1-1-2002

Alternate path routing algorithm for traffic engineering in the Internet

Shyam Subramanian
University of Nevada, Las Vegas

Follow this and additional works at: <https://digitalscholarship.unlv.edu/rtds>

Repository Citation

Subramanian, Shyam, "Alternate path routing algorithm for traffic engineering in the Internet" (2002).
UNLV Retrospective Theses & Dissertations. 1444.
<http://dx.doi.org/10.25669/m0s7-etk9>

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

14

1 3 4 9 7

U M I
MICROFILMED 2003

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

ALTERNATE PATH ROUTING ALGORITHM FOR TRAFFIC ENGINEERING IN
THE INTERNET

by

Shyam Subramanian

Bachelor of Engineering
Bharathiar University
2000

A thesis submitted in partial fulfillment of the
requirements for the

**Master of Science Degree in Electrical Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering**

**Graduate College
University of Nevada Las Vegas
December 2002**

UMI Number: 1413497

UMI[®]

UMI Microform 1413497

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

November 15, 2002

The Thesis prepared by

Shyam Subramanian

Entitled

Alternate Path Routing Algorithm For Traffic Engineering in the
Internet

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Alternate Path Routing Algorithm For Traffic Engineering In The Internet

by

Shyam Subramanian

**Dr. Muthukumar Venkatesan, Examination Committee Chair
Professor of Electrical and Computer Engineering
University of Nevada, Las Vegas**

Data flowing through the internet continues to grow every year. Increase in traffic demands increase in bandwidth. But bandwidth¹ does not grow as fast as the traffic. This leads to congestion in the network and performance degradation. One way to avoid this problem is to use efficient routing algorithm that efficiently maps the flow of data onto the network.

The most often used routing algorithm in the internet is the shortest path algorithm (Dijkstra's algorithm). This algorithm is simple and easy to implement. But this algorithm leads to over-utilization of part of the network, while the other part remains under-utilized. In this study, a new algorithm to route traffic efficiently is proposed. This

¹ Bandwidth of a link is defined as the maximum traffic that the link can accommodate at any given time. In other words it is the capacity of the link.

algorithm is as simple as the Dijkstra's algorithm, but routes traffic more efficiently and reduces the discrepancy in the network utilization.

The proposed algorithm identifies the critical links in the network that are responsible for congestion and finds additional routes in a network which do not contain critical links. The traffic in the network is routed via paths that have no or least number of critical links. This avoids congestion in the critical links and increases the performance of the network. The performance of the algorithm is compared with the shortest path (Dijkstra's) algorithm. The simulation is done using myNetSim which was developed for this thesis.

ACKNOWLEDGEMENTS

I am indebted to my advisor and thesis chair, Dr. Muthukumar Venkatesan for his valuable guidance and his valuable time. His profound knowledge and patient instructions with a warm smile, have given me a great help in completing my thesis. I wish to thank the members of my Thesis committee Dr. Henry Selvaraj, Dr. Emma Regentova and Dr. John Wang, who have given me valuable suggestions on the earlier versions of this thesis. A great help has been extended by OPNET technologies for letting me use the OPNET modeler. I would like to extend thanks to my colleagues at University of Nevada, Las Vegas for their unconditional support in helping me.

Last but certainly not least, I am forever indebted to the love and care of my family, who made me capable to accomplish their expectations.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENTS.....	v
LIST OF FIGURES	ix
LIST OF TABLES.....	xii
CHAPTER1 INTRODUCTION: INTERNET TRAFFIC	1
1.1 Multi-Protocol Label Switching Networks.....	2
1.1.1 Traffic Engineering with MPLS	4
1.2 Enhanced Interior Gateway Protocol	5
1.2.1 OSPF-TE.....	6
1.3 Constraint Based Routing and Quality of Service (QoS)	6
1.4 Organization of This Report	7
CHAPTER 2 TRAFFIC ENGINEERING.....	8
2.1 Traffic Engineering Algorithms.....	8
2.1.1 Shortest Path Algorithm – Dijkstra Algorithm.....	9
2.1.1.1 Widest-Shortest Path.....	10
2.1.1.2 Shortest-Widest Path.....	11
2.1.1.3 Dynamic Alternative Routing.....	12
2.1.2 Minimum Interference Routing with Application to MPLS – TE.....	13
2.1.3 Profile Based Routing.....	14
2.2 Routing Requirements	15
2.3 Evaluation Of Existing Routing Algorithm.....	17
CHAPTER 3 ALTERNATE PATH ROUTING ALGORITHM	19
3.1 Definitions	19
3.2 The Proposed Algorithm.....	21
3.2.1 Pre-Processing Phase	22
3.2.1.1 Algorithm of Pre-Processing Phase	22
3.2.1.2 Pseudo Code for Pre-Processing Phase	23
3.2.2 Online Routing Phase	23
3.2.2.1 Control Plane	24
3.2.2.1.1 Algorithm of Control Plane	25
3.2.2.1.2 Pseudo Code for the Control Plane.....	26

3.2.2.2 Forwarding Plane	28
3.2.2.2.1 Algorithm of the Forwarding Plane	29
3.2.2.2.2 Pseudo Code for the Forwarding Plane	29
3.3 Example	30
3.3.1. Example Network	30
3.3.2 Pre-Processing Phase	32
3.3.3 Online Routing Phase	34
3.3.3.1 The Control Plane	34
3.3.3.2 Forwarding Plane	39
 CHAPTER4 SIMULATION RESULTS	 42
4.1 MyNetSim – A Network Simulator	42
4.1.1 Network Input Module.....	42
4.1.1.1 Random Network Generation	43
4.1.1.2 Loading the network into the linked list	43
4.1.1.2.1 Example Network Format.....	44
4.1.2 Routing Module	46
4.1.3 Traffic Generation & Flow Module.....	48
4.1.3.1 Traffic Generation Phase	48
4.1.3.1.1 Example Event File.....	50
4.1.3.2 Flow Module Phase	51
4.2 Simulation Results	52
4.2.1 Network1	53
4.2.1.1 Event0	55
4.2.1.1.1 Event0 – using Dijkstra’s Algorithm	56
4.2.1.1.2 Event0 – using Alternate Path Routing Algorithm	59
4.2.1.2 Event10	61
4.2.1.2.1 Event10 – using Dijkstra’s Algorithm	62
4.2.1.2.2 Event10 – using Alternate Path Routing Algorithm	66
4.2.1.3 Event20	71
4.2.1.3.1 Event20 – using Dijkstra’s Algorithm	71
4.2.1.3.2 Event20 – using Alternate Path Routing Algorithm	77
4.2.1.4 Event50	81
4.2.1.4.1 Event50 – using Dijkstra’s Algorithm	82
4.2.1.4.2 Event50 – using Alternate Path Routing Algorithm	91
4.2.2 Network2	97
4.2.2.1 Event0	102
4.2.2.1.1 Event0 – using Dijkstra’s Algorithm	102
4.2.2.1.2 Event0 – using Alternate Path Routing Algorithm	106
4.2.2.2 Event10	110
4.2.2.2.1 Event10 – using Dijkstra’s Algorithm	110
4.2.2.2.2 Event10 – using Alternate Path Routing Algorithm	114
4.2.2.3 Event20	118

4.2.2.3.1 Event20 – using Dijkstra’s Algorithm	118
4.2.2.3.2 Event20 – using Alternate Path Routing Algorithm	122
4.2.2.4 Event50	126
4.2.2.4.1 Event50 – using Dijkstra’s Algorithm	127
4.2.2.4.2 Event50 –using Alternate Path Routing Algorithm	131
4.2.3 Network3	135
4.2.3.1 Event0	140
4.2.3.1.1 Event0 – using Dijkstra’s Algorithm	140
4.2.3.1.2 Event0 – using Alternate Path Routing Algorithm	143
4.2.3.2 Event10	146
4.2.3.2.1 Event10 – using Dijkstra’s Algorithm	146
4.2.3.2.2 Event10 – using Alternate Path Routing Algorithm	149
4.2.3.3 Event20	152
4.2.3.3.1 Event20 – using Dijkstra’s Algorithm	153
4.2.3.3.2 Event20 – using Alternate Path Routing Algorithm	156
4.2.3.4 Event50	159
4.2.3.4.1 Event50 – using Dijkstra’s Algorithm	159
4.2.3.4.2 Event50 – using Alternate Path Routing Algorithm	162
 CHAPTER 5 CONCLUSION AND RECOMMENDATIONS	 166
 BIBLIOGRAPHY	 169
 APPENDIX 1 C++ CODE AND DOCUMENTATION	 171
 APPENDIX 2 LIST OF ACRONYMS USED	 200
 VITA	 201

LIST OF FIGURES

Figure 1.1 MPLS.....	3
Figure 2.1 A Sample Network	9
Figure 2.2 Shortest Path from Node0 to Node6.....	10
Figure 2.3. Widest-Shortest Path from Node0 to Node6	11
Figure 2.4. Shortest-Widest Path from Node0 to Node6	12
Figure 2.5. Dynamic-Alternative Path from Node0 to Node6.....	12
Figure 2.6. Path from Node2 to Node8 got using the MIRA Algorithm.....	13
Figure 2.7. Path from Node2 to Node8 got using the Profile Based Routing.....	15
Figure 3.1 A Sample Network	31
Figure 3.2. Switching of traffic in the network.....	33
Figure 4.1. Network loaded into the linked list	45
Figure 4.2. Network1	53
Figure 4.3. Network1–Event0: Throughput of Link 13 – Dijkstra’s Algorithm	58
Figure 4.4. Network1–Event0: Delay of Link 13 – Dijkstra’s Algorithm.....	58
Figure 4.5. Network1–Event0: Throughput of link 13–Alternate Path Routing	61
Figure 4.6. Network1–Event10: Throughput of Link 13 – Dijkstra’s Algorithm	63
Figure 4.7. Network1–Event10: Throughput of Link 6 – Dijkstra’s Algorithm	64
Figure 4.8. Network1–Event10: Throughput of Link 3 – Dijkstra’s Algorithm	64
Figure 4.9. Network1–Event10: Delay of Link 13 – Dijkstra’s Algorithm.....	65
Figure 4.10. Network1–Event10: Delay of Link 6 – Dijkstra’s Algorithm.....	65
Figure 4.11. Network1–Event10: Delay of Link 3 – Dijkstra’s Algorithm.....	66
Figure 4.12. Network1–Event10: Throughput of link 13–Alternate Path Routing	68
Figure 4.13. Network1–Event10: Throughput of link 6–Alternate Path Routing	68
Figure 4.14. Network1–Event10: Throughput of link 3–Alternate Path Routing	69
Figure 4.15. Network1–Event10: Throughput of link 5–Alternate Path Routing	69
Figure 4.16. Network1–Event10: Throughput of link 8–Alternate Path Routing	70
Figure 4.17. Network1–Event20: Throughput of Link 13 – Dijkstra’s Algorithm	73
Figure 4.18. Network1–Event20: Throughput of Link 12 – Dijkstra’s Algorithm	73
Figure 4.19. Network1–Event20: Throughput of Link 6 – Dijkstra’s Algorithm	74
Figure 4.20. Network1–Event20: Throughput of Link 3 – Dijkstra’s Algorithm	74
Figure 4.21. Network1–Event20: Delay of Link 13 – Dijkstra’s Algorithm.....	75
Figure 4.22. Network1–Event20: Delay of Link 12 – Dijkstra’s Algorithm.....	75
Figure 4.23. Network1–Event20: Delay of Link 6 – Dijkstra’s Algorithm.....	76
Figure 4.24. Network1–Event20: Delay of Link 3 – Dijkstra’s Algorithm.....	76
Figure 4.25. Network1–Event20: Throughput of link 13–Alternate Path Routing	78
Figure 4.26. Network1–Event20: Throughput of link 12–Alternate Path Routing	79

Figure 4.27. Network1–Event20: Throughput of link 6–Alternate Path Routing	79
Figure 4.28. Network1–Event20: Throughput of link 3–Alternate Path Routing	80
Figure 4.29. Network1–Event20: Throughput of link 5–Alternate Path Routing	80
Figure 4.30. Network1–Event20: Throughput of link 8–Alternate Path Routing	81
Figure 4.31. Network1–Event50: Throughput of Link 13 – Dijkstra’s Algorithm	84
Figure 4.32. Network1–Event50: Throughput of Link 12 – Dijkstra’s Algorithm	84
Figure 4.33. Network1–Event50: Throughput of Link 11 – Dijkstra’s Algorithm	85
Figure 4.34. Network1–Event50: Throughput of Link 6 – Dijkstra’s Algorithm	85
Figure 4.35. Network1–Event50: Throughput of Link 3 – Dijkstra’s Algorithm	86
Figure 4.36. Network1–Event50: Throughput of Link 2 – Dijkstra’s Algorithm	86
Figure 4.37. Network1–Event50: Throughput of Link 1 – Dijkstra’s Algorithm	87
Figure 4.38. Network1–Event50: Delay of Link 13 – Dijkstra’s Algorithm.....	87
Figure 4.39. Network1–Event50: Delay of Link 12 – Dijkstra’s Algorithm.....	88
Figure 4.40. Network1–Event50: Delay of Link 11 – Dijkstra’s Algorithm.....	88
Figure 4.41. Network1–Event50: Delay of Link 6 – Dijkstra’s Algorithm.....	89
Figure 4.42. Network1–Event50: Delay of Link 3 – Dijkstra’s Algorithm.....	89
Figure 4.43. Network1–Event50: Delay of Link 2 – Dijkstra’s Algorithm.....	90
Figure 4.44. Network1–Event50: Delay of Link 1 – Dijkstra’s Algorithm.....	90
Figure 4.45. Network1–Event50: Throughput of link 13–Alternate Path Routing	92
Figure 4.46. Network1–Event50: Throughput of link 12–Alternate Path Routing	93
Figure 4.47. Network1–Event50: Throughput of link 11–Alternate Path Routing	93
Figure 4.48. Network1–Event50: Throughput of link 6–Alternate Path Routing	94
Figure 4.49. Network1–Event50: Throughput of link 3–Alternate Path Routing	94
Figure 4.50. Network1–Event50: Throughput of link 2–Alternate Path Routing	95
Figure 4.51. Network1–Event50: Throughput of link 1–Alternate Path Routing	95
Figure 4.52. Network1–Event50: Throughput of link 5–Alternate Path Routing	96
Figure 4.53. Network1–Event50: Throughput of link 8–Alternate Path Routing	96
Figure 4.54. Network2	97
Figure 4.55. Network2–Event0: Throughput of Link 2 – Dijkstra’s Algorithm	104
Figure 4.56. Network2–Event0: Throughput of Link 3 – Dijkstra’s Algorithm	104
Figure 4.57. Network2–Event0: Delay of Link 3 – Dijkstra’s Algorithm.....	105
Figure 4.58. Network2–Event0: Delay of Link 2 – Dijkstra’s Algorithm.....	105
Figure 4.59. Network2–Event0: Throughput of link 3–Alternate Path Routing	107
Figure 4.60. Network2–Event0: Throughput of link 2–Alternate Path Routing	108
Figure 4.61. Network2–Event0: Throughput of link 8–Alternate Path Routing	108
Figure 4.62. Network2–Event0: Throughput of link 10–Alternate Path Routing	109
Figure 4.63. Network2–Event0: Throughput of link 13–Alternate Path Routing	109
Figure 4.64. Network2–Event10: Throughput of Link 2 – Dijkstra’s Algorithm	112
Figure 4.65. Network2–Event10: Throughput of Link 3 – Dijkstra’s Algorithm	112
Figure 4.66. Network2–Event10: Delay of Link 3 – Dijkstra’s Algorithm.....	113
Figure 4.67. Network2–Event10: Delay of Link 2 – Dijkstra’s Algorithm.....	113
Figure 4.68. Network2–Event10: Throughput of link 3–Alternate Path Routing	115
Figure 4.69. Network2–Event10: Throughput of link 2–Alternate Path Routing	116

Figure 4.70. Network2–Event10: Throughput of link 8–Alternate Path Routing	116
Figure 4.71. Network2–Event10: Throughput of link 10–Alternate Path Routing	117
Figure 4.72. Network2–Event10: Throughput of link 13–Alternate Path Routing	117
Figure 4.73. Network2–Event20: Throughput of Link 2 – Dijkstra’s Algorithm	120
Figure 4.74. Network2–Event20: Throughput of Link 3 – Dijkstra’s Algorithm	120
Figure 4.75. Network2–Event20: Delay of Link 3 – Dijkstra’s Algorithm.....	121
Figure 4.76. Network2–Event20: Delay of Link 2 – Dijkstra’s Algorithm.....	122
Figure 4.77. Network2–Event20: Throughput of link 3–Alternate Path Routing	124
Figure 4.78. Network2–Event20: Throughput of link 2–Alternate Path Routing	124
Figure 4.79. Network2–Event20: Throughput of link 8–Alternate Path Routing	125
Figure 4.80. Network2–Event20: Throughput of link 10–Alternate Path Routing	125
Figure 4.81. Network2–Event20: Throughput of link 13–Alternate Path Routing	126
Figure 4.82. Network2–Event50: Throughput of Link 2 – Dijkstra’s Algorithm	129
Figure 4.83. Network2–Event50: Throughput of Link 3 – Dijkstra’s Algorithm	129
Figure 4.84. Network2–Event50: Delay of Link 3 – Dijkstra’s Algorithm.....	130
Figure 4.85. Network2–Event50: Delay of Link 2 – Dijkstra’s Algorithm.....	131
Figure 4.86. Network2–Event50: Throughput of link 3–Alternate Path Routing	133
Figure 4.87. Network2–Event50: Throughput of link 2–Alternate Path Routing	133
Figure 4.88. Network2–Event50: Throughput of link 8–Alternate Path Routing	134
Figure 4.89. Network2–Event50: Throughput of link 10–Alternate Path Routing	134
Figure 4.90. Network2–Event50: Throughput of link 13–Alternate Path Routing	135
Figure 4.91. Network3	135
Figure 4.92. Network3–Event0: Throughput of Link 10 – Dijkstra’s Algorithm	142
Figure 4.93. Network3–Event0: Delay of Link 10 – Dijkstra’s Algorithm.....	142
Figure 4.94. Network3–Event0: Throughput of link 10–Alternate Path Routing	144
Figure 4.95. Network3–Event0: Throughput of link 9–Alternate Path Routing	145
Figure 4.96. Network3–Event0: Throughput of link 11–Alternate Path Routing	145
Figure 4.97. Network3–Event10: Throughput of Link 10 – Dijkstra’s Algorithm	148
Figure 4.98. Network3–Event10: Delay of Link 10 – Dijkstra’s Algorithm.....	149
Figure 4.99. Network3–Event10: Throughput of link 10–Alternate Path Routing	151
Figure 4.100. Network3–Event10: Throughput of link 9–Alternate Path Routing	151
Figure 4.101. Network3–Event10: Throughput of link 11–Alternate Path Routing	152
Figure 4.102. Network3–Event20: Throughput of Link 10 – Dijkstra’s Algorithm	155
Figure 4.103. Network3–Event20: Delay of Link 10 – Dijkstra’s Algorithm.....	155
Figure 4.104. Network3–Event20: Throughput of link 10–Alternate Path Routing	157
Figure 4.105. Network3–Event20: Throughput of link 9–Alternate Path Routing	158
Figure 4.106. Network3–Event20: Throughput of link 11–Alternate Path Routing	158
Figure 4.107. Network3–Event50: Throughput of Link 10 – Dijkstra’s Algorithm	161
Figure 4.108. Network3–Event50: Delay of Link 10 – Dijkstra’s Algorithm.....	162
Figure 4.109. Network3–Event50: Throughput of link 10–Alternate Path Routing	164
Figure 4.110. Network3–Event50: Throughput of link 9–Alternate Path Routing	164
Figure 4.111. Network3–Event50: Throughput of link 11–Alternate Path Routing	165

LIST OF TABLE

Table 3.1 An example network with link bandwidth.....	31
Table 3.2. Routing Table of Node_0	32
Table 3.3. Traffic flow setup between edge nodes	33
Table 3.4. Traffic flow split in each link of the network	34
Table 3.5. Routing_Table List after computation of shortest path	35
Table 3.6. Alternate Paths for node_6 from node_0.....	35
Table 3.7. Routing_Table List with the alternate routes.....	36
Table 3.8. Routing_Table List with the alternate routes.....	37
Table 3.9. Routing_Table List with paths arranged in ascending order	38
Table 3.10. Routing_Table List with paths arranged in ascending order	38
Table 3.11. Set of Traffic flow to be setup	39
Table 3.12. Amount of traffic flowing in each link after first flow is handled.....	40
Table 3.13. Amount of traffic flowing in each link After second flow is handled	41
Table 3.14. Amount of traffic flowing in each link After third flow is handled	41
Table 4.1. Node Attributes.....	45
Table 4.2. Node Attributes.....	46
Table 4.3. Cost ordered list attributes	47
Table 4.4. Routing table list attributes	47
Table 4.5. Flow Result description	52
Table 4.6. Routing table of node1 of Network1–Dijkstra’s Algorithm.....	54
Table 4.7. Routing table of node1 of Network1–Alternate Path Routing Algorithm.....	54
Table 4.8. Network1-Event0: Set of Flows	55
Table 4.9. Network1-Event0: Set of Critical links	56
Table 4.10. Network1-Event0: Flow Results–Using Dijkstra’s Algorithm.....	57
Table 4.11. Network1-Event0: Flow Results–Using Alternate Path Routing	60
Table 4.12. Network1-Event10: Set of Flows	61
Table 4.13. Network1-Event10: Set of Critical links	62
Table 4.14. Network1-Event10: Flow Results–Using Dijkstra’s Algorithm.....	63
Table 4.15. Network1-Event10: Flow Results–Using Alternate Path Routing	67
Table 4.16. Network1-Event20: Set of Flows	71
Table 4.17. Network1-Event20: Set of Critical links	71
Table 4.18: Network1-Event20: Flow Results–Using Dijkstra’s Algorithm.....	72
Table 4.19. Network1-Event20: Flow Results–Using Alternate Path Routing	77
Table 4.20. Network1-Event50: Set of Flows	82
Table 4.21. Network1-Event50: Set of Critical links	82
Table 4.22. Network1-Event50: Flow Results–Using Dijkstra’s Algorithm.....	83
Table 4.23. Network1-Event50: Flow Results–Using Alternate Path Routing	91

Table 4.24. Routing Table of Node1 of Network2 –Dijkstra’s Algorithm.....	98
Table 4.25. Routing Table of Node1 of Network2 –Alternate Path Routing Algorithm	100
Table 4.26. Network2-Event0: Set of Flows	102
Table 4.27. Network2-Event0: Set of Critical links	103
Table 4.28. Network2-Event0: Flow Results–Using Dijkstra’s Algorithm.....	103
Table 4.29. Network2-Event0: Flow Results–Using Alternate Path Routing	106
Table 4.30. Network2-Event10: Set of Flows	110
Table 4.31. Network2-Event10: Set of Critical links	111
Table 4.32. Network2-Event10: Flow Results–Using Dijkstra’s Algorithm.....	111
Table 4.33. Network2-Event10: Flow Results–Using Alternate Path Routing	114
Table 4.34. Network2-Event20: Set of Flows	118
Table 4.35. Network2-Event20: Set of Critical links	119
Table 4.36. Network2-Event20: Flow Results–Using Dijkstra’s Algorithm.....	119
Table 4.37. Network2-Event20: Flow Results–Using Alternate Path Routing	123
Table 4.38. Network2-Event50: Set of Flows	126
Table 4.39. Network2-Event50: Set of Critical links	127
Table 4.40. Network2-Event50: Flow Results–Using Dijkstra’s Algorithm.....	128
Table 4.41. Network2-Event50: Flow Results–Using Alternate Path Routing	132
Table 4.42. Routing Table of Node1 of Network3 –Dijkstra’s Algorithm.....	136
Table 4.43. Routing Table of Node1 of Network3 –Dijkstra’s Algorithm.....	138
Table 4.44. Network3-Event0: Set of Flows	140
Table 4.45. Network3-Event0: Set of Critical links	140
Table 4.46. Network3-Event0: Flow Results–Using Dijkstra’s Algorithm.....	141
Table 4.47. Network3-Event0: Flow Results–Using Alternate Path Routing	143
Table 4.48. Network3-Event10: Set of Flows	146
Table 4.49. Network3-Event10: Set of Critical links	147
Table 4.50. Network3-Event10: Flow Results–Using Dijkstra’s Algorithm.....	147
Table 4.51. Network3-Event10: Flow Results–Using Alternate Path Routing	150
Table 4.52. Network3-Event20: Set of Flows	153
Table 4.53. Network3-Event20: Set of Critical links	153
Table 4.54. Network3-Event20: Flow Results–Using Dijkstra’s Algorithm.....	154
Table 4.55. Network3-Event20: Flow Results–Using Alternate Path Routing	156
Table 4.56. Network3-Event50: Set of Flows	159
Table 4.57. Network3-Event50: Set of Critical links	160
Table 4.58. Network3-Event50: Flow Results–Using Dijkstra’s Algorithm.....	160
Table 4.59. Network3-Event50: Flow Results–Using Alternate Path Routing	163

CHAPTER 1

INTRODUCTION: INTERNET TRAFFIC

The Internet came into existence in late 70's. Since then the Internet has been growing at an unprecedented rate. Traffic in the Internet continues to double every year [1]. This leads to the need for more bandwidth, better quality and security. This chapter discusses about the growth of traffic in Internet and the measures to accommodate it.

Increase in traffic demands increase in bandwidth. To increase bandwidth new transmission lines and equipments have to be installed by the *Internet Service Provider* (ISP). The cost of installation and maintenance of transmission lines and equipments are huge and they prohibit the rapid expansion of bandwidth available to the end-user. The speed of the Internet is not a function of the speed of the modem or the bandwidth available it is a function of both the end users connectivity and the infrastructure. So the ISP's are reluctant to spend money on bandwidth. The above factor leads to slower growth rate in bandwidth. More Internet traffic flowing through less bandwidth leads to congestion, delay, packet loss etc.

