DELAY-CONSTRAINED ONLINE MULTICAST ROUTING HEURISTICS FOR REAL-TIME COMMUNICATION

by

Murat Şensoy

B.S., Chemical Engineering, Boğaziçi University, 2001

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2004

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Assist. Prof. Murat Zeren for his guidance and his invaluable support during my graduate study. I am also profoundly thankful for my family's patience and encouragement not only during my M.S. study but also all through my life.

ABSTRACT

DELAY-CONSTRAINED ONLINE MULTICAST ROUTING HEURISTICS FOR REAL-TIME COMMUNICATION

Real-time communications is becoming essential. As applications of real-time communications with multiple participants become widespread, multicast routing is becoming more popular. Although there is a vast amount of literature on multicast routing, online multicast routing is a relatively unexplored area. In online multicast sessions, multicast group members join and leave the multicast session frequently. When real-time communications is concerned, online multicast routing becomes tougher. Real-time communications imposes its additional constraints, such as end-to-end delay, to multicast routing. Rigid delay constraints of real-time communications restrict computational effort for online update of multicast trees.

The motivation of this work is to develop fast and efficient online multicast routing heuristics that can be used in real-time communications for online computation of delay-constrained multicast trees throughout a multicast session without significantly sacrificing optimality for speed. For this purpose, fast and efficient delay-constrained least-cost path heuristics are also required to be used as an underlying unicast routing heuristic in multicast routing.

In this thesis, two online multicast routing heuristics and one unicast routing heuristic is proposed. Performance of proposed heuristics is evaluated using online and offline multicast routing heuristics from literature. Simulations indicate that our heuristics offer the best balance among performance and computational effort.

ÖZET

GERÇEK-ZAMANLI İLETİŞİM İÇİN GECİKME-KISITLAMALI ÇEVRİMİÇİ ÇOKLU-YAYIN YOL-ATAMA BİLİŞSEL YÖNTEMLERİ

Gerçek-zamanlı iletişim önem kazanmakta. Çoklu kullanıcılı gerçek-zamanlı iletişim uygulamaları yaygınlaşırken, çoklu-yayın yol-atama daha popüler olmaktadır. Çoklu-yayın yol-atama literatürü oldukça geniş olmasına rağmen, çevrimiçi çoklu-yayın yol-atama göreceli olarak fazla çalışılmamış bir alandır. Çevrimiçi çoklu-yayın oturumlarında çoklu-yayın grup üyeleri oturuma sıklıkla girer ve çıkar. Gerçek-zamanlı iletişim söz konusu olduğunda, çevrimiçi çoklu-yayın yol-atama daha zor olmaktadır. Gerçek-zamanlı iletişim uçtan-uca gecikme gibi ilave kısıtlamaları çoklu-yayın yol-atama üzerine empoze eder. Gerçek-zamanlı iletişimin katı gecikme kısıtlamaları çoklu-yayın ağaçlarının yenilenmesi için gösterilecek hesapsal çabayı sınırlar.

Bu çalışmanın motivasyonu gerçek-zamanlı iletişimde oturum boyunca gecikme-kısıtlamalı çoklu-yayın ağaçlarının çevrimiçi hesaplamalarında, hız için eniyilikten belirgin bir şekilde ödün vermeksizin kullanılabilecek hızlı ve etkili çevrimiçi çoklu-yayın yol-atama buluşsal yöntemleri geliştirmektir. Bu amaç için çoklu-yayın yol-atama altyapısında kullanılabilecek hızlı ve etkili gecikme-kısıtlamalı tekli-yayın yol-atama yöntemleri ayrıca gereklidir.

Bu tezde, iki çevrimiçi çoklu-yayın yol-atama bilişsel yöntemi ve bir tekli-yayın yol-atama bilişsel yöntemi önerilmiştir. Önerilen bilişsel yöntemlerin başarımı literatürdeki çevrimiçi ve çevrimdışı bilişsel yöntemler kullanılarak değerlendirilmiştir. Benzetimlerin de işaret ettiği üzere, önerilen bilişsel yöntemler başarım ile hesapsal çaba arasındaki en iyi dengeyi sunmaktadır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

$C(e)$	Cost of the edge e
d	Destination node
$d(u,v)$	Euclidian distance between nodes u and v
$D(e)$	Delay of the edge e
e	An edge
e'	Reverse of an edge e
E	Set of edges
G	Graph
k	Parameter of k-shortest path algorithm
K	Scale factor for random graph model
L	Maximum possible distance between two nodes in a graph
m	Multicast member size
n	Network size
P	Path in a graph
$Pr()$	Probability function
$P_{set}(s,d)$	Set of paths from s to d
$P^{\Delta}_{set}(s,d)$	Set of delay-constrained paths from s to d
P_T	Path in a tree
r_i	i^{th} request in the request vector
R	Request vector
s	Source node
T	Multicast tree
V	Set of nodes
W	Set of multicast members
α	Tunable parameter of Waxman's random graph model
β	Tunable parameter of Waxman's random graph model
γ	Tunable parameter of random request model
Δ	Delay constraint

ε	Number of edges in a graph
$\overline{\varepsilon}$	Mean degree of a node
Θ	Number of marked nodes in an m-region of CRCDM
κ	Parameter of CRCDM
ρ	Utilization threshold for rearrangements in CRCDM
τ	$InflationTolerance$ parameter of CoBaCROM
Ψ	Number of multicast members in an m-region of CRCDM
$\Re$	An m-region defined in CRCDM

ARIES	A rearrangeable online multicast routing heuristic
BSMA	Bounded shortest multicast algorithm
CC	Cost competitiveness
CDKS	Constrained Dijkstra algorithm
CoBaCROM	Proposed heuristic for rearrangeable online multicasting
CPR	Change per rearrangement
CPRQ	Change per request
CRCDM	A rearrangeable constrained online multicasting heuristic
DDCLCMR	A delay-constrained online multicast routing heuristic
DeCoNROM	Proposed heuristic for nonrearrangeable online multicasting
DMBDCLC	Proposed delay-constrained least-cost path heuristic
FBSMA	Fast-BSMA
GPR	Gain per rearrangement
ITU	International Telecommunication Union
LC	Least cost tree produced by Dijkstra algorithm
LD	Least delay tree produced by Dijkstra algorithm
NR	Number of rearrangements
NSPR	Number of subtrees per rearrangement
SMT	Steiner minimal tree

1. INTRODUCTION

Real-time communications is becoming essential. Some well-known applications of real-time communications are teleconferencing, videoconferencing, e-learning, and radio/tv broadcasting services over IP networks. A substantial portion of those applications concerns multiple receivers. Multicasting well fits into basic requirements of such group communications. Hence, multicasting is becoming more and more popular, as real-time communications with multiple participants becomes widespread.

Construction of optimal multicast trees for static multicast groups can be modeled as the *NP-complete Steiner problem* [1, 2]. However, real-time communications imposes its additional constraints such as end-to-end delay as a part of its QoS requirements. This QoS constraint, end-to-end delay, requires the computation of delay-constrained least-cost multicast trees. Network delay is the composition of different delay components such as transmission delay, queuing delay and processing delay. Processing delay may be the principle component of delay if computational requirements are high on the router site. When real-time communications is concerned, delay bounds are tight and computation of delay-constrained least-cost multicast trees is required to create acceptable multicast trees with respect to predefined delay bounds. However, computation of such a multicast tree is NP-complete and it may result in unacceptable delays at the start or during a multicast session for large networks. Delay sensitivity of those applications requires limiting the processing time for the computation of such trees by using heuristics instead of explicit algorithms and the result is a trade of between optimality and computational effort. Some of those heuristics are explained shortly in this thesis and their time complexities are usually proportional to optimality of the resultant multicast trees.

Many of the multicasting applications also require the network to support online or dynamic multicast sessions, in which the membership of the multicast group changes frequently. In order to support online multicast sessions, an existing multicast tree should be altered to accommodate membership changes throughout a multicast session

as members join and leave. There is a vast literature on the subject of establishing static or offline multicast trees in point-to-point networks. However, online multicasting is a relatively unexplored area of research especially for real-time communications.

