Dimension	Stretch Cost			Congestion Cost		
	SPT cost	HT cost	ratio	SPT cost	HT cost	ratio
1	1	1	1	1	1	1
2	6	6	1	10	10	1
3	24	28	1.2	68	84	1.2
4	80	120	1.5	392	680	1.7
5	240	502	2.1	2064	5456	2.6
6	672	2110	3.1	10272	43680	4.3
7	1792	8500	4.7	49216	349504	7.1
8	4608	34046	7.3	229504	2796160	12.2
9	11520	138482	12	1048832	22369536	21.3
10	28160	568054	20.2	4719104	178956800	37.9

optimum solutions.

Table 5.7 shows the congestion costs for the shortest-path spanning trees and hamiltonian spanning trees for  $H_1, H_2, \dots, H_{10}$ . The table also displays the *ratio* between the congestion costs for the hamiltonian spanning tree and the shortest-path spanning tree, for the specified dimensions. The ratio is basically the percentage ratio between the minimum and presumably maximum congestion cost values. These values can assist in evaluating the performance of the ContractedRandomSpan algorithm.

In the case of the butterfly, we found that the congestion costs for the shortest-path spanning trees generated for each dimension might vary. This



Table 5.8: The congestion costs for a shortest-path spanning tree (SPT) for the butterfly (dimension = 1, 2 . . . , 8).

Dimension	SPT Stretch Cost	SPT Congestion Cost
1	6	10
2	34	212
3	130	2576
4	434	23664
5	1266	183200
6	3538	1266112
7	9554	8055296
8	30202	48676096

might be due to the fact that a vertex might have  $degree = 2$  or  $4$ . We believe that a shortest-path spanning tree on the butterfly can approximate the optimum congestion cost spanning tree to some degree. The butterfly, however, does not have a hamiltonian spanning tree for  $dimension > 1$ . Therefore, it is difficult to determine the maximum congestion costs for this topology.

We were able to generate all the shortest-path spanning trees for  $BF_1, BF_2, \dots, BF_6$  and record the minimum congestion cost value for each dimension. However, only one shortest-path spanning tree was generated for  $BF_7$  and  $BF_8$  and was assumed to be the minimum. Table 5.8 shows the congestion costs for the minimum shortest-path spanning tree gathered for  $BF_1, BF_2, \dots, BF_8$ .

## 5.4 Experiment Results

The experiments performed were conducted using several types of special graphs. The special graphs used were: three randomly generated graphs  $G_1$ ,  $G_2$ , and  $G_3$ , in addition to the network topologies  $H_1, H_2, \dots, H_{10}$  and  $BF_1, BF_2, \dots, BF_8$ . The ContractedRandomSpan algorithm was used to generate several spanning trees for these special graphs. The corresponding congestion costs for the spanning trees were calculated, and the ones with the lowest congestion costs were recorded.

Let  $T_{low}$  be a spanning tree with the lowest congestion cost amongst all the spanning trees generated using our approach for a particular graph. Let  $T_{opt}$  be a spanning tree that has the optimum congestion cost. As discussed in the experiment setup,  $T_{opt}$  has been obtained for the graphs  $G_1$ ,  $G_2$ , and  $G_3$  by using the brute-force method. In the case of a hypercube,  $T_{opt}$  is considered to be any shortest-path spanning tree for the required dimension. We extend this observation to the butterfly and assume that a shortest-path spanning tree is a good approximation for an optimum solution. We have recorded the values of  $T_{opt}$  for  $H_1, H_2, \dots, H_{10}$  and  $BF_1, BF_2, \dots, BF_8$  which are listed earlier.

The performance of our approach can be evaluated by calculating the *performance ratio*  $= \frac{T_{low}}{T_{opt}}$ . The performance ratio can be used to show how far a specific spanning tree is from the optimum solution for a given graph. Using our approach, we managed to get near-optimum solutions for both the optimum congestion cost problem and the optimum stretch cost problem for the graphs  $G_1$ ,  $G_2$ , and  $G_3$ . The results are displayed in Table 5.9 and Table 5.10. Similar results have been obtained in the case of the hypercube and butterfly. The results for the hypercube are shown in Table 5.11 and Table 5.12, and the results for the butterfly are shown in Table 5.13 and Table 5.14.

Table 5.9: The congestion costs for the randomly generated graphs  $G_1$ ,  $G_2$  and  $G_3$ . (The results are produced by the ContractedRandomSpan algorithm)

<b>Graph</b>	<b>min. cost</b>	<b>max. cost</b>	<b># of trees generated</b>	<b>ratio</b>
$G_1$	103	165	26930	1.17
$G_2$	108	165	44800	1.23
$G_3$	108	165	10700	1.23

Table 5.10: The stretch costs for the randomly generated graphs  $G_1$ ,  $G_2$  and  $G_3$ . (The results are produced by the ContractedRandomSpan algorithm)

<b>Graph</b>	<b>min. cost</b>	<b>max. cost</b>	<b># of trees generated</b>	<b>ratio</b>
$G_1$	57	101	24860	1.08
$G_2$	66	118	4900	1.4
$G_3$	60	111	1700	1.17

Table 5.11: The congestion costs for the hypercube (dimension = 1, 2 ..., 10) generated by the ContractedRandomSpan algorithm.