One way to accommodate the increase in Internet traffic without increase in bandwidth is to route the traffic effectively via the existing bandwidth. In most of the networks present today not all the links are over-utilized. Only few links are over-utilized and only those links become congested. This is mainly because the current *Internet Gateway Protocols* (IGP) always use the shortest paths to forward (route) traffic. Using

shortest paths conserves network resources, but it causes some resources of the network to be over utilized while the others remain under utilized. The process of effectively routing the traffic in the internet and optimizing resource efficiency is called *Traffic Engineering*. By employing traffic engineering methodologies one can control the mapping and distribution of traffic over the existing network infrastructure to:

- 1) avoid and/or relieve congestion,
- 2) to assure satisfactory service delivery, and
- 3) to optimize resource efficiency.

In order to do Traffic Engineering effectively, the *Internet Engineering Task Force* (IETF) introduced the following algorithms and protocols:

1. *Multi Protocol Label Switching* (MPLS),
2. Constraint-based Routing, and
3. Enhanced Link State Interior Gateway Protocol.

In the following section (Section 1.1) the above algorithms and protocols are discussed in brief.

1.1 Multi-Protocol Label Switching Networks

MPLS is an advanced forwarding scheme. It extends routing with respect to packet forwarding and path controlling [2]. Packet forwarding is a process by which the data packets are switched from the source to the destination node in a network, while path controlling controls the path taken by each packet.

In MPLS a label identifies each packet. In an Asynchronous Transfer Mode (ATM) [3] this label is inserted in the Virtual Path Identifier (VPI) [3] and Virtual Control Identifier

(VCI) [3] fields. In all other environments the label is present in a header called the Shim Header. Shim header is 32 bits in length. It resides between the layer 2² (L_2) and layer 3 (L_3) headers. The shim header consists of

- Label – Label value, 20 bits in length. This value contains the MPLS label.
- Exp – Experimental use, 3-bits in length. This field is not yet fully defined.
- S – Stacking bit, 1-bits in length. Used to indicate stacking of multiple labels and
- TTL – TTL field, 8-bits in length. Places a limit on how many hops the MPLS packet can be forwarded.

Figure 1.1: MPLS

Figure 1.1 depicts a MPLS domain. There are 4 hosts and 3 routers, which switch packets between them. Routers in MPLS domain are referred to as Label Switched

² Layer 2 is also called the datalink layer and Layer 3 is also called the network layer

Routers (LSR). The first LSR a packet encounters in a MPLS domain is called the ingress LSR, while the last LSR is known as the egress LSR. At the ingress LSR, IP packets are classified based on a combination of the information carried in the IP header of the packets and the local routing information maintained by the LSR and a label is assigned to them. The assigned label is then distributed to the neighboring LSR in the MPLS domain. The neighboring LSR then associates a label with this incoming label and distributes it to its neighbors. This process is known as label distribution and it ends at the egress LSR. The incoming label and corresponding outgoing labels are stored in a table, known as the forwarding table.

At the ingress LSR a label is then inserted into each packet. Within an MPLS-capable domain, each LSR uses the label to forward the packet. At each LSR (except the egress LSR) the outgoing label replaces the incoming label and the packet is switched to the next LSR. The process of switching the label is known as Label Swapping. At the egress LSR the label is removed and the packet is forwarded to the next domain. This whole process is showed in Figure 1.1. The path between the ingress LSR and the egress LSR is referred to as label switched paths (LSP). MPLS uses some signaling protocol like Resource reservation protocol (RSVP) [4] [5], Label Distribution protocol (LDP) [5] or Constraint based LDP (CR-LDP) [6] to setup a LSP and distribute the label.

1.1.1. Traffic Engineering with MPLS

The most significant application of MPLS is in traffic engineering. In MPLS traffic engineering is done effectively using the LSP's which can be manipulated and managed by the network administrator to direct the traffic. A LSP can be formed in two ways

control-driven or Explicitly Routed LSP (ER-LSP). In a control driven LSP each and every LSR decides the next interface through which the LSP should be setup based on its Layer 3 topology database.

An ER-LSP is setup on a predetermined path. So each LSR decides the next interface based on the predetermined path specified in the path setup message. The control-driven LSP normally takes the path a layer 3 routed packets will take, while the ER-LSP can take a path completely independent of the layer 3 topology. The ER-LSP can be specified by the network operator or the network management application and in this way used to achieve traffic engineering.

1.2 Enhanced Interior Gateway Protocol

An Interior Gateway Protocol (IGP) is a way of exchanging information within a domain or an Autonomous System (AS)³. The two commonly used IGP's are Open Shortest Path First (OSPF) [7] and the Routing Information Protocol (RIP) [8]. OSPF is the most commonly used protocol for traffic engineering and its operation is briefly explained below.

The IGP is a network protocol used to get the topological information of an AS. Using the information provided by the IGP a routing algorithm computes the routes to all nodes in an AS. As an example Dijkstra Algorithm is commonly used with the OSPF to compute the routes for different nodes in an AS.

An IGP, like OSPF passes the information about the nodes and links present in a network. It does not pass the information about the amount of traffic flowing in each link

³ An Autonomous System (AS) or a routing domain is a network controlled by a single network administrator.

or the delay associated with a particular node or link. This information passed by the IGP is insufficient to do effective traffic engineering. So the IGP was enhanced to include the traffic engineering parameters and extension OSPF-TE [9] was defined.

1.2.1 OSPF-TE

This extension provides a way of describing the traffic engineering topology (including bandwidth and administrative constraints). The information made available by these extensions can be used to build an extended link state database⁴. The extended link state database (referred to below as the traffic engineering database) has additional link attributes describing the amount of bandwidth available, delay associated with a link etc. This extension makes use of the Opaque Link State Advertisements (LSA) [10]. One new LSA is defined, the Traffic Engineering LSA. This LSA describes routers, point-to-point links, and connections to multi-access networks (similar to a Router LSA).

Using the information provided by the OSPF-TE traffic engineering can be effectively done. More complex routing algorithm that routes traffic effectively, avoiding the more congested links can be implemented.

1.3 Constraint Based Routing and Quality of Service (QoS)

The present days Internet is transforming into a multi-service medium leading to the convergence of voice, video and data communications. Each type of service has a particular constraint and it has to be satisfied for the communication to be effective. For

⁴ A link state database is a repository for flooded topology link state information learned from the IGPs

example a voice or video data is delay sensitive and has to be transmitted with in a certain delay.

So the service for each and every type of data needs to be met. Traditionally the internet only provided the best effort service, where the traffic is processed as quickly as possible, but there is no guarantee to the timeliness and assurance of actual delivery. This type of single service can no longer meet the need of the present day constraints.

Integrated service (Intserv) [12] and Differentiated Service (Diffserv) [11] are the two models defined which provide QoS. The essence of Intserv is to reserve traffic from each and every flow, while that of diffserv is to divide the traffic into different types and process them according to their classification.

1.4 Organization of This Report

This report discusses how traffic engineering is done in the Internet. The focus is on the development of a new routing algorithm that routes traffic more efficiently and increases the network through put.

In Chapter 2, a brief introduction to traffic engineering and the more commonly used algorithms is given. In chapter 3, the new algorithm is proposed and compared with the more commonly used algorithms. In chapter 4, the simulation results are shown and the results are compared with the simulation results of the shortest path algorithm. The future work and conclusion is given in chapter 5.

CHAPTER 2

TRAFFIC ENGINEERING

The task of mapping traffic flows onto an existing physical topology is called Traffic Engineering (TE). Methodologies to improve traffic flows include 1) Routing Algorithms, 2) Network Planning and 3) Network Optimization. The purpose of routing algorithms is to optimize resource efficiency and network utilization. Network planning is to improve the architecture (topology and link capacity) of a network in a systematic way so that the network is robust, adaptive and easy to operate. Network optimization is to control the mapping and distribution of traffic over the existing network infrastructure to avoid and/or relieve congestion, to assure satisfactory service delivery, and to optimize resource efficiency.

2.1 Traffic Engineering Algorithms

This section describes briefly the various traffic engineering algorithm commonly used, then a description on the various routing requirements that a flow routing algorithm must satisfy is given followed by the evaluation of the current routing algorithms. The network shown below would be used to describe the various algorithms.

Figure 2.1: A Sample Network

2.1.1 Shortest Path Algorithm – Dijkstra Algorithm

This routing algorithm is based on the link state Routing Protocol⁵. This algorithm computes the shortest distance path for all the nodes in a network. Shortest-distance path [13] is defined as a feasible path with the shortest distance. The distance function is defined by

$$Dist(p) = \sum_{j=1}^k 1/LB_j$$

where, LB_j is the bandwidth available on link L_j ,

L_j is the link identifier, $j=1$ to k (k =number of links).

The shortest distance path is one of the earliest routing algorithms and it is one of the most commonly used algorithms, due to its ease in implementation and less computation intensive nature.

⁵ Link State Routing Protocol: This type of routing protocol requires each router to maintain at least a partial map of the network. When a network link changes state (up to down, or vice versa), a notification, called a *link state advertisement* (LSA) is *flooded* throughout the network

Figure 2.2: Shortest Path from Node0 to Node6

The above Figure shows the shortest path between node0 and node6. There are a lot of extensions to this algorithm. Few of them are discussed below.

2.1.1.1 Widest-Shortest Path

Widest – Shortest Path [13] algorithm uses the shortest path algorithm to compute a path with minimum hop count among all the feasible paths. If there are several such paths, the one with the maximum reservable bandwidth is selected.

The Figure 2.3 shows the widest shortest path between node0 and node6. Here there is only one shortest path between the two nodes so the shortest path and the widest shortest path are one and the same for this network.

Figure 2.3: Widest-Shortest Path from Node0 to Node6

2.1.1.2 Shortest-Widest Path

Figure 2.4: Shortest-Widest Path from Node0 to Node6

Shortest-Widest Path [13] routing algorithm computes a path with the maximum reservable bandwidth among all feasible paths. If there are several such paths one with the minimum hop count is selected.

The Figure 2.4 shows the widest shortest path between node0 and node6. Here the 4 Mbps link (one in green) is assumed to have the maximum reservable bandwidth.

2.1.1.3 Dynamic Alternative Routing

Dynamic Alternative Routing Algorithm [13] is an extension of the widest-shortest path algorithm. Let n be the hop count of a minimum-hop path when the network is idle. A dynamic alternative path is a widest-shortest path with no more than $n+1$ hops.

Figure 2.5: Dynamic-Alternative Path from Node0 to Node6

The Figure2.5 shows the path between node0 and node6 setup using the dynamic alternative algorithm. Since the dynamic alternative path is a path which can have one hop count greater than the widest-shortest path the path marked in black is chosen.

2.1.2 Minimum Interference Routing with Application to MPLS Traffic Engineering (MIRA)

This is an online algorithm, which does not have any priori knowledge of the traffic requests. It just routes request as they come in. This algorithm chooses a path for the request that causes minimum interference for the future potential LSP. In other words a path that maximizes the minimum open capacity between every other ingress-egress pairs (Potential nodes where flow can generate and leave the MPLS domain) is chosen.

Figure 2.6: Path from Node2 to Node8 got using the MIRA Algorithm

A detailed mathematical formulation of the algorithm and simulation results can be found in [14].

Assume that there is a traffic flow from node2 to node8 in the above network which requests a bandwidth of 1.5Mbps. MIRA algorithm chooses a path which least interferes with other potential paths. The other potential nodes where a traffic request can originate are node0, node1, node6, node7 and node8. If this traffic flow is setup in any of the other 2 paths, then the maximum reservable bandwidth between the other nodes will get affected maximally. So MIRA chooses a path which has minimal effect on the maximum reservable bandwidth between other potential traffic originating nodes. This path is marked in the Figure 2.6.

2.1.3 Profile Based Routing

This is an algorithm [15] and framework for dynamic routing of bandwidth guaranteed flows. The main idea of the algorithm is to use the “traffic profile” of the network obtained by measurements or service level agreements⁶ (SLA), as a rough predictor of the future traffic distribution. This algorithm has two phases, viz. Preprocessing phase and Online routing phase.

The algorithm uses the “traffic profile” in the preprocessing step (one multi-commodity flow computation) to determine certain bandwidth allocation on the links of the network. The online phase of the routing algorithm then routes the request using a “shortest path” (Shortest Path Forwarding) like algorithm but with the additional information given by the preprocessing phase. The multi-commodity preprocessing phase

⁶ SLA is a means by which the service provider assures a certain type of service to the user.

allows the online algorithm to exercise admission control by rejecting some request because of their blocking effects in the network.

The path between node2 to node8 got using the profile based routing is shown in Figure 2.7.

Figure 2.7: Path from Node2 to Node8 got using the Profile Based Routing

2.2 Routing Requirements

In this section a brief description about the routing requirements of a flow routing algorithm is given. A detailed list of routing requirements is given in [16].

a. Routing without splitting flows

Each flow request must be routed through a single route, without splitting. Many flows may involve traffic that is unsplittable (circuit emulation, voice, video etc.). So for each request the routing algorithm should find a path or say that the flow is unroutable.

b. Online Routing

A routing algorithm should be able to route traffic as the request comes in. It should not require knowledge of any dynamic data, like other traffic requests, their bandwidth, time of arrival of their request etc.

c. Computational requirements

The routing algorithm should be fast and scalable. The path to the destination for each flow is determined in the ingress router, which is heavily loaded and has less computational power. So the routing algorithm should be as simple as possible.

d. Policy Constraint

A good routing algorithm should be able to incorporate various policy constraints. For example an SLA for video traffic may require that the traffic should be able to flow through a given minimum number of links.

e. Traffic Profile

The history of traffic flow between the edge routers is a good estimator of today's traffic flow. So the traffic history or the statistics of the data flow between the edge routers could be used as a rough predictor of the future traffic flow while determining traffic route to the destination.

f. Use Knowledge of edge routers

The knowledge of edge routers or potential points from which traffic request can originate and terminate if known should be used by the routing algorithm. It is not good to assume that all the routers in the network are potential edge routers; this will lead to unnecessary computations and increase the computation intensity of the algorithm.

g. Good rerouting upon link failure

This is clearly a performance metric. A routing algorithm should be able to find alternate route for traffic flows when there is a link failure. This rerouting should be done as soon as possible and should be done for as many link failures that happen.

h. Feasibility of distributed implementation

It is desirable that the routing algorithms be able to compute routes without it having to contact a centralized server for information. It should be able to use the information distributed by the routing protocols alone and determine the routes.

i. Re-Optimization

Re-routing traffic for optimizing the flow in the network is called re-optimization. Frequent re-routing is not desirable, but occasional re-routing to optimize traffic flow in a network is acceptable. When such re-optimization occurs it should be consistent with the rest of the network. This is important, as otherwise routing loops may get formed.

2.3 Evaluation Of Existing Routing Algorithm

In this section an informal description of the shortcomings of the existing algorithms are illustrated. The parameters throughput, delay, complexity of routing algorithm, ease of implementation and number of request accepted are being used to evaluate the routing algorithms.

The shortest path algorithm is one of the earliest and simplest routing algorithms. The computation of the shortest path is simple and less intensive. But, the Shortest Path algorithms routes traffic via the shortest path. This implies that all the traffic in the network will be routed via the shortest path even when other longer path exists. This

leaves part of the network underutilized while the other part is over utilized. The over utilized part of the network gets congested often. Due to the congestion the delay increases, throughput decreases. The shortest path algorithms also do not impose any form of admission control. This implies that any single large flow request can potentially lead to the degradation of all other future requests.

MIRA is a routing algorithm which routes traffic through a path which causes minimum interference to potential flows. This implies that the traffic gets routed via the path that will lessen the chance of congestion for all other potential flows. Unlike the shortest path algorithms this algorithm ensures that there is no discrepancy in utilization within the network. But this algorithm is highly computation intensive, it requires the max flow computations. This algorithm requires a router or switch to know the list of edge routers. This algorithm also does not impose any form of admission control. This implies that any single large flow request can potentially lead to block all other future request.

Profile based routing algorithm is similar to the MIRA algorithm. Here the computational intensive part in MIRA is computed in the offline phase of the algorithm. The online phase of the routing algorithm keeps track of the residual capacity in each link and per traffic class. The online phase also imposes admission control. This admission control is based on the residual capacity of each link. So this routing algorithm is less computation intensive. But this algorithm requires the knowledge of all the potential flows in the form of SLA's. If this detail is not accurate then the performance of this algorithm degrades. This requirement also makes it difficult to implement this routing algorithm when the SLA's are not available.

CHAPTER 3

ALTERNATE PATH ROUTING ALGORITHM

The problem of routing is to optimize flow of packets through a network (Traffic Engineering) with various constraints such as,

- 1) Optimize network utilization,
- 2) Minimize cost of switching,
- 3) Minimize the number of hop counts and
- 4) Optimize use of link bandwidth.

The basic problem of routing is equivalent to the max-flow or shortest path problem and can be solved in polynomial time. The routing problem that optimizes many constraints is classified as a NP-Hard problem. A heuristic algorithm for solving the bandwidth constraint routing is proposed. The definitions used are described below.

3.1 Definitions

a) Node

A node in a network is defined as a router, switch or any other device capable of switching a data packet.

b) Network

A network is represented by a graph $G(v,e)$, where v is the vertex representing each node in the network and e is the set of edge or link interconnecting each vertex.

c) Bandwidth of Link

Bandwidth of a link is defined as the maximum traffic that the link can accommodate at any given time. In other words it is the capacity of the link.

d) Link

A link is a bus or a cable connecting two or more nodes.

e) Link Number or Interface Number

The link number or the interface number is a unique number that is used to identify a link connecting 2 nodes in a network

f) Critical Link (CL)

This is defined as the link that is responsible for congestion. In other words this is the link in which more traffic than the bandwidth of the link tries to flow, leading to congestion.

g) Hop Count

In data communication networks hop count is defined as the number of legs traversed by a data packet from source to destination.

h) Traffic Spike or burst

Traffic spike or burst is a sudden surge of traffic flowing in the network which lasts for a very small time.

i) Link State Advertisements

Each router in an Autonomous system originates one or more link state advertisements. There are 5 types of LSA that a router can originate and each type has its own function [7].

j) Link State Database

A list of Link State Advertisements (LSA's) is stored in the Link State database.

k) Path

A path to a destination from a source is defined as the list of intermediate nodes or nodes that a packet has to traverse in order to reach the destination from the source.

l) Path Cost

It is defined as the number of hops or links that a packet traverses in the described path in order to reach the destination. In other words it is the hop count of the path.

3.2 The Proposed Algorithm

The algorithm consists of two phases

- 1) Pre-processing phase and
- 2) Online routing phase.

In the pre-processing phase the critical links in a network is identified. The online routing phase consists of two planes

- 1) The control plane and
- 2) The forwarding plane.

In the control plane the paths between all nodes from a start node are computed, while in the forwarding plane the data packets are switched from the source node to the destination node via the best available path. The two phases in the algorithm are described below.

3.2.1 Pre-Processing Phase

In this pre-processing phase the critical links in a network are identified. The existing routing algorithm or the shortest path algorithm is run on the network and this algorithm is allowed to switch traffic or move data from one node to another for a period of time. During this period the data traffic pattern flowing in each link of the network is tabulated.

This data traffic flow pattern can be obtained from the linkstate database of a node. The linkstate database stores the raw data of the amount of traffic flowing in each link over a network. This data varies significantly at different time due to traffic spike or burst. This traffic burst or spike should not be taken for congestion as it is for a very short period. So it must be decided how to use the raw data. One common choice is the 95th-percentile of all rates measured every 5 minutes over a period of time. This value is used as it is close to the real peak value as opposed to the traffic spike.

This data is used to find the amount of traffic flowing in each link. From the amount of traffic flowing in each link the critical links are identified as those link in which the amount of traffic flowing is equal to the capacity of the link. In other words the critical link is a link in a network that is responsible for congestion. The interface number or the link number of all such critical links is noted.

3.2.1.1 Algorithm of the Pre-Processing Phase

Step 1:- Create an empty list "*Critical Link List*",

Step 2:- Switch traffic in the network using the current routing algorithm or use the dijkstra algorithm for a period of time,

Step 3:- Collect data from the linkstate database,

Step 4:- Compute the peak rate of traffic flowing in each link, from the linkstate database,

Step 5:- Then the critical links are identified as the links whose peak rate is equal to the link capacity. These links are added into the critical link list.

3.2.1.2 Pseudo Code for Pre-Processing Phase.

AIM: Switch traffic in the network using dijkstra's algorithm or the current routing algorithm used over a period of time. This builds the linkstate database in every node in the network. This linkstate database is used as the input for this phase to determine the critical links in the network

INPUT: Get the input node number.

OUTPUT: Create "Linkstate_List" and "Critical_Link_List"

INITIALIZE: Load linkstate database of the input node into a "Linkstate_List"

INITIALIZE: make "Critical_Link_List" = NULL;

WHILE "Linkstate_List" != Null DO

COMPUTE utilization for each link UL_i , for each link L_i

IF $UL_i = \text{Link Capacity}$ THEN

"Critical_Link_List".insert(L_i)

END IF

END WHILE

3.2.2 Online Routing Phase

The online routing phase can be divided into two planes

- 1) Control plane and

2) The forwarding plane.

In the control plane the routes to all the destinations are computed and in the forwarding plane the request to switch traffic to destinations are handled. The two planes are described in detail as follows.

3.2.2.1 Control Plane

The details distributed by the linkstate protocols like OSPF, IS-IS etc. are the input to this plane. This information distributed by these protocols is stored in the linkstate database.

A list of nodes and the links connecting them is determined from the linkstate database. The shortest path to all nodes in the network is computed. Let this shortest path have a cost say shortest path cost (SPC). The main idea in this algorithm is that there might be additional paths to the destination which remain non-congested, but have a higher hop count than the shortest path. These paths are called the *alternate Paths*. The additional hops that the *alternate paths* traverse should not be longer so as to utilize all the network resources. When the *alternate paths* utilize most of the network resources the throughput and performance of the network is reduced. So the additional hops traversed by the new paths should be limited. The additional number of hops traversed by the *alternate paths* is represented in terms of the *Shortest Path Cost* (SPC). This is done so that the additional hops traversed by the *alternate paths* increase linearly with the distance between the two nodes.

The cost of an *alternate paths* = $(1+\alpha) * \text{SPC}$,

where “ α ” is the *alternate path parameter*, which varies from 0 to 1.

The value of “ α ” varies from network to network and is left for the network administrator to decide what value to use.

Alternate paths are now computed to all the nodes in the network. One common choice for “ α ” while computing alternate paths is 0.1. This implies that the cost of the *alternate paths* is 10% greater than the SPC. All the paths (both shortest path and the *alternate paths*) to the nodes are now computed. Let the paths be named P_i , where i vary from 1 to total number of paths to a node.

The list of critical links for a network is got from the pre-processing phase. This list is used to identify the number of critical links present in each path. Let CL_i , represent the number of critical links in path P_i . After computing CL_i , the paths P_i to a destination are sorted in ascending order according to the number of critical links (CL_i). If there is more than one path with the same number of critical links (CL_i), then those paths are sorted according to the path cost.

The path P_1 that is the first path after sorting is named the primary path (PP). The primary path is a path, which has least number of critical links in it. If there are multiple such paths then the one with least path cost becomes the primary path. The rest of the paths are named as the additional paths (AP_i , where i varies from 1 to $d^7 - 1$).

3.2.2.1.1 Algorithm for Control Plane.

Step 1: Load the input network from the linkstate database,

Step 2: Get the input node number from which the paths to all other nodes in the network are to be computed,

⁷ d is total paths to a destination

Step 3: Compute the shortest path and the path cost (say SPC) from the start node to all other nodes in the network,

Step 4: Get the input " α ",

Step 5: Compute the alternate paths which have a path cost less than or equal to $(1+\alpha) * \text{SPC}$,

Step 6: Compute the path cost of each path. Let the path cost of each path be C_i for each path P_i ,

Step 7: Compute the number of critical links in each path. Let the number of critical links be CL_i for each path P_i ,

Step 8: Sort the paths according to the number of critical links CL_i present in them,

Step 9: If there is more than one path with the same number of critical links CL_i , then sort them in ascending order according to the path cost of each path C_i ,

Step 10: Name path P_1 as the primary path PP,

Step 11: Name path P_2 to $P_{\text{total_number_of_path}}$ as Additional paths (AP_i , where i varies from 1 to $d - 1$).

3.2.2.1.2 Pseudo Code for the Control Plane

AIM: To find the routes to the nodes in a network from the given start node.

INPUT: Load input network from linkstate database (Add nodes into "NODE_LIST").

INPUT: Start NODE_NUMBER and " α ".

OUTPUT: Create "ROUTING_TABLE_LIST".

INITIALIZE: Set INFINITY = Flow_Max.

INITIALIZE: "ROUTING_TABLE_LIST" = Null.


```

WHILE (Node_List.empty() != True) DO

    To_Node_Number = "NODE_LIST".Present.Node_Number

    COMPUTE shortest path SP to To_Node_Number

    SPC = path cost of SP

    "ROUTING_TABLE_LIST".Addroute(To_Node_Number, SP, SPC)

    "NODE_LIST".Present++

END WHILE

WHILE ("ROUTING_TABLE_LIST".Empty() != True) DO

    To_Node_No = "ROUTING_TABLE_LIST".Present.Node_Number

    SPC = "ROUTING_TABLE_LIST".Present.SPC

    COMPUTE alternate path cost, APCost =  $(1+\alpha) * SPC$ 

    COMPUTE Alternate path AP to To_Node_No with path cost  $\leq$  APCost

    APC = path cost of AP

    "ROUTING_TABLE_LIST".Addroute(To_Node_No, AP, APC)

    "ROUTING_TABLE_LIST".Present++

END WHILE

WHILE ("ROUTING_TABLE_LIST".Empty() != True) DO

    Path[i] = "ROUTING_TABLE_LIST".Present.Getpath().

    WHILE (Path[i] != Null) DO

        COMPUTE number of critical links  $CL_i$  for each Path[i]

        COMPUTE path cost  $C_i$  for each Path[i]

        i++

    END WHILE

END WHILE

```

```

FOR (i=1, i<totalNumberOfPaths - 1, i++)
    IF (CLi > CLi+1) THEN
        SWAP (Path[i], Path[i+1])
    END IF
END FOR

FOR (i=1, i<totalNumberOfPaths - 1, i++)
    IF (CLi = CLi+1) THEN
        IF (Ci > Ci+1) THEN
            SWAP (Path[i], Path[i+1])
        END IF
    END IF
END FOR

"ROUTING_TABLE_LIST".Present.Deleteallpaths()

FOR (i=1, i<totalNumberOfPaths - 1, i++)
    "ROUTING_TABLE_LIST".Present.Addnewpath(Path[i])
END FOR

"ROUTING_TABLE_LIST".Present++

END WHILE

```

3.2.2.2 Forwarding Plane

In the forwarding plane of the online phase the traffic is routed via the primary path, *PP*. If the primary path, *PP* is congested or gets congested, then the traffic is routed via the additional path, *AP₁*. If additional path, *AP₁*, is also congested then traffic is switched

via the other additional paths present. If no non-congested paths are available then the traffic flow request is rejected. This is done so because, it is better to reject a request than to accept it and let the performance of all other flows degrade along with it.

