2010

An Efficient Scheme for Non-bifurcated Traffic Grooming in WDM Networks

Syed Jabbar
University of Windsor

AN EFFICIENT SCHEME FOR NON-BIFURCATED TRAFFIC GROOMING IN WDM NETWORKS

by
Syed Jabbar

A Thesis
Submitted to the Faculty of Graduate Studies
through School of Computer Science
in Partial Fulfillment of the Requirements for
The Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2010

AN EFFICIENT SCHEME FOR NON-BIFURCATED TRAFFIC GROOMING IN WDM NETWORKS

by

Syed Jabbar

APPROVED BY:

_____

Dr. Jagdish Pathak
Odette School of Business

_____

Dr. Dan Wu
School of Computer Science

_____

Dr. Subir Bandyopadhyay, Advisor
School of Computer Science

_____

Dr. Christie Ezeife, Chair of Defense
School of Computer Science

May 17, 2010

Declaration of Co-Authorship

I hereby declare that this thesis incorporates material that is result of joint research, as follow:

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Mr. Quazi Rahman under the supervision of Dr. Subir Bandyopadhyay. The collaboration is covered in Chapter 3 of the thesis. The author was responsible for adding eta factorization to the arc chain solver developed by Mr Quazi Rahman. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation for eta factorization were performed by the author and the contribution of co-authors was primarily through the provision of constructive comments.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from the co-author to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or

otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

In optical networks, non-bifurcated traffic grooming is known to be a difficult problem, intractable for large networks. One approach is to use the branch and price technique, using the Arc-Chain representation. In this approach the GUB decomposition and implicit column generation can be used to speed up the optimization process. Our objective is to further optimize this approach using Eta Factorization to avoid inverting the basis during each iteration of the revised simplex method.

Dedication

This thesis is dedicated to my parents, my father-in-law, my mother-in-law, my wife Fahima, my son Inan and a special uncle Reaz Salim for their endless support. Also, it is dedicated to my friends at work who support me all the time in every possible way.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr. Subir Bandyopadhyay for his help and guidance during my master's study. His encouragement and support enabled me to complete this research and write the thesis.

I am grateful as well to my external reader Dr. Jagdish Pathak, my internal reader Dr. Dan Wu and my thesis committee chair Dr. Christie Ezeife for their valuable time and suggestions.

Also, I want to thank Mr. Quazi Rahman who helped me a lot during my thesis work.

Lastly, I would like to express my deep and sincere gratitude to my parents, my wife Fahima and my son Inan who have always encouraged and supported me.

Table of Contents

DECLARATION OF CO-AUTHORSHIP ..... **III**
ABSTRACT ..... **V**
DEDICATION ..... **VI**
ACKNOWLEDGEMENTS ..... **VII**
CHAPTER 1: INTRODUCTION **1**
1.1 OBJECTIVE OF THIS THESIS 4
1.2 MOTIVATION 5
1.3 ORGANIZATION OF THESIS 6
CHAPTER 2: BACKGROUND REVIEW **7**
2.1 OPTICAL NETWORKS 7
2.2 OPTICAL FIBER 7
2.3 WAVELENGTH DIVISION MULTIPLEXING (WDM) 9
2.3.1 Advantages of WDM 10
2.3.2 WDM Network Architecture 10
2.3.3 Physical Topology 11
2.3.4 Lightpath 12
2.3.5 Logical Topology 13
2.3.6 Routing in WDM Networks 15
2.4 LINEAR PROGRAMMING (LP) 17
2.5 REVISED SIMPLEX METHOD 19
2.5.1 Steps in one iteration of the Revised Simplex Method 20
2.6 CONGESTION 21
2.7 TRAFFIC GROOMING IN WDM 22
2.7.1 Optical Carrier Level Notation (OC-n) 22
2.7.2 Traffic Grooming Example 22
2.7.3 Bifurcated Traffic Grooming 25
2.7.4 Non-Bifurcated Traffic Grooming 26
2.8 FORMULATIONS FOR TRAFFIC GROOMING 27
2.8.1 Node-Arc Formulation 27
2.8.2 Arc-Chain Formulation 28
2.8.3 Implicit Column Generation 31
2.9 LP FORMULATION USING ARC-CHAIN REPRESENTATION 32
2.9.1 Solving LP using Arc-Chain Representation 34
2.9.2 Finding an Initial Feasible Solution 37
2.9.3 Finding the Entering Column 40
2.9.4 LP using Arc-Chain satisfies GUB Structure 41
2.10 GENERALIZED UPPER BOUNDING (GUB) 42
2.10.1 Updating the matrices R, S, T 47
2.11 ETA MATRIX 52
2.12 ETA FACTORIZATION 52

2.13 REFACTORIZATIONS 57
2.14 BRANCH AND PRICE TECHNIQUE 58
2.14.1 Branch and Bound 58
2.14.2 Branch and Price 59

CHAPTER 3: AN EFFICIENT SCHEME FOR NON-BIFURCATED TRAFFIC GROOMING 61

3.1 ETA FACTORIZATION WITH GUB STRUCTURE IN BRANCH AND PRICE 62
3.2 REVISED SIMPLEX METHOD WITHOUT INVERTING (R–ST) 62
3.3 CALCULATING L_I, P_I, U_I 65
3.4 REPRESENTATION OF L_I, P_I, U_I 72
3.5 CALCULATING J_K, F_K 74
3.6 REPRESENTATION OF J_K, F_K 74
3.7 SOLVING EQUATIONS WITHOUT INVERTING MATRICES 75
3.8 ALGORITHM 78
3.8.1 Algorithm for getting y′ 78
3.8.2 Algorithm for getting d′ 79
3.9 USE OF ETA FACTORIZATION IN THE ALGORITHM 80

CHAPTER 4: EXPERIMENTAL RESULTS 81

4.1 COMPARISON OF SIMPLEX MULTIPLIERS 82
4.2 EXPERIMENTS WITH THE DIFFERENT INTERVAL OF REFACTORIZATIONS 83
4.3 COMPARISON EXPERIMENTS 87

CHAPTER 5: CONCLUSIONS 90

BIBLIOGRAPHY 92

VITA AUCTORIS 95

List of Figures and Tables

FIGURE 2.1: FIBER CABLE AND CROSS-SECTION OF A FIBER [1] 8

FIGURE 2.2: LIGHT PROPAGATION THROUGH A FIBER USING TOTAL INTERNAL REFLECTIONS [1] 9

FIGURE 2.3: WAVELENGTH DIVISION MULTIPLEXING (WDM) ARCHITECTURE [1] 11

FIGURE 2.4: PHYSICAL TOPOLOGY 12

FIGURE 2.5: PHYSICAL TOPOLOGY WITH A LIGHTPATH 13

FIGURE 2.6: LOGICAL TOPOLOGY 14

FIGURE 2.7 AN ALTERNATE ROUTE OF THE LIGHTPATH FROM E_1 TO E_3 SHOWN IN FIGURE 2.5. 15

FIGURE 2.8: A 4-NODE LOGICAL TOPOLOGY AND ITS TRAFFIC MATRIX 16

FIGURE 2.9: A PHYSICAL TOPOLOGY 23

TABLE 2.1: TRAFFIC REQUESTS 23

FIGURE 2.10: A LOGICAL TOPOLOGY 24

FIGURE 2.11: BIFURCATED TRAFFIC GROOMING 25

FIGURE 2.12: NON-BIFURCATED TRAFFIC GROOMING 26

FIGURE 2.13: A 4-END-NODE NETWORK 27

TABLE 2.2: NODE-ARC INCIDENCE MATRIX 28

FIGURE 2.14: A 4-END-NODE NETWORK 29

TABLE 2.3: SOURCE AND DESTINATION FOR THE COMMODITIES IN FIGURE 2.14 29

TABLE 2.4: LOGICAL PATH AND CHAIN (VECTOR) FOR THE COMMODITIES IN FIGURE 2.14 30

FIGURE 2.15: ARC-CHAIN INCIDENCE MATRIX 31

FIGURE 2.16: NETWORK SHOWN IN FIGURE 2.14 WITH THE TRAFFIC FLOWS 38

FIGURE 2.17: A MATRIX WITH GUB STRUCTURE 42

FIGURE 2.18: BASIS SATISFYING GUB STRUCTURE AFTER PERMUTATION 44

FIGURE 2.19: BASIS SATISFYING GUB STRUCTURE AFTER K^{TH} ITERATION 45

FIGURE 2.20: ETA MATRIX 52

FIGURE 2.21: B_K AND B_{K-1} DIFFERS ONLY IN ONE COLUMN WHICH IS THE ENTERING COLUMN A 53

FIGURE 2.22: $B_{K-1}E_K = B_K$ 54

FIGURE 3.1: MATRICES J_K AND F_K 74

TABLE 4.1: COMPARISON OF EXECUTION TIMES WITH DIFFERENT INTERVALS BETWEEN REFACTORIZATIONS WITH AROUND 400 USER REQUESTS 84

FIGURE 4.1: GRAPH OBTAINED FROM TABLE 4.1 85

TABLE 4.2: COMPARISON OF EXECUTION TIMES WITH DIFFERENT INTERVALS BETWEEN REFACTORIZATIONS WITH AROUND 1000 USER REQUESTS 86

FIGURE 4.2: GRAPH OBTAINED FROM TABLE 4.2 87

TABLE 4.3: EXPERIMENTAL RESULTS OF TWO APPROACHES 89

Chapter 1: Introduction

The vision of Information Technology is that data can be stored electronically in any electronic device but can be accessible from everywhere by any other electronic device. These electronic devices can be computers or any other devices that can generate or store data in an electronic form. A network is an interconnection of a group of computers or electronic devices connected for data transfer or data sharing. To achieve fast data transfer or data sharing, optical network is one of the key solutions to this increasing demand for bandwidth [12]. Optical networks are widely used because of its high bandwidth. The first generation of optical networks just replaced copper wires with optical fibers that connect electronic devices such as computers. Second generation optical networks take into account some important issues such as optimizing the use of optical network resources, using the same fiber for carrying multiple optical signals simultaneously, and handling the speed difference between optical signals and electronic circuits as the communication speed of optical signals is far greater than the processing speed of electronic circuits.

Wavelength Division Multiplexing (WDM) is the technology of sending multiple optical signals having different carrier wavelengths through a single fiber. The *physical topology* of a WDM network shows the major physical components of the network with the connections that includes *end-nodes* (typically computers or any other devices that can generate or store data in an electronic form), *routers* that determine how the optical signals are sent to their respective destinations, and *optical fiber links* that connect the routers physically. A *lightpath* is an optical connection from one end-node to another in a

WDM network. A *logical topology* of a WDM network defines how the end-nodes are connected by the lightpaths in a network. A logical topology shows directed edges from one end-node to another. If there is a lightpath from end-node *a* to end-node *b*, it is denoted by $a \rightarrow b$.

The *congestion* of a WDM network is the load on the logical edge that carries the maximum amount of traffic as data. *Minimizing congestion* is a very common and popular objective for optimization. For example, if in a network there are three logical paths p_1, p_2 and p_3 available to send user requests from source end-node *a* to destination end-node *b*, and the loads on p_1, p_2 and p_3 are *0.3*, *0.8* and *0.4* respectively then, in this network, the congestion is *0.8* which is the load on the logical path p_2. If we can minimize this congestion, which is *0.8*, taking into account and balancing the loads on the other logical paths p_1 and p_3, then this network can have more room to handle increases in the traffic from other requests.

Linear programs (*LP*) are often used to solve this kind of optimization problems. An LP formulation includes a linear objective function which has to be either minimized or maximized. The fact that shared resources of the network are limited is expressed using a set of linear constraints in an LP. In our optimization problem, the linear objective function is minimizing the congestion. A frequently used method for solving a linear programming formulation is the *revised simplex method*. In the revised simplex method, in each iteration, one feasible solution is replaced by another "better" solution until an optimal solution is found.

Traffic grooming is a technique in WDM networks that combines a number of low-speed traffic streams, so that the high capacity of each lightpath can be used as efficiently as possible [DR2002, HM2004, ZM2002, ZZM2003]. The capability of handling large networks depends on improving the efficiency of traffic grooming. There are two traffic grooming models, the *bifurcated traffic grooming model* and the *non-bifurcated traffic grooming model*. In the bifurcated traffic grooming model, the traffic from a source to a destination may be divided into as many components as necessary to optimize the network performance and communicated using different logical paths. In the non-bifurcated traffic grooming model, the entire traffic from source to destination must be routed on the same path and be communicated using a single logical path. For this reason, non-bifurcated traffic grooming is an ideal choice for real-time applications that require that their traffic be kept intact.

In Operations Research, a representation called the *arc-chain* representation can be used efficiently to minimize the congestion. Solving the problem by the revised simplex method using the arc-chain representation involves a basis (matrix) of size $(m+q)\times(m+q)$ where m is the number of logical edges and q is the number of user requests. Here, for N end-nodes, the number of user requests or commodities is close to $N(N-1)$, since most end-nodes communicate with each other. If there are, on average, 3 inbound and outbound edges to and from each end-node, then the number of logical edges $m = 3\times N$ and $q = N(N–1)$. As $q >> m$, when the number of end-nodes increases, the value of q increases rapidly.

An arc-chain solver for bifurcated traffic grooming with GUB structure has implemented in our lab by Mr. Quazi Rahman a candidate for the Ph.D (CS) degree of this University. Then a technique called *Branch and Price* has been used to achieve *non-bifurcated traffic grooming* by repeatedly performing optimal *bifurcated traffic grooming*. During the process of branch and price technique, the arc-chain solver is executed again and again to get the optimal solution for the bifurcated traffic grooming.

1.1 Objective of this Thesis

In this thesis, our objective is to optimize the approach implemented by Mr. Quazi Rahman further by using the technique *Eta Factorization* [5] to improve the time required to determine the strategy for traffic grooming. We will discuss how to incorporate the eta factorization with the arc-chain solver implemented by Mr. Quazi Rahman to improve the performance of it and hence the cumulative performance of the approach will be improved.

After implementing our approach, we will describe the experiments to evaluate our approach and compare it with the approach implemented by Mr. Quazi Rahman. We will describe some experiments with our approach to analyze different functionalities of our approach and to see how those functionalities can affect the performance of our approach.

1.2 Motivation

The most expensive operation of revised simplex method is inverting a basis (matrix) in each iteration. So, in each iteration of the revised simplex method using arc-chain representation, the matrix inversion is done on the matrix of size $(m+q)\times(m+q)$.

It is established that the constraints in the arc-chain representation satisfy a special structure called the *Generalized Upper Bounding (GUB)* structure [5]. In operations research, it is well known that if a LP satisfies the GUB structure, operations can be done on a matrix of size $(m\times m)$ instead of carrying out operations on the entire basis which is a matrix of size $(m+q)\times(m+q)$. So, in each iteration of the revised simplex method, with a GUB structure, the most expensive operation of *inverting a matrix* can be done on a matrix of size $(m\times m)$ instead of operations on a matrix of size $(m+q)\times(m+q)$. This improves the time it takes to perform an iteration of the revised simplex method significantly, as $q >> m$.

In the each iteration of the revised simplex method using the arc-chain representation with the GUB structure, the expensive operation of matrix inversion still has to be done, although it is done on a smaller matrix size. If the matrix inversion can be avoided, the time to perform each iteration of revised simplex method can be improved further. Eta factorization is a technique by which matrix operations in the revised simplex method can be done without calculating the matrix from the scratch and without inverting the matrix. Using eta factorization with GUB structure can improve the performance of revised

simplex method further, since no matrix inversion is needed at all in eta factorization [5] [1].

The arc-chain solver implemented by Mr. Quazi can be extended by combining eta factorization with the GUB structure in it. So, in each iteration of the revised simplex method, the matrix operations can be done without calculating the matrix from scratch and without inverting the matrix at all. It improves the time to perform each iteration of the revised simplex method and, as a result, the cumulative time to execute the arc-chain solver is improved further.

Using these approaches we have formulated our algorithm for non-bifurcated traffic grooming in WDM networks and have implemented our algorithm in C. We have tested our program on various network sizes with different traffic loads.

1.3 Organization of Thesis

This thesis is organized as follows. In Chapter 2, we have given a background review in the area of WDM network and optimization. In Chapter 3, we have outlined our formulation for the optimization using eta factorization combined with GUB structure. In Chapter 4, we have given the experimental results we have found after implementing our formulation. In Chapter 5, we have drawn some conclusions and have made some suggestions for future research in this area.

Chapter 2: Background Review

2.1 Optical Networks

The first generation of optical network just replaced copper wires with optical fibers that connect electronic devices such as computers. Optical fibers have become an essential for internet and data-networking infrastructure because of the fast transmission rate (50 terabits per second), low signal attenuation, low signal distortion, low power requirement, low material usage, small space requirement and cost efficiency of such fibers [11].