<b>Dimension</b>	<b>min. cost</b>	<b>max. cost</b>	<b># of trees generated</b>	<b>ratio</b>
1	1	1	1	1
2	10	10	1	1
3	68	84	28	1
4	392	680	11500	1
5	2294	4682	15000	1.1
6	13112	23084	8000	1.3
7	70088	126620	8000	1.4
8	404774	506280	100	1.8
9	1959320	2532534	100	1.9
10	9747306	12122242	30	2.1

Table 5.12: The stretch costs for the hypercube (dimension = 1, 2 . . . , 10) generated by the ContractedRandomSpan algorithm.

Dimension	min. cost	max. cost	# of trees generated	ratio
1	1	1	1	1
2	6	6	1	1
3	24	30	36	1
4	80	156	11500	1
5	284	570	15000	1.2
6	946	1650	8000	1.4
7	2878	5154	8000	1.6
8	9736	11972	100	2.1
9	26248	34164	100	2.3
10	74548	93462	30	2.6
10	74548	93462	30	2.6

Table 5.13: The congestion costs for the butterfly (dimension = 1, 2 . . . , 8) generated by the ContractedRandomSpan algorithm.

Dimension	min. cost	max. cost	# of trees generated	ratio
1	10	10	1	1
2	212	244	60	1
3	2568	4124	10000	0.99
4	25678	37774	4000	1.1
5	214424	299212	1000	1.2
6	1599808	2051166	200	1.3
7	11216140	12541424	10	1.4
8	71365298	74727996	2	1.5

Table 5.14: The stretch costs for the butterfly (dimension = 1, 2 . . . , 8) generated by the ContractedRandomSpan algorithm.

<b>Dimension</b>	<b>min. cost</b>	<b>max. cost</b>	<b># of trees generated</b>	<b>ratio</b>
1	6	6	1	1
2	34	34	1	1
3	130	182	40	1
4	438	646	4000	1.01
5	1382	1948	1000	1.1
6	4363	5620	200	1.2
7	12790	14250	10	1.3
8	35372	36730	2	1.2

From the experiments conducted, we find that the ContractedRandomSpan algorithm can be used to generate near-optimum solutions. During the experiments, it was noticed that a near-optimum solution can be found quickly by only generating a few number of spanning trees. It was also observed that the quality of the results improve, to some extent, as the number of spanning trees generated increase. The experiment results support the fact that not all the spanning trees for a given graph need to be enumerated in order to find an optimum solution. From the results obtained, it is clear that we managed to scope the whole search space of possible solutions by only generating a small fraction of the possible solutions. The reason behind this is because the amount of possible spanning trees is large compared to the range of possible congestion cost values. From table Table 5.12, it is clear that the minimum and maximum congestion cost values have been generated for  $H_4$  by only generating a small number of spanning trees. As a result, our approach could have been used

instead of the brute-force method for finding the extreme congestion costs for a particular graph.

Another point worth mentioning is regarding the assumption that a shortest-path spanning might be an optimum solution for the optimum congestion cost problem on the butterfly. It turns out that this assumption is invalid and has been proven by counter example. From the experimental results, it is clear that the solution obtained from the ContractedRandomSpan algorithm for  $BF_3$  is less than the congestion cost of any of the shortest-path spanning trees for  $BF_3$ . Therefore, an optimum solution for the butterfly can't be obtained using a shortest-path spanning tree, like the hypercube.

# Chapter 6

## CONCLUSION

The ODST problem is a special case of the OCST problem and has been proven to be NP-hard. Several approaches currently exist that try to find an optimum solution for the ODST problem. Some of these approaches consider approximating the optimum solution by using either deterministic or randomized approximation algorithms. Other approaches attempt to reduce the ODST problem to other well known NP-hard problems that already have algorithms implemented for them. Lately, several approaches take advantage of genetic algorithms to find an optimum solution for the problem. In the genetic algorithms approach, it is necessary to generate spanning trees that are biased, in some way, in order to efficiently scope the search space.

### 6.1 Contribution

In this thesis work, we proposed a randomized algorithm that can generate the spanning trees for a given graph randomly. The proposed algorithm is biased towards generating shortest-path spanning trees. It is biased in the sense that it does not generate random spanning trees with equal probability, but generates



spanning trees that are similar to the shortest-path spanning trees with higher probability. We believe that the proposed algorithm has the potential of being applied to help solve a variety of problems, especially those that deal with spanning trees for finding their solutions. An example of such problems is the ODST problem.

The work presented in this thesis demonstrates how randomization can be harnessed to efficiently solve two special cases of the ODST problem. The results obtained from the experimental study illustrate the fact that near-optimum solutions can be obtained by only performing a few runs. The findings of this thesis support the fact that randomized algorithms can efficiently solve complex problems in a simple manner. The experiment results also prove that the assumption regarding a shortest-path spanning tree being a possible optimum solution for the butterfly is invalid by counter example.

## 6.2 Future Work

The algorithm proposed in this thesis has been applied to efficiently solve the optimum congestion cost problem. In this problem, the costs of the edges for a given graph are constant and equal to one. This is in contrast to the ODST problem where the costs of the edges are arbitrary. However, it would be interesting to solve the more general case, where the costs for the edges are arbitrary. In this case, the problem would be similar to the ODST problem except that it would be applicable to general graphs opposed to limiting it to complete graphs only.