The purpose of this thesis is to develop fast and efficient heuristics that can be used in real-time communications for the computation of delay-constrained online multicast trees without significantly sacrificing optimality of resulting multicast tree for speed. Contribution of this thesis is depicted in Figure 1.1.

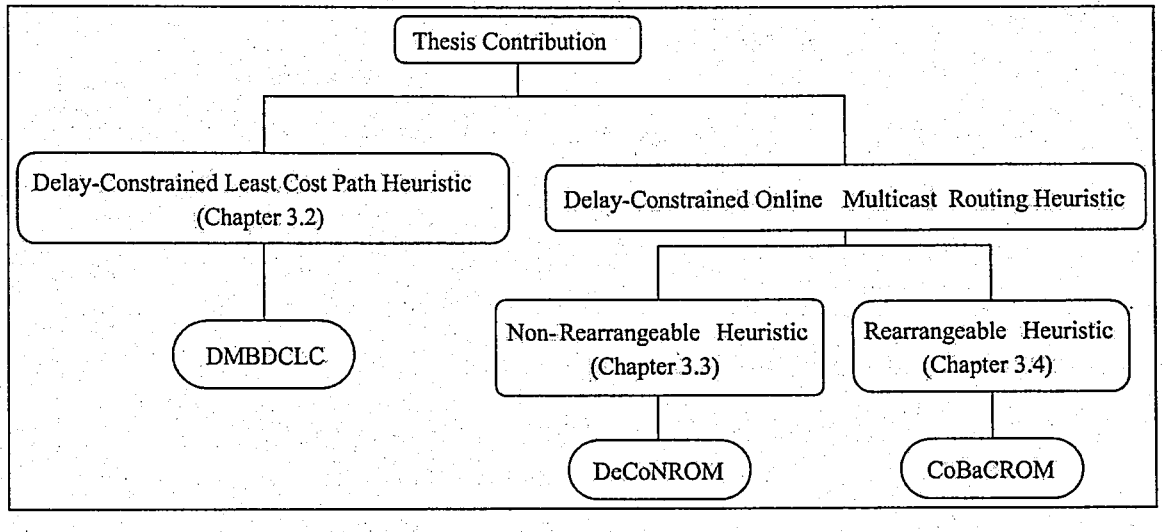

Figure 1.1. Contribution of thesis

2. MULTICAST ROUTING AND RELATED WORK

Multicasting is the capability of delivering a packet to multiple receivers. A multicast tree is used for this purpose. In a multicast session, exactly one copy of the packet traverses each link in the multicast tree. There are numerous examples of multicast applications. Some of those applications include videoconferencing, teleconferencing, whiteboard applications, e-learning applications, and radio/tv broadcasting over IP networks.

Multicast routing heuristics can roughly be classified into two categories. The first category is the *shortest path based heuristics.* These heuristics construct a multicast tree that minimizes the length of paths from the source node to each multicast group member. The other category is the *Steiner Minimal Tree (SMT) based heuristics.* The purpose of these heuristics is to minimize the total cost of the multicast tree. SMT problem is NP-hard [3]. If the set of receivers of a Steiner minimal tree includes all nodes in the network, the problem becomes a minimum spanning tree problem and it is solvable in polynomial time [4].

Figure 2.1 shows the taxonomy of multicast routing heuristics. SMT based heuristics can be further divided as online and offline heuristics. In the offline multicasting, members of the multicast group are known a priori, whereas multicast members can join or leave the multicast tree dynamically in the online multicasting, while the multicast session is in progress. Most of multicasting applications may require ability of supporting dynamic sessions. Online multicast routing heuristics concern dynamic features of multicast sessions and update multicast tree accordingly, while the multicast session is in progress. A trivial solution to dynamic multicast sessions is to rebuild the multicast tree using an offline heuristic whenever multicast group membership is changed. However, this solution results in disturbance of unchanged group members in each change of membership and may be costly in terms of computation time and packet loss.

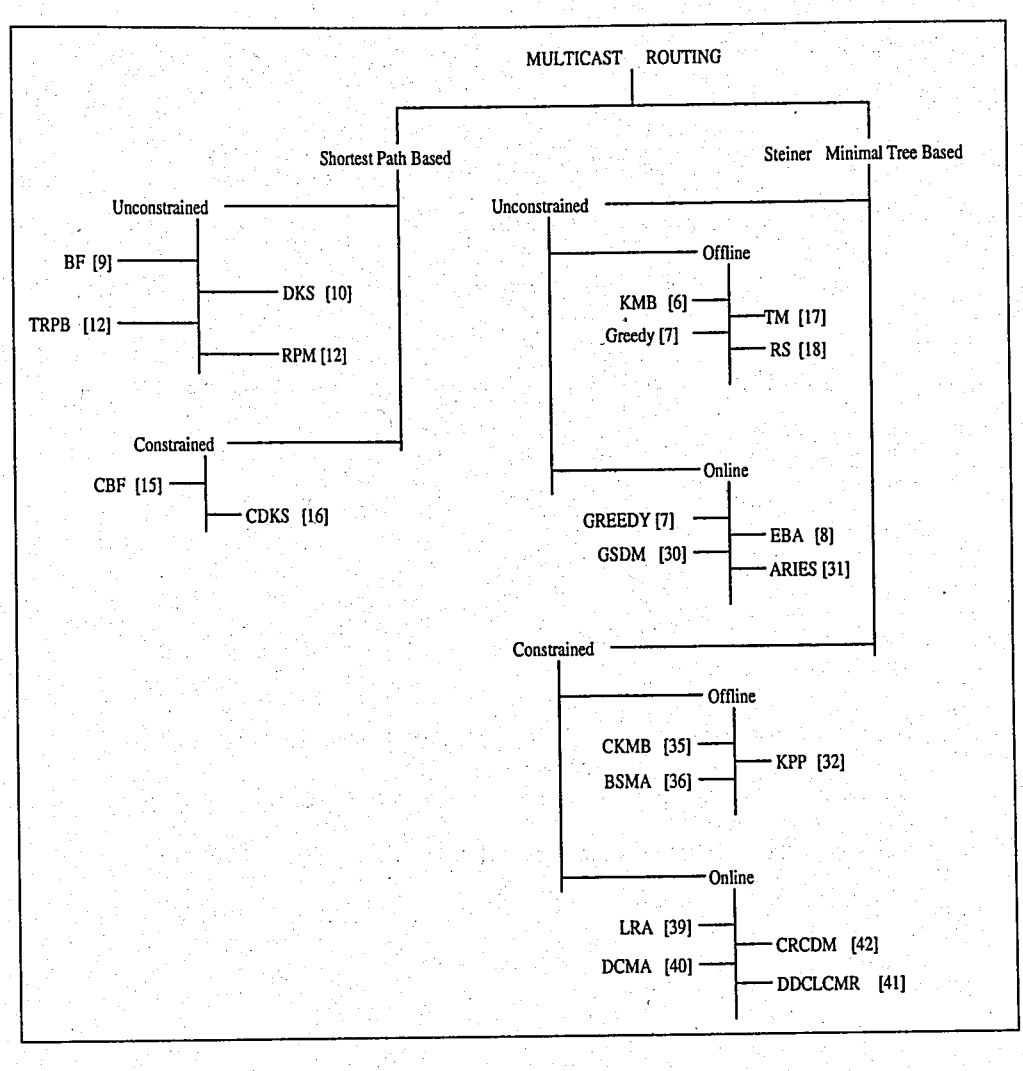

Figure 2.1. Taxonomy of multicast routing heuristics

Unlike SMT based heuristics, shortest path based heuristics concern paths to multicast group members instead of whole tree. So, they can simply add and remove paths, as group members join and leave the session. Therefore shortest path based heuristics are both offline and online heuristics. Doar and Leslie [5] investigated a naive approach that connects a joining node to the existing tree via the least cost path from the source node in a dynamic multicast session. They made simulations on 100-node networks. Their simulations showed that this naive approach constructs trees that are on the average 50 per cent more expensive than the costs of multicast trees constructed using KMB heuristic [6], which is an unconstrained SMT based multicast routing heuristic (see Section 2.2.1).