The second generation optical network takes into account the following issues:

- Optimization – optimizing the use of optical network resources because optical devices are more expensive than electronic devices.
- Use of fibers – using same fiber for carrying multiple optical signals at different carrier wavelengths simultaneously.
- Speed difference – handling the speed difference between optical signals and electronic circuits. Since the communication speed of optical signals is far greater than the processing speed of electronic circuits, this is an important issue.

2.2 Optical Fiber

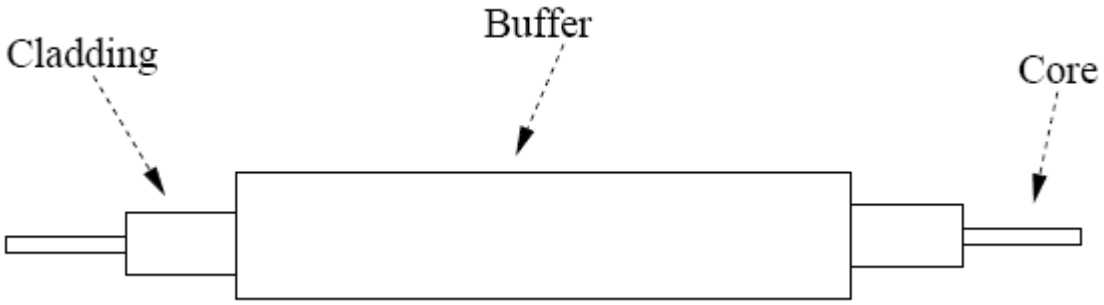
An optical fiber is a very thin glass cylinder or filament that carries optical signals in the form of optical signals [1]. An optical fiber consists of:

Core – cylinder made of silica (SiO_2) with a refractive index μ_1.

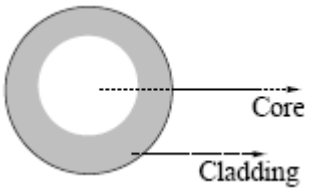
Cladding – cylinder made of silica (SiO_2) with a lower refractive index μ_2.

Buffer – protects and isolates the fiber by encapsulating it surrounding the cladding.

[1] [12]



Fiber Cable



Cross-section of a Fiber

Figure 2.1: Fiber Cable and cross-section of a fiber [1]

Light can travel along a fiber with a relatively low attenuation because of the physical phenomenon of total internal reflection. The core and the cladding have different refractive indexes - μ_1 and μ_2 respectively. As μ_2 is less than μ_1, total internal reflection can occur at the core if the angle of incidence is properly chosen and data as a light signal can propagate through the fiber [12]. Light propagates in optical fibers due to a series of

total internal reflections that occur at the core-cladding interface. Figure 2.2 shows the different refractive indices of core and cladding.

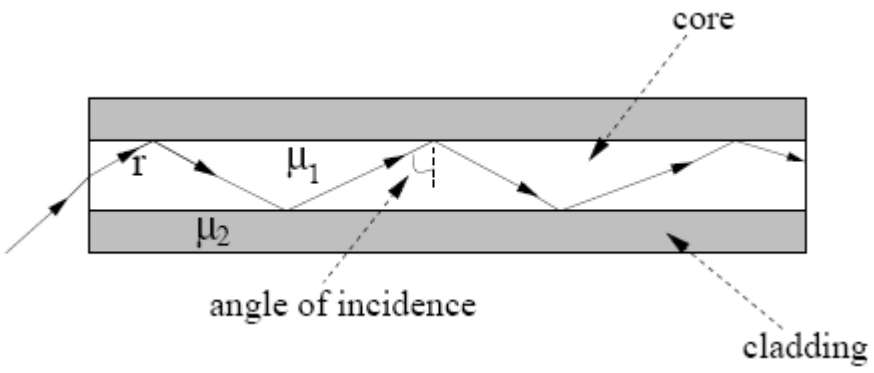


Figure 2.2: Light propagation through a fiber using total internal reflections [1]

The differences between copper wires and fibers as communication media are:

"1. optical devices are much more expensive compared to electronic devices. So it is important to optimize the use of optical network resources,

2. a number of optical signals at different carrier wavelengths may be simultaneously carried by the same fiber,

3. the speed at which optical signals may be communicated is far greater than the speed at which data can be processed by electronic circuits". [1]

2.3 Wavelength Division Multiplexing (WDM)

Wavelength Division Multiplexing (WDM) is a technology that uses multiple optical signals on the same fiber [1]. There is a huge bandwidth mismatch between optical data rate and electronic data rate. WDM uses this mismatch to utilize the capability of a fiber as much as possible by dividing the huge bandwidth of a fiber into many channels (non-

overlapping bands of wavelengths), each operating at a desirable speed, e.g., the peak electronic speed of a few Gb/s [11]. As a result, the transmission capacity of a fiber is improved by having multiple channels at different carrier wavelengths.

2.3.1 Advantages of WDM

The key advantage of optical technology is speed. Other advantages of WDM include [1]:

1. Low signal attenuation: the strength of signal propagating through fibers goes down at a low rate. As a result, the number of optical amplifiers needed is relatively small.
2. Low signal distortion.
3. Low power requirement.
4. Low material usage.
5. Small space requirements.
6. Low cost.

2.3.2 WDM Network Architecture

The architecture of a wavelength-routed WDM network has been discussed in several papers: [12], [13] [15], [4], [8]. Using the WDM network technology, end users can communicate via all-optical WDM channels that may span multiple fiber links. The WDM architecture consists of nodes interconnected by a pair of unidirectional fiber links. Each node has a set of transmitters and a set of receivers to send or receive optical signals at specified carrier wavelengths. Data at the source end-node, represented by an electronic signal, modulates an optical signal at the carrier wavelength corresponding to

the channel. The transmitter directs the modulated signal to the optical fiber connected to the transmitter. At the end-node, a receiver tuned to the same carrier wavelength extracts the data from the incoming optical signal. Router nodes are responsible to re-direct data to the appropriate output port. Figure 2.3 shows an overview of a WDM network.

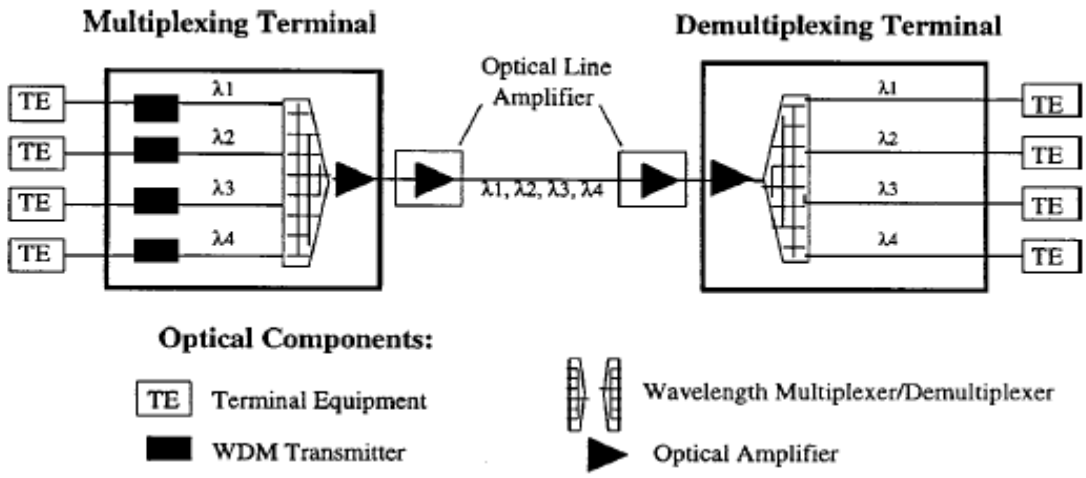


Figure 2.3: Wavelength Division Multiplexing (WDM) Architecture [1]

2.3.3 Physical Topology

A physical topology of a WDM network shows the major physical components of the network with the connections. This includes end-nodes and routers connected by optical fiber links [1].

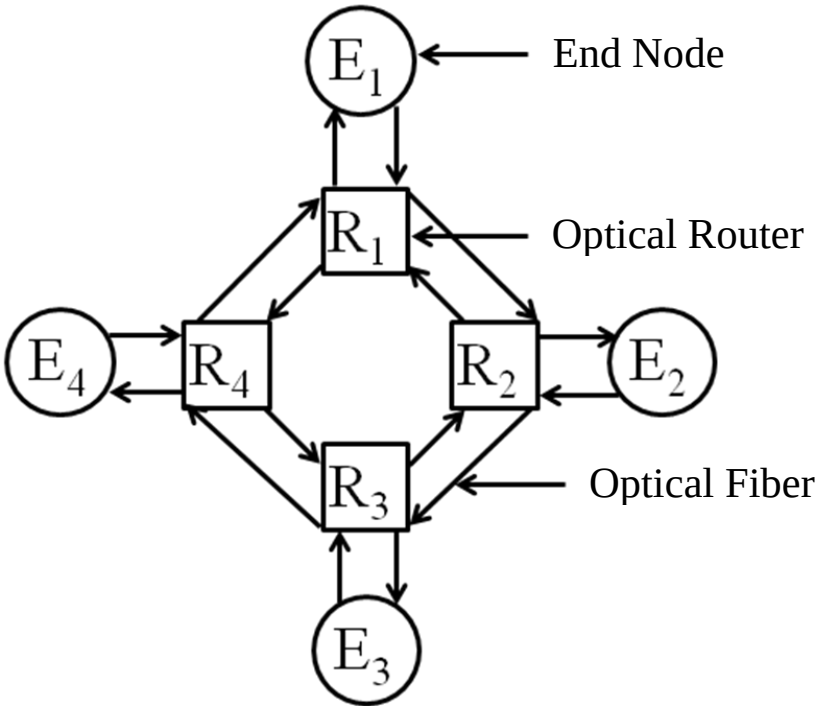


Figure 2.4: Physical Topology

Figure 2.4 shows a physical topology with four end nodes and four routers connected with optical fibers.

2.3.4 Lightpath

A lightpath is an optical connection from one end-node to another. Once the lightpaths are set up, the physical topology is irrelevant for determining a strategy for data communication. A lightpath is defined by a path between end nodes and a wavelength on that path in a network. It provides a "pipe" between end nodes with a bandwidth equal to the bandwidth of the channel. Two lightpaths that share a path in the network must use different wavelengths [15].

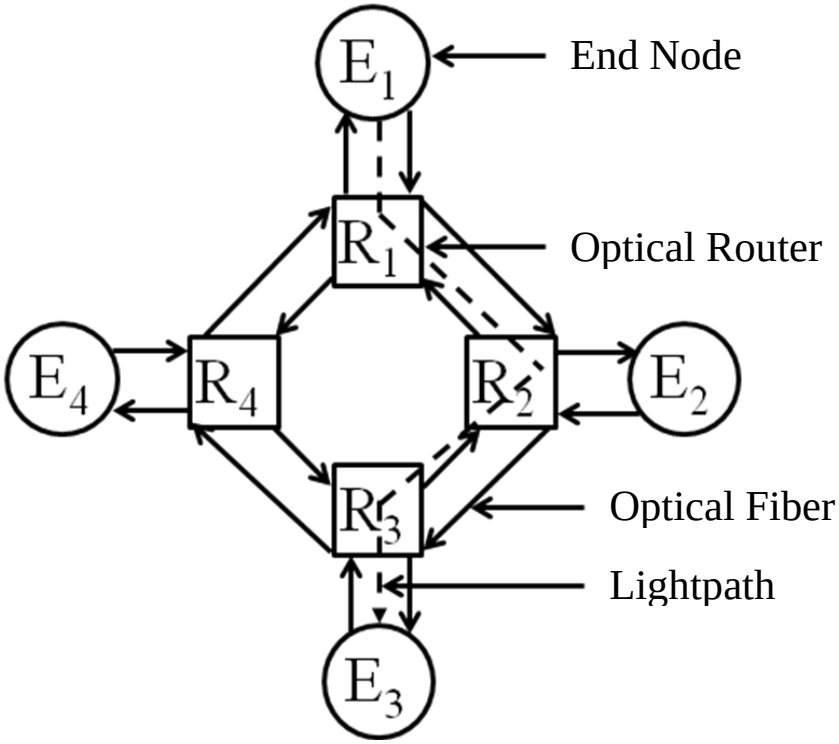


Figure 2.5: Physical Topology with a lightpath

Figure 2.5 shows the physical topology shown in Figure 2.4 with a lightpath. In this figure, the lightpath is established from end-node E_1 to end-node E_3.

2.3.5 Logical Topology

A logical topology defines how the end-nodes are connected by lightpaths in a network. A logical topology shows directed edges from one end-node to another. If there is a lightpath from node *a* to node *b*, it is denoted by $a \rightarrow b$. The logical topology of a WDM network is represented as a directed graph $G_L = (V_L, E_L)$, where V_L is a set of the end-

nodes of the physical topology and E_L is the set of the lightpaths. For routing data, the actual route of a lightpath through physical topology is irrelevant [12].

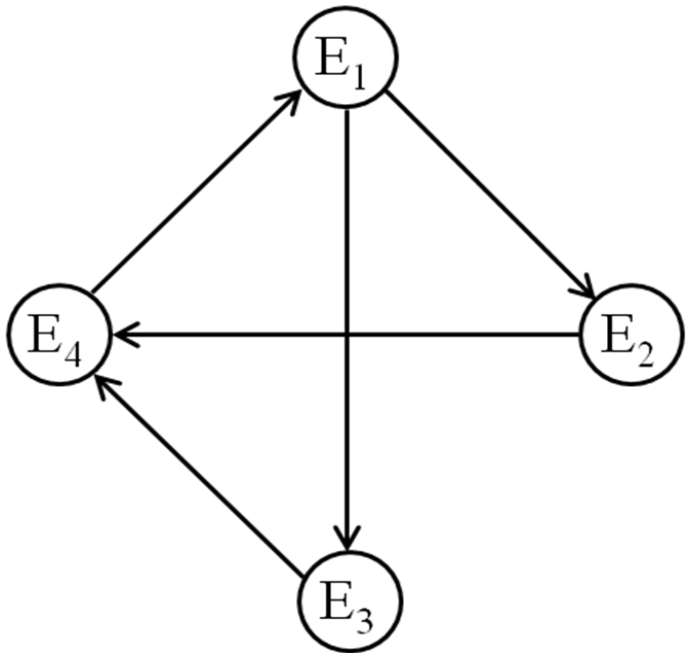


Figure 2.6: Logical Topology

Figure 2.6 shows a logical topology with directed edges. In this logical topology, the directed arc $E_1 \rightarrow E_3$ represents the light path shown in Figure 2.5. Figure 2.7 shows an alternate route of the lightpath from E_1 to E_3 shown in Figure 2.5. Use of the alternate route in Figure 2.7 instead of the route shown in Figure 2.5 does not affect the logical topology.

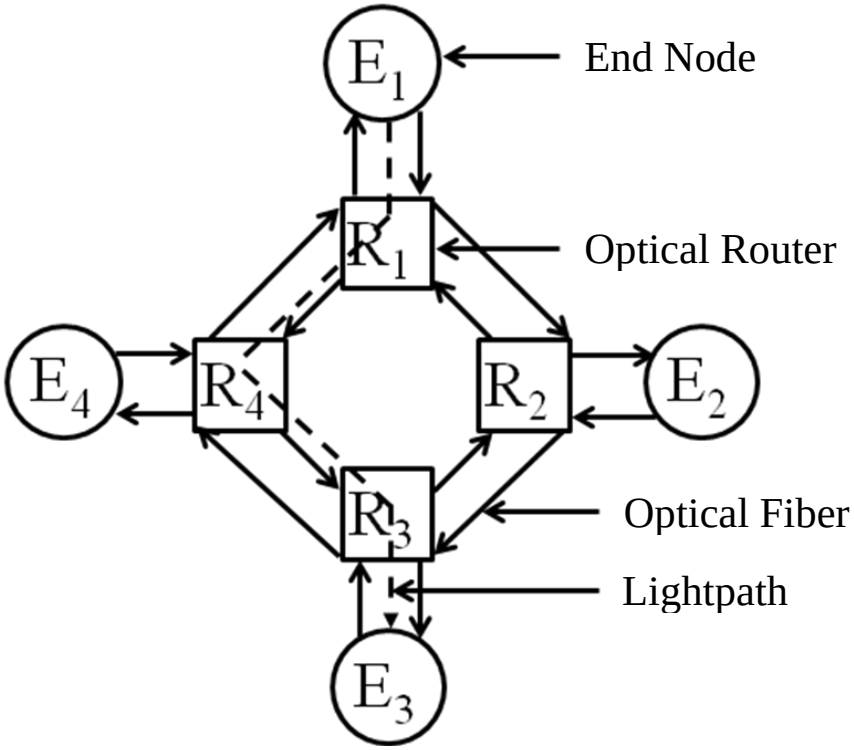
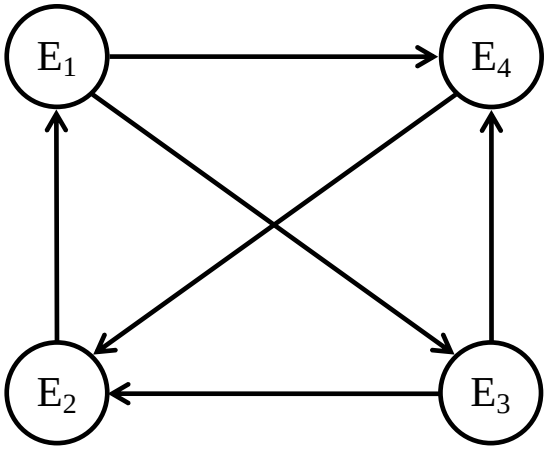


Figure 2.7 An alternate route of the lightpath from E_1 to E_3 shown in Figure 2.5.