In our approach, we randomly generate spanning trees that are biased towards shortest-path spanning trees. By generating spanning trees with such properties, we manage to scope the search space of possible solutions, for the

optimum congestion cost problem, efficiently. It might be useful to adapt other biasing techniques to help solve other types of problems, such as the OCST problem. In the OCST problem, in addition to the arbitrary costs of the edges, there are arbitrary requirement variables between each pair of vertices. In this case, it might be useful to generate spanning trees that are biased to the minimum cost spanning tree. The requirement variables between the vertices could be used to compute the minimum cost spanning tree.

The objective of the experimental study performed in this thesis is to measure the performance of the proposed algorithm when trying to solve the optimum congestion cost problem. The experiments were conducted using three randomly generated graphs, in addition to the hypercube and butterfly topologies. By knowing the optimum solutions for the hypercube in advance, we managed to precisely measure the performance of the proposed algorithm for high dimensions for this topology. However, other types of network topologies might exist with properties that might help reveal their optimum solutions. An example of a popular network topology is the cube-connected cycle [22]. Therefore, it is encouraged to explore the properties of other network topologies and try to determine their optimum solutions. This will be of significant value to the work presented in this thesis, as well as having the potential of influencing the work of other researchers interested in the area of topology design problems. Additional future work is to analyze the proposed algorithm and find the expected number of runs needed to get the optimum solution.

# **APPENDIX**

## **Source Code**

```

(* The ContractedRandomSpan algorithm *)
createEnhancedRandomSpanningTree[network_]:=
Module[{i, j, flag, added, spanTreeNodes, networkEdges, spanTree, adjLists,
  randomPosition, randomNode, randomNodeEdgeSet, randomEdge,
  neighbourNode, neighbourNodes, contractedNode, edge, n1, n2,
  selectedNode},
(
  If[DEBUG \[Equal] True,
    Print["Entered createEnhancedRandomSpanningTree"];

    (* Strip the network from its edges *)
    adjLists = ToAdjacencyLists[network];
    If[DEBUG \[Equal] True, Print["Adjacency list ", adjLists]];

    networkEdges = ToUnorderedPairs[network];
    spanTree = DeleteEdges[network, networkEdges];
    spanTreeNodes = {};

    (*-----*)
    (* Pick a node Randomly and add it to spanTreeNodes *)
    randomCount++;
    randomNode = Random[Integer, V[network]-1]+1;
    AppendTo[spanTreeNodes, randomNode];

    (* get the neighbours to the randomly picked node*)
    neighbourNodes = adjLists[[randomNode]];
    contractedNode = {};

    (* create the edge list that will be used to create the span tree *)
    For[i=1, i\[LessEqual] Length [neighbourNodes], i++,
      (
        edge = {neighbourNodes[[i]], {randomNode, neighbourNodes[[i]}};
        AppendTo[contractedNode, edge];
      )];

    (*-----*)

    While[Length[spanTreeNodes]<V[network],
      (
        (* Pick a node randomly from the contracted nodes edge list *)
        randomCount++;
        randomPosition = Random[Integer, Length[contractedNode]-1]+1;
        randomNodeEdgeSet = contractedNode[[randomPosition]];

        (* delete the randomly picked node from the contracted nodes edge list *)
        contractedNode = Delete[contractedNode, randomPosition];

        (* Select an edge from the list of edges for the selected neighbour node *)
        randomPosition = Random[Integer, Length[randomNodeEdgeSet]-2]+2;
        randomEdge = randomNodeEdgeSet[[randomPosition]];

        (* add the edge to the spanning tree *)
        spanTree = AddEdge[spanTree, randomEdge];

        (* Contract the node selected *)
        neighbourNode = randomNodeEdgeSet[[1]];
        AppendTo[spanTreeNodes, neighbourNode];

        (* get the neighbours for the node to be contracted *)
        neighbourNodes = adjLists[[neighbourNode]];

```

```

For[i=1, i\[LessEqual] Length[neighbourNodes], i++,
  (
    node = neighbourNodes[[i]];

    (* if the node has not been contracted before *)
    If[!MemberQ[spanTreeNodes, node], (
      (* check whether this node is already part of the contracted list *)
      added = False;

      For[j=1, j\[LessEqual] Length[contractedNode], j++, (
        selectedNode = contractedNode[[j, 1]];

        If[selectedNode \[Equal] node, (
          added = True;
          contractedNode = Delete[contractedNode, j];
          edge = {node, {node, neighbourNode}};
          AppendTo[contractedNode, edge];
          break[];
        )];
      )];

      If[added \[Equal] False, (
        edge = {node, {node, neighbourNode}};
        AppendTo[contractedNode, edge];
      )];
    )];
  );

If[DEBUG \[Equal] True,
  Print["Finished createEnhancedRandomSpanningTree"];
Return[spanTree];
)];