Waxman was one of the first researchers investigating online multicasting problem [7, 8]. Waxman partitioned online heuristics into two categories as rearrangeable and non-rearrangeable. Rearrangeable heuristics allow reconfiguration, when degradation of the tree exceeds some limit. This degradation is originated from join/leave requests. Non-rearrangeable heuristics do not allow reconfiguration.

There may also be heuristics and categories other than those shown in Figure 2.1. However, those heuristics and categories in the figure best fit to the scope of this thesis and are explained in the following sections.

2.1. Shortest Path Based Heuristics

The objective of shortest path based multicast routing heuristics is to minimize the cost of each path from the source to each destination individually. In order to construct shortest path trees, shortest paths from source node to destination nodes are combined. Shortest path based multicast routing heuristics are further classified as unconstrained and constrained.

2.1.1. Unconstrained Heuristics

There are well-known shortest path algorithms to construct unconstrained shortest path trees; those are Bellman-Ford algorithm (BF) [9] and Dijkstra algorithm [10]. Those algorithms compute the optimal solution in polynomial time. Time complexity of BF is $O(n^3)$ and that of Dijkstra is $O(n^2)$, where n is the number of nodes in the network. Reverse path forwarding is another algorithm for the construction of unconstrained least cost paths and trees [11]. RPF works and can find optimum paths in networks with symmetric links. In RPF, each packet is forwarded from the source to the receivers using knowledge of the shortest path from the receivers back to source.

Shortest path based trees produced by BF, Dijkstra or RPF algorithms carries out broadcasting. These algorithms can be used to carry a multicast packet to all links in the network using shortest paths. However, this is not acceptable, because

unwanted multicast packets are sent to all nodes in a network. So, leaf nodes not desiring to get multicast packets should be truncated from shortest path tree. For this purpose, Deering *et al.* [12] generalized RPF algorithm to the multicast case and proposed two distributed algorithms; the truncated reverse path broadcasting (TRPB) algorithm and the reverse path multicasting (RPM) algorithm.

Link cost or end-to-end delay is usually used as metric in shortest path algorithms. Resulting tree is a least cost tree and called as LC if the metric is cost and it is a least delay tree and called as LD if the metric is end-to-end delay. LD is usually computed in cases where end-to-end delay is crucial, such as in real-time applications. If LD does not satisfy the delay constraint, no other heuristics or algorithms can do.

2.1.2. Constrained Heuristics

Multimedia and real-time multicast applications motivate the delay-constrained multicast routing research. However, finding constrained paths or constrained trees is an NP-complete problem [13, 14]. Widyono [15] proposed the Constrained Bellman-Ford (CBF), which is an optimal algorithm basing on BF algorithm. CBF finds optimum solution, but its running time grows exponentially with network size. So, CBF is not practical for real life. However, it is usually used as a comparison basis for delay constrained Steiner minimal tree heuristics.

Sun and Langendorefer [16] proposed a delay-constrained shortest path tree heuristic, Constrained Dijkstra (CDKS). CDKS runs Dijkstra algorithm using link cost metric to construct LC. If LC satisfies the delay constraint, then CDKS has the optimal tree, otherwise, CDKS runs Dijkstra algorithm again using link delay metric to construct LD. Finally, CDKS combines LC and LD by cancelling the paths from the LC that do not satisfy delay constraint and replacing them by the paths from the LD. The worst-case time complexity of CDKS is $O(n^2)$, since Dijkstra algorithm dominates the algorithm. It can be further reduced to $O(n \log(n))$ using proper data structures for Dijkstra algorithm.

2.2. Steiner Minimal Tree Based Heuristics

SMT based heuristics use the Steiner tree problem as a model for the construction of multicast trees. This technique is one of the most studied techniques for multicast tree construction. There are many algorithms and heuristics available for this class of multicast routing technique. In the following subsections, important SMT based heuristics are explained shortly.

2.2.1. Unconstrained Heuristics

Unconstrained SMT based heuristics do not attempt to optimize the end-to-end delay at all. Therefore they may not be suitable for real-time applications. However, these heuristics have come to be very popular. This set of heuristics is further divided as offline (or static) and online (or dynamic) heuristics.

2.2.1.1. Offline Heuristics. The best-known SMT based unconstrained offline heuristics were proposed by Kou, Markowski, and Berman (KMB heuristic) [6], Takahashi and Matsuyama (TM heuristic) [17], and Rayward-Smith (RS heuristic) [18].

The KMB heuristic constructs a complete graph consisting of multicast group members only. Each edge between two nodes in this graph represents the least cost path between those nodes in original graph. Then, KMB heuristic uses Prim's algorithm [4] to construct a spanning tree of all multicast group members. Lastly, edges of this tree replaced by the least cost paths to construct multicast tree. Prim's algorithm is optimal only for networks with symmetric links. Thus, the cost performance of the KMB heuristic may be degraded, if it is applied to networks with asymmetric links. The worst-case time complexity of the KMB heuristic is $O(m \cdot n^2)$, where m is the size of the multicast group and n is network size. Wall [19, 20] proposed a distributed version of the KMB heuristic. The cost of trees generated using KMB heuristic in symmetric networks is compared [5, 21] and result of this comparison is, on the average, only five per cent worse than the cost of the optimal Steiner minimal trees.

The TM heuristic starts with a tree containing only of the source node. TM iteratively adds the nearest multicast group member to the tree using the least cost path, until no multicast group member remains out of the tree. The worst-case time complexity of the TM heuristic is $O(m \cdot n^2)$.

The RS heuristic starts with a forest of trees where each multicast group member is represented by a tree. RS heuristic uses an algorithm similar to Kruskal's minimum spanning tree algorithm. It iteratively combines nearest two trees using the shortest path between the trees, until one tree is left. Rayward-Smith and Clare [22] showed that performance of RS heuristic in terms of tree cost is better than that of KMB and TM heuristics. However, RS heuristic was designed for symmetric networks, and may not produce good solutions for asymmetric networks.

Another unconstrained offline multicast routing heuristic is Greedy heuristic [7, 23]. It is similar to TM. However, it adds multicast group members to the tree randomly instead of adding nearest nodes first. It starts with a randomly chosen multicast node as initial multicast tree. Then, it iteratively chooses another random multicast group member and adds this node to the multicast tree using the shortest path from the tree to this node. This approach can give bad results but it usually gives reasonable approximations, on average not worse than 20 per cent of optimal solution [23]. The worst-case time complexity of the Greedy heuristic is $O(m \cdot n^2)$.

Jiang [24] proposed modified versions of KMB heuristic and RS heuristic that produces better trees than the trees produced by original heuristics. Jiang also proposed a distributed Steiner minimal tree heuristic in [25].

The heuristics explained until now are working only with symmetric links. A heuristic working in asymmetric networks was proposed by Ramanathan [26]. This heuristic is parameterized to select between low tree cost and fast execution time. Ramanathan showed that Dijkstra's shortest path algorithm, KMB and TM are particular cases of the proposed heuristic with different values of heuristic's parameter. There are also other SMT based heuristics to construct unconstrained multicast trees. Further

examples of those heuristics can be found in [27, 28, 29].

2.2.1.2. Online Heuristics. Waxman was one of the first researchers investigating online multicast routing problem [7, 8]. First online multicast routing heuristic proposed by Waxman is GREEDY heuristic [7]. GREEDY is a non-rearrangeable online multicast routing heuristic. In case of a join request, new multicast group member is added to multicast tree using the shortest path between the tree and the new member node. In case of a leave request, the member node to be removed is pruned only if it is a leaf node; otherwise this node becomes a relay node. Imase and Waxman proved that worst-case cost of the multicast tree produced by GREEDY heuristic is not worse than twice the cost of multicast tree produced by the best non-rearrangeable algorithm for unconstrained online multicast routing problem [8].