2.3.6 Routing in WDM Networks

In the logical topology of a WDM network, the routing strategy is the policy of sending data from all source nodes to their respective destination nodes using appropriate lightpaths. Traffic is routed from a source node *s* to a destination node *d* using one or more sequences of lightpaths from *s* to *d* where each path consists of one or more lightpaths.



Logical Topology

Nodes	E_1	E_2	E_3	E_4
E_1	0	0	5	25
E_2	30	0	0	0
E_3	0	10	0	80
E_4	0	15	0	0

Figure 2.8: A 4-Node logical topology and its traffic matrix

In Figure 2.8, 25 units of data needs to be sent from node E_1 to E_4. This amount of data can be sent using one or both of the following two paths:

Path 1): from node E_1 to node E_4 using logical link $E_1 \rightarrow E_4$,

Path 2): from node E_1 to node E_3 using logical link $E_1 \rightarrow E_3$ and then from node E_3 to node E_4 using the logical link $E_3 \rightarrow E_4$.

We can use either one of the two paths or can distribute the traffic over both paths. For every pair of source destination nodes that has some traffic to route, every possible routes have to be considered in the logical topology. The objective is to route all the traffic in a way that minimize the congestion in the network and thus optimize the use of optical resources in the network.

2.4 Linear Programming (LP)

Linear programs are often used to solve optimization problems. A linear programming formulation includes a linear objective function which has to be either minimized or maximized, a set of linear constraints and a set of restrictions imposed for the underlying decision variables [5].

Following is an example of a linear programming formulation:

Minimize $c_1x_1 + c_2x_2 + \ldots\ldots + c_nx_n$

Subject to $a_{11}x_1 + a_{12}x_2 + \ldots\ldots + a_{1n}x_n \leq b_1$

$a_{21}x_1 + a_{22}x_2 + \ldots\ldots + a_{2n}x_n \leq b_2$

$\ldots\ldots\ldots\ldots\ldots\ldots$

$a_{m1}x_1 + a_{m2}x_2 + \ldots\ldots + a_{mn}x_n \leq b_m$

$x_1, x_2, \ldots\ldots, x_n \geq 0$

In this example,

$c_1x_1 + c_2x_2 + \ldots\ldots + c_nx_n$ is the objective function to be determined,

$\boldsymbol{x}$ represents the vector of decision variables that to be determined,

$\boldsymbol{c}$ and $\boldsymbol{b}$ are vectors of known coefficients,

m inequality constraints that can be represented as a constraints matrix A such that:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

The objective function is of the form: $\sum_{j=1}^{n} c_j x_j$

The constraints are of the form: $\sum_{j} a_{ij} x_j \leq b_i$

So, using the matrix and vector notations, the formulation can be written as:

Minimize $\boldsymbol{cx}$

Subject to $A\boldsymbol{x} \leq \boldsymbol{b}$

Before solving any LP problem, the first step is to transform all the constraints into equality constraints. So, the constraints can be written into:

$$a_{11}x_1 + a_{12}x_2 + \dots\dots + a_{1n}x_n + x_s^{\,1} = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots\dots + a_{2n}x_n + x_s^{\,2} = b_2$$

$$\dots\dots\dots\dots\dots$$

$$a_{m1}x_1 + a_{m2}x_2 + \dots\dots + a_{mn}x_n + x_s^{\,m} = b_m$$

$$x_1, x_2, \dots\dots, x_n \geq 0$$

Here, to transform into equality constraints, non-negative slack variables x_s^i are added where $1 \leq i \leq m$. There are m constraints, so m slack variables were added one for each constraint.

Now, we have an equation of form $A\boldsymbol{x}=\boldsymbol{b}$. In this equation:

A is a $m \times (n+m)$ matrix,

$\boldsymbol{b}$ is a column vector of m known coefficients, and

$\boldsymbol{x}$ is a vector consisting of the n decision variables $x_1, x_2, \ldots, x_n$ followed by the m slack variables $x_s^1, x_s^2, \ldots, x_s^m$.

In the equation $A\boldsymbol{x}=\boldsymbol{b}$, the values for $x_1, x_2, \ldots, x_n$ that satisfy all the constraints of the LP problem is called the *feasible solution* of the problem. A feasible solution that minimizes or maximizes the objective function is called an *optimal solution* and the corresponding value of the objective function is called the *optimal value*. Not every LP problem necessarily has a unique optimal solution. Some problems may have many different solution and some do not have any optimal solution at all [5].

2.5 Revised Simplex Method

The *simplex method* is the first method for solving linear programming formulation introduced by G. B Dantzig in 1947 in which one basic feasible solution is replaced by an adjacent solution. In each iteration of the simplex method, one feasible solution is replaced by another solution. The column in the solution being replaced is called the *leaving column* and the column replacing it is called the *entering column*. The variable

corresponding to the leaving column is called the *leaving variable* and the variable corresponding to the entering column is called the *entering variable*. These resulting implementations of the simplex method are called the *revised simplex method* [5].

2.5.1 Steps in one iteration of the Revised Simplex Method

Notation used:

n_c = number of constraints.

n_v = number of variables.

A = a matrix of size $n_c \times n_v$ where each element in the matrix is a constant, used to specify the constraints.

$[AI]$ = a matrix of size $n_c \times (n_v + n_c)$ where first n_v columns of $[AI]$ are taken from A and the remaining n_c columns of $[AI]$ are taken from identity matrix I.

$\boldsymbol{a}$ = a column of $[AI]$

$\boldsymbol{b}$ = a column vector of n_c nonnegative constants.

$\boldsymbol{c}$ = a row vector of n_v constants.

$\boldsymbol{c_B}$ = a row vector of n_c constants (cost vector).

$\boldsymbol{y}$ = a row vector of n_c variables.

$\boldsymbol{d}$ = a column vector of n_c variables.

$\boldsymbol{x_B}$ = a column vector of n_c variables, called basic variables.

B = a nonsingular matrix of size $n_c \times n_c$.

I = identity matrix of size $n_c \times n_c$.

The following steps are done in each iteration of a revised simplex method:

Step 1: Solve the equation $\boldsymbol{y}.B = \boldsymbol{c_B}$.

Step 2: Find, if possible, an entering column. The entering column may be any column $\boldsymbol{a}$ of *[AI]* such that $\boldsymbol{y}.\boldsymbol{a} < \boldsymbol{c}$. If no entering column found, then the current solution is optimal.

Step 3: Solve the equation $B.\boldsymbol{d} = \boldsymbol{a}$.

Step 4: Find the largest t such that $\boldsymbol{x_B} - t\boldsymbol{d} \geq 0$. If no t found, then the problem is unbounded, otherwise, at least one component of $\boldsymbol{x_B} - t\boldsymbol{d}$ equals 0. That corresponds to the leaving variable and the corresponding column in B is the leaving column.

Step 5: In the basis B, replace the leaving column by the entering column. Recalculate $\boldsymbol{x_B}$ using the formula by $\boldsymbol{x_B} = B^{-1}\boldsymbol{b}$. Replace the cost of the leaving column in $\boldsymbol{c_B}$ by the cost of the entering column.

2.6 Congestion

The congestion is the load Λ_{max} on the logical edge that carries maximum amount of data [1]. Minimizing congestion is a very common and popular objective for optimization for the following reasons:

- Higher value of Λ_{max} means more data is transported by the lightpath carrying that traffic and to handle more data, more time and/or complex electronic hardware is needed.

- If Λ_{max} is less than the capacity of the lightpath, all logical edges may be realized by one single lightpath which reduces the network cost, because each lightpath adds additional cost because of its optical and electronic hardware.

- Lower value of Λ_{max} allows greater possibility of scaling up the traffic without changing the routing strategy in the network [1].

2.7 Traffic Grooming in WDM

Traffic Grooming is a technique in WDM networks that combines a number of low-speed traffic streams from users so that the high capacity of each lightpath may be used as efficiently as possible. Traffic grooming minimizes the network cost in terms of the number of transmitters, receivers, and optical switches [DR2002, HM2004, ZM2002, ZZM2003].

2.7.1 Optical Carrier Level Notation (OC-n)

In traffic grooming, the data communication rate is specified using the Optical Carrier level notation (OC-n). OC-n means n×51.84 Mbps. If the capacity of a lightpath is 10 Gbps or 2.5 Gbps, it is specified as OC-192 or OC-48.

2.7.2 Traffic Grooming Example

For this example, it is assumed that the data communication capacity of a lightpath is 2.5 Gbps or OC-48.

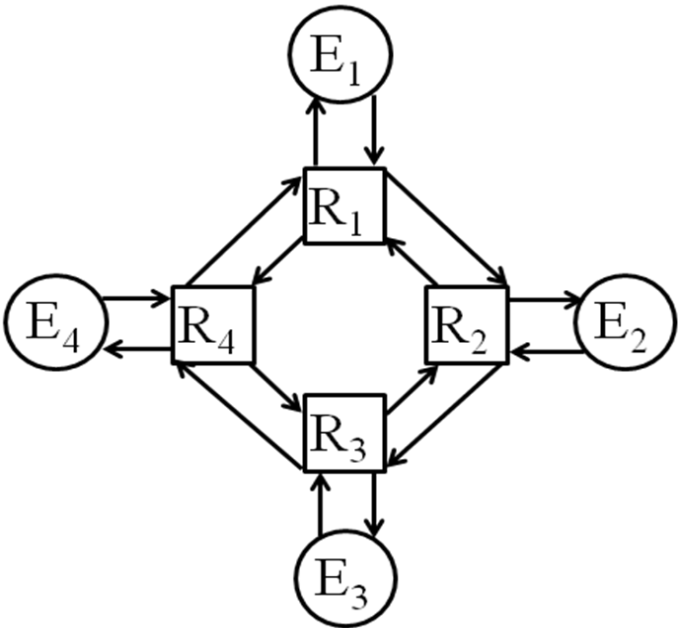


Figure 2.9: A Physical Topology

	E_1	E_2	E_3	E_4
E_1		OC-12 OC-24	OC-12 OC-3 OC-6	OC-6 OC-3 OC-3
E_2	OC-6		OC-3	OC-6 OC-3 OC-3
E_3	OC-12	OC-3 OC-6		OC-6
E_4	OC-3	OC-3		

Table 2.1: Traffic Requests

A possible logical topology to support all requests in Table 2.1 is shown in Figure 2.10. Here lightpath L_1 may be selected to serve the requests:

OC-12 from E_1 to E_3,

OC-3 from E_1 to E_3,

OC-6 from E_1 to E_3,

OC-6 from E_1 to E_4,

OC-3 from E_1 to E_4,

OC-3 from E_2 to E_3,

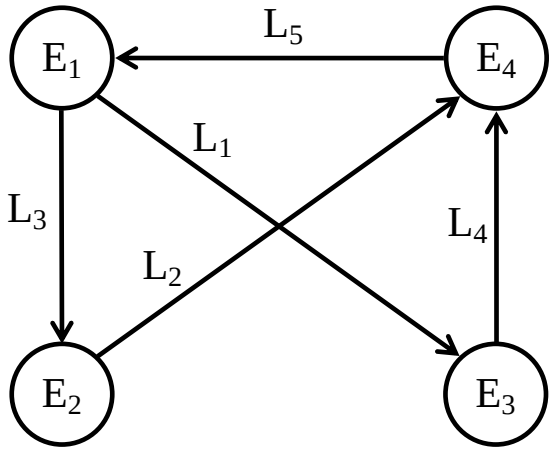


Figure 2.10: A Logical Topology

In this example, the OC-6 request from E_1 to E_4 used the logical path E_1 => E_3 => E_4. But there may be other choices for a valid logical path to handle this OC-6 request from E_1 to E_4. For instance, this request could be handled by using the logical path E_1 => E_2 => E_4 as well.

2.7.3 Bifurcated Traffic Grooming

In the Bifurcated Traffic Grooming model, the traffic $t(s, d)$ from end node E_s to end node E_d may be divided into as many components as necessary to optimize the network performance [7]. That means, a part of the total traffic of an individual request may be split into a number of components and the different components of that request may be communicated using different logical paths.

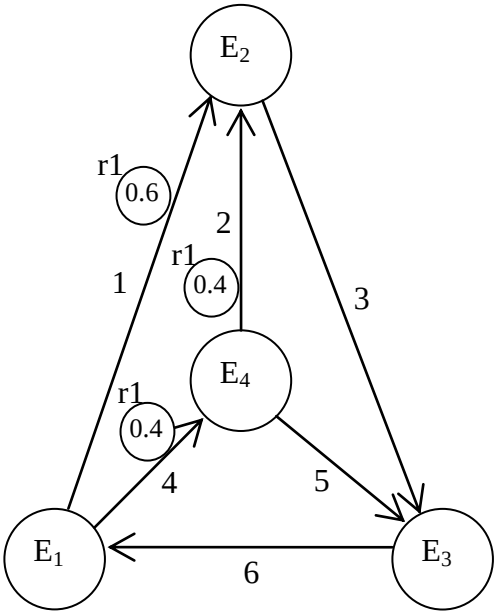


Figure 2.11: Bifurcated Traffic Grooming

Bifurcated Traffic Grooming increases the complexity and the cost of traffic reassembly. It may also introduce delay jitter at the application layer, as many applications, especially real-time applications, require that their traffic be kept intact [10].

2.7.4 Non-Bifurcated Traffic Grooming

In the Non-Bifurcated Traffic Grooming model, the entire traffic $t(s, d)$ from end node E_s to end node E_d must be routed on the same path. That means, the total traffic of an individual request must be communicated using a single logical path.

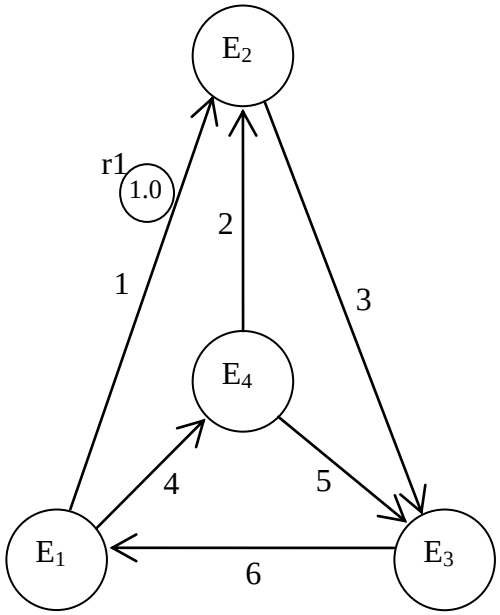


Figure 2.12: Non-Bifurcated Traffic Grooming

Non-Bifurcated Traffic Grooming is also more complex and time consuming as it has to satisfy an additional requirement that each request has to be routed using a single logical path. That means, one additional formulation involving binary variables is needed [1]. Non-Bifurcated Traffic Grooming is an ideal choice for real-time applications that require their traffic be kept intact.

2.8 Formulations for Traffic Grooming

In this section we will describe two possible formulations for traffic grooming – the Node-Arc formulation and the Arc-Chain formulation.