```

```

(* The RandomSpan algorithm *)

createRandomSpanningTree[network_]:=
Module[{i, j, spanTreeNodes, networkEdges, spanTree, adjLists,
  randomPosition, randomNode, neighbourNode}, (

  (* Strip the network from its edges *)
  adjLists = ToAdjacencyLists[network];
  networkEdges = ToUnorderedPairs[network];
  spanTree = DeleteEdges[network, networkEdges];
  spanTreeNodes = {};

  (* Pick a node Randomly and add it to spanTreeNodes*)
  randomCount++;
  AppendTo[spanTreeNodes, Random[Integer, V[network]-1]+1];

  While[Length[spanTreeNodes]<V[network],
  (
    (* Pick a node randomly from the list of selected nodes *)
    randomCount++;
    randomPosition = Random[Integer, Length[spanTreeNodes]-1]+1;
    randomNode = spanTreeNodes[[randomPosition]];

    (* Pick a neighbor to this node from the adjacency lists *)
    randomCount++;

    randomPosition =
      Random[Integer, Length[adjLists[[randomNode]]]-1]+1;
    neighbourNode = adjLists[[randomNode, randomPosition]];

    If [Count[spanTreeNodes, neighbourNode]\[Equal]0,
      (
        AppendTo[spanTreeNodes,neighbourNode];
        spanTree = AddEdge[spanTree, {randomNode, neighbourNode}];
      )];
  )];

  Return[spanTree];
)]

```

```

<<DiscreteMath`Combinatorica`

(*Generate the spanning trees deterministically using the STU algorithm*)

MyCut[g_, t_, e_] :=
  Module[{i, j, v1, v2, myG, myT, myE, CompGraph, comp1, comp2, cutSpan,
    AdjacencyLists, bridgeEdges},
    (
      bridgeEdges = {};
      cutSpan = DeleteEdge[t, e];
      CompGraph = ConnectedComponents[cutSpan];

      If[
        Length[CompGraph] !=
          2, (Print["Error--> Cut[]: two components Not found = ",
            CompGraph]; Return[{}])];

      comp1 = CompGraph[[1]];
      comp2 = CompGraph[[2]];

      AdjacencyLists = ToAdjacencyLists[g];

      For[i=1, i<=Length[comp1], i++,
        (
          v1 = comp1[[i]];
          Vlists = AdjacencyLists[[v1]];

          For[j=1, j<=Length[Vlists], j++,
            (
              v2 = Vlists[[j]];
              If[Count[comp2, v2]!=0, AppendTo[bridgeEdges, {v1, v2}]]
            )
          ]];

      Return [bridgeEdges];
    )];

MyEntr[g_, ti_, tp_, e_] := Module[{result, a, b},
  (
    a = MyCut[g, tp, e];
    b = MyCut[g, ti, e];

    result = Intersection[a, b];
    result = Complement[result, {e}];

    Return[result];
  )];

```

```

FindChildren[g_, ti_, tp_, initEdges_, k_]:=
Module[{myEdges, addEdge, delEdge, tc, i, cost},
(
  If[k <= 0, Return[]];

  delEdge = initEdges[[k]];
  myEdges = MyEntr[g, ti, tp, delEdge];
  For[i=1, i <= Length[myEdges], i++,
    (
      addEdge = myEdges[[i]];
      tc = AddEdge[tp, addEdge];
      tc = DeleteEdge[tc, delEdge];

      (* calculate the communication cost for the span tree*)
      numSpanTrees ++;
      spMatrix = AllPairsShortestPath[tc];

      costAll = communicationCostAllPair[ti, spMatrix];
      costNetwork = communicationCost[g, ti, spMatrix];

      If[minCostAll > costAll, minCostAll = costAll];
      If[maxCostAll < costAll, maxCostAll = costAll];
      If[minCostNetwork > costNetwork, minCostNetwork = costNetwork];
      If[maxCostNetwork < costNetwork, maxCostNetwork = costNetwork];

      If[DEBUG == True, (Print["Span Tree (", numSpanTrees, " "];
        Print[Edges[tc]]);

      If[DRAW == True,
        ShowLabeledGraph[Highlight[g, {Edges[tc]}]]];

      If[DEBUG \[Equal] True,
        Print["-----"];

      (* find the children for this span tree *)
      FindChildren[g, ti, tc, initEdges, k-1];
    )];
  FindChildren[g, ti, tp, initEdges, k-1];

  (*Print["The edges found are: ", myEdges];*)
);

```



```

communicationCostAllPair[spanTree_, spMatrix_] := Module[{i, x, y, cost, },
  (
    mLength = Length[spMatrix];

    cost = 0;
    (*e = Edges[network];*)

    For[i=1, i\[[LessEqual] mLength, i++,
      (
        For[j=1, j\[[LessEqual] mLength, j++,
          (
            cost+= spMatrix[[i,j]];
          )]
        )];

    (* Return the communication cost for the all pairs *)

    Return[cost/2];
  )]

```

```

communicationCost[network_, spanTree_, spMatrix_] := Module[{i, x, y, cost},
  (
    cost = 0;
    e = Edges[network];

    For[i=1, i <= Length[e], i++,
      (
        x = e[[i,1]];
        y = e[[i,2]];
        cost+= spMatrix[[x,y]];
      )];

    Return[cost];
  )]