Edge Bounded Algorithm (EBA) is a rearrangeable online multicast routing heuristic and proposed by Imase and Waxman [8]. In EBA, partial reconfiguration of the tree is permitted, in case of a change in multicast group membership. EBA bounds the worst-case cost performance of the generated multicast tree to $4 \cdot \alpha$ times the cost of the optimal Steiner tree, where α is a user-defined variable [8]. Number of rearrangements in EBA is limited to $O(K^{\frac{3}{2}})$, where K is the number of join/leave requests. EBA creates multicast trees in a way similar to KMB heuristic. It creates a distance graph G' of multicast members. Distance graph is a complete graph where edges between any two nodes are weighted by the shortest path between the two nodes in the original graph. Then, it runs minimum spanning tree algorithm on G' and replaces edges with shortest paths to compute a tree T' connecting all of the multicast group members. For each join request, EBA connects the new member node v to the multicast tree via the closest node w in T'. EBA verifies that the added path is α-bounded bounded by ensuring that the cost of the maximum-cost edge on the path between v and any node u in T' does not exceed α times the cost of edge (v, u) in G'. If the path from v to u in the tree exceeds that limit, v and u are connected using the least cost path between v and u. For a leave request, if the degree of the node to be removed is one, it is simply pruned just as in GREEDY heuristic. If the node has

degree three or more, the node is marked as deleted and no further action is taken. If the degree of the node is two then the node is removed along with the parent and child nodes. This creates two subtrees and these two subtrees are connected using the least cost path between two subtrees. After any request, EBA verifies that T' is still an extension tree, which is a tree in which the degree of any non-multicast group member is greater than two. If the degree of a non-multicast group member node is two, this node is removed and resulting subtrees are connected similarly.

The Geographic Spread Dynamic Multicast Heuristic (GSDM) is an unconstrained rearrangeable online multicast routing heuristic proposed by Kadirire [30]. The GSDM introduces the idea of Geographic Spread (GS). GSDM spreads out the multicast connections geographically in the network and lowers the cost of the resulting multicast tree. For each add request, GSDM identifies the closest tree node to the new member, let it be node X, and two other tree nodes, which are closest to node X. Then, GSDM chooses the least cost configuration among four possible ways to connect the new member to those three nodes. GSDM chooses configuration with greatest GS, if more than one cheapest configuration exists. For each delete request, GSDM deletes the nodes only if it is a leaf node. Kadirire evaluated the performance of GSDM and compared it with Waxman's GREEDY heuristic [8] and Doar and Leslie's shortest path based heuristic, NAIVE heuristic [5], using simulations on random networks with 100 nodes. He showed that GSDM's performance is similar to performance of GREEDY heuristic and performance of GSDM is better than that of Doar and Leslie's NAIVE heuristic.

Bauer and Verma proposed an unconstrained rearrangeable online multicast routing heuristic, ARIES [31]. ARIES behaves just like GREEDY heuristic as it adds and removes members at each request. So, it makes the minimum necessary modifications to the existing multicast tree. For each join request, ARIES adds the new member node to the existing multicast tree using the shortest path between the new member node and the tree. For each remove request, ARIES deletes the member node only if it is a leaf node and ARIES just cancels its membership if it is not a leaf node. However, ARIES always monitors the cumulative damage happened in the multicast tree due to the changing membership. It marks joining member nodes and leaving member nodes,

which are kept as relay nodes in the tree. Those marked nodes define regions. These regions encapsulate the degraded portions of the tree. Degradation is due to join/leave requests. If the number of marked nodes in a region is greater than a threshold, this means the region is degraded too much and that region is removed. The resulting subtrees are connected using a method similar to RS heuristic. Source subtree, which is a subtree containing the source node, is iteratively merged with the nearest subtree, until only one tree is left. Bauer and Verma compared the performance of ARIES with that of EBA and GREEDY. They showed that ARIES is much better than both GREEDY and EBA.

2.2.2. Constrained Heuristics

Kompella, Pasquale, and Polyzos [32, 33] formulated the delay-constrained SMT problem for the first time. The objective of the problem is to minimize the total cost of the tree, without violating the delay constraints. Kompella *et al.* also proved the NP-completeness of the problem in their studies [32, 33]. There are optimal algorithms for the problem [34], but these algorithms are complex and they are useful only as a reference to evaluate a heuristic solution for the problem.

2.2.2.1. Offline Heuristics. Kompella, Pasquale, and Polyzos [32, 33] proposed the first heuristic for the delay-constrained Steiner minimal tree problem. This heuristic is cited as KPP in the literature. KPP assumes that the link delays and the delay constraint, Δ, are integers, but the link costs may be any positive real number. KPP is dominated by computing a constrained distance graph in $O(\Delta \cdot n^3)$. KPP uses Prim's algorithm [4] to obtain a minimum spanning tree of the distance graph. Prim's algorithm is only optimal for symmetric networks. This might affect the performance of KPP in asymmetric networks. Time complexity of KPP highly depends on Δ and it is polynomial only if Δ has a fixed integer value. When the link delays and Δ are not integer, Kompella, Pasquale, and Polyzos proposed to multiply out fractional values to get integers. So, KPP is guaranteed to construct a constrained tree if one exists. However, the granularity of the delay constraint becomes very small in some cases,

and the number of bits required to represent delay constraint increases considerably. This results in very high time complexities. Fixed granularity may be used to avoid very high time complexities. However, KPP's performance is compromised and it may fail to construct a constrained multicast tree if the granularity is comparable to the average link delays.

Sun and Langendorfer proposed a delay-constrained least-cost multicast routing heuristic, CKMB [35]. CKMB bases on the KMB heuristic [6], which is proposed by Kou, Markowsky, and Berman. CKMB first finds the least cost, delay-constrained paths between every two multicast members by first finding least cost path and replaces it with least delay path if it violates delay constraint as in CDKS [16]. Using these paths, the heuristic builds a distance graph consisting of the multicast members. The cost and delay of an edge between two nodes in the distance graph is the cost and delay of the least-cost delay-constrained path between these two nodes in the original graph. From the distance graph, the heuristic uses KMB to build the multicast tree [35]. The worst-case time complexity of CKMB heuristic is $O(m \cdot n^2)$.

Parsa *et al.* proposed the Bounded Shortest Multicast Algorithm (BSMA) [36]. BSMA is an iterative heuristic. It starts by computing LD using Dijkstra algorithm for a given source and multicast group members. If this tree violates delay constraint, then delay constraint must be renegotiated. BSMA iteratively replaces superedges in the tree with cheaper superedges, without violating the delay constraint. A superedge is a path in the tree between two branching nodes, or two multicast group members, or a branching node and a multicast group member. This iterative procedure stops, when the total cost of the tree cannot be reduced any further. BSMA uses a k-shortest path algorithm to find cheaper superedges. Number of iterations is $O(n\log(n))$ and time complexity of BSMA is $O(KSTP(n) \cdot n\log(n))$, where $KSTP(n)$ is the time complexity of k-shortest path algorithm used in the BSMA. Time complexity of the k-shortest path algorithm used in the implementation of BSMA is $O(k \cdot n^3)$, where k is the number of paths produced by the k-shortest path algorithm [37]. The k-shortest path algorithms find paths between two nodes. Parsa *et al.* made fine modifications on the k-shortest path algorithm to find paths between subtrees resulting from removal of

superedges. In case of large, densely connected networks, k may be very large, and it may be difficult to achieve acceptable running times. However, it is possible to trade off multicast tree cost for fast execution speed by either limiting the value of k in the k-shortest path algorithm or by limiting the number of superedge replacements. BSMA always finds a constrained multicast tree, if one exists, because it starts with LD, and replaces superedges, only if it finds a superedge, which is not violating the delay constraint and resulting in a lower cost tree. Unlike most of the other heuristics, BSMA also works with asymmetric links. Salama *et al.* evaluated the performance of delay-constrained least-cost multicast routing heuristics [38]. BSMA was shown to be the best heuristic in terms of the tree cost among other heuristics proposed for the problem [38]. Hence, in the literature, BSMA is frequently used for the performance evaluation of other heuristics, because for large networks it is not feasible to use optimal algorithms for comparison.