3.2.2.2.1 Algorithm for the Forwarding Plane

Step 1: Load request for into the Request_List,

Step 2: For first request in the Request_List find route to the destination node,

Step 3: Switch the data packets via the primary path *PP* if it is not congested,

Step 4: Find a non congested additional path if primary path is congested and switch traffic via that path,

Step 5: If all paths are congested or if no path to destination exist say request could not be handled,

Step 6: Delete the request handled from Request_List,

Step 7: Do steps 2 through 6 for all the requests in the Request_List.

3.2.2.2.2 Pseudo Code for the Forwarding Plane

AIM: To map all the data flow request onto the underlying network efficiently.

INPUT: Load request into the "REQUEST_LIST".

WHILE ("REQUEST_LIST".Empty() != True) DO

To_Node_Number = "REQUEST_LIST".Present.Destination_Node

Path = "ROUTING_TABLE_LIST".Getroute(To_Node_Number)

WHILE (Path[i] != Null) DO

IF (Path[i] not Congested) THEN

```

        switch traffic via this path

        Request_Handled_Sucessfully = 1

        BREAK from WHILE LOOP

    END IF

    i++

END WHILE

IF (Request_Handled_Sucessfully = 1) THEN

    PRINT "Traffic Switched successfully"

ELSE

    PRINT "Traffic can not be Switched"

END IF

    "REQUEST_LIST".Present++

END WHILE

```

3.3 Example

3.3.1 Example Network

Let us consider the example network described by the Figure 3.1. The link bandwidth and various other characteristics of the network are shown below in Table 3.1. The network shown below contains 12 nodes, 6 of which are edge nodes. The cost of each link in the network is assumed to be 1.

Figure 3.1: A example network

Table 3.1: An example network with link bandwidth

Link ID	Start Node	End Node	Link bandwidth in Mbps	Is Start Node Edge Node
1	Node_0	Node_3	3	Yes
2	Node_1	Node_3	4	Yes
3	Node_2	Node_3	2	Yes
4	Node_3	Node_4	4	No
5	Node_3	Node_11	2	No
6	Node_3	Node_5	3	No
7	Node_4	Node_5	4	No
8	Node_5	Node_9	2	No
9	Node_6	Node_5	3	Yes
10	Node_7	Node_5	4	Yes
11	Node_8	Node_5	2	Yes
12	Node_9	Node_10	2	No
13	Node_10	Node_11	2	No
13	Node_11	Node_10	2	No

3.3.2 Pre-Processing Phase

The pre-processing phase is used to identify the critical links in the network. Assume that the shortest path routing algorithm (Dijkstra's algorithm) is used to build the linkstate database of this network. Let the start node, for which the routing table is to be computed be node_0. The routing table of node_0 is given in Table 3.2. The routing table gives a list of routes to all other edge nodes in the network.

Table 3.2: Routing Table of Node_0

Destination Node	Cost	Shortest Path ⁸
Node_1	2	n0_n3_n1
Node_2	2	n0_n3_n2
Node_6	3	n0_n3_n5_n6
Node_7	3	n0_n3_n5_n7
Node_8	3	n0_n3_n5_n8

To identify the critical links in the network traffic has to be switched between the edge nodes and the linkstate database has to be formed. The list of traffic flow to be setup for the above network is shown in the Table 3.3.

As the traffic in the network is switched, as shown in Figure 3.2, the linkstate database is formed. The linkstate database has other data. The information computed from the linkstate database is the amount of traffic flowing in each link of the network.

⁸ The paths in the routing table are represented as n_i where i varies from 0 to 11. node_0 is represented as n_0 to make the path simple. The paths in the routing table list the nodes thro which the packets have to traverse in order to reach the destination. Each node in the path is separated by an underscore

Table 3.3: Traffic flow setup between edge nodes

Flow ID	Source Node	Destination Node	Amount of traffic in Mbps	Path through which the traffic flows
1	Node_0	Node_6	2	n0_n3_n5_n6
2	Node_1	Node_7	3	n1_n3_n5_n7
3	Node_2	Node_8	1	n2_n3_n5_n8

Figure 3.2: Switching of traffic in the network

This information is calculated for the above network-flow combination and is shown in Table 3.4. All the links except the link from node_3 to node_5 are non-congested. For link 6 the amount of traffic flowing is equal to the link bandwidth. It can be seen from the Table 3.4 that the amount of traffic trying to flow via the link 6 is 6Mbps while its capacity is 3 Mbps. This leads to congestion in the link 6.

Table 3.4: Traffic flow split in each link of the network

Link ID	Start Node	End Node	Link bandwidth in Mbps	95% peak rate in Mbps	Amount of Traffic trying to flow in Mbps	Critical Link
1	Node_0	Node_3	3	2	2	No
2	Node_1	Node_3	4	3	3	No
3	Node_2	Node_3	2	1	1	No
6	Node_3	Node_5	3	3	6	Yes
9	Node_5	Node_6	3	2	2	No
10	Node_5	Node_7	4	3	3	No
11	Node_5	Node_8	2	1	1	No

Since link 6 is the only link in the network that is responsible for congestion it is named as the critical link and it is added into the critical link list.

3.3.3 Online Routing Phase

In the online routing phase the routes to the destination nodes are computed and the data packets are switched in the computed routes. This phase is divided into two planes,

- 1) The control plane and
- 2) The forwarding plane.

3.3.3.1 The Control Plane

In the control plane the routes to the destination nodes are computed from the start node. The input to this plane is the way in which each node is connected (i.e. the network topology is the input to this plane). This input is loaded into a linked list (Node_List) from the linkstate database. The node for which the routes are to be computed is also got as an input. Let this node be node_0. Set the value of " α "=1 for this example.

Compute the shortest path to all the nodes in the Node_List. Add the shortest path to the routing table, which is another linked list "ROUTING_TABLE_LIST". The "ROUTING_TABLE_LIST" now contains the shortest path to all the destination nodes in the network and it is shown in Table 3.5.

Table 3.5: "ROUTING_TABLE_LIST" after computation of shortest path

Destination Node	Path Number	Cost	Path
Node_1	1	2	n0_n3_n1
Node_2	1	2	n0_n3_n2
Node_6	1	3	n0_n3_n5_n6
Node_7	1	3	n0_n3_n5_n7
Node_8	1	3	n0_n3_n5_n8

Table 3.6: Alternate Paths for node_6 from node_0

Alternate Path Number	Alternate Path Cost	Alternate Path
1	4	n0_n3_n4_n5_n6
2	6	n0_n3_n11_n10_n9_n5_n6

Now the alternate paths and the alternate paths for all the edge nodes in the network are computed. Compute the alternate paths for node_6. The shortest path for node_6 from node_0 is listed in Table 3.5. The path cost of the shortest path is 3 (SPC = 3). The " α " input for this network is 1. So the alternate path cost,

$$APCost = (1+\alpha) * SPC$$

$$APCost = (1+1) * 3 = 6$$

APCost for node_6 is 6. The alternate paths to node_6 can have a maximum path cost of 6. All such paths are computed and are added to the routing table. The alternate paths for node_6 are given in Table 3.6.

The alternate paths for all the edge nodes in the network are computed and are added to the routing table. The Routing_Table_List with the alternate routes are shown in Table 3.7. From Table 3.7 it can be seen that the edge nodes node_6, node_7 and node_8 have 2 alternate paths (path 2 and 3) apart from one shortest path. So now those three nodes have 3 routes to the destination.

Table 3.7: Routing_Table List with the alternate routes

Destination Node	Path Number	Cost	Path
Node_1	1	2	n0_n3_n1
Node_2	1	2	n0_n3_n2
Node_6	1	3	n0_n3_n5_n6
	2	4	n0_n3_n4_n5_n6
	3	6	n0_n3_n11_n10_n9_n5_n6
Node_7	1	3	n0_n3_n5_n7
	2	4	n0_n3_n4_n5_n7
	3	6	n0_n3_n11_n10_n9_n5_n7
Node_8	1	3	n0_n3_n5_n8
	2	4	n0_n3_n4_n5_n8
	3	6	n0_n3_n11_n10_n9_n5_n8

The critical links which were computed in the preprocessing phase is loaded into the “CRITICAL LINK LIST”. Link 6 connecting node_3 and node_5 is the only link in the “CRITICAL LINK LIST”. The number of critical links for each path is computed. The number of critical link for path 1 of Node_6 is computed below.

The path 1 for destination 6 is n0_n3_n5_n6. The links in this path are n0_n3, n3_n5 and n5_n6. The corresponding link numbers for those links are 1, 6 and 9. These numbers are stored in an array and compared with the critical link list. The number of critical link for this path is calculated as 1.

Table 3.8: Routing_Table List with the alternate routes

Destination Node	Path Number	Cost	Path	Number of Critical Links CL_i
Node_1	1	2	n0_n3_n1	0
Node_2	1	2	n0_n3_n2	0
Node_6	1	3	n0_n3_n5_n6	1
	2	4	n0_n3_n4_n5_n6	0
	3	6	n0_n3_n11_n10_n9_n5_n6	0
Node_7	1	3	n0_n3_n5_n7	1
	2	4	n0_n3_n4_n5_n7	0
	3	6	n0_n3_n11_n10_n9_n5_n7	0
Node_8	1	3	n0_n3_n5_n8	1
	2	4	n0_n3_n4_n5_n8	0
	3	6	n0_n3_n11_n10_n9_n5_n8	0

Similarly the critical links for each path to all the edge nodes are computed. The number of critical link for each path to all the edge nodes is shown in Table 3.8.

After computing the number of critical links in each path, the paths are sorted according to the number of critical links in ascending order. The paths arranged in ascending order according to the number of critical links are shown in Table 3.9

After sorting the path according to the number of critical links, the paths which have the same number of critical links, path 1 and 2 to node_6 have the same number of

critical links in them ($CL_i = 0$). Path 1 and 2 have to be interchanged as path 2 has a smaller path cost than path 1.

Table 3.9: Routing_Table List with paths arranged in ascending order

Destination Node	Path Number	Cost	Path	Number of Critical Links CL_i
Node_1	1	2	n0_n3_n1	0
Node_2	1	2	n0_n3_n2	0
Node_6	1	6	n0_n3_n11_n10_n9_n5_n6	0
	2	4	n0_n3_n4_n5_n6	0
	3	3	n0_n3_n5_n6	1
Node_7	1	6	n0_n3_n11_n10_n9_n5_n7	0
	2	4	n0_n3_n4_n5_n7	0
	3	3	n0_n3_n5_n7	1
Node_8	1	6	n0_n3_n11_n10_n9_n5_n8	0
	2	4	n0_n3_n4_n5_n8	0
	3	3	n0_n3_n5_n8	1

Table 3.10: Routing_Table List with paths arranged in ascending order

Destination Node	Path Number	Cost	Path	Number of Critical Links CL_i
Node_1	1	2	n0_n3_n1	0
Node_2	1	2	n0_n3_n2	0
Node_6	1	4	n0_n3_n4_n5_n6	0
	2	6	n0_n3_n11_n10_n9_n5_n6	0
	3	3	n0_n3_n5_n6	1
Node_7	1	4	n0_n3_n4_n5_n7	0
	2	6	n0_n3_n11_n10_n9_n5_n7	0
	3	3	n0_n3_n5_n7	1
Node_8	1	4	n0_n3_n4_n5_n8	0
	2	6	n0_n3_n11_n10_n9_n5_n8	0
	3	3	n0_n3_n5_n8	1

The paths with the same path cost are sorted and the sorted list is given in Table 3.10. Table 3.10 is the final routing table, where path 1 is the primary path and the rest of the paths (if present) are the alternate paths.

3.3.3.2 Forwarding Plane

In this plane the data flow request which are to be setup from the requested source to the destination nodes are handled. The input to this plane is the routing table computed in the control plane and the set of traffic request. The set of traffic request that are to be switched is given in Table 3.11. Assume that there is no traffic flowing in the network when these request come in.

Table 3.11: Set of Traffic flow to be setup

Source Node	Destination Node	Amount of traffic in Mbps
Node_0	Node_6	2
Node_1	Node_7	3
Node_2	Node_8	1

The first set of traffic flow is from node_0 to node_6 and the amount of traffic to be switched is 2Mbps. The route from node_0 to node_6 is determined from the routing table. The primary route is got and the traffic is switched thro the primary path (n0_n3_n4_n5_n6), as there is no congestion in the primary path.

The amount of traffic flowing in each link after the traffic flow is setup is determined from the linkstate database. This information is shown in Table 3.12.

Table 3.12: Amount of traffic flowing in each link after first flow is handled

Link ID	Start Node	End Node	Link bandwidth in Mbps	Amount of Traffic Flowing in Mbps
1	Node_0	Node_3	3	2
2	Node_1	Node_3	4	0
3	Node_2	Node_3	2	0
4	Node_3	Node_4	4	2
5	Node_3	Node_11	2	0
6	Node_3	Node_5	3	0
7	Node_4	Node_5	4	2
8	Node_5	Node_9	2	0
9	Node_6	Node_5	3	2
10	Node_7	Node_5	4	0
11	Node_8	Node_5	2	0
12	Node_9	Node_10	2	0
13	Node_10	Node_11	2	0

Now the second traffic flow for 3 Mbps request is to be setup. The primary route for node_7 from node_1 is determined. The traffic is tried to switch via the primary route, n1_n3_n4_n5_n7, but the link n3_n4 will become congested, as there is already a 2Mbps traffic is flowing via that link. So the next best route the alternate path1 (n1_n3_n11_n10_n9_n5_n7) is selected. The traffic flow via this route is also not possible as the link capacity from n3_n11 is only 2Mbps, so the final and alternate path2 is (n1_n3_n5_n7) is tried. The traffic is finally switched via the alternate path2, since it leads to no congestion. The amount of traffic flowing in each link is shown in Table 3.13.

Now the third and last flow request from node_2 to node_8 for 1 Mbps request is to be setup. The primary path for node_8 from node_2 is n2_n3_n4_n5_n8. See if this traffic setup will create congestion in any of the links in the primary path. No traffic congestion will happen when this traffic is routed via the primary path, so the traffic is

switched via the primary path. The amount of traffic flowing in each link after the final traffic request is handled is shown in Table 3.14.

Table 3.13: Amount of traffic flowing in each link after second flow is handled

Link ID	Start Node	End Node	Link bandwidth in Mbps	Amount of Traffic Flowing in Mbps
1	Node_0	Node_3	3	2
2	Node_1	Node_3	4	3
3	Node_2	Node_3	2	0
4	Node_3	Node_4	4	2
5	Node_3	Node_11	2	0
6	Node_3	Node_5	3	3
7	Node_4	Node_5	4	2
8	Node_5	Node_9	2	0
9	Node_6	Node_5	3	2
10	Node_7	Node_5	4	3
11	Node_8	Node_5	2	0
12	Node_9	Node_10	2	0
13	Node_10	Node_11	2	0

Table 3.14: Amount of traffic flowing in each link After third flow is handled

Link ID	Start Node	End Node	Link bandwidth in Mbps	Amount of Traffic Flowing in Mbps
1	Node_0	Node_3	3	2
2	Node_1	Node_3	4	3
3	Node_2	Node_3	2	1
4	Node_3	Node_4	4	3
5	Node_3	Node_11	2	0
6	Node_3	Node_5	3	3
7	Node_4	Node_5	4	3
8	Node_5	Node_9	2	0
9	Node_6	Node_5	3	2
10	Node_7	Node_5	4	3
11	Node_8	Node_5	2	1
12	Node_9	Node_10	2	0
13	Node_10	Node_11	2	0

CHAPTER 4

SIMULATION RESULTS

In this chapter the simulation results are given. The simulation was done on a simulator 'myNetSim', which was designed for this thesis. In the first section a brief description about the simulator is given, which is followed by the simulation results.

4.1 MyNetSim – A Network Simulator

This simulator was used to do simulation and evaluate the proposed algorithm. The code for the simulator was written in C++ and Perl. This simulator has 3 modules.

- 1) Network input module,
- 2) Routing module and
- 3) Traffic generation & flow module.

The function of each module is explained below.

4.1.1 Network Input Module

This module has 2 phases in it

- 1) Random Network Generation and
- 2) Loading Network.

The Random Network Generation phase generates a random network. In the Loading Network phase the random network generated into the linked list so that it can be used for the rest of the modules of the simulator.

4.1.1.1 Random Network Generation

In this phase random networks are generated. To generate random networks Tiers Topology Generator [17] is used. The network generated by tiers is converted to the format used in this simulator by a perl code. The perl code apart from converting the format, gives an option to randomize the networks bandwidth. The bandwidth is randomized because the tiers network uses fixed bandwidth to the links, this will lead to congestion in a few links as the fixed bandwidth used vary significantly in size (10 to 1000 Mbps).

These generated networks are taken as the input to the second phase of this module.

4.1.1.2 Loading Network

In this phase the network (Randomly generated in the previous phase) is received as an input from the file and loaded in to the list. The format in which the network is stored in the file is given below

lnodeIDledgeladdresslinterfacel

where

nodeID = A unique number to identify the node in the network

edge = Specifies whether the node is an LER or LSR (when edge is 1 the node is an LER).

address = Gives the nodes ip address

interface = this describes a list of an interface. It is further split as follows.

interface = intf1&intf2&....

where,

intf1 = describes the first interface of the node. The intf1 is further split as follows

intf1 = linkID\$destID\$cost\$capacity

where

linkID = unique number used to identify each link

destID = The node id that can be reached via the link

cost = Cost of the link

capacity = link bandwidth in kbps.

4.1.1.2.1 Example Network Format

A part of the network file is shown below. It describes the characteristics of a single node in the network.

```
1111130.130.12.1.11$4$1$4000$&|
```

From the above line the following information about the node can be inferred.

nodeID = 1

edge = 1 (meaning the node represents an LER)

IP address of node = 130.130.12.1

This node has only one interface and the details of the interface are given below

linkID = 1

destID = 4

cost = 1

capacity of link is = 4000 kbps.

The network in the above format is the input for this phase. The network is stored as a set of nodes in a linked list and each node has a linked list, describing the interface or the links that is connected to it. An example, a network having four nodes and each node having 3 interface each, that is loaded into the list is shown in Figure 4.1.

Figure 4.1: Network loaded into the linked list

This module is the base for all other modules, i.e. all other modules operate on the list that is created by this module. The node has a set of attributes that are shown in Table 4.1. The attributes describe the node characteristics.

Table 4.1: Node Attributes

S. No	Node Attributes	Description of the attributes
1.	nodeID	A unique number by which the node is identified in the node list
2.	edge	This attribute specifies whether a node is a edge node or not. (LER or LSR)
3.	nAddress	The IP address of the node.
4.	intf	Pointer to the interface list
5.	next.	Pointer to the next node

The set of attributes for an interface is shown in Table 4.2. In this simulator only point to point links can be simulated.

Table 4.2: Node Attributes

S. No	Node Attributes	Description of the attributes
1.	linkID	A Unique number identifying each and every link in the network
2.	destID	The nodeID which can be reached via that link
3.	cost	Cost of the link. It is generally one
4.	linkCapacity	Bandwidth of the link
5.	linkUtilized	The amount of traffic flowing in the link.
6.	currentTime	The time at which the utilization was last changed.
7.	next	Pointer to the next interface

So the linked list stores the complete information about a network. This list is used by the other two modules of this simulator.

4.1.2 Routing Module

In this module, routes are computed from a start node to all the destination nodes in the network. The network stored in a linked list from the network input module is taken and routes are computed. The start node for which the routes are to be computed is also given as an input.

There are three routing algorithms that can be used to compute the routes, viz. the shortest path algorithm (Dijkstra Algorithm), widest shortest path and the alternate routing algorithm. The way in which the routes are computed using the alternate path routing algorithm was explained in the previous chapter, while the shortest path and the widest shortest path were explained briefly in chapter 2.

Table 4.3. Cost ordered list attributes

S. No	Node Attributes	Description of the attributes
1.	nodeID	A destination node id.
2.	totalCost	The path cost for reaching the destination node id from the start node
3.	path	The list of paths by which a destination node can be reached.
4.	next	Pointer to next route in the routing table

Table 4.4. Routing table list attributes

S. No	Node Attributes	Description of the attributes
1.	nodeID	A destination node id.
2.	totalCost	The path cost for reaching the destination node id from the start node
3.	path	The list of paths by which a destination node can be reached.
4.	interfaceID	Identifies the interface number for each path
5.	bandwidth	Identifies the amount of bandwidth left unutilized in that path
6.	next	Pointer to next route in the routing table

A cost ordered list for all the nodes is formed in this module. The attributes of the cost ordered list is shown in Table 4.3.

The routes and the path cost to reach each node in the network is computed and stored in the cost ordered list. Each node in the cost ordered list is deleted when certain condition is satisfied. The condition depends on the routing algorithm used. Before deleting the node from the cost ordered list a routing table list is created and the values at each node (destID, path and path cost) are copied into that list. The attributes of the routing table are given in Table 4.4.

4.1.3 Traffic Generation & Flow Module

This module can be sub-divided into 2 phases, viz. the traffic generation phase and the flow module phase.

4.1.3.1 Traffic Generation Phase

In this phase traffic is generated from one edge node to another and it is stored in a file. To generate random traffic between edge nodes a perl code is used. This perl code takes the total simulation time (say simTime) and the network file as input. It generates the traffic for the network described by the network file. From the network file the edge nodes and the link capacity⁹ of the edge node are noted. Let the total number of edge nodes in the network be (NoEN =) 5 and the capacity of the edge node be CoEN_i. The traffic is set between all possible pairs of edge nodes. The amount of traffic between each node is

⁹ The edge node will have only one link connected to that, so the capacity or bandwidth of that link is also noted

Amount of traffic between edge nodes = $\text{rand}(\text{CoEN}_i / \text{NoEN})$.,

where,

$\text{rand}(\text{seed})$ is a random number generator, which generates a random between 0 to seed,
 i in CoEN_i = start node id. In other words CoEN_i represents the capacity of the start node.

The from time of the flows is

$$\text{fromTime} = \text{rand}(\text{simTime}/2)$$

While the end time of the flow is

$$\text{toTime} = \text{simTime}/2 + \text{rand}(\text{simTime}/2).$$

The set of events to be simulated is stored in a file. The model of a typical event file is shown below.

#Comments####

\$FileType FileName

lspID fromNode toNode amountOfTraffic fromTime toTime

There are three types of commands in the event files, viz. a comment line, file input and the lsp's to be setup.

a) Comment Line

A line beginning with # indicates that it is a comment line and it is not processed by the simulator.

b) File Name Input

A line beginning with a \$ indicates that it is a line to be processed and it gives information about the file type and the name of the file. The file type is a flag which tells how the file name that follows should be interpreted. Currently there is only one value for the file type and it is nodeFileName. This indicates that the input node file or the file

name where the network input data is to be taken from is given by the file name following it. This file name is the input to the module 1.

c) LSP's Details.

The details of a LSP are given by the lines beginning with numbers. They are of the following format

lspID fromNode toNode amountOfTraffic fromTime toTime [er];

where,

lspID = A unique number to identify the flow.

fromNode = this is the node id of the node from which the flow of traffic originates

toNode = this is the node id of the destination node where the traffic ends.

amountOfTraffic = this gives the amount of traffic that is set up by this flow

fromTime = the time at which this flow starts

toTime = the time at which this flow ends.

er = this is an optional parameter which gives the explicit route through which the traffic flows.

4.1.3.1.1 Example Event File

```
$nodeFileName node1.txt
1000 1 2 409 4 17;
1001 1 3 604 0 27;
1002 1 10 564 4 29;
```

The first line gives the input network file name. This file name is the input to module 1. The second, third and the fourth line describe three flows to be setup in the network. The first flow in line 2 has an id 1000. This flow begins at node 1 and ends at node 2. It

carries traffic of 409kbps. The start time of this flow is 4 seconds from the beginning of the simulation and the end time is 17 seconds.

4.1.3.2 Flow Module Phase

The input for this phase is the file name which contains the list of events to be simulated (generated in the previous phase). The network file name is got from the event file name and is given as the input to module 1.

The list of events to be simulated is loaded into a linked list (called the eventList). This list is sorted in ascending order according to the start time of the simulation. The head of the list is the first event to be simulated. This event is first simulated and it is added to another linked list called the stopList. The stopList is sorted in ascending order according to the stop time of the flow. This head of this list is first event to be stopped.

If the start time is less than the stop time then the first event in the eventList is executed, else the first flow in the stopList is stopped and that node is deleted from the list. The set of resources allocated for the flow is released when an event is stopped. When both the list is empty the simulation is completed and the program exits.

The output of this phase is a set of trace files. There are three types of trace file. The first set of trace file indicates how each flow was setup and through what route the flow was setup, etc. This type of trace file have an extension MDres or APres. The extension is MDres when the dijkstra (Shortest path) algorithm is used to switch the traffic, while the extension is APres when alternate path routing is used.

Table 4.5: Flow Result description

S.No.	Flow Result	Description of the result
1	-1	NO ROUTE EXIST
2	1	SUCCESSFUL TRANSMISSION ON NON CONGESTED LINK
3	2	PRIMARY ROUTE CONGESTED
4	21	SECONDARY EXIST
5	-21	NO SECONDARY EXIST
6	3	TRAFFIC FLOWING VIA CONGESTED PATH

These trace file give the result of each flow. A flow might get setup through the primary path, or a congested path, or no path – meaning the flow was failed to be setup etc. These results are given a numeric numbers and the meaning of these numbers is given in Table 4.5.

The second set of trace files list the critical links in the network. These files have an extension cl. The third set of trace files gives the details about the amount of link utilized in each link in the network. These trace files have an extension tr.

Using these trace files the link utilization of the network, delay at a node, number of flows setup etc., can be calculated. These result are plotted in a graph using the x-graph [17] software.