2.8.1 Node-Arc Formulation

A node-arc *incidence matrix* represents a network with N_e end-nodes and m arcs as a - $N_e \times m$ matrix, where the i^{th} row corresponds to the i^{th} end-node and the j^{th} column corresponds to the j^{th} arc. The column corresponding to the arc $i \rightarrow j$ has

- +1 in the row corresponding to node i
- -1 in the row corresponding to node j
- 0 in all other rows

Table 2.2 is the Node-Arc representation for the network shown in Figure 2.13

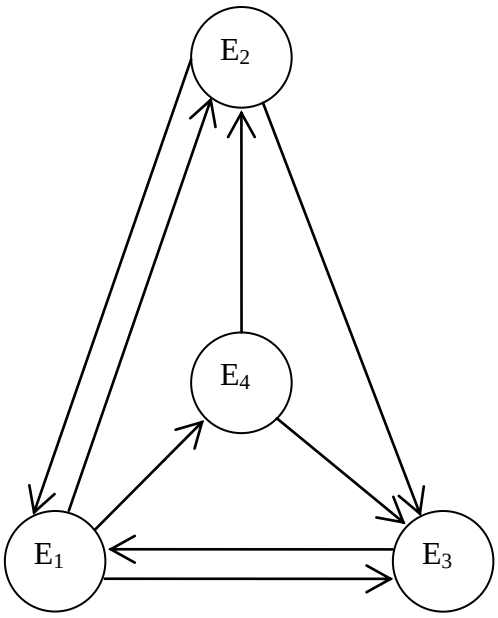


Figure 2.13: A 4-End-Node Network

	(E_1,E_2)	(E_2,E_1)	(E_1,E_3)	(E_3,E_1)	(E_1,E_4)	(E_2,E_3)	(E_4,E_2)	(E_4,E_3)
E_1	1	-1	1	-1	1	0	0	0
E_2	-1	1	0	0	0	1	-1	0
E_3	0	0	-1	1	0	-1	0	-1
E_4	0	0	0	0	-1	0	1	1

Table 2.2: Node-Arc incidence matrix

2.8.2 Arc-Chain Formulation

Definition: "A chain [2] from a source *s* to a destination *d* is a sequence of logical edges $[(s = i_0 \rightarrow i_1), (i_1 \rightarrow i_2),\ldots, (i_{p-1} \rightarrow i_p = d)]$. The logical path described by the chain above is $[(s = i_0 \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{p-1} \rightarrow i_p = d]$." [1]

In a network with *m* logical edges numbered 1, 2,…, *m*, a chain may be represented by a vector of *m 1*'s and *0*'s so that, if the i^{th} element in the chain is 1, then the i^{th} logical edge appears in the chain, and if the i^{th} element in the chain is 0, then the i^{th} logical edge does not appear in the chain for all *i*, where $1 \leq i \leq m$.

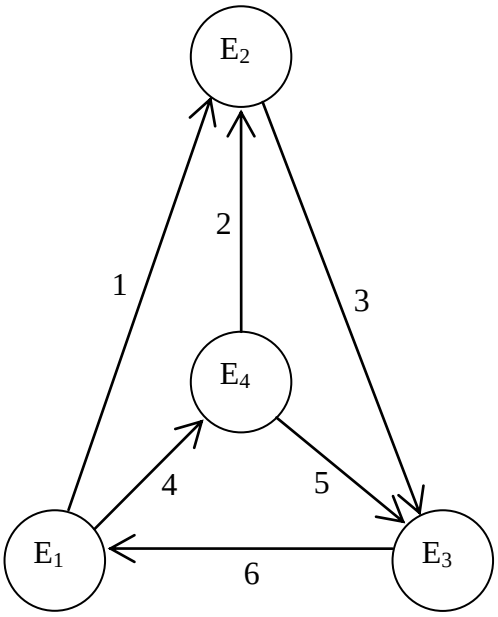


Figure 2.14: A 4-End-Node Network

Figure 2.14 shows a simple logical topology with four end nodes and six logical edges. Each logical edge is assigned a number from 1 to 6 as shown. This network carries three commodities K_1, K_2, and K_3. The source and destination for the commodities are shown in Table 2.3.

Commodity	Source	Destination
K_1	E_1	E_3
K_2	E_2	E_3
K_3	E_4	E_3

Table 2.3: Source and Destination for the Commodities in Figure 2.14

There are three logical paths for K_1, one logical path for K_2, and two logical paths for K_3. The logical paths and chains (vectors) are shown in the Table 2.4.

Commodity	Logical Paths	Chains (Vectors)
K_1	$E_1 \rightarrow E_2 \rightarrow E_3$	[1, 0, 1, 0, 0, 0]
	$E_1 \rightarrow E_4 \rightarrow E_3$	[0, 0, 0, 1, 1, 0]
	$E_1 \rightarrow E_4 \rightarrow E_2 \rightarrow E_3$	[0, 1, 1, 1, 0, 0]
K_2	$E_2 \rightarrow E_3$	[0, 0, 1, 0, 0, 0]
K_3	$E_4 \rightarrow E_3$	[0, 0, 0, 0, 1, 0]
	$E_4 \rightarrow E_2 \rightarrow E_3$	[0, 1, 1, 0, 0, 0]

Table 2.4: Logical Path and Chain (Vector) for the Commodities in Figure 2.14

Arc-Chain Incidence Matrix: "A network having *m* edges and *q* commodities may be represented by an *arc-chain incidence matrix AC*. If there are n^k chains for the k^{th} commodity K^k, for all *k*, $1 \leq k \leq q$, *AC* is an $m \times n$ matrix where $n^1 + n^2 + \ldots + n^q$ is the total number of chains for all commodities. Each chain of a commodity corresponds to a column in matrix *AC* so that the first n^1 columns of *AC* correspond to chains for commodity K^1, the n^2 columns correspond to chains for commodity K^2, and so on." [1]

Figure 2.15 shows the arc-chain incidence matrix *AC* of the network shown in Figure 2.14 for three commodities K_1, K_2, and K_3.

$$AC = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.15: Arc-Chain Incidence Matrix

In Figure 2.15, the rows of the arc-chain matrix AC corresponds to the edges 1, 2, 3, 4, 5, 6 of the network shown in Figure 2.12. Columns 1, 2, and 3 correspond to the chain for commodity K_1, column 4 corresponds to the chain for commodity K_2, and the columns 5 and 6 correspond to the chain for commodity K_3.

2.8.3 Implicit Column Generation

In arc-chain formulation, for each commodity there are may be many possible chains available. So, storing all possible chains for all commodities is not feasible. This problem can be handled by adapting Tomlin's approach for solving a minimum cost multi-commodity flow problem [16]. Instead of explicitly storing the constraints as done in an LP solver, Tomlin's approach implicitly keeps track of the constraints and generates a chain on the fly, only when it is established that the chain should be part of the column entering the basis.

2.9 LP Formulation using Arc-Chain Representation

The linear programming problem using arc-chain representation is as follows [1]:

$$\textit{Minimize} \quad \Lambda_{max} \qquad \ldots \text{(eq. 2.1)}$$

$$\textit{Subject to} \quad \begin{aligned} AC_1x_1 + AC_2x_2 + \ldots\ldots + AC_qx_q &\leq \Lambda \\ e_1x_1 &= \Gamma_1 \\ e_2x_2 &= \Gamma_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ e_qx_q &= \Gamma_q \end{aligned} \qquad \ldots \text{(eq. 2.2)}$$

Here, the objective function is Λ_{max}, the congestion,

m = number of logical edges.

q = number of commodities (requests).

K_k = k^{th} commodity.

n = total number of chains for all commodities.

n_k = number of chains for k^{th} commodity.

AC = arc-chain incidence matrix of size $(m+q)\times n$.

AC_k = sub-matrix of AC of size $(m\times n_k)$ that corresponds to all the chains of commodity K_k.

ac_k^{ij} = i^{th} element of j^{th} chain of commodity K_k where $ac_k^{ij} = 1$ if edge i appears in chain j of commodity K_k and $ac_k^{ij} = 0$ otherwise.

Γ_k = traffic $t(src(k), dest(k))$ for commodity k, for all k, $1 \leq k \leq q$.

x_k = column vector of variables containing n_k elements.

x_k^j = variable denoting the flow over j^{th} chain of K_k commodity,

for all k, $0 \leq x_k^j \leq \Gamma_k$.

e_k = row vector of n_k *1*'s, ([1, 1, …, 1]) .

Λ = column vector with m occurrences of Λ_{max} ([Λ_{max}, Λ_{max}, …, Λ_{max}]).

x_s = column vector of m slack variables.

x_s^i = i^{th} slack variable.

B = basis matrix for the revised simplex method of size $(m+q)\times(m+q)$.

c_B = vector of $(m+q)$ cost coefficients.

The first line of the equation (eq. 2.2) corresponds to the edges in the logical topology. The right hand side of the equation (eq. 2.2) represents the column vector of Λ_{max} of size m which is [Λ_{max}, Λ_{max}, …, Λ_{max}]. The left hand side is $\sum_{k=1}^{q} AC_k x_k$. where AC_k is a sub-matrix of AC having size of $(m\times n_k)$ and x_k is a column vector of size m. For logical edge i, $\sum_{k=1}^{q}\sum_{j=1}^{n_k} ac_k^{ij}.\ x_k^j \leq \Lambda_{max}$, which is the sum of i^{th} elements of all the products $AC_k x_k$ should be less than or equal to the congestion Λ_{max}.

Lines *2* to $q+1$ of the equation (eq. 2.2) correspond to the commodities (requests). So, the constraint for commodity k is $e_k.x_k = \Gamma_k$ for all i where $1 \leq k \leq q$. But e_k is a row vector of n_k *1*'s ([1, 1, …, 1]). So, the constraints become $\sum_{j=1}^{n_k} x_k^j = \Gamma_k$, which is the sum of the flows for k^{th} commodity using all chains for k^{th} commodity must be equal to the requirement Γ_k.

2.9.1 Solving LP using Arc-Chain Representation

In the revised simplex method, the first step to solve a LP problem is to remove the inequalities by adding slack variables. So, after adding slack variable to remove the inequality constraints in the first line of the equation (eq. 2.2), it becomes:

$$\left.\begin{array}{lcl} AC_1x_1 + AC_2x_2 + \ldots\ldots + AC_qx_q \;\; + \;\; x_s & = & \Lambda \\ e_1x_1 & = & \Gamma_1 \\ \quad e_2x_2 & = & \Gamma_2 \\ \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & \\ \quad\quad e_qx_q & = & \Gamma_q \end{array}\right| \qquad \ldots \text{(eq. 2.3)}$$

Considering the network shown in Figure 2.14 and the three commodities described in Table 2.4, let the traffic demand Γ_1, Γ_2, and Γ_3 for commodities K_1, K_2, and K_3 are 0.3, 0.5, and 0.7 respectively. So, the constraints in the equation (eq. 2.3), in matrix form is:

$$\left[\begin{array}{cccc|cc|cccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] . \left[\begin{array}{c} x_1^{\,1} \\ x_1^{\,2} \\ x_1^{\,3} \\ x_2^{\,1} \\ x_3^{\,1} \\ x_3^{\,2} \\ x_s^{\,1} \\ x_s^{\,2} \\ x_s^{\,3} \\ x_s^{\,4} \\ x_s^{\,5} \\ x_s^{\,6} \end{array}\right] = \left[\begin{array}{c} \Lambda_{max} \\ \Lambda_{max} \\ \Lambda_{max} \\ \Lambda_{max} \\ \Lambda_{max} \\ \Lambda_{max} \\ 0.3 \\ 0.5 \\ 0.7 \end{array}\right]$$

$\ldots$ (eq. 2.4)

In this constraints matrix (eq. 2.4):

- The first m rows correspond to the first line of the equation (eq. 2.3) that corresponds to 6 logical edges in this example.
- The last q rows correspond to the remaining lines of the equation (eq. 2.3) that correspond to the 3 commodities in this example.
- For commodity K_1:
  - The columns 1, 2 and 3 correspond to the three chains of this commodity.
  - The variables x_1^1, x_1^2 and x_1^3 correspond to the flows for these three chains.
- For commodity K_2:
  - Column 4 corresponds to the one chain of this commodity.
  - Variable x_2^1 corresponds to the flows for this one chain.
- For commodity K_3:
  - Columns 5 and 6 correspond to the two chains of this commodity.
  - Variables x_3^1 and x_3^2 correspond to the flows for these two chains.
- Variables x_s^1, x_s^2, x_s^3, x_s^4, x_s^5 and x_s^6 are the slack variables.
- The identity matrix in rows 1 to 6 and columns 7 to 12 correspond to the slack variables.

After taking the variable Λ to the left hand side in the equation (eq. 2.3), the constraints become:

$$\left.\begin{array}{rcl} -\Lambda + AC_1x_1 + AC_2x_2 + \ldots\ldots + AC_qx_q + x_s &=& 0 \\ e_1x_1 &=& \Gamma_1 \\ e_2x_2 &=& \Gamma_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & & \\ e_qx_q &=& \Gamma_q \end{array}\right| \quad \ldots \text{(eq. 2.5)}$$

The matrix form of the equation (eq. 2.5) is as follows:

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} \Lambda_{max} \\ x_1^{\,1} \\ x_1^{\,2} \\ x_1^{\,3} \\ x_2^{\,1} \\ x_3^{\,1} \\ x_3^{\,2} \\ x_s^{\,1} \\ x_s^{\,2} \\ x_s^{\,3} \\ x_s^{\,4} \\ x_s^{\,5} \\ x_s^{\,6} \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}$$

… (eq. 2.6)

In this constraints matrix (eq. 2.6):

- In each column, for the values in the first m positions corresponding to the flow variable x_k^j is coming from the chain C_k^j in AC_k. The values in the remaining q positions are all 0's except for the position $m+k$ where the value is 1.
- In the column for Λ_{max}, the values in the first m positions are –1 and in the remaining q positions the values are 0.
- In the column for slack variable x_s^i, the values are all 0's except for the position i where the value is 1.

2.9.2 Finding an Initial Feasible Solution

The total number of constraints in the equation (eq. 2.3) is $m+q$. So, the basis in the equation (eq. 2.6) should be a $(m+q)\times(m+q)$ matrix. But in (eq. 2.6), the basis is not a $(m+q)\times(m+q)$ matrix. A technique [1] can be used to create the basis B of size $(m+q)\times(m+q)$ and the x_B denoting the vector corresponding to the basis variables. To find the *initial feasible solution*, the idea of the technique is to select only one chain from each commodity in the basis B and adjust the slack variables accordingly.

For the network shown in Figure 2.14 with the six logical edges ($1, 2, 3, 4, 5, 6$) and three commodities (K_1, K_2, K_3) having the traffic demands 0.3, 0.5 and 0.7 will have the basis of size $(m+q)\times(m+q)$ which is $(6+3)\times(6+3) = 9\times9$ matrix. The steps for finding the basis corresponding to the *initial feasible solution* are as follows:

Step 1: Choose any one chain from each commodity and send the entire traffic for the commodity through the selected chain. In this example, let's select the chains:

$E_1 \rightarrow E_2 \rightarrow E_3$ ([1, 0, 1, 0, 0, 0]) from commodity K_1,

$E_2 \rightarrow E_3$ ([0, 0, 1, 0, 0, 0]) from commodity K_2 and

$E_4 \rightarrow E_2 \rightarrow E_3$ ([0, 1, 1, 0, 0, 0]) from commodity K_3.

Figure 2.16 shows the traffic flow on each logical edge.

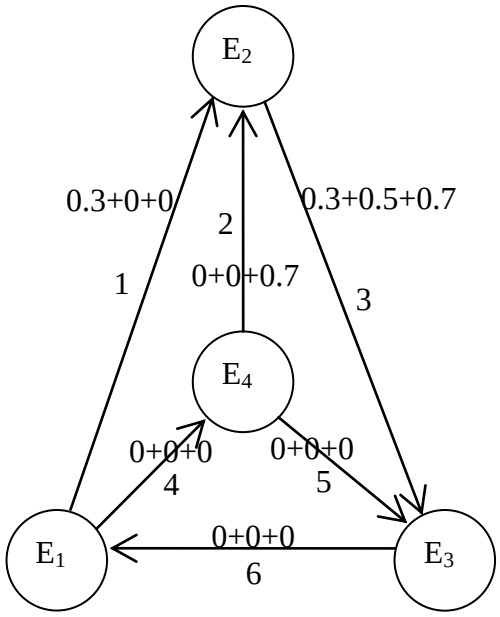


Figure 2.16: Network shown in Figure 2.14 with the traffic flows

Step 2: Calculate the sum of the flows on each logical edge and find the logical edge carrying the maximum flow Λ_{max}. In this example, the logical edge 3 is carrying the maximum flow of *1.5*. So, the value of Λ_{max} is *1.5*.

Step 3: In the basis, create the columns for Λ_{max} and the three chains selected from the three commodities K_1, K_2 and K_3. The values for the corresponding basis variables are *1.5*, *0.3*, *0.5* and *0.7* respectively.