2.2.2.2. Online Heuristics. Hong *et al.* proposed a non-rearrangeable online multicast routing heuristic; Lagrangian Relaxation-Based Algorithm (LRA), [39]. LRA tries to minimize the cost of multicast tree, while bounding the delay. For each join request, the heuristic first tries to add the new member node using the least-cost path between the source node and the new member node. If the least-cost path violates the delay constraint, the heuristic creates a new metric for path calculations. Value of new metric on an edge bases on edge's cost, edge's delay and both of least cost path and least delay path. Then, LRA iteratively changes the new metric and searches for a least-cost, delay-constrained path. For this purpose, LRA uses a lagrangian relaxation based technique with dynamic parameters combining cost and delay. Leave requests are handled as in GREEDY heuristic [7]. If the member node to be removed is a leaf node, it is simply pruned, otherwise nothing is done. Time complexity of the heuristic is not presented in [39] and there is no known upper bound for time complexity of this heuristic.

Baoxian *et al.* [40] proposed a heuristic, DCMA, for non-rearrangeable online multicast routing problem. DCMA uses information available from unicast protocol,

OSPF, to simplify the addition of new multicast group members to multicast tree. For each join request, new member node is connected to tree via the least-cost delay-constrained path between source node and the new member node. Least-cost delay-constrained path between source node, s, and the new member node, d, is computed using an iterative procedure. Initially, node d is the first node on constrained path. Node d is set as current active node, then recursively one predecessor of active node is added to constrained path, if the predecessor node, say w, satisfies Equation 2.1.

$$D(P_{LC}(s,w)) + delay\_so\_far < \Delta \tag{2.1}$$

In the Equation 2.1, $delay\_so\_far$ is the delay of the path already constructed and Δ is the delay constraint. Then, node w is set as current active node and the process is repeated recursively until s is reached. In this way, the delay-constrained unicast path from s to d is constructed. This path is simply merged with the tree. There can be loops in the resulting tree. These loops are removed in such a way that the resulting tree is connected, loop-free and none of delay constraints is violated. Leave requests are handled as in most of the other non-rearrangeable online multicast routing heuristics. If the member node to be removed is a leaf node, it is simply pruned, otherwise nothing is done.

Zhengying *et al.* [41] proposed DDCLCMR, a delay-constrained least-cost multicast routing heuristic for dynamic groups. This heuristic is a non-rearrangeable online multicast routing heuristic. For each join request, new member node is connected to the tree using the procedure explained in [41]. In this procedure, DDCLCMR creates a virtual node, v_a, which is connected to each tree node with a virtual edge whose cost and delay are the cost and the delay of the path in the tree from source node to that tree node respectively. Virtual node is used to prevent cycles in the tree after addition of the new member node. Current node is set to v_a. Then, current node iteratively selects one of its neighbors according to a selection function and the selected node is set as current node. This procedure is repeated until the new member node is selected. Selected nodes constitute the path connecting the new member to the tree. Selection function is critical. It is a nonlinear function of cost and delay. DDCLCMR always

finds a delay-constrained path from source to the new member node. Because selection value of a neighbor node becomes infinite if the neighbor node of current node is not on any delay-constrained path. However, a neighbor node with lower selection value is selected by the current node. So, a delay-constrained path is always found. However, this path is usually not the delay-constrained least-cost path. The worst-case time complexity of DDCLCMR heuristic is $O(n \cdot \varepsilon)$, where ε is the number of edges and n is the number of nodes in the graph.

Raghavan *et al.* proposed a delay-constrained rearrangeable online multicast routing heuristic, CRCDM [42]. CRCDM behaves just like ARIES heuristic as it adds and removes members at each request. Moreover, CRCDM's rearrangement decisions are made very similar to that of ARIES. The major differences between those heuristics are that CRCDM is a delay-constrained heuristic but ARIES is not, and metrics for rearrangement in the two heuristics are slightly different. For each join request, CRCDM adds the new member to the existing multicast tree by a delay-constrained least-cost path between the tree and the node. Any delay-constrained least-cost path heuristic can be used for this, but preferred-link based delay-constrained least-cost routing heuristic [43] is used in the original implementation. CRCDM is also a parameterized heuristic like ARIES. CRCDM's parameter κ defines the number of tree nodes from which delay-constrained least-cost paths to the new member node are computed. Firstly all tree nodes are sorted according to decreasing priorities. Tree nodes with less delay from source node have higher priority. Then, first κ number of these nodes is selected and delay-constrained paths from those nodes to new member node are computed. Lowest cost path among those κ paths is chosen and new member is added to the tree using this path. For the best performance in terms of tree cost, κ should be set to the number of tree nodes. For each remove request, CRCDM deletes the member node, only if it is a leaf node, and CRCDM cancels its membership if it is not a leaf node. However, CRCDM always monitors the cumulative damage happened in the multicast tree due to the changing membership. CRCDM marks leaving member nodes, which are kept as relay nodes in the tree. These marked nodes define regions. These regions encapsulate the degraded portions of the tree. Degradation is due to leave requests.

Let $\Re$ be a region, Θ be the number of marked nodes in $\Re$, and Ψ be the number of multicast group members serviced by those marked nodes in $\Re$. In ARIES, if Θ is greater than a threshold, this means that the region, $\Re$ is degraded too much and this region is removed. In CRCDM, definition of the region, $\Re$, is almost the same as that of ARIES, but rearrangement criterion does not depend only on the number of marked nodes in the region. CRCDM makes rearrangement if the ratio $\frac{\Psi}{\Theta}$ decreases below a predefined threshold, ρ. After the removal of the degraded region $\Re$, the resulting subtrees are connected using delay-constrained least-cost paths between subtrees. Source subtree, which is a subtree containing the source node, is iteratively merged with the nearest subtree using the delay-constrained path between subtrees until only one tree is left. Raghavan *et al.* compared the performance of CRCDM with LRA [39] and ARIES [31]. They showed that CRCDM is better than LRA and ARIES.

3. PROPOSED HEURISTICS FOR DELAY-CONSTRAINED ONLINE MULTICAST ROUTING PROBLEM

3.1. Formulation of Delay-Constrained Online Multicast Routing Problem

The network is modeled as a directed, connected graph $G = (V, E)$, where V is the set of vertices representing nodes in the network and E is the set of edges representing links in the network. An edge $e \in E$ connecting nodes u and v will be denoted as (u, v) and has a reverse edge $e' \in E$ denoted as (v, u). Each edge has two nonnegative metrics associated with it: a cost $C(e)$ and a delay $D(e)$. A path $P = (v_0, v_1, v_2, ..., v_n)$, in the network, has also two associated characteristics: a cost $C(P)$ and a delay $D(P)$.

$$C(P) = \sum_{i=0}^{n-1} C((v_i, v_{i+1})) \tag{3.1}$$

$$D(P) = \sum_{i=0}^{n-1} D((v_i, v_{i+1})) \tag{3.2}$$

Similarly, a tree $T = (V_T, E_T)$, which is a subgraph of G such that $V_T \in V$ and $E_T \in E$, has an associated cost defined as in Equation 3.3.

$$C(P) = \sum_{e \in E_T} C(e) \tag{3.3}$$

The path $P_T(v_i, v_j)$ denotes the path in tree T, which is between nodes v_i and v_j belonging the tree T.