4.2 Simulation Results

In this section the simulation results of the alternate path routing algorithm is shown. These simulation results are compared with the simulation results of the shortest path (Dijkstra's) algorithm.

4.2.1 Network1

The first network that was simulated contained 12 nodes. The network is shown in Figure 4.2. This network contains 6 edge node and 6 LSRs to switch traffic between the edge nodes.

Figure 4.2: Network1

The routing table for the node1 of this network computed using shortest path algorithm (Dijkstra's algorithm) is shown in Table 4.6.

The routing table of node1 of this network computed using Alternate path routing algorithm (proposed algorithm) is shown in Table 4.7. The routing table is for $\alpha=0.1$.

Table 4.6: Routing table of node1 of Network1–Dijkstra's Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	4	p0	1	1	1_4_
2	3	p0	2	1	1_4_3_
3	2	p0	2	1	1_4_2_
4	5	p0	2	1	1_4_5_
5	6	p0	2	1	1_4_6_
6	9	p0	2	1	1_4_9_
7	7	p0	3	1	1_4_6_7_
8	8	p0	3	1	1_4_9_8_
9	10	p0	3	1	1_4_9_10_
10	11	p0	3	1	1_4_9_11_
11	12	p0	3	1	1_4_9_12_

Table 4.7: Routing table of node1 of Network1–Alternate Path Routing Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	4	p0	1	1	1_4_
2	3	p0	2	1	1_4_3_
3	2	p0	2	1	1_4_2_
4	5	p0	2	1	1_4_5_
		p1	3	1	1_4_9_5_
5	6	p0	2	1	1_4_6_
6	9	p0	2	1	1_4_9_
		p1	3	1	1_4_5_9_
7	7	p0	3	1	1_4_6_7_
		p1	4	1	1_4_9_8_7_
8	8	p0	3	1	1_4_9_8_
		p1	4	1	1_4_5_9_8_
		p2	4	1	1_4_6_7_8_
9	10	p0	3	1	1_4_9_10_
		p1	3	1	1_4_5_9_
10	11	p0	3	1	1_4_9_11_
		p1	4	1	1_4_5_9_11_
11	12	p0	3	1	1_4_9_12_
		p1	4	1	1_4_5_9_12_

From the two routing tables (Table 4.6 and Table4.7) it can be seen that the Alternate path routing algorithm gives more number of routes to the destination nodes. Since there are more number of routes to the destination nodes, there is more option of choosing a route when data traffic is switched.

There were four events being setup in this network, viz. event0, event10, event20 and event50. Each event sets up a set of flow between the edge nodes. These events are first switched using the shortest path algorithm (Dijkstra's algorithm) and then they are switched using the Alternate path routing algorithm (proposed algorithm). The simulation results of each of the events are discussed below.

4.2.1.1 Event0

Table 4.8: Network1-Event0: Set of Flows

S. No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	1	2	409	4	17
2	1001	1	3	604	0	27
3	1002	1	10	564	4	29
4	1003	1	11	513	11	21
5	1004	1	12	787	1	17
6	1005	2	3	300	0	20
7	1006	2	10	400	1	16
8	1007	2	11	310	8	24
9	1008	2	12	319	3	28
10	1009	3	10	293	7	23
11	1010	3	11	274	9	23
12	1011	3	12	245	2	24
13	1012	10	11	709	7	21
14	1013	10	12	484	11	24
15	1014	11	12	410	4	19

The set of flows that are to be switched in the network for event0 are shown in Table 4.8. These flows are first switched using the shortest path algorithm (Dijkstra’s algorithm).

4.2.1.1.1 Event0 – using Dijkstra’s Algorithm

These set of flow is first switched using the shortest path algorithm (Dijkstra’s Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.9.

Table 4.9: Network1-Event0: Set of Critical links

S.No.	From Node	To Node	Link ID	Bandwidth in kbps
1	9	12	13	2000

There is only one critical link in this network for this set of flow. The critical link has a bandwidth of 2000kbps. The path each flow has been setup in and the nature of those paths are given in Table 4.10.

All the flows are setup along the primary path. The flow result 1 means that the flow is setup via the primary path which is non-congested. The flow result 3 means that the flow is setup via the primary path that is congested. A flow result of -21 indicates that the destination node is unreachable from the source node and hence the flow could not be setup.

Event0 consisted of 15 flows in all. Out of the 15 flows 14, had a flow result of 1 and were switched via the primary non congested route. The flow (with lspID = 1013) alone,

had a flow result 3 and was setup through a congested primary route. The link that is congested is given in Table 4.9. The throughput of the critical link is shown in Figure 4.3. The throughput is plotted between simulation time and link utilization in bps.

Table 4.10: Network1-Event0: Flow Results–Using Dijkstra’s Algorithm

S.No.	Lsp ID	Flow Result	Route
1	1001	1	1_4_3_
2	1005	1	2_4_3_
3	1004	1	1_4_9_12_
4	1006	1	2_4_9_10_
5	1011	1	3_4_9_12_
6	1008	1	2_4_9_12_
7	1002	1	1_4_9_10_
8	1000	1	1_4_2_
9	1014	1	11_9_12_
10	1012	1	10_9_11_
11	1009	1	3_4_9_10_
12	1007	1	2_4_9_11_
13	1010	1	3_4_9_11_
14	1013	3	10_9_12_
15	1003	1	1_4_9_11_

The delay of the critical link is shown in Figure 4.4. The delay is plotted against the simulation time. It is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

Figure 4.3: Network1-Event0: Throughput of Link 13 – Dijkstra's Algorithm

Figure 4.4: Network1-Event0: Delay of Link 13 – Dijkstra's Algorithm

4.2.1.1.2 Event0 – using Alternate Path Routing Algorithm

The traffic flows of event0 (described in Table 4.8) is now switched using the Alternate path routing algorithm. The result of the event0 is summarized in Table 11.

The flows, which have a flow result 1, are transmitted via the non congested path successfully. For the flows which have a flow result 2 (meaning that the path in which the flow is currently setup is congested) an alternate route is looked up and if an alternate path exist the flow is setup via that path. The flows, which have a flow result of -21 can not be transmitted via any route, since there is no further route to the destination which is non congested.

Event0 consisted of 15 flows in all. Out of the 15 flows 14 of them, have a flow result 1 and were switched via the primary non congested route. For the flow 1013, the first flow result was 2, which meant that the primary path was congested and the flow can not be transmitted via that route. As a result an alternate route is looked up for in the routing Table, for the flow 1013, from node 10 to 12. But there is no alternate route from node 10 to 12 (second flow result for flow 1013 was -21). Hence this event is not set up and the flow request for 1013 is rejected. If the flow 1013 had been allowed it would have led to congestion in link 13. Congestion would have led to the performance degradation of other flows in the network, which are flowing via the link 13.

The throughput of link 13 is shown in Figure 4.5. The throughput is plotted against simulation time. The spike in the graph (at time = 11sec) corresponds to the flow 1013. The flow 1013 is initially been admitted and then the congestion bit is set, as soon as the congestion in the link is noted the flow is withdrawn by the edge node and this results in a spike. The throughput of the network for this set of flow is lower when compared with

the dijkstra’s algorithm, but the flow that is switched is done efficiently, without any performance degradation.

Table 4.11: Network1-Event0: Flow Results–Using Alternate Path Routing Algorithm

S.No.	Lsp ID	Flow Result	Route
1	1001	1	1_4_3_
2	1005	1	2_4_3_
3	1004	1	1_4_9_12_
4	1006	1	2_4_9_10_
5	1011	1	3_4_9_12_
6	1008	1	2_4_9_12_
7	1002	1	1_4_9_10_
8	1000	1	1_4_2_
9	1014	1	11_9_12_
10	1012	1	10_9_11_
11	1009	1	3_4_9_10_
12	1007	1	2_4_9_11_
13	1010	1	3_4_9_11_
14	1013	2	10_9_12_
15	1013	-21	NO_ROUTE
16	1003	1	1_4_9_11_

The philosophy of the alternating path routing algorithm is not to allow any flow that will lead to congestion and eventually degrade the performance of the flow and the network. This is one of the important aspect of the alternate path routing algorithm. It can be seen from the results in Table 4.11 that there is flow that is flowing through the congested path. So the delay in buffer for the flows is zero, as there is no congestion and no packet loss.

Figure 4.5: Network1 – Event0: Throughput of link 13 – Alternate Path Routing

Algorithm

4.2.1.2 Event10

Table 4.12: Network1-Event10: Set of Flows

S.No.	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	1	2	755	9	26
2	1001	1	3	739	7	20
3	1002	1	10	554	11	28
4	1003	1	11	861	12	21
5	1004	1	12	670	0	29
6	1005	2	3	578	5	22
7	1006	2	10	443	2	29
8	1007	2	11	574	3	27
9	1008	2	12	423	10	24
10	1009	3	10	387	7	19
11	1010	3	11	286	5	18
12	1011	3	12	327	5	22
13	1012	10	11	685	8	16
14	1013	10	12	599	3	26
15	1014	11	12	353	4	24

The set of flows that are to be switched in the network for event10 are shown in Table 4.12. These flows are first switched using the shortest path algorithm (Dijkstra’s algorithm).

4.2.1.2.1 Event10 – using Dijkstra’s Algorithm

The set of flow given in Table 4.12 is first switched using the shortest path algorithm (Dijkstra’s Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.13.

Table 4.13: Network1-Event10: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	3	4	3	2000
2	4	9	6	4000
3	9	12	13	2000

There are three critical links identified in this network for this set of flow. The critical links along with their characteristics are shown in Table 4.13. The path each flow has been setup in and the nature of those paths are given in Table 4.14.

Event10 consisted of 15 flows in all. Out of the 15 flows 12 of them, had a flow result 1 and were switched via the primary non congested route. The flows (with lspID = 1009, 1008 and 1003), had a flow result 3 and were setup through the primary route, which was congested. The links that are congested is given in Table 4.13.

Table 4.14: Network1-Event10: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1004	1	1_4_9_12_
2	1006	1	2_4_9_10_
3	1007	1	2_4_9_11_
4	1013	1	10_9_12_
5	1014	1	11_9_12_
6	1011	1	3_4_9_12_
7	1005	1	2_4_3_
8	1010	1	3_4_9_11_
9	1001	1	1_4_3_
10	1009	3	3_4_9_10_
11	1012	1	10_9_11_
12	1000	1	1_4_2_
13	1008	3	2_4_9_12_
14	1002	1	1_4_9_10_
15	1003	3	1_4_9_11_

Figure 4.6: Network1-Event10: Throughput of Link 13 – Dijkstra’s Algorithm

The throughput of the critical links is shown in Figure 4.6, 4.7 and 4.8. The throughput is plotted between simulation time and link utilization in bps.

Figure 4.7: Network1-Event10: Throughput of Link 6 – Dijkstra's Algorithm

Figure 4.8: Network1-Event10: Throughput of Link 3 – Dijkstra's Algorithm

The delay of the critical links is shown in Figure 4.9, 4.10 and 4.11. The delay is plotted against the simulation time.

Figure 4.9: Network1-Event10: Delay of Link 13 – Dijkstra's Algorithm

Figure 4.10: Network1-Event10: Delay of Link 6 – Dijkstra's Algorithm

Figure 4.11: Network1–Event10: Delay of Link 3 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.1.2.2 Event10 – using Alternate Path Routing Algorithm

The traffic flows of event10 (described in Table 4.12) is now switched using the Alternate path routing algorithm. The result of the event10 is summarized in Table 15.

Event10 consisted of 15 flows in all. Out of the 15 flows 12 of them, had a flow result 1 and were switched via the primary non congested route. The flows 1009, 1008 and 1003 had a flow result 2 and could not be transmitted via the primary route, since this flow congests one of the links in the network. As a result an alternate route is looked up for in the routing table from node 3 to10, 2 to 12 and 1 to 11. The first alternate route is

also congested for flows 1008 and 1009 (corresponding flow result was 2) and there is no alternate route from node 3 to 10 and 2 to 12. Hence this event is not set up and the flow request for 1009 and 1008 were rejected. If the flow 1013 had been allowed it would have led to congestion in link 13. Congestion would have led to the performance degradation of other flows in the network, which are flowing via the link 13. The flow 1003 had a non-congested alternate path and the flow was setup via the first alternate path.

Table 4.15: Network1-Event10: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1004	1	1_4_5_9_12_
2	1006	1	2_4_5_9_10_
3	1007	1	2_4_5_9_11_
4	1013	1	10_9_12_
5	1014	1	11_9_12_
6	1011	1	3_4_5_9_12_
7	1005	1	2_4_3_
8	1010	1	3_4_5_9_11_
9	1001	1	1_4_3_
10	1009	2	3_4_5_9_10_
11	1009	21	3_4_9_10_
12	1009	2	3_4_9_10_
13	1009	-21	NO_ROUTE
14	1012	1	10_9_11_
15	1000	1	1_4_2_
16	1008	2	2_4_5_9_12_
17	1008	21	2_4_9_12_
18	1008	2	2_4_9_12_
19	1008	-21	NO_ROUTE
20	1002	1	1_4_5_9_10_
21	1003	2	1_4_5_9_11_
22	1003	21	1_4_9_11_
23	1003	1	1_4_9_11_

Figure 4.12: Network1-Event10: Throughput of link 13-Alternate Path Routing

Figure 4.13: Network1-Event10: Throughput of link 6-Alternate Path Routing

The throughput of critical links is shown in Figure 4.12, 4.13 and 4.14. The throughput is plotted against simulation time.

Figure 4.14: Network1-Event10: Throughput of link 3-Alternate Path Routing

Figure 4.15: Network1-Event10: Throughput of link 5-Alternate Path Routing

Traffic flows via the links 5 and 8, instead of the link 6. The throughput of those links is shown in Figure 4.15 and 4.16.

Figure 4.16: Network1-Event10: Throughput of link 8-Alternate Path Routing

The amount of traffic flowing via the non-congested link is increased for the Alternate Path routing algorithm. The traffic flow 1003 in the dijkstra's algorithm was responsible for congesting the link 6. The same flow here creates congestion in the primary path 1_4_5_9_11_, congesting the links 5 and 8. So the flow is now being switched via the alternate path1 (1_4_9_11_), which remains non-congested. Thus the amount of traffic flowing via the non-congested link is increased at the expense of traversing an additional hop.

4.2.1.3 Event20

The set of flows that are to be switched in the network for event20 are shown in Table 4.16. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.16: Network1-Event20: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	1	2	610	1	17
2	1001	1	3	724	3	16
3	1002	1	10	643	9	22
4	1003	1	11	894	1	20
5	1004	1	12	590	5	18
6	1005	2	3	505	13	17
7	1006	2	10	451	5	27
8	1007	2	11	560	3	26
9	1008	2	12	427	1	26
10	1009	3	10	438	1	21
11	1010	3	11	389	2	16
12	1011	3	12	257	4	28
13	1012	10	11	638	13	28
14	1013	10	12	499	14	20
15	1014	11	12	658	10	15

4.2.1.3.1 Event20 – using Dijkstra's Algorithm

Table 4.17: Network1-Event20: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	4	9	6	4000
2	9	12	13	2000
3	4	3	3	2000
4	9	11	12	3000

The set of flow given in Table 4.16 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.17.

There are four critical links identified in this network. The critical links along with their characteristics are shown in Table 4.17. The path each flow has been setup in and the nature of those paths are given in Table 4.18.

Table 4.18: Network1-Event20: Flow Results–Using Dijkstra's Algorithm

S.No	Lsp ID	Flow Result	Route
1	1000	1	1_4_2_
2	1003	1	1_4_9_11_
3	1008	1	2_4_9_12_
4	1009	1	3_4_9_10_
5	1010	1	3_4_9_11_
6	1007	1	2_4_9_11_
7	1001	1	1_4_3_
8	1011	1	3_4_9_12_
9	1006	1	2_4_9_10_
10	1004	3	1_4_9_12_
11	1002	3	1_4_9_10_
12	1014	1	11_9_12_
13	1012	3	10_9_11_
14	1005	3	2_4_3_
15	1013	3	10_9_12_

Event20 consisted of 15 flows in all. Out of the 15 flows 10 of them, had a flow result of 1 and were switched via the primary non congested route. The flows (with lspID = 1004, 1002, 1012, 1005 and 1013), had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.17.

Figure 4.17: Network1–Event20: Throughput of Link 13 – Dijkstra’s Algorithm

Figure 4.18: Network1–Event20: Throughput of Link 12 – Dijkstra’s Algorithm

The throughput of the critical links is shown in Figure 4.17, 4.18, 4.19 and 4.20. The throughput is plotted between simulation time and link utilization in bps.

Figure 4.19: Network1-Event20: Throughput of Link 6 – Dijkstra's Algorithm

Figure 4.20: Network1-Event20: Throughput of Link 3 – Dijkstra's Algorithm

The delay of the critical links is shown in Figure 4.21, 4.22, 4.23, and 4.24. The delay is plotted against the simulation time.

Figure 4.21: Network1–Event20: Delay of Link 13 – Dijkstra’s Algorithm

Figure 4.22: Network1–Event20: Delay of Link 12 – Dijkstra’s Algorithm

Figure 4.23: Network1-Event20: Delay of Link 6 – Dijkstra's Algorithm

Figure 4.24: Network1-Event20: Delay of Link 3 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.1.3.2 Event20 – using Alternate Path Routing Algorithm

Table 4.19: Network1-Event20: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1000	1	1_4_2_
2	1003	1	1_4_5_9_11_
3	1008	1	2_4_5_9_12_
4	1009	1	3_4_5_9_10_
5	1010	1	3_4_5_9_11_
6	1007	1	2_4_5_9_11_
7	1001	1	1_4_3_
8	1011	1	3_4_5_9_12_
9	1006	2	2_4_5_9_10_
10	1006	21	2_4_9_10_
11	1006	1	2_4_9_10_
12	1004	2	1_4_5_9_12_
13	1004	21	1_4_9_12_
14	1004	1	1_4_9_12_
15	1002	2	1_4_5_9_10_
16	1002	21	1_4_9_10_
17	1002	1	1_4_9_10_
18	1014	1	11_9_12_
19	1012	2	10_9_11_
20	1012	-21	NO_ROUTE
21	1005	2	2_4_3_
22	1005	-21	NO_ROUTE
23	1013	2	10_9_12_
24	1013	-21	NO_ROUTE

The traffic flows of event20 (described in Table 4.16) is now switched using the Alternate path routing algorithm. The result of the event20 is summarized in Table 4.19. Event20 consisted of 15 flows in all. Out of the 15 flows 9 of them, had a flow result 1 and were switched via the primary non congested route. The flows 1013, 1012, 1005, 1006, 1004 and 1002 had a flow result 2 and could not be transmitted via the primary route, since this flow congests the one of the links in the network. As a result an alternate route is looked up for those three flows. There is no alternate route for the flows 1013, 1012 and 1005 (flow result -21). Hence these event are not set up and the flow request for are rejected. For the flows 1002, 1004 and 1006 there exist and non-congested alternate path, and that flow is setup via that path.

The throughput of critical links is shown in Figure 4.25, 4.26, 4.27 and 4.28. The throughput is plotted against simulation time.

Figure 4.25: Network1-Event20: Throughput of link 13-Alternate Path Routing

Figure 4.26: Network1-Event20: Throughput of link 12-Alternate Path Routing

Figure 4.27: Network1-Event20: Throughput of link 6-Alternate Path Routing

Figure 4.28: Network1-Event20: Throughput of link 3-Alternate Path Routing

Traffic flows via the links 5 and 8, instead of the link 6. The throughput of those links is shown in Figure 4.29 and 4.30.

Figure 4.29: Network1-Event20: Throughput of link 5-Alternate Path Routing

Figure 4.30: Network1–Event20: Throughput of link 8–Alternate Path Routing

The amount of traffic flowing via the non-congested link is increased for the Alternate Path routing algorithm. The traffic flows 1006 and 1002 in the dijkstra’s algorithm was responsible for congesting the one of the links in the network. So the flow is now being switched via the alternate path, which remains non-congested. Thus the performance of the network is increased.

4.2.1.4 Event50

The set of flows that are to be switched in the network for event50 are shown in Table 4.20. These flows are first switched using the shortest path algorithm (Dijkstra’s algorithm).

Table 4.20: Network1-Event50: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	1	2	1070	6	16
2	1001	1	3	1015	10	29
3	1002	1	10	1164	8	16
4	1003	1	11	952	1	23
5	1004	1	12	1062	3	27
6	1005	2	3	662	0	19
7	1006	2	10	542	4	19
8	1007	2	11	856	4	21
9	1008	2	12	586	14	22
10	1009	3	10	483	9	17
11	1010	3	11	401	13	25
12	1011	3	12	499	14	24
13	1012	10	11	1092	3	17
14	1013	10	12	777	13	21
15	1014	11	12	686	13	26

4.2.1.4.1 Event50 – using Dijkstra’s Algorithm

The set of flow given in Table 4.20 is first switched using the shortest path algorithm (Dijkstra’s Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.21.

Table 4.21: Network1-Event50: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	3	4	13	2000
2	4	9	1	4000
3	9	10	2	3000
4	1	4	3	2000
5	9	11	11	4000
6	9	12	6	4000
7	2	4	12	3000

There are seven critical links identified in this network. The critical links along with their characteristics are shown in Table 4.21. The path each flow has been setup in and the nature of those paths are given in Table 4.22.

Table 4.22: Network1-Event50: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1005	1	2_4_3_
2	1003	1	1_4_9_11_
3	1004	1	1_4_9_12_
4	1012	1	10_9_11_
5	1007	1	2_4_9_11_
6	1006	1	2_4_9_10_
7	1000	3	1_4_2_
8	1002	3	1_4_9_10_
9	1009	3	3_4_9_10_
10	1001	3	1_4_3_
11	1010	3	3_4_9_11_
12	1013	3	10_9_12_
13	1014	3	11_9_12_
14	1011	3	3_4_9_12_
15	1008	3	2_4_9_12_

Event50 consisted of 15 flows in all. Out of the 15 flows 6 of them, had a flow result 1 and were switched via the primary non congested route. The flows, had a flow result 3 (with lspID = 1000, 1002, 1009, 1001, 1010, 1014, 1011, 1008 and 1013) and were setup through a congested route, the links that are congested is given in Table 4.21.

The throughput of the critical links is shown in Figures 4.31 to 4.37. The throughput is plotted against simulation time.

Figure 4.31: Network1-Event50: Throughput of Link 13 – Dijkstra's Algorithm

Figure 4.32: Network1-Event50: Throughput of Link 12 – Dijkstra's Algorithm

Figure 4.33: Network1–Event50: Throughput of Link 11 – Dijkstra’s Algorithm

Figure 4.34: Network1–Event50: Throughput of Link 6 – Dijkstra’s Algorithm

Figure 4.35: Network1-Event50: Throughput of Link 3 – Dijkstra's Algorithm

Figure 4.36: Network1-Event50: Throughput of Link 2 – Dijkstra's Algorithm

Figure 4.37: Network1-Event50: Throughput of Link 1 – Dijkstra's Algorithm

The delay of the critical links is shown in Figures 4.38 to 4.44. The delay is plotted against the simulation time.

Figure 4.38: Network1-Event50: Delay of Link 13 – Dijkstra's Algorithm

Figure 4.39: Network1-Event50: Delay of Link 12 – Dijkstra's Algorithm

Figure 4.40: Network1-Event50: Delay of Link 11 – Dijkstra's Algorithm

Figure 4.41: Network1-Event50: Delay of Link 6 – Dijkstra's Algorithm

Figure 4.42: Network1-Event50: Delay of Link 3 – Dijkstra's Algorithm

Figure 4.43: Network1-Event50: Delay of Link 2 – Dijkstra's Algorithm

Figure 4.44: Network1-Event50: Delay of Link 1 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.1.4.2 Event50 – using Alternate Path Routing Algorithm

The traffic flows of event50 (described in Table 4.20) is now switched using the Alternate path routing algorithm. The result of the event50 is summarized in Table 4.23.

Table 4.23. Network1-Event50: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1005	1	2_4_3_
2	1003	1	1_4_5_9_11_
3	1004	1	1_4_5_9_12_
4	1012	1	10_9_11_
5	1007	1	2_4_5_9_11_
6	1006	2	2_4_5_9_10_
7	1006	21	2_4_9_10_
8	1006	1	2_4_9_10_
9	1000	2	1_4_2_
10	1000	-21	NO_ROUTE
11	1002	2	1_4_5_9_10_
12	1002	21	1_4_9_10_
13	1002	1	1_4_9_10_
14	1009	2	3_4_5_9_10_
15	1009	21	3_4_9_10_
16	1009	1	3_4_9_10_
17	1001	2	1_4_3_
18	1001	-21	NO_ROUTE
19	1010	2	3_4_5_9_11_
20	1010	21	3_4_9_11_
21	1010	2	3_4_9_11_
22	1010	-21	NO_ROUTE
23	1013	2	10_9_12_
24	1013	-21	NO_ROUTE
25	1014	2	11_9_12_
26	1014	-21	NO_ROUTE
27	1011	2	3_4_5_9_12_
28	1011	21	3_4_9_12_
29	1011	1	3_4_9_12_
30	1008	2	2_4_5_9_12_
31	1008	21	2_4_9_12_
32	1008	2	2_4_9_12_
33	1008	-21	NO_ROUTE

Event50 consisted of 15 flows in all. Out of the 15 flows 5 of them, had a flow result 1 and were switched via the primary non congested route. The flows 1006, 1002, 1009, 1011, 1000, 1001, 1010, 1013, 1014 and 1008, had a flow result 2 and could not be transmitted via the primary route. An alternate route is looked up for, for those flows in the routing table. The flows 1006, 1002, 1009 and 1011 had a flow result 1, and were setup via the non congested alternate route. The other set of flows had a flow result -21, indicating that there was no alternate route or all the alternate routes, which exist was congested. Hence those flows were not set up. If those flows had been allowed it would have led to congestion in one of the network links. Congestion would have led to the performance degradation of other flows in the network, which are flowing via those links.

The throughput of critical links is shown in Figures 4.45 to 4.51. The throughput is plotted against simulation time.