```

The main program

```

DEBUG = False;
DRAW = False;
numRandomSpanTrees = 1000000;
counter = 0;

network = RandomGraph[10, 0.6];
Print["Number of Spanning Trees = ", NumberOfSpanningTrees[network]];

network =
  SetGraphOptions[network, VertexColor\[[Rule]Blue,
    VertexStyle\[[Rule]Disk[Small], EdgeColor\[[Rule] Red,
    EdgeStyle\[[Rule]Thin] ;
Print["-----"];
Print["The original network"];
Print["Expected number of Span Trees = ", NumberOfSpanningTrees[network]];
ShowLabeledGraph[network];
Print["-----"];

(* Find the BFS tree for the randomly generated graph *)
networkEdges = ToUnorderedPairs[network];
bfsTree = DeleteEdges[network, networkEdges];
bfsTree =
  SetGraphOptions[bfsTree, VertexColor\[[Rule]Blue,
    VertexStyle\[[Rule]Disk[Small], EdgeColor\[[Rule] Red,
    EdgeStyle\[[Rule]Thin] ;

bfsEdges = BreadthFirstTraversal[network, 1, Edge];
If[DRAW \[[Equal] True, ShowLabeledGraph[Highlight[network, {bfsEdges}]]];

bfsTree = AddEdges[bfsTree, bfsEdges];
spMatrix = AllPairsShortestPath[bfsTree];

costNetwork = communicationCost[network, bfsTree, spMatrix];
costAll = communicationCostAllPair[bfsTree, spMatrix];

Print["----> BFS Tree All-Pair Costs = ", costAll];
Print["----> BFS Tree Network-Pair Costs = ", costNetwork ];

```

```

Print["-----"];

Print["*****\
];
Print["--> Enumerating all the spanning trees for the graph using the \
STU-algorithm"];
Print["*****\
];

numSpanTrees = 0;
minCostAll = Infinity;
minCostNetwork = Infinity;
maxCostAll=0;
maxCostNetwork = 0;

(* create a DFS spanning Tree for the graph *)
initSpanTreeEdges = DepthFirstTraversal[network, 1, Edge];
ti= network;
ti = DeleteEdges[ti, Edges[network]];
ti = AddEdges[ti,initSpanTreeEdges];
ti = SetGraphOptions[ti, VertexColor\[Rule]Blue,
VertexStyle\[Rule]Disk[Small],EdgeColor\[Rule] Red,
EdgeStyle\[Rule]Thin] ;

(* calculate the communication cost for the DFS tree*)
numSpanTrees ++;
spMatrix = AllPairsShortestPath[ti];

costAll = communicationCostAllPair[ti, spMatrix];
costNetwork = communicationCost[network, ti, spMatrix];

minCostAll= costAll;
maxCostAll = costAll;
minCostNetwork = costNetwork;
maxCostNetwork = costNetwork;

(* find all the children spanning trees for the DFS span tree *)
numSpanTreeEdges=Length[initSpanTreeEdges];
FindChildren[network, ti, ti, initSpanTreeEdges,numSpanTreeEdges];

Print["Expected number of Span Trees = ",NumberOfSpanningTrees[network]];
Print["Number of Span Trees generated = ",numSpanTrees];

Print["--> Results for enumerating all the spanning trees are:"];
Print["----> All-Pair Costs"];
Print["      Min cost = ", minCostAll];
Print["      Max cost = ", maxCostAll];
Print["----> Network-Pair Costs"];
Print["      Min cost = ", minCostNetwork];
Print["      Max cost = ", maxCostNetwork];
Print["-----"]\
;
Print[""];

```

```

Print["*****"];
Print["--> Generating the randomly constructed spanning trees "];
Print["*****"];

minCostAll= Infinity;
minCostNetwork = Infinity;
maxCostAll = 0;
maxCostNetwork = 0;
NumberOfTreesGenerated =0;
For[i=1,i<numRandomSpanTrees,i++,(

  If[counter \[Equal]10,
    (
      counter=0;
      Print["Number of span trees generated = ",i];
      Print["----> All-Pair Costs"];
      Print["      Min cost = ", minCostAll];
      Print["      Max cost = ", maxCostAll];
      Print["----> Network-Pair Costs"];
      Print["      Min cost = ", minCostNetwork];
      Print["      Max cost = ", maxCostNetwork];
      Print["-----"];
      Print[""];
    )];

    (* create a spanning tree randomly *)
    randomSpanTree=createEnhancedRandomSpanningTree[network];
    If[DRAW \[Equal] True,
      ShowLabeledGraph[Highlight[network, {Edges[randomSpanTree]}]];

    (* calculate the communication cost for the span tree *)
    spMatrix = AllPairsShortestPath[randomSpanTree];

    costNetwork = communicationCost[network, randomSpanTree, spMatrix];
    costAll = communicationCostAllPair[randomSpanTree, spMatrix];

    If[minCostAll> costAll, minCostAll= costAll];
    If[maxCostAll< costAll, maxCostAll= costAll];
    If[minCostNetwork> costNetwork, minCostNetwork= costNetwork];
    If[maxCostNetwork< costNetwork, maxCostNetwork= costNetwork];

    counter ++;
  )];