- **GIVEN:** A graph $G = (V, E)$, a set of multicast members $W \in V$, a request vector $R = (r_0, r_1, r_2, ..., r_m)$, where each element of R is a join or a leave request concerning a single member node, an initial multicast tree T_0, a multicast source s and a positive delay constraint Δ.
- **FIND:** Multicast trees $T_1, T_2, ..., T_m$ such that the member nodes of each tree T_i

are those of the tree T_0 modified by the requests $r_0, r_1, r_2, ..., r_i$ and its cost is the minimum among all possible choices for tree T_i. For any $v \in T_i$ in each tree, path delay from source s to v must be less than or equal to the delay constraint Δ. This means that Equation 3.4 must be correct.

$$D(P_{T_i}(s,v)) \leq \Delta \tag{3.4}$$

This problem is NP-complete [13, 14, 33, 32]. However, heuristics can find approximate solutions in polynomial time. Waxman partitioned this problem into two categories as rearrangeable and non-rearrangeable [7, 8]. Rearrangeable heuristics allow reconfiguration, when degradation of the tree exceeds some limit. This degradation is originated from join/leave requests. Non-rearrangeable heuristics do not allow reconfiguration. Rearrangements should not take too much time and difference between consecutive trees should be minimum to prevent the loss of packets on the fly. Those are some basic requirements for real-time communications and should be considered carefully in a solution for the problem.

3.2. Heuristic DMBDCLC

Unconstrained unicast routing protocols characterize network links by a single metric, usually cost. Well-known algorithms, Dijkstra's algorithm and the BF algorithm, find the shortest paths based on a single metric. However, routing with explicit quality of service (QoS) requirements characterizes network links with additional metrics such as bandwidth, end-to-end-delay, jitter or congestion. These additional metrics make routing more complicated. The problem of finding a path subject to two or more additive metrics is NP-complete [13].

Although the problem is NP-complete, some polynomial time heuristics are proposed to approximate the solution. A substantial portion of those heuristics depends on k-shortest path algorithms [44, 45, 46, 47, 48, 49]. Performance of such heuristics highly depends on k parameter and the performance of underlying k-shortest path algorithm. They can find near optimum paths for appropriately large k values with a

drawback of high computation time. Constrained versions of well-known single metric routing algorithms are also proposed. Widyono proposed the CBF algorithm [15]. Although CBF algorithm is exact, its worst-case running time grows exponentially with the network size. There are also heuristics combining two or more metrics accordingly as a single metric to find constrained paths. In [50], a Lagrange relaxation based heuristic (LARAC) was proposed for this purpose. LARAC linearly combines cost and delay. Another heuristic using non-linear combinations of cost and delay to find delay-constrained least-cost paths was proposed in [51]. However, none of these heuristics guarantees the optimal solution.

For real-time communications, such as teleconferencing, videoconferencing or real-time online multicast routing applications, connection establishment time can be crucial. Especially for online multicast sessions where new members join and some members leave the session, immediate establishment of new admissible paths may be nearly as important as constructing low cost delay-constrained paths. Dual Memory Based Delay-Constrained Least-Cost Path Heuristic (DMBDCLC) is a heuristic proposed to find best effort least-cost delay-constrained paths within a worst-case time complexity of $O(n \cdot \log(n))$ for the applications, which require fast computation of new delay-constrained paths without trading off performance for speed.

3.2.1. Formulation of Delay-Constrained Least-Cost Path Problem

The network is modeled as a directed, connected graph $G = (V, E)$, where V is the set of vertices representing nodes and E is the set of directed edges representing links in the network. Each link, e, is characterized by a cost, $C(e)$, and a delay, $D(e)$, and n is the number of nodes. An example graph is shown in Figure 3.1. Edges in the figure are labelled according to $(cost, delay)$ format. Given a source node $s \in V$, a destination node $d \in V$, and a positive delay constraint Δ, the delay-constrained least-cost path problem is summarized in Equation 3.5.

$$\min_{P \in P^{\Delta}_{set}(s,d)} \sum_{e \in P} C(e) \tag{3.5}$$

In Equation 3.5, P_{set}^{Δ} is the set of paths from s to d for which the end-to-end delay is bounded by Δ. With P_{set} denoting the set of all paths from s to d, a path $P \in P_{set}(s,d)$ belongs to P_{set}^{Δ} if Equation 3.6, is valid for P.

$$\sum_{e \in P} D(e) \leq \Delta \tag{3.6}$$

This problem is NP complete and we will present a fast approximation to the optimal solution in the next section. Performance of the heuristic is evaluated and results are presented in Chapter 4.

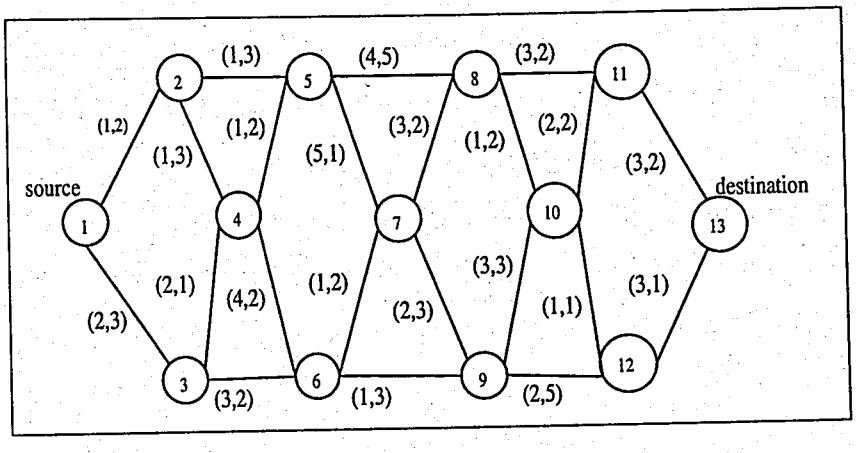

Figure 3.1. An example graph

3.2.2. Proposed Heuristic

DMBDCLC bases on Dijkstra and Reverse-Dijkstra algorithms. In Dijkstra algorithm, objective is to find the shortest paths from a source node to other nodes. Each node keeps the least cost path from source node to itself by keeping previous node on that path. By back tracking this previous node, this path can be constructed. In other words, each node remembers the least cost path to itself. This node also keeps the total cost and the total delay of the path. In Figure 3.2, result of one run of Dijkstra algorithm with cost metric from a source node to other nodes is shown for the graph in Figure 3.1. In the figure, nodes are labelled with $[pathcost, pathdelay], PredecessorNodeID$ format in order to keep path information at each node.

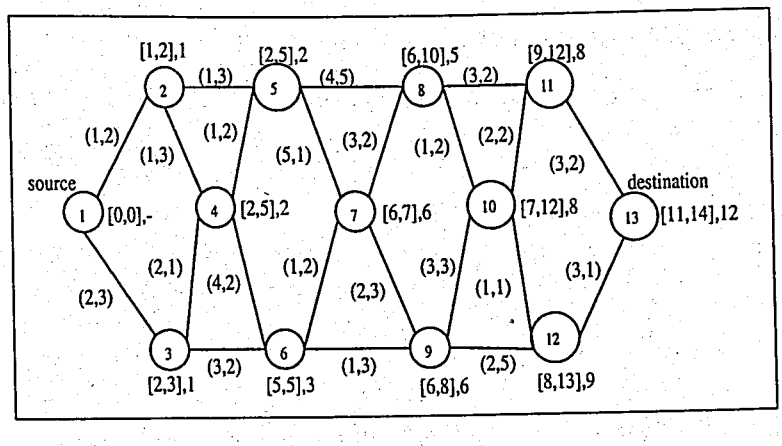

Figure 3.2. Cost metric Dijkstra from source node to other nodes

Similarly Reverse-Dijkstra algorithm can be used to find paths to a destination from other nodes. This time each node remembers a path from itself to destination by keeping forward node information. This node also keeps total cost and total delay of the path. In Figure 3.3, result of Reverse-Dijkstra algorithm with cost metric from nodes to a destination is shown for the graph in Figure 3.1. In the figure, nodes are labelled with $[pathcost, pathdelay], ForwardNodeID$ format in order to keep path information at each node. A pseudo-code of Reverse-Dijkstra algorithm can be found in Figure A.2.