Figure 4.45: Network1–Event50: Throughput of link 13–Alternate Path Routing

Figure 4.46: Network1–Event50: Throughput of link 12–Alternate Path Routing

Figure 4.47: Network1–Event50: Throughput of link 11–Alternate Path Routing

Figure 4.48: Network1-Event50: Throughput of link 6-Alternate Path Routing

Figure 4.49: Network1-Event50: Throughput of link 3-Alternate Path Routing

Figure 4.50: Network1–Event50: Throughput of link 2–Alternate Path Routing

Figure 4.51: Network1–Event50: Throughput of link 1–Alternate Path Routing

Traffic flows via the links 5 and 8, instead of the link 6. The throughput of those links is shown in Figure 4.52 and 4.53.

Figure 4.52: Network1-Event50: Throughput of link 5-Alternate Path Routing

Figure 4.53: Network1-Event50: Throughput of link 8-Alternate Path Routing

The traffic flows 1009, 1011 and 1012 in the dijkstra's algorithm was responsible for congesting the one of the links in the network. So the flow is now being switched via the alternate path, which remains non-congested. So the amount of traffic flowing via the non-congested link is increased for the Alternate Path routing algorithm.

4.2.2 Network2

The second network that was simulated contained 30 nodes. This network was generated randomly by the random network generator tiers [17]. The network is shown in Figure 4.54. This network contains 5 edge node and 25 LSRs to switch traffic between the edge nodes.

Figure 4.54: Network2

The routing table for the node1 of this network computed using shortest path algorithm (Dijkstra's algorithm) is shown in Table 4.24.

Table 4.24: Routing Table of Node1 of Network2 –Dijkstra's Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	2	p0	4	1	1_8_4_3_2_
2	3	p0	3	1	1_8_4_3_
3	4	p0	2	1	1_8_4_
4	5	p0	4	1	1_8_4_7_5_
5	6	p0	6	1	1_8_4_3_2_9_6_
6	7	p0	3	1	1_8_4_7_
7	8	p0	1	1	1_8_
8	9	p0	5	1	1_8_4_3_2_9_
9	10	p0	1	2	1_10_
10	11	p0	1	3	1_11_
11	12	p0	3	3	1_11_20_12_
12	13	p0	2	3	1_11_13_
13	14	p0	2	1	1_8_14_
14	15	p0	3	1	1_8_14_15_
15	16	p0	3	3	1_11_20_16_
16	17	p0	2	3	1_11_17_
17	18	p0	3	1	1_8_14_18_
18	19	p0	3	1	1_8_14_19_
		p1	3	3	1_11_17_19_
19	20	p0	2	3	1_11_20_
20	21	p0	2	1	1_8_21_
21	22	p0	5	1	1_8_21_24_30_22_
22	23	p0	3	1	1_8_21_23_
23	24	p0	3	1	1_8_21_24_
24	25	p0	3	1	1_8_21_25_
		p1	3	4	1_29_28_25_
25	26	p0	3	4	1_29_28_26_
26	27	p0	4	4	1_29_28_26_27_
27	28	p0	2	4	1_29_28_
28	29	p0	1	4	1_29_
29	30	p0	4	1	1_8_21_24_30_

The routing table of node1 of this network computed using Alternate path routing algorithm (proposed algorithm) is shown in Table 4.25. The routing table is for $\alpha=0.1$.

From the two routing tables (Table 4.24 and Table4.25) it can be seen that the Alternate path routing algorithm gives more number of routes to the destination nodes. Since there is more number of routes to the destination nodes, there is more option of choosing a route when data traffic is switched.

There were four events being setup in this network, viz. event0, event10, event20 and event50. Each event sets up a set of flow between the edge nodes. These events are first switched using the shortest path algorithm (Dijkstra's algorithm) and then they are switched using the Alternate path routing algorithm (proposed algorithm). The simulation results of each of the events are discussed below.

Table 4.25: Routing Table of Node1 of Network2 –Alternate Path Routing Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	2	p0	4	1	1_8_4_3_2_
		p1	5	2	1_10_8_4_3_2_
		p2	5	1	1_8_4_7_5_2_
2	3	p0	3	1	1_8_4_3_
		p1	4	2	1_10_8_4_3_
3	4	p0	2	1	1_8_4_
		p1	3	2	1_10_8_4_
4	5	p0	4	1	1_8_4_7_5_
		p1	5	2	1_10_8_4_7_5_
		p2	5	1	1_8_4_3_2_5_
5	6	p0	6	1	1_8_4_3_2_9_6_
		p1	7	2	1_10_8_4_3_2_9_6_
		p2	7	1	1_8_4_7_5_2_9_6_
6	7	p0	3	1	1_8_4_7_
		p1	4	2	1_10_8_4_7_
7	8	p0	1	1	1_8_
		p1	2	2	1_10_8_
8	9	p0	5	1	1_8_4_3_2_9_
		p1	6	2	1_10_8_4_3_2_9_
		p2	6	1	1_8_4_7_5_2_9_
9	10	p0	1	2	1_10_
		p1	2	1	1_8_10_
10	11	p0	1	3	1_11_
11	12	p0	3	3	1_11_20_12_
		p1	4	3	1_11_20_16_12_
12	13	p0	2	3	1_11_13_
13	14	p0	2	1	1_8_14_
		p1	3	2	1_10_8_14_
14	15	p0	3	1	1_8_14_15_
		p1	4	2	1_10_8_14_15_
		p2	4	3	1_11_17_19_15_
15	16	p0	3	3	1_11_20_16_
		p1	4	1	1_8_14_18_16_
		p2	4	3	1_11_20_12_16_
16	17	p0	2	3	1_11_17_
17	18	p0	3	1	1_8_14_18_
		p1	4	2	1_10_8_14_18_
		p2	4	3	1_11_20_16_18_
18	19	p0	3	3	1_11_17_19_
		p1	4	1	1_8_14_15_19_

19	20	p0	2	3	1_11_20_
20	21	p0	2	1	1_8_21_
		p1	3	2	1_10_8_21_
21	22	p0	5	1	1_8_21_24_30_22_
		p1	6	2	1_10_8_21_24_30_22_
22	23	p0	3	1	1_8_21_23_
		p1	4	2	1_10_8_21_23_
23	24	p0	3	1	1_8_21_24_
		p1	4	2	1_10_8_21_24_
24	25	p0	3	4	1_29_28_25_
25	26	p0	3	4	1_29_28_26_
26	27	p0	4	4	1_29_28_26_27_
27	28	p0	2	4	1_29_28_
28	29	p0	1	4	1_29_
29	30	p0	4	1	1_8_21_24_30_
		p1	5	2	1_10_8_21_24_30_

4.2.2.1 Event0

The set of flows that are to be switched in the network for event0 are shown in Table 4.26. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.26: Network2-Event0: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	22	23	1317	0	26
2	1001	22	2	1356	8	19
3	1002	22	25	1048	9	18
4	1003	22	18	1066	13	19
5	1004	23	2	1594	7	27
6	1005	23	25	1314	5	26
7	1006	23	18	1185	7	21
8	1007	2	25	1050	11	28
9	1008	2	18	1018	14	22
10	1009	25	18	1045	11	20

4.2.2.1.1 Event0 – using Dijkstra's Algorithm

The set of flow given in Table 4.26 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.27.

There are eight critical links identified in this network. The critical links along with their characteristics are shown in Table 4.27. The path each flow has been setup in and the nature of those paths are given in Table 4.28.

Table 4.27: Network2-Event0: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	1	5	2	6000
2	1	6	3	6000
3	5	7	11	7000
4	9	3	6	6000
5	9	6	12	5000
6	26	27	34	5000
7	26	29	35	5000
8	29	28	36	6000

Event0 consisted of 10 flows in all. Out of the 10 flows 4 of them, had a flow result 1 and were switched via the primary non congested route. The other flows (with lspID = 1001, 1002, 1003, 1006, 1007 and 1009), had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.27.

Table 4.28: Network2-Event0: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1000	1	22_26_27_24_30_23_
2	1005	1	23_30_24_27_26_29_28_21_25_
3	1004	1	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
4	1006	3	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
5	1001	3	22_26_29_28_7_5_1_6_9_3_8_2_
6	1002	3	22_26_29_28_21_25_
7	1007	3	2_8_3_9_6_1_5_7_28_21_25_
8	1009	3	25_21_28_7_5_1_6_9_3_20_11_18_
9	1003	3	22_26_29_28_7_5_1_6_9_3_20_11_18_
10	1008	1	2_8_3_20_11_18_

The throughput of few of the critical links is shown in Figures 4.55 and 4.56. The throughput is plotted against simulation time.

Figure 4.55: Network2-Event0: Throughput of Link 2 – Dijkstra's Algorithm

Figure 4.56: Network2-Event0: Throughput of Link 3 – Dijkstra's Algorithm

The delay of the critical links is shown in Figures 4.57 and 4.58. The delay is plotted against the simulation time.

Figure 4.57: Network2-Event0: Delay of Link 3 – Dijkstra's Algorithm

Figure 4.58: Network2-Event0: Delay of Link 2 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.2.1.2 Event0 – using Alternate Path Routing Algorithm

Table 4.29: Network2-Event0: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1000	1	22_26_27_24_30_23_
2	1005	1	23_30_24_27_26_29_28_21_25_
3	1004	1	23_30_24_27_26_29_28_7_5_4_10_6_9_3_8_2_
4	1006	2	23_30_24_27_26_29_28_7_5_4_10_6_9_3_20_11_18_
5	1006	21	23_30_24_27_26_29_28_7_5_4_1_6_9_3_20_11_18_
6	1006	2	23_30_24_27_26_29_28_7_5_4_1_6_9_3_20_11_18_
7	1006	21	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
8	1006	2	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
9	1006	-21	NO_ROUTE
10	1001	1	22_26_29_28_7_5_4_10_6_9_3_8_2_
11	1002	2	22_26_29_28_21_25_
12	1002	-21	NO_ROUTE
13	1007	1	2_8_3_9_6_4_1_5_7_28_21_25_
14	1009	2	25_21_28_7_5_4_10_6_9_3_20_11_18_
15	1009	21	25_21_28_7_5_4_1_6_9_3_20_11_18_
16	1009	2	25_21_28_7_5_4_1_6_9_3_20_11_18_
17	1009	21	25_21_28_7_5_1_6_9_3_20_11_18_
18	1009	2	25_21_28_7_5_1_6_9_3_20_11_18_
19	1009	-21	NO_ROUTE
20	1003	2	22_26_29_28_7_5_4_10_6_9_3_20_11_18_
21	1003	21	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
22	1003	2	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
23	1003	21	22_26_29_28_7_5_1_6_9_3_20_11_18_
24	1003	2	22_26_29_28_7_5_1_6_9_3_20_11_18_
25	1003	-21	NO_ROUTE
26	1008	1	2_8_3_20_11_18_

The traffic flows of event0 (described in Table 4.26) is now switched using the Alternate path routing algorithm. The result of the event0 is summarized in Table 4.29.

Event0 consisted of 10 flows in all. Out of the 10 flows 5 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1007 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. The reason for that is the primary path in the alternate routing algorithm has more capacity than the shortest path. Though the traffic is carried via an extra hop the amount of traffic flowing through the non congested links in the network is increased

The remaining flows are not transmitted via the primary path as it is congested. Further there is no alternate paths that is non congested, so those flows are not transmitted to prevent congestion in the network links.

Figure 4.59: Network2–Event0: Throughput of link 3–Alternate Path Routing

The throughput of few critical links is shown in Figures 4.59 and 4.60. The throughput is plotted against simulation time.

Figure 4.60: Network2-Event0: Throughput of link 2-Alternate Path Routing

Figure 4.61: Network2-Event0: Throughput of link 8-Alternate Path Routing

Traffic flows via the links 8, 10 and 13, instead of the links 2 and 3. The throughput of those links is shown in Figure 4.61, 4.62 and 4.63.

Figure 4.62: Network2-Event0: Throughput of link 10-Alternate Path Routing

Figure 4.63: Network2-Event0: Throughput of link 13-Alternate Path Routing

4.2.2.2 Event10

The set of flows that are to be switched in the network for event10 are shown in Table 4.30. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.30: Network2-Event10: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	22	23	1626	8	15
2	1001	22	2	1566	11	18
3	1002	22	25	1270	14	29
4	1003	22	18	1314	7	18
5	1004	23	2	1321	0	15
6	1005	23	25	1251	7	24
7	1006	23	18	1859	7	22
8	1007	2	25	1136	8	28
9	1008	2	18	1698	0	26
10	1009	25	18	1845	8	19

4.2.2.2.1 Event10 – using Dijkstra's Algorithm

The set of flow given in Table 4.30 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.31.

There are 13 critical links identified in this network. The critical links along with their characteristics are shown in Table 4.31. The path each flow has been setup in and the nature of those paths are given in Table 4.32.

Table 4.31: Network2-Event10: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	1	5	2	6000
2	1	6	3	6000
3	5	7	11	7000
4	9	3	6	6000
5	9	6	12	5000
6	26	27	34	5000
7	26	29	35	5000
8	29	28	36	6000
9	21	28	29	5000
10	3	20	7	6000
11	24	27	32	6000
12	11	20	16	6000
13	25	21	28	5000

Table 4.32: Network2-Event10: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1004	1	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
2	1008	1	2_8_3_20_11_18_
3	1005	1	23_30_24_27_26_29_28_21_25_
4	1006	1	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
5	1003	3	22_26_29_28_7_5_1_6_9_3_20_11_18_
6	1007	3	2_8_3_9_6_1_5_7_28_21_25_
7	1000	3	22_26_27_24_30_23_
8	1009	3	25_21_28_7_5_1_6_9_3_20_11_18_
9	1001	3	22_26_29_28_7_5_1_6_9_3_8_2_
10	1002	3	22_26_29_28_21_25_.

Event10 consisted of 10 flows in all. Out of the 10 flows 4 of them, had a flow result 1 and were switched via the primary non congested route. The other flows (with lspID =

1001, 1002, 1003, 1000, 1007 and 1009), had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.31.

Figure 4.64: Network2–Event10: Throughput of Link 2 – Dijkstra's Algorithm

Figure 4.65: Network2–Event10: Throughput of Link 3 – Dijkstra's Algorithm

The throughput of few critical links is shown in Figures 4.64 and 4.65. The throughput is plotted against simulation time.

Figure 4.66: Network2-Event10: Delay of Link 3 – Dijkstra's Algorithm

Figure 4.67: Network2-Event10: Delay of Link 2 – Dijkstra's Algorithm

The delay of the critical links is shown in Figures 4.66 and 4.67. The delay is plotted against the simulation time. The delay is calculated under the assumption that there is

infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.2.2 Event10 – using Alternate Path Routing Algorithm

Table 4.33: Network2-Event10: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1004	1	23_30_24_27_26_29_28_7_5_4_10_6_9_3_8_2_
2	1008	1	2_8_3_20_11_18_
3	1005	1	23_30_24_27_26_29_28_21_25_
4	1006	1	23_30_24_27_26_29_28_7_5_4_10_6_9_3_20_11_18_
5	1003	2	22_26_29_28_7_5_4_10_6_9_3_20_11_18_
6	1003	21	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
7	1003	2	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
8	1003	21	22_26_29_28_7_5_1_6_9_3_20_11_18_
9	1003	2	22_26_29_28_7_5_1_6_9_3_20_11_18_
10	1003	-21	NO_ROUTE
11	1007	1	2_8_3_9_6_4_1_5_7_28_21_25_
12	1000	2	22_26_27_24_30_23_
13	1000	-21	NO_ROUTE
14	1009	2	25_21_28_7_5_4_10_6_9_3_20_11_18_
15	1009	21	25_21_28_7_5_4_1_6_9_3_20_11_18_
16	1009	2	25_21_28_7_5_4_1_6_9_3_20_11_18_
17	1009	21	25_21_28_7_5_1_6_9_3_20_11_18_
18	1009	2	25_21_28_7_5_1_6_9_3_20_11_18_
19	1009	-21	NO_ROUTE
20	1001	2	22_26_29_28_7_5_4_10_6_9_3_8_2_
21	1001	21	22_26_29_28_7_5_4_1_6_9_3_8_2_
22	1001	2	22_26_29_28_7_5_4_1_6_9_3_8_2_
23	1001	21	22_26_29_28_7_5_1_6_9_3_8_2_
24	1001	2	22_26_29_28_7_5_1_6_9_3_8_2_
25	1001	-21	NO_ROUTE
26	1002	2	22_26_29_28_21_25_
27	1002	-21	NO_ROUTE

The traffic flows of event10 (described in Table 4.30) is now switched using the Alternate path routing algorithm. The result of the event10 is summarized in Table 4.33.

Event10 consisted of 10 flows in all. Out of the 10 flows 5 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1007 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. The reason for that is the primary path in the alternate routing algorithm has more capacity than the shortest path. Though the traffic is carried via an extra hop the amount of traffic flowing through the non congested links in the network is increased.

The remaining flows are not transmitted via the primary path as it is congested. Further there is no alternate paths that is non congested, so those flows are not transmitted to prevent congestion in the network links.

Figure 4.68: Network2–Event10: Throughput of link 3–Alternate Path Routing

The throughput of few critical links is shown in Figures 4.68 and 4.69. The throughput is plotted against simulation time.

Figure 4.69: Network2-Event10: Throughput of link 2-Alternate Path Routing

Figure 4.70: Network2-Event10: Throughput of link 8-Alternate Path Routing

Traffic flows via the links 8, 10 and 3, instead of the links 2 and 3. The throughput of those links is shown in Figure 4.70, 4.71 and 4.72.

Figure 4.71: Network2-Event10: Throughput of link 10-Alternate Path Routing

Figure 4.72: Network2-Event10: Throughput of link 13-Alternate Path Routing

4.2.2.3 Event20

The set of flows that are to be switched in the network for event20 are shown in Table 4.34. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.34: Network2-Event20: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	22	23	1273	5	27
2	1001	22	2	1142	1	27
3	1002	22	25	1873	0	19
4	1003	22	18	2090	12	22
5	1004	23	2	1920	5	17
6	1005	23	25	1632	12	27
7	1006	23	18	1970	4	16
8	1007	2	25	2023	0	28
9	1008	2	18	1471	6	19
10	1009	25	18	1238	12	25

4.2.2.3.1 Event20 – using Dijkstra's Algorithm

These set of flow is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.35.

There are 14 critical links identified in this network. The critical links along with their characteristics are shown in Table 4.35. The path each flow has been setup in and the nature of those paths are given in Table 4.36.

Table 4.35: Network2-Event20: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	1	5	2	6000
2	1	6	3	6000
3	5	7	11	7000
4	9	3	6	6000
5	9	6	12	5000
6	26	27	34	5000
7	26	29	35	5000
8	29	28	36	6000
9	21	28	29	5000
10	3	20	7	6000
11	24	27	32	6000
12	11	20	16	6000
13	25	21	28	5000
14	3	8	5	6000

Table 4.36: Network2-Event20: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1002	1	22_26_29_28_21_25_
2	1007	1	2_8_3_9_6_1_5_7_28_21_25_
3	1001	1	22_26_29_28_7_5_1_6_9_3_8_2_
4	1006	3	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
5	1000	1	22_26_27_24_30_23_
6	1004	3	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
7	1008	3	2_8_3_20_11_18_
8	1005	3	23_30_24_27_26_29_28_21_25_
9	1003	3	22_26_29_28_7_5_1_6_9_3_20_11_18_
10	1009	3	25_21_28_7_5_1_6_9_3_20_11_18_

Event20 consisted of 10 flows in all. Out of the 10 flows 4 of them, had a flow result 1 and were switched via the primary non congested route. The other flows (with lspID =

1003, 1004, 1005, 1006, 1008 and 1009), had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.35.

Figure 4.73: Network2-Event20: Throughput of Link 2 – Dijkstra's Algorithm

Figure 4.74: Network2-Event20: Throughput of Link 3 – Dijkstra's Algorithm

The throughput of few of the critical links is shown in Figures 4.73 and 4.74. The throughput is plotted against simulation time.

Figure 4.75: Network2–Event20: Delay of Link 3 – Dijkstra’s Algorithm

The delay of the critical links is shown in Figures 4.75 and 4.76. The delay is plotted against the simulation time.

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

Figure 4.76: Network2–Event20: Delay of Link 2 – Dijkstra’s Algorithm

4.2.2.3.2 Event20 – using Alternate Path Routing Algorithm

The traffic flows of event20 (described in Table 4.34) is now switched using the Alternate path routing algorithm. The result of the event20 is summarized in Table 4.37.

Event20 consisted of 10 flows in all. Out of the 10 flows 5 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1000 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. The reason for that is the primary path in the alternate routing algorithm has more capacity than the shortest path. Though the traffic is carried via an extra hop the amount of traffic flowing through the non congested links in the network is increased.

The remaining flows are not transmitted via the primary path as it is congested. Further there is no alternate paths that is non congested, so those flows are not transmitted to prevent congestion in the network links.

Table 4.37: Network2-Event20: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1002	1	22_26_29_28_21_25_
2	1007	1	2_8_3_9_6_4_1_5_7_28_21_25_
3	1001	1	22_26_29_28_7_5_4_10_6_9_3_8_2_
4	1006	2	23_30_24_27_26_29_28_7_5_4_10_6_9_3_20_11_18_
5	1006	21	23_30_24_27_26_29_28_7_5_4_1_6_9_3_20_11_18_
6	1006	2	23_30_24_27_26_29_28_7_5_4_1_6_9_3_20_11_18_
7	1006	21	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
8	1006	2	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
9	1006	-21	NO_ROUTE
10	1000	1	22_26_27_24_30_23_
11	1004	2	23_30_24_27_26_29_28_7_5_4_10_6_9_3_8_2_
12	1004	21	23_30_24_27_26_29_28_7_5_4_1_6_9_3_8_2_
13	1004	2	23_30_24_27_26_29_28_7_5_4_1_6_9_3_8_2_
14	1004	21	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
15	1004	2	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
16	1004	-21	NO_ROUTE
17	1008	1	2_8_3_20_11_18_
18	1005	2	23_30_24_27_26_29_28_21_25_
19	1005	-21	NO_ROUTE
20	1003	2	22_26_29_28_7_5_4_10_6_9_3_20_11_18_
21	1003	21	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
22	1003	2	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
23	1003	21	22_26_29_28_7_5_1_6_9_3_20_11_18_
24	1003	2	22_26_29_28_7_5_1_6_9_3_20_11_18_
25	1003	-21	NO_ROUTE
26	1009	2	25_21_28_7_5_4_10_6_9_3_20_11_18_
27	1009	21	25_21_28_7_5_4_1_6_9_3_20_11_18_
28	1009	2	25_21_28_7_5_4_1_6_9_3_20_11_18_
29	1009	21	25_21_28_7_5_1_6_9_3_20_11_18_
30	1009	2	25_21_28_7_5_1_6_9_3_20_11_18_
31	1009	-21	NO_ROUTE

The throughput of few critical links is shown in Figures 4.77 and 4.78. The throughput is plotted against simulation time.

Figure 4.77: Network2-Event20: Throughput of link 3-Alternate Path Routing

Figure 4.78: Network2-Event20: Throughput of link 2-Alternate Path Routing

Traffic flows via the links 8, 10 and 3, instead of the links 2 and 3. The throughput of those links is shown in Figure 4.79, 4.80 and 4.81.

Figure 4.79: Network2-Event20: Throughput of link 8-Alternate Path Routing

Figure 4.80: Network2-Event20: Throughput of link 10-Alternate Path Routing

Figure 4.81: Network2–Event20: Throughput of link 13–Alternate Path Routing

4.2.2.4 Event50

Table 4.38: Network2-Event50: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	22	23	2001	13	23
2	1001	22	2	2311	6	27
3	1002	22	25	2037	10	25
4	1003	22	18	1632	14	19
5	1004	23	2	1750	13	26
6	1005	23	25	2426	3	19
7	1006	23	18	1639	3	19
8	1007	2	25	2316	3	24
9	1008	2	18	1323	7	26
10	1009	25	18	2472	12	15

The set of flows that are to be switched in the network for event50 are shown in Table 4.38. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

4.2.2.4.1 Event50 – using Dijkstra's Algorithm

Table 4.39: Network2-Event50: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	1	5	2	6000
2	1	6	3	6000
3	5	7	11	7000
4	9	3	6	6000
5	9	6	12	5000
6	26	27	34	5000
7	26	29	35	5000
8	29	28	36	6000
9	21	28	29	5000
10	3	20	7	6000
11	24	27	32	6000
12	11	20	16	6000
13	25	21	28	5000
14	3	8	5	6000
15	2	8	4	7000
16	22	26	30	7000
17	11	18	15	7000
18	23	30	31	7000
19	30	24	33	7000

The set of flow given in Table 4.38 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical

links in the network are identified. The critical links in this network are shown in Table 4.39.

There are 19 critical links identified in this network. The critical links along with their characteristics are shown in Table 4.39. The path each flow has been setup and the nature of those paths are given in Table 4.40.

Table 4.40: Network2-Event50: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1005	1	23_30_24_27_26_29_28_21_25_
2	1006	1	23_30_24_27_26_29_28_7_5_1_6_9_3_20_11_18_
3	1007	1	2_8_3_9_6_1_5_7_28_21_25_
4	1001	3	22_26_29_28_7_5_1_6_9_3_8_2_
5	1008	1	2_8_3_20_11_18_
6	1002	3	22_26_29_28_21_25_
7	1009	3	25_21_28_7_5_1_6_9_3_20_11_18_
8	1004	3	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
9	1000	3	22_26_27_24_30_23_
10	1003	3	22_26_29_28_7_5_1_6_9_3_20_11_18_

Event50 consisted of 10 flows in all. Out of the 10 flows 4 of them, had a flow result 1 and were switched via the primary non congested route. The other flows (with lspID = 1000, 1001, 1002, 1003, 1004 and 1009), had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.39.

The throughput of few of the critical links is shown in Figures 4.82 and 4.83. The throughput is plotted against simulation time.

Figure 4.82: Network2-Event50: Throughput of Link 2 – Dijkstra's Algorithm

Figure 4.83: Network2-Event50: Throughput of Link 3 – Dijkstra's Algorithm

The delay of the critical links is shown in Figures 4.84 and 4.85. The delay is plotted against the simulation time.

Figure 4.84: Network2–Event50: Delay of Link 3 – Dijkstra’s Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and some additional delay will be introduced due to retransmission of the lost packet.

Figure 4.85: Network2–Event50: Delay of Link 2 – Dijkstra’s Algorithm

4.2.2.4.2 Event50 –using Alternate Path Routing Algorithm

The traffic flows of event50 (described in Table 4.38) is now switched using the Alternate path routing algorithm. The result of the event50 is summarized in Table 4.41.