```

```

<<DiscreteMath`Combinatorica`

(* Brute force method to generate all the spanning trees for a graph*)
(* and compute their congestion costs*)

MyCut[g_, t_, e_] :=
Module[{i, j, v1, v2, myG, myT, myE, CompGraph, comp1, comp2, cutSpan,
AdjacencyLists, bridgeEdges},
(
bridgeEdges = {};
cutSpan = DeleteEdge[t, e];
CompGraph = ConnectedComponents[cutSpan];

If[
Length[CompGraph] !=
2, (Print["Error--> Cut[]: two components Not found = ",
CompGraph]; Return[{}])];

comp1 = CompGraph[[1]];
comp2 = CompGraph[[2]];

AdjacencyLists = ToAdjacencyLists[g];

For[i=1, i<=Length[comp1], i++,
(
v1 = comp1[[i]];
Vlists = AdjacencyLists[[v1]];

For[j=1, j<=Length[Vlists], j++,
(
v2 = Vlists[[j]];
If[Count[comp2, v2]!=0, AppendTo[bridgeEdges, {v1, v2}]]
)]
)];

Return [bridgeEdges];
)];

MyEntr[g_, ti_, tp_, e_] := Module[{result, a, b},
(
a = MyCut[g, tp, e];
b = MyCut[g, ti, e];

result = Intersection[a, b];
result = Complement[result, {e}];

Return[result];
)];

FindChildren[g_, ti_, tp_, initEdges_, k_] :=
Module[{myEdges, addEdge, delEdge, tc, i, cost},
(
If[k\[LessEqual] 0, Return[]];

delEdge = initEdges[[k]];
myEdges = MyEntr[g, ti, tp, delEdge];
For[i=1, i\[LessEqual] Length[myEdges], i++,
(
addEdge = myEdges[[i]];
tc = AddEdge[tp, addEdge];
tc = DeleteEdge[tc, delEdge];

(* calculate the communication cost for the span tree*)
numSpanTrees ++;
spMatrix = AllPairsShortestPath[tc];

costAll = communicationCostAllPair[ti, spMatrix];
costNetwork = communicationCost[g, ti, spMatrix];

If[minCostAll > costAll, minCostAll = costAll];
If[maxCostAll < costAll, maxCostAll = costAll];
If[minCostNetwork > costNetwork, minCostNetwork = costNetwork];
If[maxCostNetwork < costNetwork, maxCostNetwork = costNetwork];
)];

```

```

If[DEBUG \[Equal] True, (Print["Span Tree (", numSpanTrees, ")"];
Print[Edges[tc]]]);

If[DRAW \[Equal] True,
ShowLabeledGraph[Highlight[g, {Edges[tc]}]]];

If[DEBUG \[Equal] True,
Print["-----"];

(* find the children for this span tree *)
FindChildren[g, ti, tc, initEdges,k-1];
]);

FindChildren[g, ti, tp, initEdges,k-1];

(*Print["The edges found are: ", myEdges];*)
)];

(* The main program *)

DEBUG = False;
DRAW = False;
numRandomSpanTrees = 1000000;
counter =0;

network =RandomGraph[10, 0.6];
Print["Number of Spanning Trees = ", NumberOfSpanningTrees[network]];

network =
SetGraphOptions[network, VertexColor\[Rule]Blue,
VertexStyle\[Rule]Disk[Small],EdgeColor\[Rule] Red,
EdgeStyle\[Rule]Thin] ;
Print["-----"];
Print["The original network"];
Print["Expected number of Span Trees = ",NumberOfSpanningTrees[network]];
ShowLabeledGraph[network];
Print["-----"];

(* Find the BFS tree for the randomly generated graph *)
networkEdges = ToUnorderedPairs[network];
bfsTree = DeleteEdges[network, networkEdges];
bfsTree =
SetGraphOptions[bfsTree, VertexColor\[Rule]Blue,
VertexStyle\[Rule]Disk[Small],EdgeColor\[Rule] Red,
EdgeStyle\[Rule]Thin] ;

bfsEdges = BreadthFirstTraversal[network,1, Edge];
If[DRAW \[Equal] True,ShowLabeledGraph[Highlight[network,{bfsEdges}]]];

bfsTree=AddEdges[bfsTree, bfsEdges];
spMatrix = AllPairsShortestPath[bfsTree];

costNetwork = communicationCost[network, bfsTree, spMatrix];
costAll = communicationCostAllPair[bfsTree, spMatrix];

Print["----> BFS Tree All-Pair Costs = ", costAll];
Print["----> BFS Tree Network-Pair Costs = ", costNetwork ];
Print["-----"];

Print["*****\
"];
Print["--> Enumerating all the spanning trees for the graph using the \
STU-algorithm"];
Print["*****\
"];
];

```

```

numSpanTrees = 0;
minCostAll = Infinity;
minCostNetwork = Infinity;
maxCostAll=0;
maxCostNetwork = 0;

(* create a DFS spanning Tree for the graph *)
initSpanTreeEdges = DepthFirstTraversal[network, 1, Edge];
ti= network;
ti = DeleteEdges[ti, Edges[network]];
ti = AddEdges[ti,initSpanTreeEdges];
ti = SetGraphOptions[ti, VertexColor\[Rule]Blue,
    VertexStyle\[Rule]Disk[Small],EdgeColor\[Rule] Red,
    EdgeStyle\[Rule]Thin] ;

(* calculate the communication cost for the DFS tree*)
numSpanTrees ++;
spMatrix = AllPairsShortestPath[ti];

costAll = communicationCostAllPair[ti, spMatrix];
costNetwork = communicationCost[network, ti, spMatrix];

minCostAll= costAll;
maxCostAll = costAll;
minCostNetwork = costNetwork;
maxCostNetwork = costNetwork;

(* find all the children spanning trees for the DFS span tree *)
numSpanTreeEdges=Length[initSpanTreeEdges];
FindChildren[network, ti, ti, initSpanTreeEdges,numSpanTreeEdges];

Print["Expected number of Span Trees = ",NumberOfSpanningTrees[network]];
Print["Number of Span Trees generated = ",numSpanTrees];

Print["---> Results for enumerating all the spanning trees are:"];
Print["----> All-Pair Costs"];
Print["    Min cost = ", minCostAll];
Print["    Max cost = ", maxCostAll];
Print["----> Network-Pair Costs"];
Print["    Min cost = ", minCostNetwork];
Print["    Max cost = ", maxCostNetwork];
Print["-----"]\
;
Print[""];