Step 4: The logical edge *3* is carrying the maximum flow $\Lambda_{max} = 1.5$. So, no slack variable is needed for the logical edge *3*. All the remaining logical edges need slack variables and the values of these slack variables need to be adjusted to satisfy the constraints in the first line of the equation (eq. 2.5). So, the values of the slack variables are calculated as follows:

Slack variable for the logical edge 1

$\Rightarrow$ $-1\times1.5 + 1\times0.3 + 0\times0.5 + 0\times0.7 + 1\times x_s^1 = 0$

$\Rightarrow$ $x_s^1 = 1.2$

Slack variable for the logical edge 2

$\Rightarrow$ $-1\times1.5 + 0\times0.3 + 0\times0.5 + 1\times0.7 + 1\times x_s^2 = 0$

$\Rightarrow$ $x_s^2 = 0.8$

Slack variable is not needed for the logical edge 3

Slack variable for the logical edge 4

$\Rightarrow$ $-1\times1.5 + 0\times0.3 + 0\times0.5 + 0\times0.7 + 1\times x_s^4 = 0$

$\Rightarrow$ $x_s^4 = 1.5$

Slack variable for the logical edge 5

$\Rightarrow$ $-1\times1.5 + 0\times0.3 + 0\times0.5 + 0\times0.7 + 1\times x_s^5 = 0$

$\Rightarrow$ $x_s^5 = 1.5$

Slack variable for the logical edge 6

$\Rightarrow \quad -1\times 1.5 + 0\times 0.3 + 0\times 0.5 + 0\times 0.7 + 1\times x_s^{\,6} \quad = \quad 0$

$\Rightarrow \quad x_s^{\,6} = 1.5$

So, the resulting basis matrix for the *initial feasible solution* is as follows:

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 1.5 \\ 0.3 \\ 0.5 \\ 0.7 \\ 1.2 \\ 0.8 \\ 1.5 \\ 1.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}$$

… (eq. 2.7)

2.9.3 Finding the Entering Column

In step 2, during each iteration of revised simplex method, we have to find, if possible, an entering column. It is done by first calculating the simplex multipliers in step 1. So,

Step 1: Calculate the simplex multipliers $\boldsymbol{y}$ by solving the equation $\boldsymbol{y} = \boldsymbol{c_B}\, B^{-1}$.

Step 2: Find an entering column. The entering column may be any column $\boldsymbol{a}$ of the constraints matrix where $\boldsymbol{ya}$ is less than the corresponding component of c_N where c_N is the cost associated with the column $\boldsymbol{a}$.

Since the number of possible chains for all commodities in a large network is very high, it is not feasible to store all the chains in the constraint matrix. Instead, Tomlin's *implicit column generation* approach can be used. This approach implicitly keeps track of the constraints and generates a chain on the fly only when it is established that the chain should be part of the column entering the basis [16]. The chain is then checked whether it satisfies the condition in step 2 to be part of the entering column. If it is satisfied, then the entering column is created using this chain.

2.9.4 LP using Arc-Chain satisfies GUB Structure

After analyzing the values in the columns of the constraints matrix (eq. 2.6) that corresponds to the equation (eq. 2.5), it is found that:

- **The column corresponds to Λ_{max} –** in this column, the last q rows have all 0's.
- **The columns correspond to chains –** in these six columns, the last q rows have 1 only in the position $m+k$. In the remaining position there are all 0's.
- **The columns correspond to slack variables –** in the columns for slack variables, the last q rows have all 0's.

So, the basis matrix *B* in the *arc-chain* representation satisfies the GUB structure. Now, the matrix *B* can be re-arranged to form the structure:

$$B = \begin{bmatrix} R & S \\ T & I \end{bmatrix}$$

by applying a permutation on the columns of the matrix and re-arrange the variables vector accordingly.

2.10 Generalized Upper Bounding (GUB)

Generalized Upper Bounding (GUB) [5] is a technique that makes the revised simplex method more efficient for linear programming problems having a special structure in the basis. A $(m+q)\times(m+q)$ matrix exhibits a GUB structure, if

- q is relatively large number compared to m, and
- each column of the last q rows has at most one nonzero entry, the non-zero entry being equal to 1.

Figure 2.17 shows a constraints matrix *A* with the GUB structure.

$$A = \begin{bmatrix} 5 & 3 & 4 & 7 & 9 & 2 & 3 & 8 & 4 & 6 & 7 & 5 & 8 & 4 \\ 7 & 2 & 5 & 8 & 3 & 7 & 6 & 2 & 9 & 5 & 2 & 9 & 3 & 6 \\ 1 & 1 & 1 & & & & & & & & & & & \\ & & & 1 & 1 & 1 & 1 & & & & & & & \\ & & & & & & & 1 & 1 & 1 & & & & \\ & & & & & & & & & & 1 & 1 & & \end{bmatrix}$$

Figure 2.17: A matrix with GUB structure

In the revised simplex method outlined in section 2.5, to get the vectors $\boldsymbol{y}$ and $\boldsymbol{d}$, in each iteration in *step 1* and *step 3*, we need to solve the equations:

$$\boldsymbol{y} = \boldsymbol{c_B} B^{-1} \qquad \text{and}$$

$$\boldsymbol{d} = B^{-1}\boldsymbol{a}$$

To calculate these two equations, in each iteration, we have to invert the basis B which is a matrix of size $(m+q)\times(m+q)$. When the number of end-nodes increases, the value of q becomes a very large number as $q = N(N–1)$ where N represents end-node. So, if the basis B can be arranged so that it will satisfy the GUB structure, the most expensive operation in each iteration – inverting a matrix, can be done on a smaller matrix of size m. In the case where q is much larger than m, the GUB technique can dramatically improve the time to calculate these two equations.

If we have a non-singular basis B satisfying the GUB structure, we can always apply a permutation on the columns of the matrix so that the resulting matrix has an identity matrix of size $q\times q$ in its lower right corner:

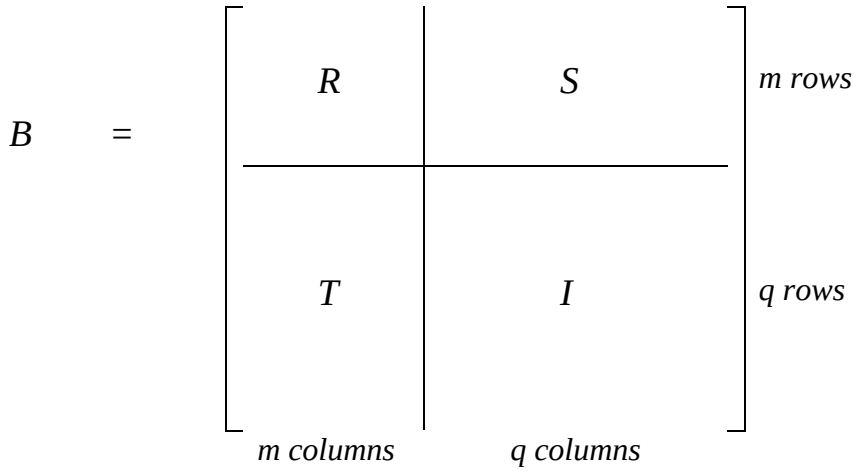


Figure 2.18: Basis satisfying GUB structure after permutation

Figure 2.18 shows the structure of basis *B* after applying the column permutation. Here *R*, *S*, *T*, and *I* are matrices of size *(m×m)*, *(m×q)*, *(q×m)*, and *(q×q)* respectively. As a result, the structure shows that the matrices *R*, *S*, *T*, and *I* are the four smaller sub-matrices of the basis *B*.

In each iteration of revised simplex method, we need to solve the equations $\boldsymbol{y} = \boldsymbol{c_B}B^{-1}$ and $\boldsymbol{d} = B^{-1}\boldsymbol{a}$ for the basis *B* of size *(m+q)×(m+q)*. Let B_k be the basis at iteration *k*. So, after k^{th} iteration, the basis B_k after applying permutation will be:

$$B_k = \left[\begin{array}{c|c} R_k & S_k \\ \hline T_k & I \end{array}\right] \begin{array}{l} m \text{ rows} \\ q \text{ rows} \end{array}$$

m columns $\quad$ q columns

Figure 2.19: Basis satisfying GUB structure after k^{th} iteration

Let $\boldsymbol{y'}$ be the vector of the first m elements and $\boldsymbol{y''}$ be the vector of the last q elements of the *simplex multipliers* $\boldsymbol{y}$, and let $\boldsymbol{c_B'}$ be the first m elements and $\boldsymbol{c_B''}$ be the last q elements of the cost vector $\boldsymbol{c_B}$. So, the equation $\boldsymbol{y}.B_k = \boldsymbol{c_B}$ can be rewritten as:

$$\left[\begin{array}{cc} y' & y'' \end{array}\right] . \left[\begin{array}{cc} R_k & S_k \\ T_k & I \end{array}\right] = \left[\begin{array}{cc} c_B' & c_B'' \end{array}\right]$$

This equation can be broken down into:

$$\boldsymbol{y'}R_k + \boldsymbol{y''}T_k = \boldsymbol{c_B'} \qquad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.8)}$$

and

$$\boldsymbol{y'}S_k + \boldsymbol{y''} = \boldsymbol{c_B''} \qquad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.9)}$$

$$\Rightarrow \quad \boldsymbol{y''} = \boldsymbol{c_B''} - \boldsymbol{y'}S_k \qquad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.10)}$$

After substituting the value for y'' from the equation (eq. 2.10) into the equation (eq. 2.8), the obtained equation is:

$$\boldsymbol{y'}(R_k - S_k T_k) = \boldsymbol{c_B'} - \boldsymbol{c_B''} T_k \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{(eq. 2.11)}$$

$$\Rightarrow \quad \boldsymbol{y'} = (\boldsymbol{c_B'} - \boldsymbol{c_B''} T_k).(R_k - S_k T_k)^{-1} \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{(eq. 2.12)}$$

So, the value of $\boldsymbol{y'}$ can be found using the equation (eq. 2.12).

Here, the size of matrix $(R_k - S_k T_k)$ is $(m \times m)$. So, to calculate $\boldsymbol{y'}$ the expensive operation of inverting a matrix can be done on the matrix of size $(m \times m)$ instead of the matrix of size $(m+q) \times (m+q)$. After calculating the value of $\boldsymbol{y'}$, the value of $\boldsymbol{y''}$ can be found using the value of $\boldsymbol{y'}$ in equation (eq. 2.10).

Similarly, let $\boldsymbol{d'}$ be the vector of the first m elements and $\boldsymbol{d''}$ be the vector of the last q elements of the *d-vector* $\boldsymbol{d}$, and let $\boldsymbol{a'}$ be the first m elements and $\boldsymbol{a''}$ be the last q elements of the entering column $\boldsymbol{a}$. So, the equation $B_k.\boldsymbol{d} = \boldsymbol{a}$ can be rewritten as:

$$\begin{bmatrix} R_k & S_k \\ T_k & I \end{bmatrix} . \begin{bmatrix} d' \\ d'' \end{bmatrix} = \begin{bmatrix} a' \\ a'' \end{bmatrix}$$

This equation can be broken down into:

$$R_k \boldsymbol{d'} + S_k \boldsymbol{d''} = \boldsymbol{a'} \quad \text{..... (eq. 2.13)}$$

and

$$T_k \boldsymbol{d'} + \boldsymbol{d''} = \boldsymbol{a''} \quad \text{..... (eq. 2.14)}$$

$$\Rightarrow \quad \boldsymbol{d''} = \boldsymbol{a''} - T_k \boldsymbol{d'} \quad \text{..... (eq. 2.15)}$$

After substituting the value for d'' from the equation (eq. 2.15) into the equation (eq. 2.13), the obtained equation is:

$$(R_k - S_k T_k)\boldsymbol{d'} = \boldsymbol{a'} - S_k \boldsymbol{a''} \quad \text{..... (eq. 2.16)}$$

$$\Rightarrow \quad \boldsymbol{d'} = (\boldsymbol{a'} - S_k \boldsymbol{a''}).(R_k - S_k T_k)^{-1} \quad \text{..... (eq. 2.17)}$$

So, the value of $\boldsymbol{d'}$ can be found using the equation (eq. 2.17).

Here, the matrix $(R_k - S_k T_k)$ was already inverted in the equation (eq. 2.12) and readily available. After calculating the value of $\boldsymbol{d'}$, the value of $\boldsymbol{d''}$ can be found using the value of $\boldsymbol{d'}$ in equation (eq. 2.15).

2.10.1 Updating the matrices R, S, T

In each iteration of the revised simplex method, the basis B is updated by replacing the leaving column with the entering column. So, the matrices R, S, T and hence $(R - ST)$ change after each iteration. Let B_k, R_k, S_k and T_k be the matrices at iteration K and B_{k+1}, R_{k+1}, S_{k+1} and T_{k+1} be the matrices at iteration $k+1$. To maintain the GUB structure after

each iteration, depending on the leaving column position, there are three cases should be considered [5]:

Case 1 - The leaving column is one of the first *m* columns of B_k:

In this case S_k and I will not be effected. The leaving column is simply replaced by the entering column and the new basis will still satisfy the GUB structure. So, the basis:

$$B_k = \begin{bmatrix} R_k & S_k \\ T_k & I \end{bmatrix}$$

will be updated into:

$$B_{k+1} = \begin{bmatrix} R_{k+1} & S_k \\ T_{k+1} & I \end{bmatrix}$$

Hence, the matrix $(R_k - S_kT_k)$ will be updated into $(R_{k+1} - S_kT_{k+1})$.

The matrices R_k and R_{k+1}, and the matrices T_k and T_{k+1}, differ only in their p^{th} column. So, the matrix $(R_k - S_kT_k)$ and the matrix $(R_{k+1} - S_kT_{k+1})$, differ only in their p^{th} column. The p^{th} column of $(R_{k+1} - S_kT_{k+1})$ is $\boldsymbol{a'} - S_k\boldsymbol{a''} = (R_k - S_kT_k)\boldsymbol{d'}$ (eq. 2.16).

So, after $k+1$ iteration:

$$R_{k+1} - S_{k+1}T_{k+1} = (R_k - S_kT_k)F_{k+1} \qquad \text{..... (eq. 2.18)}$$

where F_{k+1} stands for a *(m×m) eta matrix* [5] whose *eta column* in the p^{th} position is $\boldsymbol{d'}$.

Case 2 - The leaving column f is one of the last q columns of B_k and some other column g of B_k satisfies $g'' = f''$:

Here, g should be one of the first m columns of B_k. f' and g' denote vectors consisting of the first m elements and f'' and g'' denote vectors consisting of the last q elements of f and g. To get B_{k+1}, first f is replaced by g to preserve the $(q \times q)$ identity matrix in the lower right corner. Then the entering column $\boldsymbol{a}$ is inserted into the position formerly occupied by g. So, updating B_k into B_{k+1} is done in two stages:

Stage 1: A temporary matrix B_{k_temp} is generated by interchanging the two columns f and g in B_k:

$$B_k = \begin{bmatrix} R_k & S_k \\ T_k & I \end{bmatrix}$$

Updated into:

$$B_{k\_temp} = \begin{bmatrix} R_{k\_temp} & S_{k+1} \\ T_k & I \end{bmatrix}$$

Stage 2: B_{k+1} is generated by replacing f by the entering column $\boldsymbol{a}$ in B_{k_temp}:

$$B_{k\_temp} = \begin{bmatrix} R_{k\_temp} & S_{k+1} \\ T_k & I \end{bmatrix}$$

updated into:

$$B_{k+1} = \begin{bmatrix} R_{k+1} & S_{k+1} \\ T_{k+1} & I \end{bmatrix}$$

For stage 1:

Let f is the $(m+i)^{\text{th}}$ and g is the l^{th} column of B_k, and let r denote the i^{th} row of T_k and J_{k+1} denote the $(m \times m)$ identity matrix whose l^{th} row has been replaced by $-r$. So,

$$R_{k\_temp} - S_{k+1}T_k = (R_k - S_kT_k)J_{k+1} \qquad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.19)}$$

For stage 2:

a' denotes a vector consisting of the first m elements and a'' denotes a vector consisting of the last q elements of the entering column a. Let e_l denotes the l^{th} column of the $(m \times m)$ identity matrix. So,

$$Z = \begin{cases} J_{k+1}\boldsymbol{d'} & \text{If } a'' \neq f'' \\ J_{k+1}\boldsymbol{d'} + e_l & \text{If } a'' = f'' \end{cases}$$

Let F_{k+1} stands for the $(m \times m)$ identity matrix whose l^{th} column has been replaced by z. So,

$$R_{k+1} - S_{k+1}T_{k+1} = (R_{k\_temp} - S_{k+1}T_k)F_{k+1} \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.20)}$$

So, combining (eq. 2.19) and (eq. 2.20):

$$R_{k+1} - S_{k+1}T_{k+1} = (R_k - S_kT_k)J_{k+1}F_{k+1} \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.21)}$$

Case 3 - The leaving column *f* is one of the last *q* columns of B_k and no other column *g* of B_k satisfies $g'' = f''$:

In this case, the leaving column is simply replaced by the entering column. So, after $k+1$ iteration:

$$R_{k+1} - S_{k+1}T_{k+1} = R_k - S_kT_k \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.22)}$$

In summary, after k iterations, $R_k - S_kT_k$ may be represented as:

$$R_k - S_kT_k = (R_0 - S_0T_0)\, J_1F_1J_2F_{2\ldots\ldots\ldots}J_kF_k \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 2.23)}$$

possibly with some missing matrices J_i and F_i.