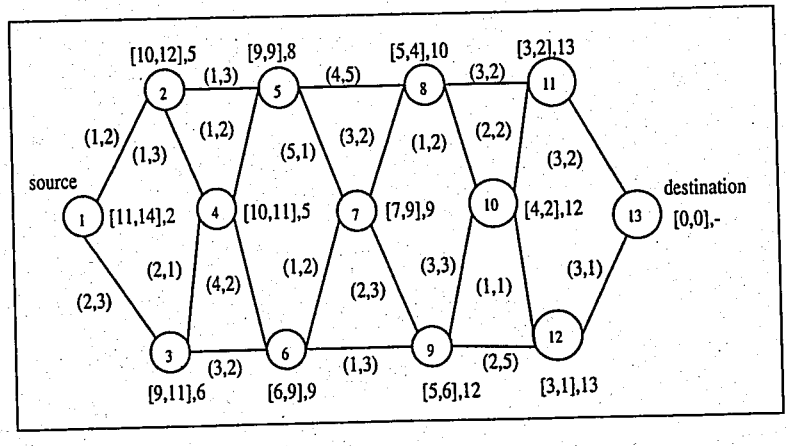

Figure 3.3. Cost metric Reverse-Dijkstra from other nodes to destination node

By combining information in node labels computed by Dijkstra and Reverse-Dijkstra, that is summing path cost and path delay values and appending values of $PredecessorNodeID$ and $ForwardNodeID$, each node remembers the shortest path from source to destination passing over itself. This is shown in Figure 3.4. In the figure, each node remembers one path passing over itself and this path is the best path contain-

ing that node. As a result, we have n paths and only a subset of them is unique. Nodes are labelled according to $[pathcost, pathdelay]$, $PredecessorNodeID$, $ForwardNodeID$ format in Figure 3.4.

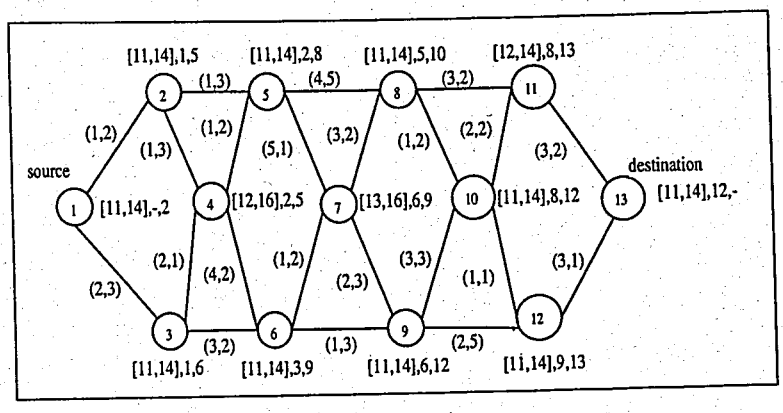

Figure 3.4. Combined labels of cost metric Dijkstra and Reverse-Dijkstra

The "Dual Memory" notation in the name of the heuristic refers that each node in the graph remembers a second path. This path is found by applying the same procedure for delay metric instead of cost metric. This time each node remembers the least delay path from the source node to the destination node, which is passing over that node. Figure 3.5 shows the results of Dijkstra algorithm for delay metric and Figure 3.6 shows the results of Reverse-Dijkstra algorithm for delay metric. In Figure 3.7, combination of the results computed by Dijkstra and Reverse-Dijkstra are shown. In Figure 3.5, nodes are labelled with $[pathcost, pathdelay]$, $PredecessorNodeID$ format and nodes in Figure 3.6 are labelled with $[pathcost, pathdelay]$, $ForwardNodeID$ format as well as nodes are labelled with $[pathcost, pathdelay]$, $PredecessorNodeID$, $ForwardNodeID$ format in Figure 3.7. Table 3.1 and Table 3.2 tabulate the derived paths from nodes for cost metric and delay metric respectively. For each node, two paths are derived; one path for cost metric and one path for delay metric.

Path selection can be done by traversing node labels and selecting the cheapest delay-constrained path. Label information and derived paths from those labels are shown in Table 3.1 and Table 3.2. Paths in Table 3.1 and Table 3.2 are not unique. Unique paths are shown in Table 3.3. Further details of DMBDCLC heuristic are stated and explained in the following Lemmas.

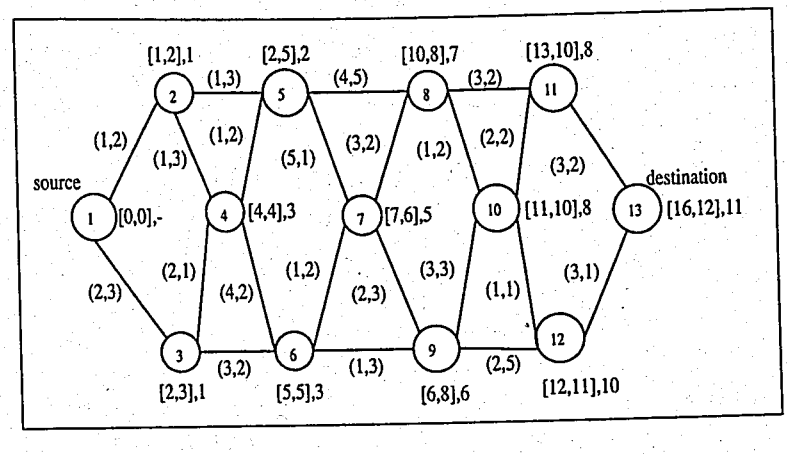

Figure 3.5. Delay metric Dijkstra from source node to other nodes

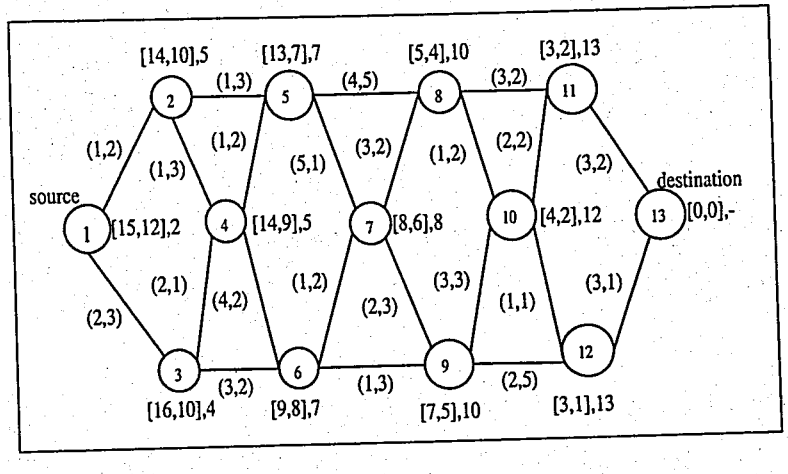

Figure 3.6. Delay metric Reverse-Dijkstra from other nodes to destination node

Lemma 3.2.1: DMBDCLC can find a delay-constrained path, if one exists, between a source node and a destination node

Proof: Least delay path between source node and destination node is always included in the derived paths. This path must be a delay-constrained path, if such a path exists. ■

Lemma 3.2.2: DMBDCLC can find a delay-constrained path, if one exists, between a source tree and a destination node (if some modifications are made).

Proof: In the preprocessing phase of classical Dijkstra algorithm, $(cost, delay)$ pair of the source node is set to zero and $(cost, delay)$ pair of other nodes is set to

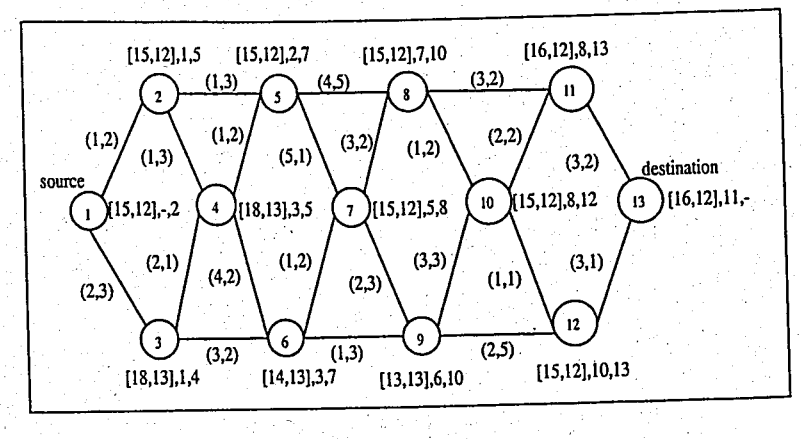

Figure 3.7. Combined labels of delay metric Dijkstra and Reverse-Dijkstra

infinity. In case of a source tree, not only the source node's $(cost, delay)$ pair but also all of the source tree nodes' $(cost, delay)$ pair is set to zero. So, at the end of delay metric calculations, each source tree node keeps a label containing the least delay path from itself to destination node. This means that, if such a path exists, DMBDCLC can always find a delay-constrained path between a source tree node and a destination node. A pseudo-code for the modified version of Dijkstra algorithm can be found in Figure A.3. ■

Lemma 3.2.3: DMBDCLC can find a delay-constrained path, if one exists, between a source tree and a destination tree (if some modifications are made).