```

```

Print["*****"];
Print["--> Generating the randomly constructed spanning trees "];
Print["*****"];

minCostAll= Infinity;
minCostNetwork = Infinity;
maxCostAll = 0;
maxCostNetwork = 0;
NumberOfTreesGenerated =0;
For[i=1,i<numRandomSpanTrees,i++,(

    If[counter \[Equal]10,
      (
        counter=0;
        Print["Number of span trees generated = ",i];
        Print["----> All-Pair Costs"];
        Print["      Min cost = ", minCostAll];
        Print["      Max cost = ", maxCostAll];
        Print["----> Network-Pair Costs"];
        Print["      Min cost = ", minCostNetwork];
        Print["      Max cost = ", maxCostNetwork];
        Print["-----"];
        Print[""];
      )];

      (* create a spanning tree randomly *)
      randomSpanTree=createEnhancedRandomSpanningTree[network];
      If[DRAW \[Equal] True,
        ShowLabeledGraph[Highlight[network, {Edges[randomSpanTree]}]];

      (* calculate the communication cost for the span tree *)
      spMatrix = AllPairsShortestPath[randomSpanTree];

      costNetwork = communicationCost[network, randomSpanTree, spMatrix];
      costAll = communicationCostAllPair[randomSpanTree, spMatrix];

      If[minCostAll> costAll, minCostAll= costAll];
      If[maxCostAll< costAll, maxCostAll= costAll];
      If[minCostNetwork> costNetwork, minCostNetwork= costNetwork];
      If[maxCostNetwork< costNetwork, maxCostNetwork= costNetwork];

      counter ++;
    )];

```

```

<<DiscreteMath`Combinatorica`

(* computation of the breadth-first spanning tree and the hamiltonian*)

DEBUG = False;
DRAW = False;
numRecursiveSpanTrees = 1;

For[dimension = 1, dimension <= 10, dimension++,
  (
    network = Hypercube[dimension];
    network =
      SetGraphOptions[network, VertexColor -> Blue,
        VertexStyle -> Disk[Small], EdgeColor -> Red, EdgeStyle -> Thin] ;

    (* create several recursive trees and compute there cost *)
    minCostAll = Infinity;
    minCostNetwork = Infinity;
    maxCostAll = 0;
    maxCostNetwork = 0;

    If[DEBUG == True,
      Print["***** Printing the Heuristically generated Spanning Trees *****"]];

    For[i = 1, i <= numRecursiveSpanTrees, i++, (

      networkEdges = ToUnorderedPairs[network];
      spanTree = DeleteEdges[network, networkEdges];

      spanTree =
        SetGraphOptions[spanTree, VertexColor -> Blue,
          VertexStyle -> Disk[Small], EdgeColor -> Red,
          EdgeStyle -> Thin] ;

      randomPosition = Random[Integer, V[network] - 1] + 1;
      bfsEdges = BreadthFirstTraversal[network, 1, Edge];
      (*ShowLabeledGraph[Highlight[network, {bfsEdges}]]);*)

      spanTree = AddEdges[spanTree, bfsEdges];

      (* calculate the communication cost for the span tree *)
      spMatrix = AllPairsShortestPath[spanTree];

      costNetwork = communicationCost[network, spanTree, spMatrix];
      costAll = communicationCostAllPair[spanTree, spMatrix];

      If[minCostAll > costAll, minCostAll = costAll];
      If[maxCostAll < costAll, maxCostAll = costAll];
      If[minCostNetwork > costNetwork, minCostNetwork = costNetwork];
      If[maxCostNetwork < costNetwork, maxCostNetwork = costNetwork];

      Print["---> Hypercube Dimension = ", dimension];
      Print["---> Recursive Span Tree Netwok Pairs Cost (", i, ") = ",
        costNetwork];

      Print["---> Recursive Span Tree All Pairs Cost (", i, ") = ",
        costAll];
      Print["-----"];
    )
  ]];

If[DEBUG == True,
  (
    Print["---> Hypercube Dimension = ", dimension];
    Print["----> All-Pair Costs"];
    Print["      Min cost = ", minCostAll];
    Print["      Max cost = ", maxCostAll];
    Print["----> Network-Pair Costs"];
    Print["      Min cost = ", minCostNetwork];
    Print["      Max cost = ", maxCostNetwork];
    Print["-----"];
    Print[""];
  )];
];

```



# Bibliography

- [1] S. Agarwal, A. K. Mittal, and P. Sharma. Constrained optimum communications trees and sensitivity analysis. *SIAM Journal on Computing*, 13:315–328, 1984.
- [2] N. Alon, R. M. Karp, D. Peleg, and D. West. A graph theoretic game and its application to the k-server problem. *SIAM J. on Computing*, pages 78–100, 1995.
- [3] M. H. Alsuwaiyel. On the average distance of the hypercube tree. *Submitted to the Journal of Computer Science*, 2005.
- [4] D. Avis and K. Fukuda. A basis enumeration algorithm for linear systems with geometric applications. *Appl. Math. Lett.*, 4:39–42, 1991.
- [5] D. Avis and K. Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete Comput. Geom.*, 8:295–313, 1992.
- [6] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65:21–46, 1996.
- [7] J. C. Bermond and C. Peyrat. De bruijn and kautz networks: a competitor for the hypercube? *In Hypercube and Distributed Computers (F. Andre and J. P. Verjus eds.)*. Horth-Holland: Elsevier Science Publishers, pages 278–293, 1989.