2.11 Eta Matrix

An *eta matrix* is a matrix differs from the identity matrix in only one column, referred to as its *eta column*. Figure 2.20 shows an eta matrix where 5th column is the eta column.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & c_1 & 0 & 0 \\ 0 & 1 & 0 & 0 & c_2 & 0 & 0 \\ 0 & 0 & 1 & 0 & c_3 & 0 & 0 \\ 0 & 0 & 0 & 1 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_6 & 1 & 0 \\ 0 & 0 & 0 & 0 & c_7 & 0 & 1 \end{bmatrix}$$

Figure 2.20: Eta Matrix

2.12 Eta Factorization

The most expensive operation of revised simplex method is inverting a basis which is done in step 1 and 3 in each iteration. So, the efficiency of the revised simplex method lies on the ease of implementing step 1 and 3. In each iteration, to calculate the value of the simplex multipliers ($\boldsymbol{y}$) and the value of d-vector ($\boldsymbol{d}$), two equations: $\boldsymbol{y} = \boldsymbol{c_B}B^{-1}$ and $\boldsymbol{d} = B^{-1}\boldsymbol{a}$ are solved, where the operation inverting a matrix involved. A technique eta factorization is used to solve these two equations without calculating the basis (matrix) from the scratch and without inverting the basis (matrix) [5].

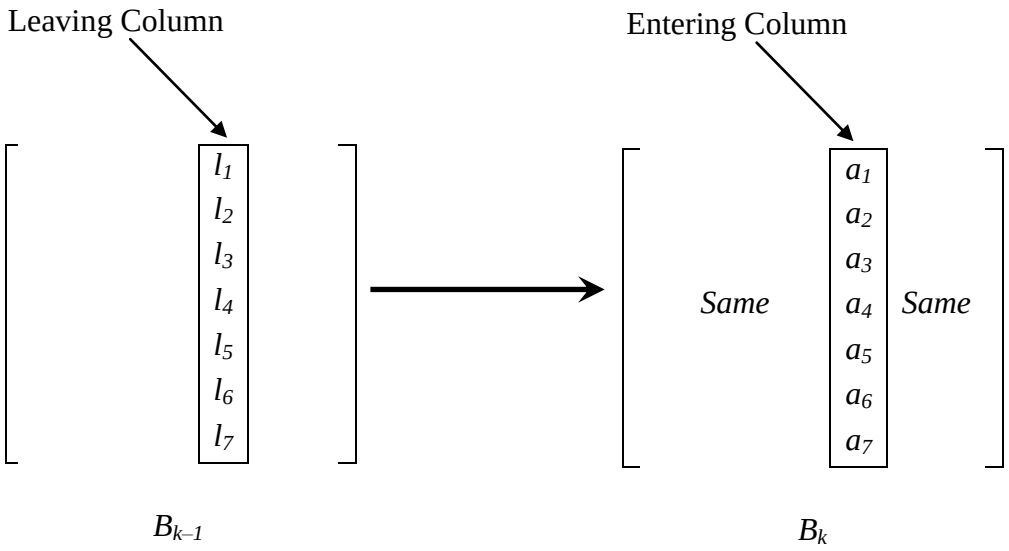


Figure 2.21: B_k and B_{k-1} differs only in one column which is the entering column a

Let B_{k-1} be the basis obtained after *k-1* iterations and B_k be the basis obtained after *k* iterations of the revised simplex method. So, each B_k differs from the preceding B_{k-1} in only one column (Figure 2.21). The rest of the basis will be same. Let that one column is the p^{th} column in which B_k differs from B_{k-1}. But the p^{th} column of B_k is the entering column $\boldsymbol{a}$ that was calculated in step 2 of k^{th} iteration. Again, this entering column $\boldsymbol{a}$ is used as the right-hand side in the equation $B_{k-1}.\boldsymbol{d} = \boldsymbol{a}$ in step 3 of the same iteration (k^{th} iteration). So, we can say that:

$$B_{k-1}E_k = B_k$$

where E_k stands for an identity matrix whose p^{th} column is replaced by $\boldsymbol{d}$ (Figure 2.22).

$$\begin{bmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \\ l_7 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & d_1 & 0 & 0 \\ 0 & 1 & 0 & 0 & d_2 & 0 & 0 \\ 0 & 0 & 1 & 0 & d_3 & 0 & 0 \\ 0 & 0 & 0 & 1 & d_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & d_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & d_6 & 1 & 0 \\ 0 & 0 & 0 & 0 & d_7 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

B_{k-1} E_k B_k

Figure 2.22: $B_{k-1}E_k = B_k$

When the initial basis consists of slack variables then the initial basis B_0 is an identity matrix. In that case:

$B_0 = I$

$B_1 = E_1$

$B_2 = E_1.E_2$

$B_3 = E_1.E_2.E_3$

.

.

.

$B_k = E_1.E_2.E_3 \ldots\ldots\ldots E_k$

This is called eta factorization of B_k. This eta factorization gives a convenient way of solving two equations: $\boldsymbol{y}B_k = \boldsymbol{c_B}$ and $B_k\boldsymbol{d} = \boldsymbol{a}$.

$\boldsymbol{y}B_k = \boldsymbol{c_B}$ can be viewed as:

$$((((\boldsymbol{y}E_1)E_2)E_3)\ldots\ldots)E_k = \boldsymbol{c_B}$$

and $B_k\boldsymbol{d} = \boldsymbol{a}$ can be viewed as:

$$E_1(E_2(E_3(\ldots\ldots(E_k\boldsymbol{d})))) = \boldsymbol{a}$$

So, the value of simplex multipliers ($\boldsymbol{y}$) and d-vector ($\boldsymbol{d}$) can be calculated by solving the equations iteratively. As E is an eta matrix, these two equations can be solved easily without inverting the matrices.

So far the initial basis B_0 was considered as an identity matrix. But if B_0 is not an identity matrix then B_k can be written as:

$$B_k = B_0.E_1.E_2.E_3\ldots\ldots\ldots E_k \qquad \ldots\ldots\ldots\ldots\ldots \text{(eq. 2.24)}$$

So, the two equations:

$\boldsymbol{y}B_k = \boldsymbol{c_B}$ can be solved as:

$$(((((\boldsymbol{y}B_0)E_1)E_2)E_3)\ldots\ldots)E_k = \boldsymbol{c_B} \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(eq. 2.25)}$$

and $B_k\boldsymbol{d} = \boldsymbol{a}$ can be solved as:

$$B_0(E_1(E_2(E_3(\ldots\ldots(E_k\boldsymbol{d})))) = \boldsymbol{a} \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(eq. 2.26)}$$

Now a triangular factorization of the initial basis B_0 can be done before the first iteration and then it can be used in conjunction with the growing sequence of $E_1, E_2, E_3, \ldots\ldots E_k$ to solve the equations:

$$L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1 B_0 = U = U_m U_{m-1} \ldots\ldots U_1 \qquad \ldots\ldots\ldots\ldots\ldots \text{(eq. 2.27)}$$

Here,

L is lower triangular matrix,

P is permutation matrix,

U is upper triangular matrix,

U_j is an eta matrix obtained by replacing j^{th} column of I by j^{th} column of U.

From (eq. 2.20),

$$L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1 B_0 = U_m U_{m-1} \ldots\ldots U_1$$

$$\Rightarrow \quad L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1 B_0 E_1 E_2 \ldots\ldots E_k = U_m U_{m-1} \ldots\ldots U_1 E_1 E_2 \ldots\ldots E_k$$

$$\Rightarrow \quad L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1 B_k = U_m U_{m-1} \ldots\ldots U_1 E_1 E_2 \ldots\ldots E_k$$

$$\Rightarrow \quad \boldsymbol{z} L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1 B_k = \boldsymbol{z} U_m U_{m-1} \ldots\ldots U_1 E_1 E_2 \ldots\ldots E_k$$

Let $\boldsymbol{y} = \boldsymbol{z} L_m P_m L_{m-1} P_{m-1} \ldots\ldots L_1 P_1$ ..... (eq. 2.28)

So, $\boldsymbol{y} B_k = \boldsymbol{z} U_m U_{m-1} \ldots\ldots U_1 E_1 E_2 \ldots\ldots E_k \ldots$ ..... (eq. 2.29)

But $\boldsymbol{y} B_k = \boldsymbol{c_B}$

So, we can say that

$$\boldsymbol{z} U_m U_{m-1} \ldots\ldots U_1 E_1 E_2 \ldots\ldots E_k = \boldsymbol{c_B} \qquad \ldots\ldots\ldots\ldots\ldots \text{(eq. 2.30)}$$

In this way the equation $\boldsymbol{y}B_k = \boldsymbol{c_B}$ can be solved by first solving the equation (eq. 2.30) and then calculating $\boldsymbol{y}$ from the equation (eq. 2.28).

Similarly, for the equation $B_k\boldsymbol{d} = \boldsymbol{a}$:

From (eq. 2.20),

$$U_mU_{m-1}\ldots\ldots U_1 = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1B_0$$

$$\Rightarrow U_mU_{m-1}\ldots\ldots U_1E_1E_2\ldots\ldots E_k = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1B_0E_1E_2\ldots\ldots E_k$$

$$\Rightarrow U_mU_{m-1}\ldots\ldots U_1E_1E_2\ldots\ldots E_k = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1B_k$$

$$\Rightarrow U_mU_{m-1}\ldots\ldots U_1E_1E_2\ldots\ldots E_k\boldsymbol{d} = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1B_k\boldsymbol{d}$$

$$\Rightarrow U_mU_{m-1}\ldots\ldots U_1E_1E_2\ldots\ldots E_k\boldsymbol{d} = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1\boldsymbol{a}$$

Let $\boldsymbol{h} = U_mU_{m-1}\ldots\ldots U_1E_1E_2\ldots\ldots E_k\boldsymbol{d}$ …… (eq. 2.31)

So, $\boldsymbol{h} = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1\boldsymbol{a}$ …… (eq. 2.32)

So, the equation $B_k\boldsymbol{d} = \boldsymbol{a}$ can be solved by first solving (eq. 2.32) and then solving (eq. 2.31).

2.13 Refactorizations

As the number of eta matrices grows with each iteration, solving the equations $\boldsymbol{y}B_k = \boldsymbol{c_B}$ and $B_k\boldsymbol{d} = \boldsymbol{a}$ become more laborious and may take longer time than to solve the two equations from the scratch. To avoid this, a fresh triangular factorization can be

computed from the basis treating B_k as B_0 and start with a new sequence of $E_1, E_2, E_3, \ldots\ldots E_k$. This technique is called *refactorizations* of the basis.

2.14 Branch and Price Technique

The successful solution of large-scale *mixed integer programming (MILP)* problems requires formulations whose linear programming '*LP*' relaxations give a good approximation to the convex hull of feasible solutions. In our problem, the solution must have binary values only (i.e., 0 or 1). Here we are including a summary of branch and price technique from [17], [3] and [20], considering the techniques applicable to our problem.

2.14.1 Branch and Bound

When the integer values are binary, this algorithm uses a divide-and-conquer strategy to partition the solution space into two *subproblems* and then recursively solve each subproblem. Let S be the set of solutions to a given discrete optimization problems (DOP), and let $c : S \rightarrow R$ be a cost function on members of S. To determine a least-cost member of S, we start with a given $s^{scs} \in S$, a "good" solution determined heuristically. In the bounding phase, we apply a LP relaxation giving a solution $s^{relaxed}$ in a superset of the feasible set S. This gives a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of S or has cost equal to that of s^{scs}, then the optimization is done, since either the new solution or s^{scs}, respectively, is the optimal value. Otherwise, we partition S into sets S_0 and S_1, using some appropriate branching rule. For instance, we may define the branching rule, so that S_0 (S_1) is the set of solutions

for $x_i = 0$ ($x_i = 1$) where x_i is one of the binary variables of the MILP formulation. We will call S_0, S_1 the *children* of S. We replace S with the children of S on the list of candidate subproblems (those that await processing). This operation is called *branching* and the candidate list is often called the *node queue*.

To continue the algorithm, we remove the "best" candidate subproblem from the node queue and process it, giving a value of $s^{scs}_{new} \in S$ and we apply a LP relaxation to the candidate subproblem, giving a solution $s^{relaxed}_{new}$ in a superset of the feasible set S.

There are four possibilities to consider:

1. If $s^{scs}_{new} < s^{scs}$, then we replace s^{scs} by s^{scs}_{new}; apply branching again and add the children of this subproblem to the node queue.
2. If the subproblem has no solutions, in which case we discard, or *fathom* it.
3. If $s^{relaxed}_{new} \geq s^{scs}$, then we may again fathom the subproblem.
4. Apply branching again and add the children of this subproblem to the node queue.

The process continues until the list of candidate subproblems is empty, at which point our current best solution stored in s^{scs} is the optimal one.

2.14.2 Branch and Price

The process of dynamically generating variables whose values should be non-zero is called *pricing* and LP-based branch and bound algorithms in which the variables are generated dynamically are known as branch and price algorithms [3]. Branch and price

techniques are useful when the number of variables is very large compared to the number of constraints. In branch and price, each LP relaxation is solved initially with only a small subset of the variables present. These variables correspond to the columns in the initial feasible solution. To find an optimal solution the *pricing problem* is solved, to try to identify a column to enter the basis. Such a column and the corresponding variables are generated as needed using, for example, the implicit column generation technique.

Chapter 3: An Efficient Scheme for Non-Bifurcated Traffic Grooming

In Chapter 2, we have described why minimizing congestion is a very common and popular objective for optimization in WDM network. An efficient technique to do this is, using arc-chain representation adapting with the implicit column generation [1]. In the arc-chain representation, the expensive operation of revised simplex method inverting a matrix can be done on the matrix of size $(m+q)\times(m+q)$ where m is the number of logical edges and q is the number of commodities. As the basis matrix in the arc-chain representation satisfies the GUB structure, inverting a matrix can be done on the matrix of size $(m\times m)$ instead of the matrix of size $(m+q)\times(m+q)$.

An arc-chain solver for bifurcated traffic grooming with GUB structure has been described in Chapter 2 to calculate the optimal solution using the revised simplex method. The branch and price technique is used in this approach to make the bifurcated traffic grooming to non-bifurcated traffic grooming [22]. Using GUB structure in this arc-chain solver, improved the performance of the revised simplex method significantly. But still matrix inversion has to be done, although it is done on a smaller matrix size.

In the section 2.12 we have shown that using eta factorization, the matrix calculation can be done without calculating the basis (matrix) from the scratch and without inverting the basis (matrix).

In this chapter we will show how to combine eta factorization with GUB structure in the arc-chain solver for the revised simplex method used in the branch and price technique [22]. This will allow calculating the simplex multipliers and the d-vector without inverting the basis and updating the basis without calculating it from the scratch, in each iteration of the revised simplex method.

3.1 Eta Factorization with GUB Structure in Branch and Price

The arc-chain solver using GUB structure improved the performance significantly since, as a result of using the GUB structure in the arc-chain solver, the expensive operation of *inverting a matrix* in each iteration can be done on a matrix of size $(m \times m)$ instead of a matrix of size $(m+q) \times (m+q)$. It is important to note that $q = N(N-1)$ where N is the number of end-nodes, since most end-nodes communicate with each other and m is $O(N)$. So, $q >> m$ and this gives substantial savings. In section 2.12, we have shown that, by using eta factorization these operations can be done without calculating the basis (matrix) from the scratch and without inverting the basis (matrix). In this section we will show how to combine eta factorization with GUB structure to improve the performance further in revised simplex method.

3.2 Revised Simplex Method without inverting (R–ST)

In section 2.10, we have shown that, using GUB structure, the simplex multipliers ($\boldsymbol{y}$) and d–vector ($\boldsymbol{d}$) can be calculated by first breaking $\boldsymbol{y}$ into $\boldsymbol{y'}$ and $\boldsymbol{y''}$ and $\boldsymbol{d}$ into $\boldsymbol{d'}$ and $\boldsymbol{d''}$. Then calculate the following equations [5]:

For $\boldsymbol{y}$**:**

$$\boldsymbol{y'} = (\boldsymbol{c_B'} - \boldsymbol{c_B''}T_k).(R_k - S_kT_k)^{-1} \quad \text{..... (eq. 2.12)}$$

$$\boldsymbol{y''} = \boldsymbol{c_B''} - \boldsymbol{y'}S_k \quad \text{..... (eq. 2.10)}$$

$$\boldsymbol{y} = (\boldsymbol{y'}, \boldsymbol{y''})$$

For $\boldsymbol{d}$:

$$\boldsymbol{d'} = (\boldsymbol{a'} - S_k\boldsymbol{a''}).(R_k - S_kT_k)^{-1} \quad \text{..... (eq. 2.17)}$$

$$\boldsymbol{d''} = \boldsymbol{a''} - T_k\boldsymbol{d'} \quad \text{..... (eq. 2.15)}$$

$$\boldsymbol{d} = (\boldsymbol{d'}, \boldsymbol{d''})$$

To avoid inverting the matrix $(R_k - S_kT_k)$ for equations (eq. 2.12) and (eq. 2.17) in each iteration of revised simplex method, eta factorization discussed in the section 2.12 can be combined with GUB structure as follows.