Event50 consisted of 10 flows in all. Out of the 10 flows 4 of them, had a flow result 1 and were switched via the primary non congested route.

This is one of the worst case results of the Alternate Path Routing Algorithm as the number of flow switched in the non congested path remains the same as that of the Dijkstra’s Algorithm. But the performances of all the flows switched are better than that of the Dijkstra’s Algorithm.

The remaining flows are not transmitted via the primary path as it is congested. Further there is no alternate paths that is non congested, so those flows are not transmitted to prevent congestion in the network links.

Table 4.41: Network2-Event50: Flow Results--Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1005	1	23_30_24_27_26_29_28_21_25_
2	1006	1	23_30_24_27_26_29_28_7_5_4_10_6_9_3_20_11_18_
3	1007	1	2_8_3_9_6_4_1_5_7_28_21_25_
4	1001	2	22_26_29_28_7_5_4_10_6_9_3_8_2_
5	1001	21	22_26_29_28_7_5_4_1_6_9_3_8_2_
6	1001	2	22_26_29_28_7_5_4_1_6_9_3_8_2_
7	1001	21	22_26_29_28_7_5_1_6_9_3_8_2_
8	1001	2	22_26_29_28_7_5_1_6_9_3_8_2_
9	1001	-21	NO_ROUTE
10	1008	1	2_8_3_20_11_18_
11	1002	2	22_26_29_28_21_25_
12	1002	-21	NO_ROUTE
13	1009	2	25_21_28_7_5_4_10_6_9_3_20_11_18_
14	1009	21	25_21_28_7_5_4_1_6_9_3_20_11_18_
15	1009	2	25_21_28_7_5_4_1_6_9_3_20_11_18_
16	1009	21	25_21_28_7_5_1_6_9_3_20_11_18_
17	1009	2	25_21_28_7_5_1_6_9_3_20_11_18_
18	1009	-21	NO_ROUTE
19	1004	2	23_30_24_27_26_29_28_7_5_4_10_6_9_3_8_2_
20	1004	21	23_30_24_27_26_29_28_7_5_4_1_6_9_3_8_2_
21	1004	2	23_30_24_27_26_29_28_7_5_4_1_6_9_3_8_2_
22	1004	21	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
23	1004	2	23_30_24_27_26_29_28_7_5_1_6_9_3_8_2_
24	1004	-21	NO_ROUTE
25	1000	2	22_26_27_24_30_23_
26	1000	-21	NO_ROUTE
27	1003	2	22_26_29_28_7_5_4_10_6_9_3_20_11_18_
28	1003	21	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
29	1003	2	22_26_29_28_7_5_4_1_6_9_3_20_11_18_
30	1003	21	22_26_29_28_7_5_1_6_9_3_20_11_18_
31	1003	2	22_26_29_28_7_5_1_6_9_3_20_11_18_
32	1003	-21	NO_ROUTE

The throughput of few critical links is shown in Figures 4.86 and 4.87. The throughput is plotted against simulation time.

Figure 4.86: Network2-Event50: Throughput of link 3-Alternate Path Routing

Figure 4.87: Network2-Event50: Throughput of link 2-Alternate Path Routing

Traffic flows via the links 8, 10 and 3, instead of the links 2 and 3. The throughput of those links is shown in Figure 4.88, 4.89 and 4.90.

Figure 4.88: Network2-Event50: Throughput of link 8-Alternate Path Routing

Figure 4.89: Network2-Event50: Throughput of link 10-Alternate Path Routing

Figure 4.90: Network2-Event50: Throughput of link 13-Alternate Path Routing

4.2.3 Network3

Figure 4.91: Network3

The third network that was simulated also contained 30 nodes. This network was generated randomly by the random network generator tiers [17]. The network is shown in Figure 4.91. This network contains 6 edge node and 24 LSRs to switch traffic between the edge nodes.

Table 4.42: Routing Table of Node1 of Network3 –Dijkstra’s Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	2	p0	5	1	1_3_7_5_4_2_
2	3	p0	1	1	1_3_
3	4	p0	4	1	1_3_7_5_4_
4	5	p0	3	1	1_3_7_5_
5	6	p0	5	1	1_3_7_5_4_6_
6	7	p0	2	1	1_3_7_
7	8	p0	1	2	1_8_
8	9	p0	3	3	1_18_17_9_
9	10	p0	2	3	1_18_10_
10	11	p0	3	3	1_18_16_11_
11	12	p0	4	2	1_8_14_19_12_
		p1	4	3	1_18_16_11_12_
12	13	p0	4	2	1_8_14_19_13_
		p1	4	3	1_18_16_11_13_
13	14	p0	2	2	1_8_14_
14	15	p0	4	3	1_18_17_9_15_
15	16	p0	2	3	1_18_16_
16	17	p0	2	3	1_18_17_
17	18	p0	1	3	1_18_
18	19	p0	3	2	1_8_14_19_
19	20	p0	4	1	1_3_7_30_20_
20	21	p0	4	1	1_3_7_30_21_
21	22	p0	5	1	1_3_7_30_21_22_
22	23	p0	6	1	1_3_7_30_20_25_23_
23	24	p0	5	1	1_3_7_30_21_24_
24	25	p0	5	1	1_3_7_30_20_25_
25	26	p0	6	1	1_3_7_30_21_22_26_
26	27	p0	7	1	1_3_7_30_20_25_23_27_
27	28	p0	6	1	1_3_7_30_20_25_28_
28	29	p0	8	1	1_3_7_30_20_25_23_27_29_
29	30	p0	3	1	1_3_7_30_

The routing Table for the node1 of this network computed using shortest path algorithm (Dijkstra's algorithm) is shown in Table 4.42.

The routing table of node1 of this network computed using Alternate path routing algorithm (proposed algorithm) is shown in Table 4.43. The routing table is for $\alpha=0.1$.

From the two routing tables (Table 4.42 and Table4.43) it can be seen that the Alternate path routing algorithm gives more number of routes to the destination nodes. Since there is more number of routes to the destination nodes, there is more option of choosing a route when data traffic is switched.

There were four events being setup in this network, viz. event0, event10, event20 and event50. Each event sets up a set of flow between the edge nodes. These events are first switched using the shortest path algorithm (Dijkstra's algorithm) and then they are switched using the Alternate path routing algorithm (proposed algorithm). The simulation results of each of the events are discussed below.

Table 4.43: Routing Table of Node1 of Network3 –Dijkstra’s Algorithm

S.No	To Node ID	Path number	Cost of Path	Output interface	Path
1	2	p0	5	1	1_3_7_5_4_2_
		p1	6	2	1_8_3_7_5_4_2_
		p2	6	1	1_3_7_30_5_4_2_
2	3	p0	1	1	1_3_
		p1	2	2	1_8_3_
3	4	p0	4	1	1_3_7_5_4_
		p1	5	2	1_8_3_7_5_4_
		p2	5	1	1_3_7_30_5_4_
4	5	p0	3	1	1_3_7_5_
		p1	4	2	1_8_3_7_5_
		p2	4	1	1_3_7_30_5_
5	6	p0	5	1	1_3_7_5_4_6_
		p1	6	2	1_8_3_7_5_4_6_
		p2	6	1	1_3_7_30_5_4_6_
6	7	p0	2	1	1_3_7_
		p1	3	2	1_8_3_7_
7	8	p0	1	2	1_8_
		p1	2	1	1_3_8_
8	9	p0	3	3	1_18_17_9_
9	10	p0	2	3	1_18_10_
		p1	3	2	1_8_14_10_
10	11	p0	3	3	1_18_16_11_
		p1	4	2	1_8_14_16_11_
11	12	p0	4	3	1_18_16_11_12_
		p1	5	2	1_8_14_16_11_12_
		p2	5	3	1_18_16_11_13_12_
12	13	p0	4	3	1_18_16_11_13_
		p1	5	2	1_8_14_16_11_13_
		p2	5	3	1_18_16_11_12_13_
		p3	5	3	1_18_17_9_15_13_
13	14	p0	2	2	1_8_14_
		p1	3	1	1_3_8_14_
		p2	3	3	1_18_10_14_
		p3	3	3	1_18_16_14_
14	15	p0	4	3	1_18_17_9_15_
		p1	5	3	1_18_16_11_13_15_
15	16	p0	2	3	1_18_16_
		p1	3	2	1_8_14_16_
16	17	p0	2	3	1_18_17_

17	18	p0	1	3	1_18_
18	19	p0	3	2	1_8_14_19_
		p1	4	1	1_3_8_14_19_
		p2	4	3	1_18_10_14_19_
		p3	4	3	1_18_16_14_19_
19	20	p0	4	1	1_3_7_30_20_
		p1	5	2	1_8_3_7_30_20_
		p2	5	1	1_3_7_5_30_20_
20	21	p0	4	1	1_3_7_30_21_
		p1	5	2	1_8_3_7_30_21_
		p2	5	1	1_3_7_5_30_21_
21	22	p0	5	1	1_3_7_30_21_22_
		p1	6	2	1_8_3_7_30_21_22_
		p2	6	1	1_3_7_5_30_21_22_
22	23	p0	6	1	1_3_7_30_20_25_23_
		p1	7	2	1_8_3_7_30_20_25_23_
		p2	7	1	1_3_7_5_30_20_25_23_
23	24	p0	5	1	1_3_7_30_21_24_
		p1	6	2	1_8_3_7_30_21_24_
		p2	6	1	1_3_7_5_30_21_24_
24	25	p0	5	1	1_3_7_30_20_25_
		p1	6	2	1_8_3_7_30_20_25_
		p2	6	1	1_3_7_5_30_20_25_
25	26	p0	6	1	1_3_7_30_21_22_26_
		p1	7	2	1_8_3_7_30_21_22_26_
		p2	7	1	1_3_7_5_30_21_22_26_
26	27	p0	7	1	1_3_7_30_20_25_23_27_
		p1	8	2	1_8_3_7_30_20_25_23_27_
		p2	8	1	1_3_7_5_30_20_25_23_27_
27	28	p0	6	1	1_3_7_30_20_25_28_
		p1	7	2	1_8_3_7_30_20_25_28_
		p2	7	1	1_3_7_5_30_20_25_28_
28	29	p0	8	1	1_3_7_30_20_25_23_27_29_
		p1	9	2	1_8_3_7_30_20_25_23_27_29_
		p2	9	1	1_3_7_5_30_20_25_23_27_29_
29	30	p0	3	1	1_3_7_30_
		p1	4	2	1_8_3_7_30_
		p2	4	1	1_3_7_5_30_

4.2.3.1 Event0

The set of flows that are to be switched in the network for event0 are shown in Table 4.44. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.44: Network3-Event0: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	29	2	958	12	19
2	1001	29	24	740	3	18
3	1002	29	26	901	11	26
4	1003	29	28	791	11	16
5	1004	29	6	1375	1	20
6	1005	2	24	823	5	23
7	1006	2	26	961	7	18
8	1007	2	28	1326	6	21
9	1008	2	6	873	1	29
10	1009	24	26	1207	8	16

4.2.3.1.1 Event0 – using Dijkstra's Algorithm

These set of flow is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.45.

Table 4.45: Network3-Event0: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	20	30	29	7000
2	4	5	7	6000
3	21	30	32	5000
4	5	30	10	6000
5	20	25	28	6000

There are five critical links identified in this network. The critical links along with their characteristics are shown in Table 4.45. The path each flow has been setup in and the nature of those paths are given in Table 4.46.

Table 4.46: Network3-Event0: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1004	1	29_27_23_25_20_30_5_4_6_
2	1008	1	2_4_6_
3	1001	1	29_27_23_25_20_30_21_24_
4	1005	1	2_4_5_30_21_24_
5	1007	1	2_4_5_30_20_25_28_
6	1011	1	24_21_30_5_4_6_
7	1006	1	2_4_5_30_21_22_26_
8	1009	1	24_21_22_26_
9	1013	3	26_22_21_30_5_4_6_
10	1014	3	28_25_20_30_5_4_6_
11	1010	3	24_21_30_20_25_28_
12	1012	3	26_22_21_30_20_25_28_
13	1003	1	29_27_23_25_28_
14	1002	3	29_27_23_25_20_30_21_22_26_
15	1000	3	29_27_23_25_20_30_5_4_2_

Event0 consisted of 15 flows in all. Out of the 15 flows 9 of them, had a flow result 1 and were switched via the primary non congested route. The other 6 flows had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.45.

The throughput of one of the critical links is shown in Figures 4.92. The throughput is plotted against simulation time.

Figure 4.92: Network3-Event0: Throughput of Link 10 – Dijkstra's Algorithm

Figure 4.93: Network3-Event0: Delay of Link 10 – Dijkstra's Algorithm

The delay of the critical link is shown in Figures 4.93. The delay is plotted against the simulation time. The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store

all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.3.1.2 Event0 – using Alternate Path Routing Algorithm

Table 4.47: Network3-Event0: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1004	1	29_27_23_25_20_30_7_5_4_6_
2	1008	1	2_4_6_
3	1001	1	29_27_23_25_20_30_21_24_
4	1005	1	2_4_5_7_30_21_24_
5	1007	1	2_4_5_7_30_20_25_28_
6	1011	1	24_21_30_7_5_4_6_
7	1006	2	2_4_5_7_30_21_22_26_
8	1006	21	2_4_5_30_21_22_26_
9	1006	1	2_4_5_30_21_22_26_
10	1009	1	24_21_22_26_
11	1013	2	26_22_21_30_7_5_4_6_
12	1013	21	26_22_21_30_5_4_6_
13	1013	2	26_22_21_30_5_4_6_
14	1013	-21	NO_ROUTE
15	1014	2	28_25_20_30_7_5_4_6_
16	1014	21	28_25_20_30_5_4_6_
17	1014	2	28_25_20_30_5_4_6_
18	1014	-21	NO_ROUTE
19	1010	1	24_21_30_20_25_28_
20	1012	2	26_22_21_30_20_25_28_
21	1012	-21	NO_ROUTE
22	1003	1	29_27_23_25_28_
23	1002	2	29_27_23_25_20_30_21_22_26_
24	1002	-21	NO_ROUTE
25	1000	2	29_27_23_25_20_30_7_5_4_2_
26	1000	21	29_27_23_25_20_30_5_4_2_
27	1000	2	29_27_23_25_20_30_5_4_2_
28	1000	-21	NO_ROUTE

The traffic flows of event0 (described in Table 4.44) is now switched using the Alternate path routing algorithm. The result of the event0 is summarized in table 4.47. Event0 consisted of 15 flows in all. Out of the 15 flows 10 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1006 could not be setup via the primary path as it is congested so it is setup through the secondary non-congested path. The flow 1010 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. So in all there are now 11 flows moving via the non-congested link. The amount of traffic flow via the non-congested path is increased in when switched using the alternate path routing algorithm.

The throughput of the critical link is shown in Figures 4.94. The throughput is plotted against simulation time.

Figure 4.94: Network3–Event0: Throughput of link 10–Alternate Path Routing

Traffic flows via the links 9 and 11, instead of the link10. The throughput of those links is shown in Figure 4.95 and 4.96.

Figure 4.95: Network3-Event0: Throughput of link 9-Alternate Path Routing

Figure 4.96: Network3-Event0: Throughput of link 11-Alternate Path Routing

4.2.3.2 Event10

Table 4.48: Network3-Event10: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	29	2	1016	11	29
2	1001	29	24	1110	4	17
3	1002	29	26	1000	3	26
4	1003	29	28	1229	5	26
5	1004	29	6	1242	6	28
6	1005	2	24	1062	13	22
7	1006	2	26	1126	2	27
8	1007	2	28	1500	8	29
9	1008	2	6	1372	13	27
10	1009	24	26	1345	4	26
11	1010	24	28	824	9	28
12	1011	24	6	1459	9	17
13	1012	26	28	811	13	28
14	1013	26	6	1071	13	22
15	1014	28	6	788	0	25

The set of flows that are to be switched in the network for event10 are shown in Table 4.48. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

4.2.3.2.1 Event10 – using Dijkstra's Algorithm

The set of flow given in Table 4.48 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.49.

Table 4.49: Network3-Event10: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	20	30	29	7000
2	4	5	7	6000
3	21	30	32	5000
4	5	30	10	6000
5	20	25	28	6000
6	22	21	30	5000
7	22	26	33	5000
8	27	23	35	5000
9	28	25	36	5000
10	29	27	37	5000

There are ten critical links identified in this network. The critical links along with their characteristics are shown in Table 4.49. The path each flow has been setup in and the nature of those paths are given in Table 4.50.

Table 4.50: Network3-Event10: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1014	1	28_25_20_30_5_4_6_
2	1006	1	2_4_5_30_21_22_26_
3	1002	1	29_27_23_25_20_30_21_22_26_
4	1009	1	24_21_22_26_
5	1001	1	29_27_23_25_20_30_21_24_
6	1003	1	29_27_23_25_28_
7	1004	1	29_27_23_25_20_30_5_4_6_
8	1007	1	2_4_5_30_20_25_28_
9	1011	3	24_21_30_5_4_6_
10	1010	3	24_21_30_20_25_28_
11	1000	3	29_27_23_25_20_30_5_4_2_
12	1005	3	2_4_5_30_21_24_
13	1012	3	26_22_21_30_20_25_28_
14	1013	3	26_22_21_30_5_4_6_
15	1008	1	2_4_6_

Event10 consisted of 15 flows in all. Out of the 15 flows 9 of them, had a flow result 1 and were switched via the primary non congested route. The other flows had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.49.

The throughput of one of the critical link is shown in Figure 4.97. The throughput is plotted against simulation time.

Figure 4.97: Network3–Event10: Throughput of Link 10 – Dijkstra’s Algorithm

The delay of the critical link is shown in Figure 4.98. The delay is plotted against the simulation time.

Figure 4.98: Network3-Event10: Delay of Link 10 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.3.2.2 Event10 – using Alternate Path Routing Algorithm

The traffic flows of event10 (described in Table 4.48) is now switched using the Alternate path routing algorithm. The result of the event10 is summarized in Table 4.51. Event0 consisted of 15 flows in all. Out of the 15 flows 10 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1005 could not be setup via the primary path as it is congested so it is setup through the secondary non-congested path. The flow 1005 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. So in all there are now 11 flows

moving via the non-congested link. The amount of traffic flow via the non-congested path is increased in when switched using the alternate path routing algorithm.

Table 4.51: Network3-Event10: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1014	1	28_25_20_30_7_5_4_6_
2	1006	1	2_4_5_7_30_21_22_26_
3	1002	1	29_27_23_25_20_30_21_22_26_
4	1009	1	24_21_22_26_
5	1001	1	29_27_23_25_20_30_21_24_
6	1003	1	29_27_23_25_28_
7	1004	1	29_27_23_25_20_30_7_5_4_6_
8	1007	1	2_4_5_7_30_20_25_28_
9	1011	2	24_21_30_7_5_4_6_
10	1011	21	24_21_30_5_4_6_
11	1011	2	24_21_30_5_4_6_
12	1011	-21	NO_ROUTE
13	1010	2	24_21_30_20_25_28_
14	1010	-21	NO_ROUTE
15	1000	2	29_27_23_25_20_30_7_5_4_2_
16	1000	21	29_27_23_25_20_30_5_4_2_
17	1000	2	29_27_23_25_20_30_5_4_2_
18	1000	-21	NO_ROUTE
19	1005	2	2_4_5_7_30_21_24_
20	1005	21	2_4_5_30_21_24_
21	1005	1	2_4_5_30_21_24_
22	1012	2	26_22_21_30_20_25_28_
23	1012	-21	NO_ROUTE
24	1013	2	26_22_21_30_7_5_4_6_
25	1013	21	26_22_21_30_5_4_6_
26	1013	2	26_22_21_30_5_4_6_
27	1013	-21	NO_ROUTE
28	1008	1	2_4_6_

The throughput of the critical link is shown in Figure 4.99. The throughput is plotted against simulation time.

Figure 4.99: Network3–Event10: Throughput of link 10–Alternate Path Routing

Figure 4.100: Network3–Event10: Throughput of link 9–Alternate Path Routing

Traffic flows via the links 9 and 11, instead of the link10. The throughput of those links is shown in Figure 4.100 and 4.101.

Figure 4.101: Network3–Event10: Throughput of link 11–Alternate Path Routing

4.2.3.3 Event20

The set of flows that are to be switched in the network for event20 are shown in Table 4.52. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm) and then the same set of flows are switched using the proposed Alternate path routing algorithm

Table 4.52: Network3-Event20: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	29	2	1210	12	27
2	1001	29	24	1271	13	23
3	1002	29	26	913	4	20
4	1003	29	28	1041	12	16
5	1004	29	6	1563	13	29
6	1005	2	24	1027	11	27
7	1006	2	26	998	6	26
8	1007	2	28	1000	2	20
9	1008	2	6	1067	0	16
10	1009	24	26	1658	3	23
11	1010	24	28	1191	8	15
12	1011	24	6	1624	6	22
13	1012	26	28	1122	6	25
14	1013	26	6	924	10	16
15	1014	28	6	1241	2	19

4.2.3.3.1 Event20 – using Dijkstra’s Algorithm

Table 4.53: Network3-Event20: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	20	30	29	7000
2	4	5	7	6000
3	21	30	32	5000
4	5	30	10	6000
5	20	25	28	6000
6	22	21	30	5000
7	22	26	33	5000
8	27	23	35	5000
9	28	25	36	5000
10	29	27	37	5000
11	24	21	31	6000
12	6	4	8	6000

The set of flow given in Table 4.52 is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.53.

There are 12 critical links identified in this network. The critical links along with their characteristics are shown in Table 4.53. The path each flow has been setup in and the nature of those paths are given in Table 4.54.

Table 4.54: Network3-Event20: Flow Results–Using Dijkstra's Algorithm

S.No	Lsp ID	Flow Result	Route
1	1008	1	2_4_6_
2	1007	1	2_4_5_30_20_25_28_
3	1014	1	28_25_20_30_5_4_6_
4	1009	1	24_21_22_26_
5	1002	1	29_27_23_25_20_30_21_22_26_
6	1011	1	24_21_30_5_4_6_
7	1012	1	26_22_21_30_20_25_28_
8	1006	1	2_4_5_30_21_22_26_
9	1010	3	24_21_30_20_25_28_
10	1013	3	26_22_21_30_5_4_6_
11	1005	3	2_4_5_30_21_24_
12	1000	3	29_27_23_25_20_30_5_4_2_
13	1003	3	29_27_23_25_28_
14	1001	3	29_27_23_25_20_30_21_24_
15	1004	3	29_27_23_25_20_30_5_4_6_

Event20 consisted of 15 flows in all. Out of the 15 flows 8 of them, had a flow result 1 and were switched via the primary non congested route. The other flows had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.53.

The throughput of one of the critical link is shown in Figure 4.102. The throughput is plotted against simulation time.

Figure 4.102: Network3-Event20: Throughput of Link 10 – Dijkstra’s Algorithm

Figure 4.103: Network3-Event20: Delay of Link 10 – Dijkstra’s Algorithm

The delay of the critical link is shown in Figure 4.103. The delay is plotted against the simulation time. The delay is calculated under the assumption that there is infinite buffer available.

4.2.3.3.2 Event20 – using Alternate Path Routing Algorithm

Table 4.55: Network3-Event20: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1008	1	2_4_6_
2	1007	1	2_4_5_7_30_20_25_28_
3	1014	1	28_25_20_30_7_5_4_6_
4	1009	1	24_21_22_26_
5	1002	1	29_27_23_25_20_30_21_22_26_
6	1011	1	24_21_30_7_5_4_6_
7	1012	1	26_22_21_30_20_25_28_
8	1006	1	2_4_5_7_30_21_22_26_
9	1010	2	24_21_30_20_25_28_
10	1010	-21	NO_ROUTE
11	1013	2	26_22_21_30_7_5_4_6_
12	1013	21	26_22_21_30_5_4_6_
13	1013	2	26_22_21_30_5_4_6_
14	1013	-21	NO_ROUTE
15	1005	2	2_4_5_7_30_21_24_
16	1005	21	2_4_5_30_21_24_
17	1005	2	2_4_5_30_21_24_
18	1005	-21	NO_ROUTE
19	1000	2	29_27_23_25_20_30_7_5_4_2_
20	1000	21	29_27_23_25_20_30_5_4_2_
21	1000	2	29_27_23_25_20_30_5_4_2_
22	1000	-21	NO_ROUTE
23	1003	1	29_27_23_25_28_
24	1001	2	29_27_23_25_20_30_21_24_
25	1001	-21	NO_ROUTE
26	1004	2	29_27_23_25_20_30_7_5_4_6_
27	1004	21	29_27_23_25_20_30_5_4_6_
28	1004	2	29_27_23_25_20_30_5_4_6_
29	1004	-21	NO_ROUTE

The traffic flows of event20 (described in Table 4.52) is now switched using the Alternate path routing algorithm. The result of the event20 is summarized in Table 4.55. Event20 consisted of 15 flows in all. Out of the 15 flows 10 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1003 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. So in all there are now 10 flows moving via the non-congested link. The amount of traffic flow via the non-congested path is increased in when switched using the alternate path routing algorithm.

The throughput of the critical link is shown in Figure 4.104. The throughput is plotted against simulation time.

Figure 4.104: Network3–Event20: Throughput of link 10–Alternate Path Routing

Traffic flows via the links 9 and 11, instead of the link10. The throughput of those links is shown in Figure 4.105 and 4.106.

Figure 4.105: Network3-Event20: Throughput of link 9-Alternate Path Routing

Figure 4.106: Network3-Event20: Throughput of link 11-Alternate Path Routing

4.2.3.4 Event50

The set of flows that are to be switched in the network for event50 are shown in Table 4.56. These flows are first switched using the shortest path algorithm (Dijkstra's algorithm).

Table 4.56: Network3-Event50: Set of Flows

S.No	Lsp ID	From Node	To Node	Traffic Rate	From Time	To Time
1	1000	29	2	1173	4	29
2	1001	29	24	1074	7	28
3	1002	29	26	1814	6	29
4	1003	29	28	1149	12	18
5	1004	29	6	2013	11	27
6	1005	2	24	1351	4	24
7	1006	2	26	1688	2	20
8	1007	2	28	1501	4	27
9	1008	2	6	1164	14	29
10	1009	24	26	1475	12	28
11	1010	24	28	1885	10	25
12	1011	24	6	1963	5	18
13	1012	26	28	1285	13	23
14	1013	26	6	1280	12	21
15	1014	28	6	1817	1	16

4.2.3.4.1 Event50 – using Dijkstra's Algorithm

These set of flow is first switched using the shortest path algorithm (Dijkstra's Algorithm). While switching using the shortest path algorithm the critical links in the network are identified. The critical links in this network are shown in Table 4.57.