- [8] R. S. Cahn. Wide area network design, concepts and tools for optimization. *San Francisco: Morgan Kaufmann Publishers*, 1998.
- [9] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378, 1989.
- [10] K. Efe. A variation on the hypercube with lower diameter. *IEEE Transactions on computers*, 40 (11):1312–1316, 1991.
- [11] A. El-Amawy and S. Latifi. Properties and performance of folded hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):31–42, 1991.
- [12] R.C. Entringer, D.E. Jackson, and P.J. Slater. Geodetic connectivity of graphs. *IEEE Transactions on Circuits Systems*, 24:460–463, 1977.
- [13] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.
- [14] H. N. Gabow and Myers E. W. Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput*, 7:280–287, 1978.
- [15] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *SIAM Journal on Applied Math*, 9:551–570, 1961.
- [16] J. P. Hayes. Computer architecture and organization. *New York: McGraw-Hill Book Company*, 1978.
- [17] T. C. Hu. Optimum communication spanning trees. *SIAM Journal of Computing*, 3:188–195, 1974.
- [18] D. S. Johnson, J. K. Lenstra, and A. H. G. Rinnooy Kan. The complexity of the network design problem. *Networks*, 8(4):279–285, 1978.

- [19] S. Kapoor and H. Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput*, 24:247–265, 1995.
- [20] A. Kershenbaum. Telecommunications network design algorithms. *New York: McGraw Hill*, 1993.
- [21] M. Kouider and P. Winkler. Mean distance and minimum degree. *Journal of Graph Theory*, 25:95–99, 1997.
- [22] F. T. Leighton. Introduction to parallel algorithms and architectures: arrays, trees and hypercubes. *San Mateo, California: Morgan Kaufmann Publishers*, 1992.
- [23] T. Matsui. An algorithm for finding all the spanning trees in undirected graphs. *Research Report, Department of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo*, 1993.
- [24] C. C. Palmer. An approach to a problem in network design using genetic algorithms. *PhD Thesis, Polytechnic University, Computer Science Department, Brookly, NewYork*, 1994.
- [25] D. Peleg. Approximating minimum communication cost spanning trees. *Proc. 4th Collaq. on Structural Information and Communication Complexity, Ascona, Switzerland*, 1997.
- [26] R. C. Read and Tarjan R. E. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.
- [27] F. Rothlauf, J. Gersticker, and A. Heinzl. On the optimal communication spanning tree problem. (*IlliGAL Report Nr. 10/2003*) *Working Papers in Information Systems*, 2003.

- [28] F. Rothlauf and A. Heinzl. Developing efficient metaheuristics for communication network problems by using problem-specific knowledge. (*Il-GAL Report Nr. 9/2004*) *Working Papers in Information Systems*, 2004.
- [29] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on computers*, 37(7):867–872, 1988.
- [30] A. Shioura and A. Tamura. Efficiently scanning all spanning trees of an undirected graph. *Journal of Operations Research Society in Japan*, 38:331–344, 1995.
- [31] A. Shioura, A. Tamura, and T. Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, 26(3):678–692, 1997.
- [32] H. Sullivan and T.R. Bashkow. A scale homogeneous full distributed parallel machine. *Proceeding of the Annual Symposium on Computer Architecture*, pages 105–117, 1977.
- [33] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.
- [34] R. Tarjan. A new approach for speeding up enumeration algorithms. *Proceedings of the 9th International Symposium on Algorithms and Computation*, pages 287–296, 1998.
- [35] M. S. Waterman. Introduction to computational biology. *Chapman & Hall, London.*, 1995.
- [36] H. Wiener. Structural determination of paraffin boiling points. *J. Am. Chem. Soc.*, 69:1–24, 1947.

- [37] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. (*Tech. Rep. No. SFI-TR-95-02-010*). Santa Fe, NM: Santa Fe Institute., 1995.
- [38] R. Wong. Worst case analysis of network design problem heuristics. *SIAM J. Algebraic Discr. Meth.*, 1:51–63, 1980.
- [39] B. Y. Wu, K. Chao, and C. Y. Tang. Approximation algorithms for some optimum communication spanning tree problems. *Discrete Applied Mathematics*, 102:245–266, 2000.
- [40] B. Y. Wu, G. Lancia, Y. Bafna, K. Chao, R. Ravi, and C. Y. Tang. A polynomial time approximation scheme for minimum routing cost spanning trees. *In Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, pages 21–32, 1998.

## VITA

- Khalid Saud Al-Zamil.
- Born in Riyadh, Saudi Arabia, on April 14, 1973.
- Completed Bachelor of Science in Computer Engineering from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in January 1997.
- Email: [khalid.zamil@aramco.com](mailto:khalid.zamil@aramco.com)