If we compare the equations (eq. 2.24) and (eq. 2.23) [5], we observe the following:

$$B_k = B_0.E_1.E_2.E_3\ldots\ldots E_k \quad \text{..... (eq. 2.24)}$$

$$R_k - S_kT_k = (R_0 - S_0T_0)\, J_1F_1J_2F_2\ldots\ldots J_kF_k \quad \text{..... (eq. 2.23)}$$

We can see that the initial basis B_0 for eta factorization is $R_0 - S_0T_0$, and the eta matrices J_k and F_k can be calculated while updating the matrices in each iteration as shown in Section 2.10.1.

Now, as shown in equation (eq. 2.27) in Section 2.12, a triangular factorization can be done on the initial basis $R_0 - S_0T_0$:

$$L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1(R_0 - S_0T_0) = U = U_mU_{m-1}\ldots\ldots U_1 \quad \ldots \text{(eq. 3.1)}$$

For equation (eq. 2.12), after k iterations, equations (eq. 2.28), (eq. 2.29) and (eq. 2.30) can be written as:

$$\boldsymbol{y'} = \boldsymbol{z}L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1 \quad \ldots \text{(eq. 3.2)}$$

$$\boldsymbol{y'}(R_k - S_kT_k) = \boldsymbol{z}U_mU_{m-1}\ldots\ldots U_1J_1F_1J_2F_2\ldots\ldots J_kF_k \quad \ldots \text{(eq. 3.3)}$$

$$\boldsymbol{z}U_mU_{m-1}\ldots\ldots U_1J_1F_1J_2F_2\ldots\ldots J_kF_k = (\boldsymbol{c_B'} - \boldsymbol{c_B''}T_k) \quad \ldots \text{(eq. 3.4)}$$

So, the value of $\boldsymbol{y'}$ can be calculated by first solving the equation (eq. 3.4) and then calculating $\boldsymbol{y'}$ from the equation (eq. 3.2). In this way, $\boldsymbol{y'}$ can be calculated without inverting the matrix as done in the equation (eq. 2.12). Once the value of $\boldsymbol{y'}$ is found, the value of $\boldsymbol{y''}$ can be calculated using the equation (eq. 2.10), and y can be calculated by $\boldsymbol{y} = (\boldsymbol{y'}, \boldsymbol{y''})$.

Similarly, for the equation (eq. 2.17), after k iteration, equations (eq. 2.31) and (eq. 2.32) can be written as:

$$\boldsymbol{h} = U_mU_{m-1}\ldots\ldots U_1J_1F_1J_2F_2\ldots\ldots J_kF_k\boldsymbol{d'} \quad \ldots \text{(eq. 3.5)}$$

$$\boldsymbol{h} = L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1(\boldsymbol{a'} - S_k\boldsymbol{a''}) \quad \ldots \text{(eq. 3.6)}$$

So, the value of $\boldsymbol{d'}$ can be calculated by first solving the equation (eq. 3.6) and then calculating $\boldsymbol{d'}$ from the equation (eq. 3.5). In this way, $\boldsymbol{d'}$ can be calculated without

inverting the matrix as done in the equation (eq. 2.17). Once the value of $\boldsymbol{d'}$ is found, the value of $\boldsymbol{d''}$ can be calculated using the equation (eq. 2.15), and $\boldsymbol{d}$ can be calculated by $\boldsymbol{d} = (\boldsymbol{d'}, \boldsymbol{d''})$.

Refactorizations can be done after every specific number of intervals by treating $R_k - S_kT_k$ as $(R_0 - S_0T_0)$, by performing the triangular factorization again on the new initial basis $R_k - S_kT_k$ and start the operations all over again. In our approach, we have done the refactorizations when the iteration number reaches the half of the number of edges.

3.3 Calculating L_i, P_i, U_i

Before starting the revised simplex method, the *lower triangular matrices* $(L_1, L_2, \ldots, L_m)$, *permutation matrices* $(P_1, P_2, \ldots, P_m)$ and *upper triangular matrices* $(U_1, U_2, \ldots, U_m)$ are calculated from the initial basis of the initial feasible solution to use those matrices in each iteration of the revised simplex method. Here, m is the number of edges in the network and $R_0 - S_0T_0$ is the initial basis of the initial feasible solution. So, L_i, P_i and U_i will be calculated from the initial basis $R_0 - S_0T_0$.

Let us consider the example described in the section 2.9.2. If we re-arrange the initial feasible solution in (eq. 2.7) as GUB structure, the equation will be as follows:

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ -1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1.5 \\ 1.2 \\ 0.8 \\ 1.5 \\ 1.5 \\ 1.5 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}$$

… (eq. 3.7)

Here, R_0, S_0 and T_0 are as follows:

$$\left[\begin{array}{cccccc|ccc} -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ -1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] . \begin{bmatrix} 1.5 \\ 1.2 \\ 0.8 \\ 1.5 \\ 1.5 \\ 1.5 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.3 \\ 0.5 \\ 0.7 \end{bmatrix}$$

$$R_0 = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$T_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

So, the initial basis R_0–S_0T_0 is:

$$R_0–S_0T_0 = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, as shown in the equation (eq. 3.1), first we can compute a triangular factorization of $R_0 - S_0T_0$:

$$L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1(R_0 - S_0T_0) = U \qquad \ldots\ldots\ldots\ldots\ldots \text{(eq. 3.8)}$$

and then from equation (eq. 3.8), we can get:

$$U = U_mU_{m-1}\ldots\ldots U_1 \qquad \ldots\ldots\ldots\ldots\ldots \text{(eq. 3.9)}$$

$$L_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad R_0-S_0T_0 = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_1P_1(R_0-S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_2P_2L_1P_1(R_0-S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}$$

$$L_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \quad P_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_3P_3L_2P_2L_1P_1(R_0-S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_4P_4L_3P_3L_2P_2L_1P_1(R_0 - S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_5P_5L_4P_4L_3P_3L_2P_2L_1P_1(R_0 - S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_6P_6L_5P_5L_4P_4L_3P_3L_2P_2L_1P_1(R_0 - S_0T_0) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

So,

$$U = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, we can get U_i by replacing the i^{th} column of an identity matrix by the i^{th} column of U:

$$U_1 = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad U_2 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad U_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$U_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad U_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad U_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4 Representation of L_i, P_i, U_i

After analyzing the matrices L_i and P_i where i is the number of logical edges, it is found that:

L_i = i^{th} lower triangular matrix.

P_i = i^{th} permutation matrix.

Instead of storing the matrices L_1, P_1, L_2, P_2, ..., L_m, P_m, we can store only the information:

For L_i : a vector of float of size m for the i^{th} column of L_i.

For P_i : an integer for the position of the row that swapped with i^{th} row in P_i.

So, L_2 and P_2 in the example in section 3.3 can be stored as:

L_P_Matrix_Store[2]{float L_Vector[]; int Position_of_P}

= *{[0 1 –1 –1 –1 –1]; 2}*

instead of:

$$L_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

By doing so, vector operations can be done, instead of matrix operations which will improve the performance substantially and at the same time memory locations can be saved.

U_i is calculated from U which is the result of $L_mP_mL_{m-1}P_{m-1}\ldots\ldots L_1P_1(R_0 - S_0T_0)$. So,

U_i = Upper triangular matrix containing i^{th} column of U.

Instead of storing the matrices $U_1, U_2, \ldots, U_m$, we can store only the information:

For U_i : a vector of float of size m for the i^{th} column of U.

So, U_3 in the example in section 3.3 can be stored as:

float U[3] = *[0 1 –1 0 0 0]*

instead of:

$$U_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

3.5 Calculating J_k, F_k

Matrices J_k and F_k are calculated after getting *entering column* and *leaving column number* and before updating the basis in each iteration of the revised simplex method. Here, *k* is the *iteration number* of the revised simplex method.

As described in Section 2.10.1, J_k and F_k can be calculated after getting the ***d'*** and the leaving column number in each iteration. So, J_k and F_k will look like the matrices shown in the Figure 3.1.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -r_{k1} & -r_{k2} & -r_{k3} & -r_{k4} & -r_{k5} & -r_{k6} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & d'_{k1} & 0 & 0 \\ 0 & 1 & 0 & d'_{k2} & 0 & 0 \\ 0 & 0 & 1 & d'_{k3} & 0 & 0 \\ 0 & 0 & 0 & d'_{k4} & 0 & 0 \\ 0 & 0 & 0 & d'_{k5} & 1 & 0 \\ 0 & 0 & 0 & d'_{k6} & 0 & 1 \end{bmatrix}$$

J_k $\qquad\qquad$ F_k

Figure 3.1: Matrices J_k and F_k

3.6 Representation of J_k, F_k

After analyzing the matrices J_k and F_k where *k* is the iteration number of the revised simplex method, it is found that:

J_k = Matrix differing from identity matrix in only one row, obtained after *k* iterations.

F_k = Eta matrix obtained after *k* iterations.

In each iteration, instead of storing the matrices $J_1, F_1, J_2, F_2, \ldots, J_k, F_k$, we can store only the information:

For J_k : an integer for the position of the row in J_k,

a vector of float of size m for the row in J_k.

For F_k : an integer for the position of the row in F_k,

a vector of float of size m for the row in F_k.

So, if the $k = 4$ in the Figure 3.1, then J_4 and F_4 can be stored as:

J_F_Matrix_Store[4]{int Position_of_J; float J_Vector[]

int Position_of_F; float F_Vector[]}

$=$ $\{5; [-r_{41} -r_{42} -r_{43} -r_{44} -r_{45} -r_{46}];$

$4; [d'_{41}\ d'_{42}\ d'_{43}\ d'_{44}\ d'_{45}\ d'_{46}\}$

3.7 Solving Equations without inverting Matrices

For any equation in the form of $\boldsymbol{z}.F = \boldsymbol{v}$

where $\boldsymbol{z}$ = unknown vector to be determined,

F = known eta matrix,

$\boldsymbol{v}$ = known vector.

In this equation, $\boldsymbol{z}$ can be calculated without inverting the matrix F.

Let's consider the following vectors and matrix representing $\boldsymbol{z}$, F and v where the vector $\boldsymbol{z}$ has to be determined and the matrix F and the vector $\boldsymbol{v}$ are known:

$$\begin{bmatrix} z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 & F_1 & 0 \\ 0 & 1 & 0 & 0 & F_2 & 0 \\ 0 & 0 & 1 & 0 & F_3 & 0 \\ 0 & 0 & 0 & 1 & F_4 & 0 \\ 0 & 0 & 0 & 0 & F_5 & 0 \\ 0 & 0 & 0 & 0 & F_6 & 1 \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{bmatrix}$$

After doing vector matrix multiplication, we get:

$z_1 = v_1$

$z_2 = v_2$

$z_3 = v_3$

$z_4 = v_4$

$z_6 = v_6$

$z_1F_1 + z_2F_2 + z_3F_3 + z_4F_4 + z_5F_5 + z_6F_6 = v_5$

$\Rightarrow \quad z_5F_5 = v_5 - (z_1F_1 + z_2F_2 + z_3F_3 + z_4F_4 + z_6F_6)$

$\Rightarrow \quad z_5 = (v_5 - (z_1F_1 + z_2F_2 + z_3F_3 + z_4F_4 + z_6F_6)) / F_5$

For z_5, after replacing the values of z_1, z_2, z_3, z_4 and z_6:

$z_5 = (v_5 - (v_1F_1 + v_2F_2 + v_3F_3 + v_4F_4 + v_6F_6)) / F_5$

Similarly, for any equation in the form of $\boldsymbol{z}.J = \boldsymbol{v}$

where $\boldsymbol{z}$ = unknown vector to be determined,

J = known matrix differing from identity matrix in only one row,

$\boldsymbol{v}$ = known vector.

In this equation, $\boldsymbol{z}$ can be calculated without inverting the matrix J.

Let's consider the following vectors and matrix representing $\boldsymbol{z}$, J and v where the vector $\boldsymbol{z}$ has to be determined and the matrix J and the vector $\boldsymbol{v}$ are known:

$$\begin{bmatrix} z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ J_1 & J_2 & J_3 & J_4 & J_5 & J_6 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{bmatrix}$$

After doing vector matrix multiplication, we get:

$z_1 + z_4J_1 = v_1 \Rightarrow z_1 = v_1 - v_4J_1 / J_4$

$z_2 + z_4J_2 = v_2 \Rightarrow z_2 = v_2 - v_4J_2 / J_4$

$z_3 + z_4J_3 = v_3 \Rightarrow z_3 = v_3 - v_4J_3 / J_4$

$z_4J_4 = v_4 \Rightarrow z_4 = v_4 / J_4$

$z_5 + z_4J_5 = v_5 \Rightarrow z_5 = v_5 - v_4J_5 / J_4$

$z_6 + z_4J_6 = v_6 \Rightarrow z_6 = v_6 - v_4J_6 / J_4$

So, $z = [z_1\ z_2\ z_3\ z_4\ z_5\ z_6]$ can be determined without inverting the matrix F or J after calculating the values of z_1, z_2, z_3, z_4, z_5 and z_6.

3.8 Algorithm

3.8.1 Algorithm for getting y′

Step 1: $i = 1$

$z = (c_B' - c_B'' T_k)$

Step 2: *while* $(i >= 1)$

{

$v = z$

Replace z by the solution of $zF_i = v$

$v = z$

Replace z by the solution of $zJ_i = v$

$i = i - 1$

}

Step 3: $j = 1$

Step 4: *while* $(j =< m)$

{

$v = z$

Replace z by the solution of $zU_j = v$

$j = j + 1$

}

Step 5: $j = m$

$y' = z$

Step 6: *while* $(j >= 1)$

{

$y' = y' L_j P_j$

$j = j - 1$

}

3.8.2 Algorithm for getting d′

Step 1: $j = 1$

$d' = (a' - S_k a'')$

Step 2: *while* $(j =< m)$

{

$d' = L_j P_j d'$

$j = j + 1$

}

Step 3: $j = m$

Step 4: *while* $(j >= 1)$

{

$v = d'$

Replace d' *by the solution of* $U_j d' = v$

$j = j - 1$

}

Step 5: $i = 1$

Step 6: *while* $(i =< k)$

{

$v = d'$

Replace d' *by the solution of* $J_i d' = v$

$v = d'$

Replace d' *by the solution of* $F_i d' = v$

$i = i + 1$

}

3.9 Use of Eta Factorization in the Algorithm

In the algorithm for getting $\boldsymbol{y'}$ (section 3.8.1), *step 2* and *step 4* are executed for the equation (eq. 3.4). At the beginning of the revised simplex method, matrices *F* and *J* are not available as they are calculated during updating the basis after getting the entering column and the leaving column in each iteration. So, to calculate the simplex multipliers (***y***) for the first time, *step 2* has to be skipped. *Step 4* will be executed to solve the equation:

$$\boldsymbol{z}U_mU_{m-1}\ldots\ldots U_1 = (\boldsymbol{c_B'} - \boldsymbol{c_B''}T_k) \qquad \ldots\ldots\ldots\ldots\ldots\ldots \text{ (eq. 3.10)}$$

In the example described in the section 2.9, the cost of the basis $\boldsymbol{c_B} = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$ as we have to minimize the Λ_{max}. So, $(\boldsymbol{c_B'} - \boldsymbol{c_B''}T_k) = [1\ 0\ 0\ 0\ 0\ 0]$. In this example, *m* is *6*. So, the loop in the *step 4* will be executed 6 times.

In the first loop of *step 4*:

$\boldsymbol{v} = \boldsymbol{z} = [1\ 0\ 0\ 0\ 0\ 0]$

Solve the solution $\boldsymbol{z}U_1 = \boldsymbol{v}$ *(assuming* $\boldsymbol{z} = \boldsymbol{z}U_6U_5U_4U_3U_2$*) and get the value of* $\boldsymbol{z}$

In the second loop of *step 4*:

$\boldsymbol{v} = \boldsymbol{z}$ *(new value of* $\boldsymbol{z}$ *found in the previous loop)*

Solve the solution $\boldsymbol{z}U_2 = \boldsymbol{v}$ *(assuming* $\boldsymbol{z} = \boldsymbol{z}U_6U_5U_4U_3$*) and get the value of* $\boldsymbol{z}$

and so on.

All the loops in the algorithm work like this using eta factorization without inverting any matrix.