Table 4.57: Network3-Event50: Set of Critical links

S.No	From Node	To Node	Link ID	Bandwidth in kbps
1	20	30	29	7000
2	4	5	7	6000
3	21	30	32	5000
4	5	30	10	6000
5	20	25	28	6000
6	22	21	30	5000
7	22	26	33	5000
8	27	23	35	5000
9	28	25	36	5000
10	29	27	37	5000
11	24	21	31	6000
12	6	4	8	6000
13	25	23	34	6000

Table 4.58: Network3-Event50: Flow Results–Using Dijkstra’s Algorithm

S.No	Lsp ID	Flow Result	Route
1	1014	1	28_25_20_30_5_4_6_
2	1006	1	2_4_5_30_21_22_26_
3	1000	1	29_27_23_25_20_30_5_4_2_
4	1007	3	2_4_5_30_20_25_28_
5	1005	3	2_4_5_30_21_24_
6	1011	3	24_21_30_5_4_6_
7	1002	3	29_27_23_25_20_30_21_22_26_
8	1001	3	29_27_23_25_20_30_21_24_
9	1010	3	24_21_30_20_25_28_
10	1004	3	29_27_23_25_20_30_5_4_6_
11	1009	3	24_21_22_26_
12	1013	3	26_22_21_30_5_4_6_
13	1003	3	29_27_23_25_28_
14	1012	3	26_22_21_30_20_25_28_
15	1008	3	2_4_6_

There are 13 critical links identified in this network. The critical links along with their characteristics are shown in Table 4.57. The path each flow has been setup in and the nature of those paths are given in Table 4.58.

Event50 consisted of 15 flows in all. Out of the 15 flows 3 of them, had a flow result 1 and were switched via the primary non congested route. The other flows had a flow result 3 and were setup through a congested route, the links that are congested is given in Table 4.57.

The throughput of one of the critical link is shown in Figure 4.107. The throughput is plotted against simulation time.

Figure 4.107: Network3–Event50: Throughput of Link 10 – Dijkstra’s Algorithm

The delay of the critical link is shown in Figures 4.108. The delay is plotted against the simulation time.

Figure 4.108. Network3–Event50: Delay of Link 10 – Dijkstra's Algorithm

The delay is calculated under the assumption that there is infinite buffer available. If infinite buffer is not present there and the buffer size is insufficient to store all the packets getting delayed, then there will be packet loss and additional delay due to retransmission.

4.2.3.4.2 Event50 – using Alternate Path Routing Algorithm

The traffic flows of event50 (described in Table 4.56) is now switched using the Alternate path routing algorithm. The result of the event50 is summarized in Table 4.59. Event20 consisted of 15 flows in all. Out of the 15 flows 7 of them, had a flow result 1 and were switched via the primary non congested route. The flow 1001, 1002, 1008 and 1009 which had been setup through a congested path in the Dijkstra algorithm is setup via non-congested path here. So in all there are now 7 flows moving via the non-

congested link. The amount of traffic flow via the non-congested path is increased in when switched using the alternate path routing algorithm.

Table 4.59: Network3-Event50: Flow Results–Using Alternate Path Routing

S.No	Lsp ID	Flow Result	Route
1	1014	1	28_25_20_30_7_5_4_6_
2	1006	1	2_4_5_7_30_21_22_26_
3	1000	1	29_27_23_25_20_30_7_5_4_2_
4	1007	2	2_4_5_7_30_20_25_28_
5	1007	21	2_4_5_30_20_25_28_
6	1007	2	2_4_5_30_20_25_28_
7	1007	-21	NO_ROUTE
8	1005	2	2_4_5_7_30_21_24_
9	1005	21	2_4_5_30_21_24_
10	1005	21	2_4_5_30_21_24_
11	1005	-21	NO_ROUTE
12	1011	2	24_21_30_7_5_4_6_
13	1011	21	24_21_30_5_4_6_
14	1011	2	24_21_30_5_4_6_
15	1011	-21	NO_ROUTE
16	1002	1	29_27_23_25_20_30_21_22_26_
17	1001	1	29_27_23_25_20_30_21_24_
18	1010	2	24_21_30_20_25_28_
19	1010	-21	NO_ROUTE
20	1004	2	29_27_23_25_20_30_7_5_4_6_
21	1004	21	29_27_23_25_20_30_5_4_6_
22	1004	2	29_27_23_25_20_30_5_4_6_
23	1004	-21	NO_ROUTE
24	1009	1	24_21_22_26_
25	1013	2	26_22_21_30_7_5_4_6_
26	1013	21	26_22_21_30_5_4_6_
27	1013	2	26_22_21_30_5_4_6_
28	1013	-21	NO_ROUTE
29	1003	2	29_27_23_25_28_
30	1003	-21	NO_ROUTE
31	1012	2	26_22_21_30_20_25_28_
32	1012	-21	NO_ROUTE
33	1008	1	2_4_6_

The throughput of the critical link is shown in Figure 4.109. The throughput is plotted against simulation time.

Figure 4.109: Network3-Event50: Throughput of link 10-Alternate Path Routing

Figure 4.110: Network3-Event50: Throughput of link 9-Alternate Path Routing

Traffic flows via the links 9 and 11, instead of the link10. The throughput of those links is shown in Figure 4.110 and 4.111.

Figure 4.111: Network3–Event50: Throughput of link 11–Alternate Path Routing

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

Traffic in the Internet grows every year but the bandwidth of the networks does not grow at the same rate, which leads to congestion in the network and performance degradation. A closer evaluation of the links in the network suggests that not all the links in the network remain congested. Only a few links (critical links) are congested. Most of the traffic in the network flows through a critical link.

The reason for only a few links getting congested is that the routing algorithm used to route traffic in the Internet are based on Dijkstra's Algorithm. This algorithm routes traffic via the shortest path conserving the network resources. But this algorithm also creates imbalance in the network utilization. This imbalance is seen in the simulation results.

In the proposed algorithm the critical links in the network are identified and the traffic in the critical links is distributed to other links. This reduces imbalance in the network utilization and increases the performance of the network. The critical links are identified from the history of the data traffic in the network.

To distribute the traffic evenly among the network links, alternate paths are found. These alternate paths may have longer path cost and could have lesser number of critical links in them than the shortest path.

The algorithm ensures that the alternate path does not have a high path cost. Higher path cost implies greater hop count. When a flow is setup through a path that has high path cost (higher hop count) it consumes more network resource. So, other future flows will have no or little network resources and hence the performance of the network will deteriorate. The alternate path parameter “ α ” ensures that the path cost of the alternate path is low.

The simulation was conducted for random networks generated by Tiers Topology Generator [17]. The performance of the proposed algorithm was compared with the shortest path algorithm (Dijkstra’s algorithm). The increase in the performance for the 30 random networks with the proposed algorithm varied from 0 to 233%. Simulation of the best and the worst case performance of networks are discussed in Chapter 4.

From the simulation results it can be seen that the performance of the proposed algorithm is better than the Dijkstra’s algorithm. The increase in the performance is at the expense of extra hop counts traversed by some flows. The computation time remains the same as the Dijkstra’s algorithm ($O(n)$), when there are no alternate paths, and about twice greater ($O(2n)$) when there are alternate paths in the network.

In conclusion this routing algorithm is easy to implement and provides good routing paths. The algorithm could be improved as follows.

- 1) The alternate path parameter “ α ” variable determines the extra path cost traversed by the alternate path. The value of “ α ” at which the performance of a network attains maximum depends on the network. In this thesis a fixed value of 0.1 was used for all the networks in the simulation. Future work would be to write an

algorithm to find the value of “ α ” at which the maximum performance of the network is attained.

- 2) The proposed algorithm performs admission control. If a flow cannot be routed via any path that is not congested it is blocked. This ensures that the performance of the already existing flows does not degrade. This admission control mechanism might lead to under utilization of the network. Future work would be to improve the admission control process so that the under utilization of the network is prevented at the expense of little or no delay.
- 3) This algorithm sets up the flow request in the primary path, if the primary path is congested then the flow is setup in the first additional path that is non-congested. It does not take into account the bandwidth of the flow request and does not try and map it into a path that exactly or closely matches the bandwidth of the request. This could be done so that the path with maximum capacity is reserved for a future request. But this should be done without increasing the complexity of the algorithm. Future work would be to device a formula that takes into account the existing data and the bandwidth of flow request and give out the ideal path for that flow.

BIBLIOGRAPHY

1. *K. G. Coffman and A. M. Odlyzko*, "Growth of the Internet", Optical Fiber Telecommunications IV, I. P. Kaminow and T. Li, eds. Academic Press, 2001.
2. *Rosen, E., Viswanathan, A. and R. Callon*, "Multiprotocol Label Switching Architecture", RFC 3031.
3. *Juha Heinanen*, Multiprotocol Encapsulation over ATM Adaptation Layer 5, RFC 1483.
4. *D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, G. Swallow*, "RSVP-TE Extension to RSVP for LSP", RFC 3209.
5. *L. Andersson, P. Doolan, N. Feldman, A. Fredette, B. Thomas* , "LDP Specification", RFC 3036.
6. *B. Jamoussi*, "Constraint-Based LSP Setup using LDP", Work in Progress
7. *J. Moy*, "OSPF Version 2", RFC 1583
8. *C. Hedrick*, "Routing Information Protocol", RFC 1058
9. *Dave Katz, Derek Yeung, Kireeti Kompella*, "Traffic Engineering Extensions to OSPF - draft-katz-yeung-ospf-traffic-06.txt ", Work in progress
10. *Coltun, R.*, "The OSPF Opaque LSA Option," RFC 2370, July 1998.
11. *K. Nichols, S. Blake, F. Baker, D. Black*, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.

12. *R. Braden, D. Clark, S. Shenker*, “Integrated Services in the Internet Architecture: an Overview”, RFC 1633.
13. *Qingming Ma, Peter Steenkiste*, “On Path Selection for Traffic with Bandwidth Guarantees”, 5th IEEE International Conference on Network Protocols, October 1997.
14. *Murali Kodialam, T.V. Lakshman*, “Minimum Interference Routing with Applications to MPLS Traffic Engineering”.
15. *Subhash Suri, Marcel Waldvogel, Priyank Ramesh Warkhede*, “Profile-Based Routing: A New Framework for MPLS Traffic Engineering”.
16. *D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, J. McManus*, “Requirements for Traffic Engineering Over MPLS”, RFC 2702.
17. *Kevin Fall and Kannan Varadhan*, editor, “The *ns* Manual”, The VINT Project.

APPENDIX 1

C++ CODES AND DOCUMENTATION

In this appendix the c codes that was written for the thesis is given. There span 14 files. The files describe a set of class and structures. The class name followed by a brief description about the member functions of the class and structure is given below. The actual code is given in the accompanying disk. The tool doxygen was used to document the code.

NODEADDRESS CLASS REFERENCE

This class stores the address of a node. This class is used to store the ip address of a node. It stores the hierarchical address of the node.

Public Methods

`nodeAddress ()`

Constructor to Initialize the address.

`~nodeAddress ()`

Dummy destructor.

`nodeAddress (int a,int b, int c, int d)`

Constructor which take 4 integers as input and uses it to store the address.

`void setAddress (int a,int b, int c, int d)`

Here a,b,c,d are the 4 parts of the heirarchical address.

`char* hierAddress ()`

Returns the hierchical address in string format 131.12.12.1.

`void dumpAddress ()`

Prints the address in the following format a.b.c.d eg 131.12.12.2.

Private Attributes

`int address [4]`

This variable stores the address.

EVENT STRUCT REFERENCE

This is a structure which stores the list of events to be executed.

This structure event is the base structure which is used to store the list of events. It stores all sets of events that are to be executed in a list.

Public Attributes

int lspID

This is the event ID.

int fromNode

The start node from which traffic originate.

int toNode

The end node where traffic sinks.

double trafficReserved

Amount of traffic to be switched from start to end node.

double fromTime

Time at which traffic flow should start.

double toTime

Time at which traffic flow ends.

char * event::path

This is an optional variable which is used to specify the path explicitly. This is the path in which traffic has to be switched. This variable is not implemented in the simulator. It is to be developed

struct event* next

This is a pointer pointing to the next event.

EVENTHANDLER CLASS REFERENCE

This is the base class which handles all the events and schedules them. This class is the base class where all the events are scheduled. This class loads the event list from a file and executes the event. Here the events are stored in 2 linked list one list contains the events that are to be started and the other list contains a list of events that are started but are to be stopped. the simulation is said to be complete when both these list are empty

Public Methods

eventHandler ()

Constructor initializing the start and stop list to NULL. It also stores the time at which the event starts.

`int loadEventFromFile (char *fName, char *nFileName)`

This function parses the input file and loads the events into the list. For a detailed description look below at the member function documentation.

`int addEvent (int lspID, int fNode, int tNode, double tReserved, double fTime, double tTime, char *path=NULL, int addInEventList = 1)`

Adds a node in the list. The new nodes parameters are given as input parameter. For a detailed description look below at the member function documentation.

`int addEvent (struct event *tra, int addInEventList = 1)`

Adds a node in the list. The new node is given as input parameter. For a detailed description look below at the member function documentation.

`int deleteEvent (int lspID, int deleteInEventList = 1)`

Deletes a node whose lspID is equal to the lspID given as input. For a detailed description look below at the member function documentation.

`struct event* getEventListHandler ()`

Gets a handle to the start list.

`struct event* getStopListHandler ()`

Gets a handle to the stop list.

`void displayEvent (struct event *tra)`

This function is used to display the list of all events stored in the list.

`int swapEvent (struct event *a, struct event *b)`

This function swaps the 2 nodes given as inputs (a, b).

`int sortEventByFromTime (struct event *tra)`

This function sorts the list with respect to the start time. For a detailed description look below at the member function documentation.

`int sortEventByToTime (struct event *tra)`

This function sorts the list with respect to the stop time. For a detailed description look below at the member function documentation.

`int setPath (int lspID, char *path, int addInEventList = 1)`

Sets the path in the start or the stop list. For a detailed description look below at the member function documentation.

`void setEventFileName (char *fName)`

Sets event file name.

`char* getEventFileName ()`

Gets events file name.

Protected Attributes

`struct event* eventList`

This list stores the list of events that are to be executed.

`struct event* stopList`

This list stores the list of events that have been started and running.

`time_t startTime`

Stores the start time of the simulation.

`char eventFileName [20]`

Stores the name of the file from which the events are loaded into the list.

Member Function Documentation

`int eventHandler::addEvent ( struct event * temp, int addInEventList = 1 )`

Add a node in the list. The new node is given as input parameter.

The node is added into the eventList or the Stop list. This is determined by the flag addInEventList. If this flag is set to 1 then the event is added to the start or the eventList, else it is added to the stop list

`int eventHandler::addEvent (int lspID, int fNode, int tNode, double tReserved, double fTime, double tTime, char * path = NULL, int addInEventList = 1 )`

Add a node in the list. The new nodes parameters are given as input parameter.

The node is added into the eventList or the Stop list. This is determined by the flag addInEventList. If this flag is set to 1 then the event is added to the start or the eventList, else it is added to the stop list

`int eventHandler::deleteEvent ( int lspID, int deleteInEventList = 1 )`

Delete a node whose lspID is equal to the lspID given as input.

The node is deleted from the eventList or the Stop list. This is determined by the flag deleteInEventList. If this flag is set to 1 then the event is deleted from the start or the eventList, else the node is deleted from the stop list

`int eventHandler::loadEventFromFile ( char * fName, char * nFileName )`

This function parses the input file and loads the events into the list.

This file inturn calls the getNodeFileName function and the parse function. Those two function parse the input string and returns the desired value

`int eventHandler::setPath ( int lspID, char * path, int addInEventList = 1 )`

Set the path in the start or the stop list. Set the path through which the traffic is to be switched for the event identified by the lspID. Since both the start list(event list) and the stop list are of the same type, instead of defining 2 functions to perform set path in the 2 list we use the same function and set an additional flag - addInEventList, which specifies whether the path is to be added in the eventList or the stopList

```
int eventHandler::sortEventByFromTime ( struct event * tra )
```

This function sorts the list with respect to the start time. This function is called with the start list(eventList). The nodes in the start list are sorted according to the start time. So the events are executed in order. When ever a event is added to the start list this function is called so as to ensure that the start list always remains sorted according to the start time

```
int eventHandler::sortEventByToTime ( struct event * tra )
```

This function sorts the list with respect to the stop time. This function is called with the stop list. The nodes in the stop list are sorted according to the to time. So the events are stopped in order. When ever a event is added to the stop list this function is called so as to ensure that the stop list always remains sorted according to the stop time

EVENTHANDLERAP CLASS REFERENCE

This is a class that is derived from eventHandler. This class switches traffic using the Alternate path routing algorithm. This class has one private variable and two functions.

Inheritance diagram for eventHandlerAP:

Public Methods

```
eventHandlerAP ()
```

A dummy constructor.

```
int executeAllEvents (char *fName)
```

This is a function which handles all the events, it setup the event and start time and stops at the stop time. For a detailed description look below at the member function documentation.

```
int setLinkUtilization (char *path, int tRate, double time, int add=1)
```

This functions alters the link utilization of the each link when the each flow is setup. For a detailed description look below at the member function documentation.

Private Attributes

class allNodeList nodeList

This is a list which stores the details about the network connectivity. For a detailed description look below at the member data documentation.

Member Function Documentation

int eventHandlerAP::executeAllEvents (char * fName)

This is a function which handles all the events, it setups the event and start time and stops at the stop time. This function is called with one input:

char *fName - This is the input file name where the various events to be scheduled are written

The events are loaded into the linked list of the parent class private members event list and the stop list. The events executed according to the start time and are stopped according to the stop time. The events after being executed are deleted from the eventList and the same event are added in the stopList. The exit condition for the function is the exit condition for the simulator. The function stops when both the eventList and the stopList are empty - meaning that all the events are scheduled and all the scheduled events are stopped. This function writes its output to a file with name fName+.APres. The output file gives a jist of information about how the various events where scheduled and thro what route the traffic was scheduled and whether the path was congested or not.

int eventHandlerAP::setLinkUtilization (char * er, int tRate, double time, int add = 1)

This functions alters the link utilization of the each link when the each flow is setup. This function takes 4 parameters as the input.

char *er - gives the path through which the traffic is to flow int tRate - gives the amount of traffic that is to flow in thro that path double time - gives the time at which the flow starts or stops int add - this is a flag that is tells whether the traffic is about to start or end The same function is used to both add and subtract the traffic, addition of traffic to a link happens when a flow is setup, while deletion of a traffic happens when the same flow is stopped

Member Data Documentation

class allNodeList eventHandlerAP::nodeList [private]

EVENTHANDLERMD CLASS REFERENCE

This is a class that is derived from eventHandler. This class swithces traffic using the Shortest Path Algorithm.

Inheritance diagram for eventHandlerMD:

Public Methods

`eventHandlerMD ()`

A dummy constructor.

`int executeAllEvents (char *fName)`

This is a function which handles all the events, it setups the event and start time and stops at the stop time. For a detailed description look below at the member function documentation.

`int setLinkUtilization (char *path, int tRate, double time, int add=1)`

This functions alters the link utilization of the each link when the each flow is setup. For a detailed description look below at the member function documentation.

Private Attributes

`class allNodeList nodeList`

This is a list which stores the details about the network connectivity. For a detailed description look below at the member data documentation.

Member Function Documentation

`int eventHandlerMD::executeAllEvents ( char * fName )`

This is a function which handles all the events, it setups the event and start time and stops at the stop time. This function is called with one input

`char *fName` - This is the input file name where the various events to be scheduled are written

The events are loaded into the linked list of the parent class private members event list and the stop list. The events executed according to the start time and are stopped according to the stop time. The events after being executed are deleted from the eventList and the same event are added in the stopList. The exit condition for the function is the exit condition for the simulator. The function stops when both the eventList and the stopList are empty - meaning that all the events are scheduled and all the scheduled events are stopped. This function writes its output to a file with name `fName+.APres`. The output file gives a jist of information about how the various events where scheduled and thro what route the traffic was scheduled and whether the path was congested or not.

`int eventHandlerMD::setLinkUtilization (char * er, int tRate, double time, int add = 1 )`

This functions alters the link utilization of the each link when the each flow is setup. This function takes 4 parameters as the input.

char *er - gives the path through which the traffic is to flow
 tRate - gives the amount of traffic that is to flow in thro that path
 double time - gives the time at which the flow starts or stops
 int add - this is a flag that tells whether the traffic is about to start or end

The same function is used to both add and subtract the traffic, addition of traffic to a link happens when a flow is setup, while deletion of a traffic happens when the same flow is stopped

Member Data Documentation

class allNodeList eventHandlerMD::nodeList [private]

This is a list which stores the details about the network connectivity. This list acts as a list of information that is got from a linkstate database. This list gives information about the connectivity of each link and also the amount of traffic flowing in each link at a particular time.

LINKUTILIZATION CLASS REFERENCE

This class updates the link utilization of the links. This class has 2 private members one describing the network and the other describing the list of lsps setup in that network. The purpose of this class is to update the link utilization of each and every link as the lsps are setup.

Public Methods

linkUtilization ()

The dummy constructor for the class.

~linkUtilization ()

The destructor for the class, which does nothing but just exits and stops the program.

int initialize (char* fName, char *teFName)

Loads LSP list and the network into the corresponding linked list, by calling the associated class functions.

int updateLinkUtilizationFromLSPList ()

This function updates the link utilization for all the links. For a detailed description look below at the member function documentation.

struct node* getNodeListHandle ()

Gets the handle to the list head of the nodeList. This function gets an handle to the node list.

void dumpList ()

Dumps the lsp list and the node list to the stdout.

Private Attributes

class teLSP teLSPList

This member describes the list of LSPs setup in the network.

class allNodeList nodeList

This nodeList describes the entire network.

Member Function Documentation

int linkUtilization::updateLinkUtilizationFromLSPList ()

This function updates the link utilization for all the links. This function runs till the Lsp list becomes null, for each entry in the lsp list the path described by the explicit route in the lsp list is taken and all the links in that explicit route the traffic utilization is changed as present utilization + the traffic rate which is to be setup and which is got from the lsp list

COSTORDEREDLIST STRUCT REFERENCE

This is a self pointing structure describing cost required to reach each node from the start node. This is the list used by the myDijkstra algorithm to find the shortest path. It contains a list of nodes which are in the network. These nodes are arranged in ascending order according to the cost of each node

Public Attributes

int nodeID

int totalCost

char* path [noOfPath]

struct costOrderedList* next

COSTLIST CLASS REFERENCE

This is a class which has a pointer to the first node of the cost list and this also stores the start node from where the shortest path to other nodes are to be computed. This class has a list of functions which helps in computing the shortest path. This class in short maintains a list of node and the path in which that node can be reached from the start node and the cost of that path

Public Methods

costList ()

The default constructor which makes the list head NULL.

`~costList ()`

This is the destructor for the class and it deletes the complete list.

`void setStartNode (int startNode)`

The start node is set by this function.

`int getStartNode ()`

This function returns the start node.

`int addNode (int nID, int tCost)`

Adds a node to the list end.

`int deleteNode (int nID)`

Deletes a node whose node id is equal to nID from this list. For a detailed description look below at the member function documentation.

`void swapNode (struct costOrderedList *a, struct costOrderedList *b)`

This function swaps the 2 given input nodes.

`int sortList ()`

This function sorts the list in ascending order according to the cost of each node.

`int setPath (int nID, char *path, int addPath = 0)`

This function sets a path to the node nID. For a detailed description look below at the member function documentation.

`int initializeList (int startNode, int numberOfNodes, int *nodeList = NULL)`

Initializes the list. The list is created here and initialized. For a detailed description look below at the member function documentation.

`int getTopNodeCost (int &nID, int &tCost)`

Gets the top node id and the cost of the node. For a detailed description look below at the member function documentation.

`int getNodeCost (int nID, int &tCost)`

Gets the cost of the node nID. For a detailed description look below at the member function documentation.

`int updateCost (int nID, int newTCost)`

Changes the cost of the node to newTCost. For a detailed description look below at the member function documentation.

`bool isEmpty ()`

Returns true if the list is empty else returns a false.

char getPath (int nID)**

Gets the path through which the node nID can be reached from the start node. returns a null if nID is not in the list.

bool isNodePresent (int nID)

Checks wheather the node nID is present in the list or not.

void dumpList ()

Prints the list to the standard output.

Private Attributes

struct costOrderedList* listHead

Pointer to the first node of the list.

int sNode

Start node of the Dijkstra algorithm.

Member Function Documentation

int costList::deleteNode (int nID)

Deletes a node whose node id is equal to nID from this list. This funcation returns a one when the node id is found and is sucessfully deleted else it returns a zero

int costList::getNodeCost (int nID, int & tCost)

Gets the cost of the node nID. The cost of the node that is to be returned is returned via the variable tCost. If the node is not present then a 0 is returned by the function indicating an error else a one is returned

int costList::getTopNodeCost (int & nID, int & tCost)

Gets the top node id and the cost of the node. The cost of the node that is to be returned is returned via the variable tCost, while the node id is returned via the variable nID. If the node is not present then a 0 is returned by the function indicating an error else a one is returned

int costList::initializeList (int startNode, int numberOfNodes, int * nodeList = NULL)

Initializes the list. The list is created here and initialized. The nodes are created with node id given by the node list or a list of nodes with node id from one to total number of nodes is created when the nodeList is left null. Node list is an optional parameter. The cost for all the nodes except the start node is set to infinity, the cost for the start node is set to 0

int costList::setPath (int nID, char * path, int addPath = 0)

This function sets a path to the node nID. The path is added to the existing path if add path flag is set else the existing paths are deleted and the path given as input is installed

```
int costList::updateCost ( int nID, int newTCost )
```

Changes the cost of the node to newTCost. If the node is not present then a 0 is returned by the function indicating an error else a one is returned

DIJKSTRAALGORITHM CLASS REFERENCE

This class implements the digkstra algorithm to find the shortest path for a given node in the network to all other nodes in the network. The input for the class is a node list which describes the entire network, while the output is the routing table list which is used to store the routes to the destination nodes in the network

Public Methods

```
dijkstraAlgorithm ()
```

A Dummy Constructor for the class.

```
~dijkstraAlgorithm ()
```

A Dummy destructor for the class.

```
void initialize (char *fName)
```

Loads node list from file and copies the file name to the private member fileName of the class.

```
int SPF (int startNode)
```

This function computes the shortest path to all the destination node and stores them in the routing table list. For a detailed description look below at the member function documentation.

```
int getInterface (char *path, int &iFaceID)
```

This function gets the interface id from the path.

```
int updateCost (int &toNID, int &tCost, int isStartNode = 0)
```

This is the function which updates the cost of the nodes in the cost list. For a detailed description look below at the member function documentation.

```
void dumpList ()
```

Dumps the routing table list of the start node to standard output.

```
char** getRoute (int toNodeID)
```

Returns paths to the destination node id.

Private Attributes

```
class allNodeList nodeList
```

This describes the input node list.

class costList cList

This is the cost ordered list used to arrange nodes in ascending order of the cost required to arranged the nodes.

class routingTable rTableList

Stores the routes to various destination nodes.

char fileName [20]

File name which describes the network connectivity.

Member Function Documentation

int dijkstraAlgorithm::SPF (int startNode)

This function computes the shortest path to all the destination node and stores them in the routing table list. The start node is given as an input to the function and the shortest path from that node to all other nodes are computed by this function

int dijkstraAlgorithm::updateCost (int & toNID, int & tCost, int isStartNode = 0)

This is the function which updates the cost of the nodes in the cost list. This function gets the first node from the cost list and its cost. Then it looks into the node list finds the list of destination node that can be reached from this node and updates the cost for those nodes

MYTEALGORITHM CLASS REFERENCE

This class extends the digkstra algorithm by finding route from one node to the all other nodes in a network, by finding the shortest path and also finding an alternate path.

Public Methods

myTEAlgorithm ()

The default constructor that makes the critical links to null. criticalLinks is a double dimensional character array.

void initialize (char* fName)

Loads node list from file, copies the file name to the private member fileName of the class and initializes the critical link pointer to NULL.

~myTEAlgorithm ()

The destructor for the class. For a detailed description look below at the member function documentation.

int alternatePath (int startNode)

This function computes the alternate path to all the destination node and stores them in the routing table list. For a detailed description look below at the member function documentation.

int updateRoutes ()

This function installs the computed routes to destination nodes in the routing table.

int getInterface (char *path, int &iFaceID)

This function gets the interface id from the path.

int updateCost (int &toNID, int *tCost, int isStartNode = 0)

This is the function which updates the cost of the nodes in the cost list. For a detailed description look below at the member function documentation.

void dumpList ()

Dumps the routing table of the start node to standard output.

int setPaths ()

This function sets the path in the right order according to the path cost and the number of critical links present in each path. For a detailed description look below at the member function documentation.

int loadCriticalLinksFromFile (int &numberOfPaths)

This function loads critical link from the file criticalLink.cl. It also returns the number of critical links added from the file.

int isCriticalLinkPresent (char *path, int numberOfCriticalLink)

Checks if critical links are present in the path given as input and also the number of critical links present. For a detailed description look below at the member function documentation.

char getRoute (int toNodeID)**

Returns paths to the destination node id.

Private Attributes

class allNodeList nodeList

A Class which holds the node list. For a detailed description look below at the member data documentation.

class costList cList

A Class containing a list of node sorted by cost. For a detailed description look below at the member data documentation.

class routingTable rTableList

Stores the final routing table. For a detailed description look below at the member data documentation.

char criticalLinks**

Stores the critical links in this variable. For a detailed description look below at the member data documentation.

char fileName [20]

Stores the File Name. For a detailed description look below at the member data documentation.

Member Function Documentation

myTEAlgorithm::~~myTEAlgorithm ()

The destructor for the class. This destructor calls the destructor off all other classes that were declared as private members of this function and also deletes the critical links

int myTEAlgorithm::alternatePath (int startNode)

This function computes the alternate path to all the destination node and stores them in the routing table list. The start node is given as an input to the function and the paths from that node to all other nodes, using the alternate path routing algorithm is computed by this function

int myTEAlgorithm::isCriticalLinkPresent (char * path, int numberOfCriticalLink)

Checks if critical links are present in the path given as input and also the number of critical links present. This function takes the path as the input in the form as a string and returns the number of critical links, if no critical links are present then this function returns a zero

int myTEAlgorithm::setPaths ()

This function sets the path in the right order according to the path cost and the number of critical links present in each path. This function rearranges the paths in the cost list. It sorts the path in ascending order according to the number of critical links and then sorts the paths with the same number of critical links according to the cost of each path

int myTEAlgorithm::updateCost (int & toNID, int * tCost, int isStartNode = 0)

This is the function which updates the cost of the nodes in the cost list. This function gets the first node from the cost list and its cost. Then it looks into the node list finds the list of destination node that can be reached from this node and updates the cost for those nodes

Member Data Documentation

`class costList myTEAlgorithm::cList [private]`

A Class containing a list of node sorted by cost. This class stores the cost required to reach a node from start node and also the path for it

`char ** myTEAlgorithm::criticalLinks [private]`

Stores the critical links in this variable. The critical links are loaded from the file and are stored in this variable

`char myTEAlgorithm::fileName[20] [private]`

Stores the File Name. Stores the file name where the network is stored

`class allNodeList myTEAlgorithm::nodeList [private]`

A Class which holds the node list. This is a class in which a node list from a test file is taken and put into a link list

`class routingTable myTEAlgorithm::rTableList [private]`

Stores the final routing table. Stores the routes to different destination nodes in a network along with their costs

COSTLIST CLASS FOR MYTEALGORITHM REFERENCE

This is a class which has a pointer to the first node of the cost list and this also stores the start node from where the path to other nodes are to be computed, using the alternate path routing algorithm. This class has a list of functions which helps in computing the path, using the alternate path routing algorithm. This class in short maintains a list of node and the path in which that node can be reached from the start node and the cost of that path

Public Methods

`costList ()`

The default constructor which makes the list head NULL.

`~costList ()`

This is the destructor for the class and it deletes the complete list.

`int addNode (int nID, int tCost[])`

Adds node to the list end.

`int deleteNode (int nID)`

Deletes the node whose node id is equal to nID from this list. For a detailed description look below at the member function documentation.

`void swapNode (struct costOrderedList *a, struct costOrderedList *b)`

This function swaps the 2 given input nodes.

int sortEntireList ()

Sorts The Entire List according to the cost. If there are multiple costs then the least among that is taken for sorting.

int sortList ()

This function sorts the list in ascending order according to the cost of each node. If there are multiple costs then the least among that is taken for sorting. For a detailed description look below at the member function documentation.

int setPath (int nID, char *path, int cost, int addPath = 0)

This function sets a path to the node nID. For a detailed description look below at the member function documentation.

int initializeList (int startNode, int numberOfNodes, int *nodeList = NULL)

Initializes the list. The list is created here and initialized. For a detailed description look below at the member function documentation.

int* getTopNodeCost (int &nID)

Gets the top node id and the cost of the node. For a detailed description look below at the member function documentation.

int* getNodeCost (int nID)

Gets the cost of the node nID. For a detailed description look below at the member function documentation.

int updateCost (int nID, int newTCost[])

Changes the cost of the node to newTCost. For a detailed description look below at the member function documentation.

bool isEmpty ()

Returns true if the list is empty else returns a false.

char getPath (int nID)**

Gets the path through which the node nID can be reached from the start node. returns a null if nID is not in the list.

bool isNodePresent (int nID)

Checks wheather the node nID is present in the list or not.

void addTopNodeToEnd ()

This function adds the top node of the list to the end of the node.

int moveNodeNextToTop (int nID)

Let the node with node id==nID, in the list be named as node1. This node1 is moved next to the top node in the cost list.

`int isNodeInSortList (int nID)`

This function is used to tell whether the node with node id == nID is present in the sort list part of the list.

`int isPathPresent (int nID, char *path)`

This function returns a true when the path given as input parameter is already present for the node nID.

`int getMinimumCost (int nID, int &cost)`

Returns the minimum cost by which a node with nID can be reached from the start node.

`void setStartNode (int startNode)`

The start nodes is set by this function.

`int getStartNode ()`

This function returns the start node.

`void dumpList ()`

Prints the list to the standard output.

Private Attributes

`struct costOrderedList* listHead`

Pointer to the first node of the list.

`int sNode`

Start node of the Alternate Path Routing algorithm.

Member Function Documentation

`int costList::deleteNode ( int nID )`

Deletes the node whose node id is equal to nID from this list. This function returns a one when the node id is found and is successfully deleted else it returns a zero

`int * costList::getNodeCost ( int nID )`

Gets the cost of the node nID. The cost of the node that is to be returned is returned as an integer array. If the node is not present then a NULL pointer is returned by the function

`int * costList::getTopNodeCost ( int & nID )`

Gets the top node id and the cost of the node. The cost of the node that is to be returned is returned as an integer pointer, while the node id is returned via the variable nID. If the node is not present then a NULL pointer is returned by the function

```
int costList::initializeList ( int startNode, int numberOfNodes, int * nodeList = NULL )
```

Initializes the list. The list is created here and initialized. The nodes are created with node id given by the node list or a list of nodes with node id from one to total number of nodes is created when the nodeList is left null. Node list is an optional parameter. The cost for all the nodes except the start node is set to infinity, the cost for the start node is set to 0

```
int costList::setPath ( int nID, char * path, int cost, int addPath = 0 )
```

This function sets a path to the node nID. The path is added to the existing path if add path flag is set else the existing paths are deleted and the path given as input is installed

```
int costList::sortList ( )
```

This function sorts the list in ascending order according to the cost of each node. If there are multiple costs then the least among that is taken for sorting. This function does not sort the entire list it sorts only the portion of the list before the start node. The rest of the list is left as such

```
int costList::updateCost ( int nID, int newTCost[] )
```

Changes the cost of the node to newTCost. If the node is not present then a 0 is returned by the function indicating an error else a one is returned

NODE STRUCT REFERENCE

This is a structure that describes the node. This is a self referential structure and it describes the node and along with its interface

Public Attributes

```
int nodeID
```

ID OF THE NODE.

```
int edge
```

Specifies whether the node is edge node(1) or LSR(0).

```
class nodeAddress nAddress
```

Heirarchical address of the node.

```
class allInterface intf
```

List of interface attached to that node.

```
struct node* next
```

Pointer to next node.

ALLNODELIST CLASS REFERENCE

Describes the network in a whole. This class describes the nodes and its connectivity. The network connectivity is loaded from the file into a linked list. The list describes the entire network

Public Methods

allNodeList ()

The default constructor making the head of the list NULL.

~allNodeList ()

The destructor for the class which deletes all the nodes defined for the class.

void addNode (int nID, int ed, int add1, int add2, int add3, int add4, int linkID, int dID, int lCapacity, int cost)

Adds a node to the end of the node list. For a detailed description look below at the member function documentation.

int addInterface (int nID, int linkID, int dID, int lCapacity, int cost)

This function adds an interface to the node with node id == nID. For a detailed description look below at the member function documentation.

int setLinkUtilized (int linkID, int changeInLinkUtilized, double cTime = -1)

Sets the link capacity currently utilized for the link with linkID == linkID and it also sets the time at which the link capacity got updated. For a detailed description look below at the member function documentation.

int setLinkUtilized (int nID, int dID, int changeInLinkUtilized, double cTime = -1, int writeCriticalLinksToFile = 0)

Sets the link capacity currently utilized for the link which connects the 2 nodes nID and dID and it also sets the time at which the link capacity got updated. For a detailed description look below at the member function documentation.

int getLinkUtilized (int linkID, int &linkUtilized)

Returns the link capacity currently utilized for the link with linkID == linkID. For a detailed description look below at the member function documentation.

int getLinkUtilized (int nID, int dID, int &linkUtilized)

Returns the link capacity currently utilized for the link which connects the 2 nodes nID and dID. For a detailed description look below at the member function documentation.

int getLinkBandwidth (int linkID, int &linkBandwidth)

Returns the bandwidth of the link with linkID == linkID. For a detailed description look below at the member function documentation.

int getLinkBandwidth (int nID, int dID, int &linkBandwidth)

Returns the bandwidth of the link which connects the 2 nodes dID and nID. For a detailed description look below at the member function documentation.

char* writeToFile ()

Writes the entire node list into a character array and returns that array. The node list is written in the array in the form that it can be written into the file directly.

int getFromFileNodeList (char *fName)

Loads the node list from the file. The name of the file is gives as input to the function.

struct node* getAllNodeListHandle ()

This function returns a handle to the node list head.

void setAllNodeListHandle (struct node *temp)

Sets the head of the list used when list is loaded in some other program or file.

int getNumberOfNodes ()

Returns the total number of nodes in the list.

struct interfac* getInterfacHandle (int nID)

This function returns a handle to the interface list of node with node id==nID.

int getLinkID (int fNode, int tNode, int &linkID)

This function gets the link id of the link which connects the nodes fNode and tNode. For a detailed description look below at the member function documentation.

int getCost (int fNode, int tNode, int &cost)

This function gets the cost of the link which connects the fNode and the tNode. For a detailed description look below at the member function documentation.

int* getNodeList ()

This function returns a list of nodes in to form of an integer array.

int dumpList ()

Prints the entire list on the standart output.

Private Attributes

struct node* firstNode

Head of the node List.

Member Function Documentation

`int allNodeList::addInterface ( int nID, int linkID, int dID, int lCapacity, int cost )`

This function adds an interface to the node with node id == nID. The characteristics of the link are given as input. This function returns a zero if no node in the list has node id == nID

`void allNodeList::addNode ( int nID, int ed, int add1, int add2, int add3, int add4, int linkID, int dID, int lCapacity, int cost )`

Adds a node to the end of the node list. The characteristics of the node are given as input parameters

`int allNodeList::getCost ( int fNode, int tNode, int & cost )`

This function gets the cost of the link which connects the fNode and the tNode. This function returns the cost via the variable cost passed as the input parameter. This function returns a zero if there is no link between the 2 nodes

`int allNodeList::getLinkBandwidth ( int nID, int dID, int & linkBandwidth )`

Returns the bandwidth of the link which connects the 2 nodes dID and nID. This function returns the bandwidth via the variable linkBandwidth, which passed as the input parameter. This function returns a zero if there is no link with link id == linkID in the network

`int allNodeList::getLinkBandwidth ( int linkID, int & linkBandwidth )`

Returns the bandwidth of the link with linkID == linkID. This function returns the bandwidth via the variable linkBandwidth, which passed as the input parameter. This function returns a zero if there is no link with link id == linkID in the network

`int allNodeList::getLinkID ( int fNode, int tNode, int & linkID )`

This function gets the link id of the link which connects the nodes fNode and tNode. This function returns the link id via the variable linkID, which passed as the input parameter. This function returns a zero if there is no link between the 2 nodes

`int allNodeList::getLinkUtilized ( int nID, int dID, int & linkUtilized )`

Returns the link capacity currently utilized for the link which connects the 2 nodes nID and dID. This function returns the link capacity currently utilized via the variable linkUtilized, which passed as the input parameter. This function returns a zero if there is no link which connects the 2 nodes in the network

`int allNodeList::getLinkUtilized ( int linkID, int & linkUtilized )`

Returns the link capacity currently utilized for the link with linkID == linkID. This function returns the link capacity currently utilized via the variable linkUtilized, which passed as the input parameter. This function returns a zero if there is no link with link id == linkID in the network

```
int allNodeList::setLinkUtilized ( int nID, int dID, int changeInLinkUtilized, double
cTime = -1, int writeCriticalLinksToFile = 0 )
```

Sets the link capacity currently utilized for the link which connects the 2 nodes nID and dID and it also sets the time at which the link capacity got updated. Apart from that it identifies the critical links in a network and writes them into a file if write critical links to file flag is set. This function returns a zero if there is no link with link id == linkID in the network

```
int allNodeList::setLinkUtilized(int linkID, int changeInLinkUtilized, double cTime = -1 )
```

Sets the link capacity currently utilized for the link with linkID == linkID and it also sets the time at which the link capacity got updated. This function returns a zero if there is no link with link id == linkID in the network

INTERFAC STRUCT REFERENCE

This structure stores the characteristics of the interface. Each member of this node describes the characteristics of the interface. This is a self referential structure, which is used in the allInterfaceList class

Public Attributes

```
int linkID
```

The id of the link.

```
int destID
```

The immediate destination that can be reached through that link.

```
int cost
```

The cost of traversing through that link.

```
int linkCapacity
```

The bandwidth of that link in kb.

```
int linkUtilized
```

The amount of bandwidth utilized at time time_t in kb.

```
time_t currentTime
```

Stores the time at which the linkutilized was updated.

```
struct interfac* next
```

Pointer to the next element in the linked list.

ALLINTERFACE CLASS REFERENCE

The all interface class stores or has the list of interface for each node. Each interface has the details of the interface and all the list of the interface are stored in the form of a

linked list. this class also list a set of function which describes the behaviour of the interface

Public Methods

`allInterface ()`

The default constructor that assigns a null pointer to the head of the list pointer.

`~allInterface ()`

This is the destructor for the class that removes all the node in the list.

`void addInterface (int linkID, int dID, int lCapacity, int cost = 1)`

Adds an interface to the list end. For a detailed description look below at the member function documentation.

`int setLinkUtilized (int linkID, int changeInLinkutilized, double cTime = -1)`

This function sets the linkUtilized and the time in which this change occurred. For a detailed description look below at the member function documentation.

`int getLinkUtilized (int linkID, int &linkUtilized)`

Returns the link utilized of the link whose link id is equal to linkID. For a detailed description look below at the member function documentation.

`int getLinkBandwidth (int linkID, int &linkBandwidth)`

Returns the bandwidth of the link whose link id is equal to linkID. For a detailed description look below at the member function documentation.

`char* writeToFile ()`

This function returns the complete list in the form of a character array or string. For a detailed description look below at the member function documentation.

`int dumpInterfaceList ()`

Prints the interface list to the stdout.

`int getLinkID (int dID, int &linkID)`

This function returns the linkID whose destination id is dID. For a detailed description look below at the member function documentation.

`int getCost (int dID, int &cost)`

This Function returns the cost of the link through which dID can be reached. For a detailed description look below at the member function documentation.

`struct interfac* getAllInterfacListHandle ()`

This function returns the first pointer.

Private Attributes

struct interfac* first

The pointer to Head of the linked list.

Member Function Documentation

void allInterface::addInterface (int linkID, int dID, int lCapacity, int cost = 1)

Adds an interface to the list end. This function adds a new element to the end of the list. The characteristics of the new element are passed as input parameters

int allInterface::getCost (int dID, int & cost)

This Function returns the cost of the link through which dID can be reached. This function returns the cost of the link, the return variable for the function is int &cost. This function returns a zero meaning a error that the dID could not be found in the interface

int allInterface::getLinkBandwidth (int linkID, int & linkBandwidth)

Returns the bandwidth of the link whose link id is equal to linkID. The return variable for the function is int &linkBandwidth. This function returns a zero meaning a error that the linkID could not be found in the interface

int allInterface::getLinkID (int dID, int & linkID)

This function returns the linkID whose destination id is dID. This function returns the linkID of the link which has a destination dID, the return variable for the function is int &linkID. This function returns a zero meaning a error that the dID could not be found in the interface

int allInterface::getLinkUtilized (int linkID, int & linkUtilized)

Returns the link utilized of the link whose link id is equal to linkID. The return variable for the function is int &linkUtilized. This function returns a zero meaning a error that the linkID could not be found in the interface

int allInterface::setLinkUtilized(int linkID, int changeInLinkUtilized, double cTime = -1)

This function sets the linkUtilized and the time in which this change occurred. The input to the function is the change in link utilized and the time in which this change happened. This same function is used to add or subtract from the link utilization. When this function returns a zero it means that the linkID could not be found in the interface list

char * allInterface::writeToFile ()

This function returns the complete list in the form of a character array or string. This function returns the entire list in the form of a string. The format of the string is intf1&intf2&... where intf1 = linkID\$DestID\$Cost\$linkCapacity. Note that the link utilization and the time at which the utilization are not added to the string as these data are not required to regenerate the topology.

RTABLE STRUCT REFERENCE

List describing the routing table of a node. This structure stores the routing table for a given node in a network

Public Attributes

int toNodeID

Destination id that can be reached via this route.

int cost [noOfPath]

Cost of the route. For a detailed description look below at the member data documentation.

int interfaceID [noOfPath]

Interface id for each path. For a detailed description look below at the member data documentation.

int bandWidth [noOfPath]

Bandwidth left unutilized in each path. For a detailed description look below at the member data documentation.

char* path [noOfPath]

The list of path through which the destination nodes can be reached.

struct rTable* next

Pointer to the next route in the list.

Member Data Documentation

int rTable::bandWidth[noOfPath]

Bandwidth left unutilized in each path. This value is currently not used

int rTable::cost[noOfPath]

Cost of the route. There can be more than one path from a source to a destination that is the reason we have the cost in an array

int rTable::interfaceID[noOfPath]

Interface id for each path. The interface through which the destination can be reached

ROUTINGTABLE CLASS REFERENCE

Class which describes the behaviour of the routing table. This class stores the output of the routing algorithms. An instance of this class will be defined in all the routing algorithms to store the output

Public Methods

routingTable ()

The default constructor making the routing list head as null.

~routingTable ()

The destructor that deletes all the routes.

int addRoute (int toNID, int cost[], int iFaceID[], char *path[noOfPath], int *bandWidth = NULL)

This function adds a route. The characteristics of the route are given as input to the function.

int deleteRoute (int toNID, char *path)

This function removes a path or all the paths to the destination node toNID. For a detailed description look below at the member function documentation.

int getInterfaceID (int toNID, int &iFaceID, int pathNumber = 0)

This function is used to get the interface id through which the destination node toNID can be reached via the path given by the pathNumber. For a detailed description look below at the member function documentation.

char** getRoute (int toNID)

Gets the set of all routes by which the destination node toNID can be reached. For a detailed description look below at the member function documentation.

void dumpList ()

Dumps the entire routing table into the standard output.

int dumpToFile (int sNode, char *fName)

Dumps the routing table list into the file. The file name is the input to this function.

void setNodeID (int nID)

Sets the value of the from node id (nodeID) in the class private variable.

int getNodeID ()

This function returns the nodeID for which the routing table is computed.

int deleteAllRoute ()

Deletes all routes to the destination. For a detailed description look below at the member function documentation.

Private Attributes

int nodeID

The nodeID for which the routing table is computed.

struct rTable* first
 Pointer to the first route in the list.

Member Function Documentation

int routingTable::deleteAllRoute ()

Deletes all routes to the destination. This function is used when it is called by a new node to compute the routing table for that node. The present routes stored for the previous nodes are deleted

int routingTable::deleteRoute (int toNID, char * path)

This function removes a path or all the paths to the destination node toNID. The char *path is used as a flag which when null indicates that all the path should be deleted for that route, else the path given as input by the char *path will alone be deleted. If the destination nodeID == toNID is not in the list then a 0 is returned by the function else a one indicating the path is deleted is returned

int routingTable::getInterfaceID (int toNID, int & iFaceID, int pathNumber = 0)

This function is used to get the interface id through which the destination node toNID can be reached via the path given by the pathNumber. We take the path number as the input as we can have multiple paths by which a destination node can be reached and each node can have different interface thorough which the destination node can be reached. If the destination nodeID == toNID is not in the list then a 0 is returned by the function else a one returned

char ** routingTable::getRoute (int toNID)

Gets the set of all routes by which the destination node toNID can be reached. The return type of this function is an array of character pointer or rather an array of strings describing the path. If the destination nodeID == toNID is not in the list then a NULL pointer is returned by the function

OTHERDEFINITIONS

Other sets of common definitions used throughout the program is given below

```
#ifndef charlen
#define charlen 300
#endif

#ifndef INFINITY
#define INFINITY 30000
#endif

#ifndef noOfPath
#define noOfPath 10
#define pathLen 35
#endif
```

```
#include <iostream.h>
#include <stdio.h>
#include <fstream.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

APPENDIX 2

LIST OF ACRONYMS USED IN THE REPORT

Acronyms	Expansion
APC	Alternate Path Cost
AS	Autonomous System
ATM	Asynchronous Transfer Mode
CR-LDP	Constraint based LDP
d	Total number of paths to a destination
Diffserv	Differentiated Service
ER-LSP	Explicitly Routed LSP
IETF	Internet Engineering Task Force
IGP	Internet Gateway Protocols
IGP	Interior Gateway Protocol
Intserv	Integrated service
ISP	Internet Service Provider
LDP	Label Distribution protocol
LSA	Link State Advertisements
LSP	Label switched paths
LSR	Label Switched Routers
MIRA	Minimum Interference Routing with Application to MPLS Traffic Engineering
MPLS	Multi Protocol Label Switching
OSPF	Open Shortest Path First
RIP	Routing Information Protocol
RSVP	Resource reservation protocol
SLA	Service level agreements
SPC	Shortest path cost
TE	Traffic Engineering
VCI	Virtual Control Identifier
VPI	Virtual Path Identifier

VITA

Graduate College
University of Nevada, Las Vegas

Shyam Subramanian

Local address:
1165 Maryland Circle #1
Las Vegas, NV 89119

Home Address:
New No: 184, Old No: 111/1,
Lake View Road,
West Mambalam,
Chennai,
Tamil Nadu,
India
ZIP – 600 033

Degrees:
Bachelor of Engineering, Electronics and Instrumentation Engineering, 2000.
Government College of Technology, Bharathiar University, India

Thesis Title: Alternate Path Routing Algorithm for Traffic Engineering in the Internet.

Publications:

- “Alternative Path Routing Algorithm for Traffic Engineering”, ICSENG 2002, *Shyam Subramanian and Venkatesan Muthukumar*.
- “Distance Spectrum of Nodes in Star Graphs”, PDPTA 2002: 1591-1595, L. Wang, S. Subramanian, Shahram Latifi, Pradip K. Srimani

Thesis Examination Committee:

Chairperson, Dr. Muthukumar Venkatesan, Ph.D.
Committee Member, Dr. Henry Selvaraj, Ph.D.
Committee Member, Dr. Emma Regentova, Ph.D.
Graduate Faculty Representative, Dr. John Wang, Ph.D.