Chapter 4: Experimental Results

In Chapter 3, we extended the arc-chain solver developed by Mr. Quazi Rahman, by incorporating eta factorization in the solver. In this chapter, we will analyze and compare the results of the arc-chain solver before and after incorporating the eta factorization and see how much improvement in the speed we have accomplished. We have done our experiments on a number of networks with sizes varying from small to large and with requests for data communication also having various sizes. The experiments were done on a Sun Fire X2200 M2 Server [21].

We used two C programs *Generate_edge_array.c* and *Generate_user_requests.c* to generate logical edges and user requests respectively, based on the given number of end-nodes. These two programs generated the logical edges and the user requests using a random number generator. We have omitted the details since these programs are quite straight-forward.

We wrote a function *arc_chain_solver_using_ETA* in C and included it in the arc-chain solver implemented by Mr. Quazi Rahman. For comparison purposes, we defined a flag. When the flag is 1, the C program does use eta factorization using our function *arc_chain_solver_using_ETA*. When the flag is 0, we don't use eta factorization, so that the program gives the same results obtained using the function *arc_chain_solver* written by Mr. Rahman.

4.1 Comparison of Simplex Multipliers

Use of eta factorization allows us to compute the simplex multipliers in a way different from traditional methods. Due to round-off errors, the values computed in the traditional way may be slightly different in the two techniques. In this section we will explore these differences.

In the revised simplex method, the condition for termination is a failure to find an entering column. In each iteration, the entering column is calculated using the values of the simplex multipliers (**y** is a vector of all the simplex multipliers). So, if the values of the simplex multipliers, using eta factorization are slightly different from the values of the same simplex multipliers, without using eta factorization, the process of finding the entering column may lead to different iteration numbers. Our experiments indicate that the small differences owing to the round-off errors do have significant repercussions.

We have done an experiment with a small network (6 end-nodes) and a small number of user requests (around 100 user requests) and compared the values of the simplex multipliers obtained using eta factorization and without eta factorization after each iteration. We have found that the some values of the simplex multipliers differ by a very small value (approximately 0.000000006) after a certain number of iterations (approximately 22 iterations). This affects the results when finding the entering column and hence the number of iterations and the times of executions change. But it does not make any difference to the value of the congestion which is the objective function. We

also have found that changes in the interval between refactorizations have an effect on the number of iterations and also the time of execution.

When using eta factorization, in each iteration of the revised simplex method, the values of the simplex multipliers are calculated by matrix multiplication with a growing number of J_k and F_k matrices where k is the iteration number. This is the reason why the values of the simplex multipliers differ after a certain number of iterations. After a certain number of iterations, when the number of J_k and F_k matrices become large, then the calculations of the simplex multipliers are done with a large number of J_k and F_k matrices and that makes the values of the simplex multipliers somewhat different compared to the values of the simplex multipliers calculated without eta factorization, where we invert the matrix. This also makes a difference in the time of execution with the eta factorization. These differences vary with the interval of the refactorizations.

4.2 Experiments with the different interval of Refactorizations

We have carried out an experiment to find out the intervals between refactorizations that gives the best results for execution time when using eta factorization. In the arc-chain solver developed by Mr. Rahman, during the process of branch and price technique, the arc-chain solver is executed repeatedly to get the optimal solution for the bifurcated traffic grooming. We tested this experiment running the arc-chain solver only once where we did not use the branch and price technique.

We tested our experiments on ten networks having 14 end-nodes. For five networks, we used five different traffic loads of around 400 user requests and for other five networks, we used five different traffic loads of around 1000 user requests. For each size of traffic load, we generated 25 sets of data (combination of five networks and five different traffic loads). For each set, we executed the program 4 times, each having refactorizations at intervals of $E/4$, $E/2$, E, $2E$ iterations, where E is the number of edges. We have reported the average of each set of 25 with 400 (1000) user requests in the Table 4.1 (Table 4.2). Graphs from the Table 4.1 (Table 4.2) are shown in the Figure 4.1 (Figure 4.2). After analyzing the Figure 4.1 and Figure 4.2, we can see that, on an average, refactorizations at intervals of $E/2$ gives the best results for the time of execution.

No. of Nodes	No. of Edges(with Traffic Loads around 400 user requests)	Time to execute (in seconds)				
		Without Eta	With Eta (Refactorizations after % of No. of Edges)			
			25%	50%	100%	200%
14	36	6.42	5.08	5.04	4.97	5.14
	35	3.59	2.48	2.46	2.58	2.59
	32	1.75	1.40	1.40	1.42	1.44
	33	0.94	0.74	0.73	0.74	0.74
	36	2.63	2.01	2.05	2.06	2.03

Table 4.1: Comparison of Execution Times with different intervals between Refactorizations with around 400 user requests

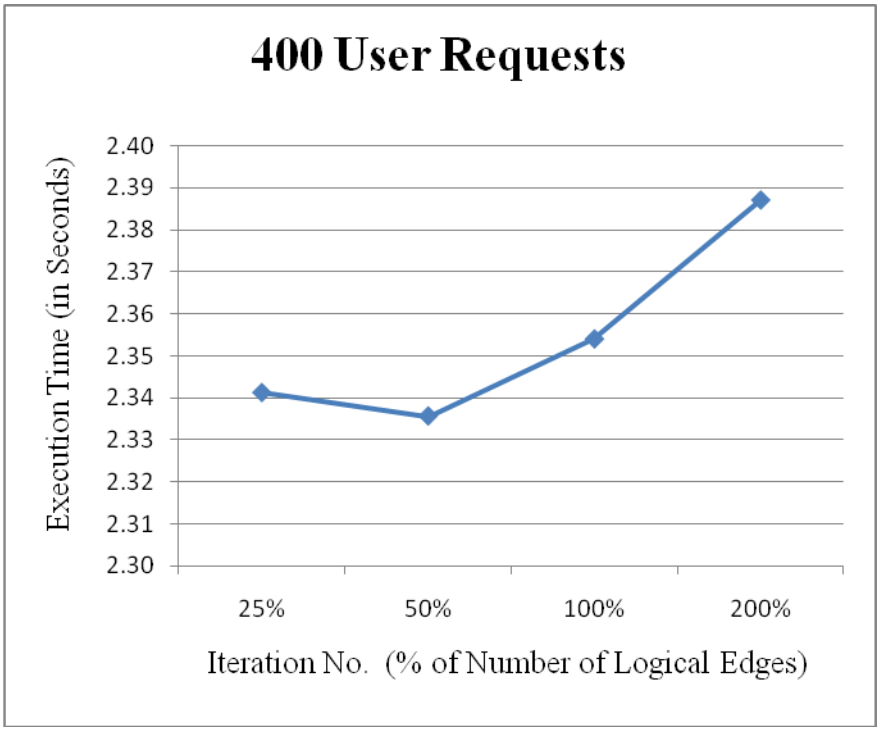


Figure 4.1: Graph obtained from Table 4.1

No. of Nodes	No. of Edges(with Traffic Loads around 1000 user requests)	Time to execute (in seconds)				
		Without Eta	With Eta (Refactorizations after % of No. of Edges)			
			25%	50%	100%	200%
14	36	19.30	16.63	16.69	16.70	16.92
	35	15.97	13.82	13.53	13.94	14.03
	32	6.14	5.50	5.59	5.58	5.54
	33	3.72	3.28	3.27	3.26	3.32
	36	10.74	9.32	9.31	9.50	9.37

Table 4.2: Comparison of Execution Times with different intervals between Refactorizations with around 1000 user requests

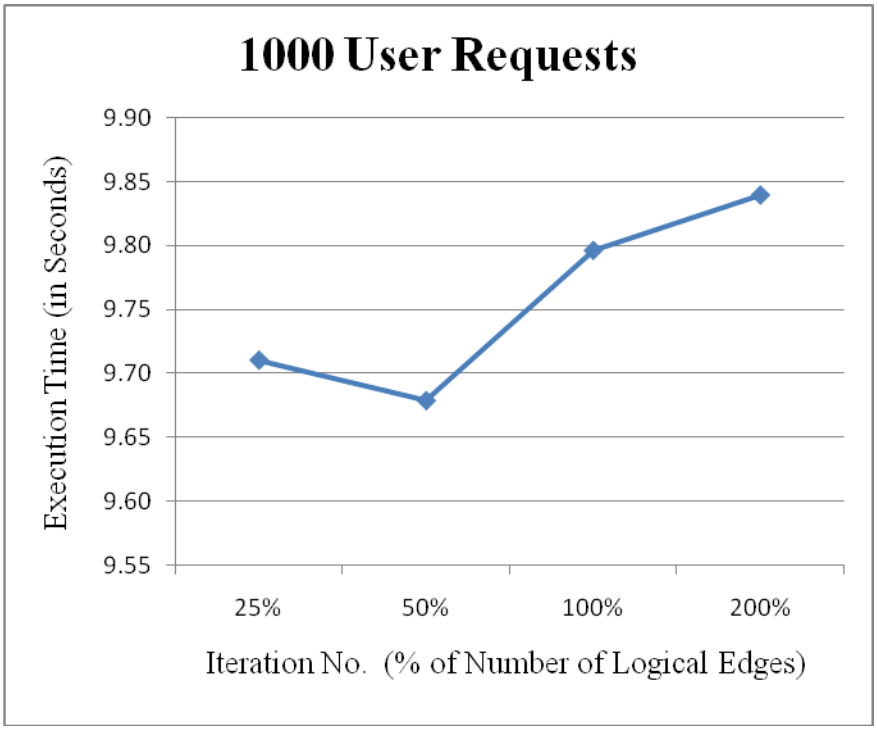


Figure 4.2: Graph obtained from Table 4.2

4.3 Comparison Experiments

We tested our experiments on three networks having 10, 14 and 20 end-nodes. For each network, we used traffic loads of around 100, 200, 400 and 1000 user requests. For each size of traffic, we generated 10 sets of data. We executed the program for each traffic load and for each network size. We measured the execution time twice. In the first (second) run, we set the flag mentioned above to 0 (1). Then we divided the time,in seconds, it took to execute the program without eta factorization by the time (in seconds) it took to execute the program with eta factorization to see how much faster the algorithm runs when we use eta factorization. We have reported the average of these 10 ratios in the table below. In the section 4.2, we found that refactorization with intervals of $E/2$ gives

the best results for the time of execution. So, here we used the refactorizations after $E/2$ iterations.

Table 4.3 shows how much faster, on an average, the program runs with the eta factorization compared to the program running without the eta factorization. The table gives the ratio of the execution time using eta factorization to the corresponding time without using eta factorization so that

- If the ratio = 1, then the execution time is same for both running without eta factorization and running with eta factorization.
- If the ratio < 1, then running with eta factorization is slower than running without eta factorization.
- If the ratio > 1, then running with eta factorization is faster than running without eta factorization.

For example, the table shows that, for the network with 10 end-nodes with 100 user requests, running with eta factorization is 31% faster, on an average, than running without eta factorization.

Number of End–nodes	Using Eta Factorization (average times faster)			
	Number of User Requests			
	100	200	400	1000
10	1.31	2.55	2.28	1.35
14	3.50	3.48	2.67	1.22
20	1.96	2.03	2.20	1.50

Table 4.3: Experimental results of two approaches

Chapter 5: Conclusions

In this thesis we have combined eta factorization with GUB structure in arc-chain solver and studied how much improvement in performance can be accomplished for optimization problem. An arc-chain solver for bifurcated traffic grooming with GUB structure was done by Mr. Quazi Rahman. A branch and price technique was used in that arc-chain solver to convert the bifurcated traffic grooming into optimal non-bifurcated traffic grooming. The performance of the revised simplex method improved significantly after using GUB structure in this approach.

We extended the approach done by Mr. Quazi by attaching eta factorization with it. We implemented an efficient scheme to improve the performance of the revised simplex method of the arc-chain solver by combining the eta factorization with the GUB structure.

We have done various experiments with our approach and the approach done by Mr. Quazi Rahman. First, we have done an experiment to compare the values of the simplex multipliers calculated with eta factorization and without eta factorization. From this experiment we have found that after certain number of iterations, some values of the simplex multipliers differ by a very small value due to round-off errors which affects the results when finding the entering column and hence the number of iterations. But it does not affect the objective function - the congestion.

We have done another experiment to see which interval between refactorizations gives the best improvements in terms of the time of execution when using eta factorization. From this experiment we have found that, performing refactorizations at the interval of when iteration number of the revised simplex method reaches the half of the number of the logical edges in the network, gives the best results for the time of execution. We used this result as the optimum interval between refactorizations in the other experiment we have done.

We have done our final experiment with our approach with the eta factorization and the approach without eta factorization. We analyzed the experimental results of both approaches and compared these two results to see how much improvement in performance we have accomplished using our approach. We have done this experiment on different size of networks with different traffic loads. We have found that in each case our approach with eta factorization is faster than the approach without eta factorization.

Our approach of using eta factorization can be pluggable to any network flow problem that satisfies GUB structure.

In our approach, using an appropriate compression algorithm for the vectors, it may be possible to reduce the space requirement and also avoid the explicit multiplications whenever possible. This may improve the performance further. It can be a good research for future.

Bibliography

1. Subir Bandyopadhyay, “Dissemination of Information in Optical Networks”, Springer-Verlag Berlin Heidelberg, 2008.
2. M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, “Linear Programming and Network Flows”, Wiley, 1990.
3. C. Barnhart, E. D. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh and P. H. Vance, “Branch and Price: Column Generation for Solving Huge Integer Programs”, Operations Research, Vol. 46, No. 3, pp. 316-329, May–June 1998.
4. V. W. S Chan, K. L. Hall, E. Modiano and K. A. Rauschenbach, “Architecture and Technologies for High-Speed Optical Data Networks”, IEEE Journals of Lightwave Technology, Vol. 16, No. 12, December 1998.
5. V. Chvatal, “Linear Programming”, W. H. Freeman and Company, New York, 1983.
6. R. Dutta and G. N. Rouskas, “On Optimal Traffic Grooming in WDM Rings”, IEEE Journal on Selected Areas in Communications, 20(1):110–121, January 2002.
7. R. Dutta and G. N. Rouskas, “Traffic Grooming in WDM Networks: Past and Future”, IEEE Network, 16(6):46–56, November 2002.
8. P. Green, “Progress in Optical Networking”, IEEE Communications Magazine, 39(1):54–61, January 2001.

9. J. Q. Hu and E. Modiano, “Optical WDM Networks: Principles and Practice”, Volume II, Chapter: Traffic Grooming in WDM Networks, Kluwer Academic Publishers, 2004.
10. R. Ul-Mustafa and A. E. Kamal, “Design and Provisioning of WDM Networks with Multicast Traffic Grooming”, IEEE Journal on Selected Areas in Communications, 24(4):37–53, April 2006.
11. B. Mukherjee, “WDM Optical Communication Networks: Progress and Challenges”, IEEE Journals on areas in communications, Vol. 18, Issue 10, pp. 1810-1824, October 2000.
12. B. Mukherjee, “Optical Communication Networks”, McGraw-Hill, New York, 1997.
13. S. Ramamurthy and B. Mukherjee, “Survivable WDM Mesh Networks”, Part I-Protection, In IEEE International Conference on Computer Communications (INFOCOM), volume 2, pages 744–751, March 1999.
14. R. Ramaswami and K. N. Sivaranjan, “Optical Networks – A Practical Perspective”, Morgan Kaufman Publishers, Optical Network Magazine, Vol. 3, May 2002.
15. R. Ramaswami and K. N. Sivarajan, “Design of Logical Topologies for Wavelength-routed Optical Networks”, IEEE JSAC, Vol. 14, pp. 840-851, June 1996.
16. J. Tomlin, “Minimum-cost Multicommodity Network Flows”, Operations Research, 14(1):45–51, January–February 1966.

17. L. A. Wolsey, “Integer Programming”, John Wiley, New York, 1998.
18. K. Zhu and B. Mukherjee, “Traffic Grooming in an Optical WDM Mesh Network”, IEEE Journal on Selected Areas in Communications, 20(1):122–133, January 2002.
19. K. Zhu, H. Zang, and B. Mukherjee, “A Comprehensive Study on Next-generation Optical Grooming Switches”, IEEE Journal on Selected Areas in Communications, 21(7):1173–1186, September 2003.
20. “Linear programming”, http://en.wikipedia.org/wiki/Linear_programming.
21. http://www.sun.com/servers/x64/x2200/.
22. Quazi Rahman, Subir Bandyopadhyay and Yash Aneja, “A Branch, Price and Cut approach to optimal non-bifurcated traffic grooming in WDM networks”, Technical Report 10-009, School of Computer Science, University of Windsor, 2010.

Vita Auctoris

Name:	Syed Jabbar
Place of Birth:	Chittagong, Bangladesh
Year of Birth:	1975
Education:	Independent University, Bangladesh (IUB), Dhaka, Bangladesh
	1997 B.Sc.
	University of Windsor, Windsor, Ontario, Canada
	2010 M.Sc.