Proof: Algorithmic differences explained for Dijkstra algorithm in the proof of Lemma 3.2.2 are also made and the same procedure is repeated for Reverse-Dijkstra algorithm using destination tree instead of source tree. In addition to those algorithmic differences, some others are made on Dijkstra algorithm. These differences are explained in the following lines.

Let node u not be in the source tree, and node v be any node in the source tree, and there exists an edge (v, u) combining v and u in the graph. If $(cost_u, delay_u)$ pair of node u is updated using the node v, $(cost_v, delay_v)$ pair of node v, which is $(0, 0)$, is not used to update $(cost_u, delay_u)$ during relaxation phase. However, $C(P_T(s, v))$ and $D(P_T(s, v))$ are used, instead of $cost_v$ and $delay_v$, respectively. $C(P_T(s, v))$ and

Table 3.1. Cost metric paths derived from labels

Node	Cost	Delay	Path
1	11	14	1,2,5,8,10,12,13
2	11	14	1,2,5,8,10,12,13
3	11	14	1,3,6,9,12,13
4	12	16	1,2,4,5,8,10,12,13
5	11	14	1,2,5,8,10,12,13
6	11	14	1,3,6,9,12,13
7	13	16	1,3,6,7,9,12,13
8	11	14	1,2,5,8,10,12,13
9	11	14	1,3,6,9,12,13
10	11	14	1,2,5,8,10,12,13
11	12	14	1,2,5,8,11,13
12	11	14	1,2,5,8,10,12,13
13	11	14	1,2,5,8,10,12,13

$D(P_T(s,v))$ are cost and delay of the path in the tree from source node to node v. Let cost and delay of the edge (v,u) be ς and φ, respectively. This exceptional procedure is shown in Equation 3.7 and in Equation 3.8. If node v were not in the source tree, procedure would be as in Equation 3.9 and in Equation 3.10.

$$cost_u = C(P_T(s,v)) + \varsigma \tag{3.7}$$

$$delay_u = D(P_T(s,v)) + \varphi \tag{3.8}$$

$$cost_u = cost_v + \varsigma \tag{3.9}$$

$$delay_u = delay_v + \varphi \tag{3.10}$$

After this exceptional procedure, at the end of delay metric calculations of DMBDCLC, each destination tree node keeps a label containing the least-delay path from source node to itself. This means that, if such a path exists, DMBDCLC can always find a delay-constrained path between a source tree and a destination tree without violating

Table 3.2. Delay metric paths derived from labels

Node	Cost	Delay	Path
1	15	12	1,2,5,7,8,10,12,13
2	15	12	1,2,5,7,8,10,12,13
3	18	13	1,3,4,5,7,8,10,12,13
4	18	13	1,3,4,5,7,8,10,12,13
5	15	12	1,2,5,7,8,10,12,13
6	14	13	1,3,6,7,8,10,12,13
7	15	12	1,2,5,7,8,10,12,13
8	15	12	1,2,5,7,8,10,12,13
9	13	13	1,3,6,9,10,12,13
10	15	12	1,2,5,7,8,10,12,13
11	16	12	1,2,5,7,8,11,13
12	15	12	1,2,5,7,8,10,12,13
13	16	12	1,2,5,7,8,11,13

the delay constraints. Pseudo-codes for the modified versions of Reverse-Dijkstra and Dijkstra algorithms can be found in Figure A.4 and in Figure A.5 respectively. After Lemma 3.2.3, DMBDCLC can also find a constrained path between a source tree and a destination node without violating the delay constraints. ■

Lemma 3.2.4: The worst-case time complexity of DMBDCLC is $O(n \cdot \log(n))$ if the number of edges in the graph is $O(n)$.

Proof: In the heuristic, Dijkstra is run two times and Reverse-Dijkstra is also run two times. Complexity of Dijkstra and Reverse-Dijkstra algorithms are $O(n \cdot \log(n))$ if Fibonacci heap is used in the implementation and the number of edges in the graph is $O(n)$ [52]. So, the total complexity of runs of Dijkstra and Reverse-Dijkstra algorithms is $O(4 \cdot (n \cdot \log(n)))$. Traversal of node labels requires $O(n)$. So, total complexity of the heuristic becomes $O(4 \cdot (n \cdot \log(n)) + n) = O(n \cdot \log(n))$. ■

Table 3.3. Unique paths sorted by cost

Node	Cost	Delay	Path	Metric
1	11	14	1,2,5,8,10,12,13	COST
3	11	14	1,3,6,9,12,13	COST
11	12	14	1,2,5,8,11,13	COST
7	13	16	1,3,6,7,9,12,13	COST
9	13	13	1,3,6,9,10,12,13	DELAY
6	14	13	1,3,6,7,8,10,12,13	DELAY
1	15	12	1,2,5,7,8,10,12,13	DELAY
11	16	12	1,2,5,7,8,11,13	DELAY
4	18	13	1,3,4,5,7,8,10,12,13	DELAY

Implementation of DMBDCLC heuristic contains all of the algorithmic differences explained in the lemmas. Pseudo-codes of modified algorithms can be found in appendix.

3.2.3. Fast-BSMA

In Chapter 2, BSMA heuristic is explained shortly. It is an iterative delay-constrained offline multicast routing heuristic proposed by Parsa *et al.* [36]. It uses a k-shortest path algorithm to find constrained paths. Parsa *et al.* made some differences on the k-shortest path algorithm to find paths between subtrees instead of nodes. The k-shortest path algorithms usually have high computational complexities. Therefore BSMA heuristic has a high time complexity. Especially in case of large and densely connected networks, k may be very large, and it may be difficult to achieve acceptable running times.

Fast-BSMA (FBSMA) is a version of BSMA heuristic in which DMBDCLC is used instead of a k-shortest path algorithm based delay-constrained least cost path heuristic. FBSMA is a good environment to evaluate the performance of DMBDCLC both as a stand-alone unicast routing heuristic and an underlying unicast routing

heuristic in multicast routing. Because BSMA is known to be the best polynomial-time heuristic in terms of the multicast tree cost in the literature and its high time complexity is its only bottleneck [38]. This bottleneck is relaxed by using DMBDCLC instead of a k-shortest path algorithm based heuristic without compromising tree cost performance significantly. Moreover, DMBDCLC is more convenient for use in BSMA heuristic than most of the other delay-constrained least-cost path heuristics. Because BSMA connects subtrees during the replacement of superedges and DMBDCLC is capable of finding delay-constrained paths between two trees. So, FBSMA heuristic is implemented and used for the evaluation of DMBDCLC heuristic. Implementations of BSMA and FBSMA are exactly the same except DMBDCLC is used instead of the k-shortest path based delay-constrained least-cost path heuristic. The same Dijkstra routines are used in BSMA and FBSMA heuristics. The worst-case time complexity of FBSMA heuristic is $O([n\log(n)]^2)$. Because expected number of iterations in FBSMA is $O(n\log(n))$ as in BSMA and worst-case time complexity of DMBDCLC is $O(n\log(n))$. So, the total time complexity is $O([n\log(n)]\cdot[n\log(n)]) = O([n\log(n)]^2)$.

3.3. Heuristic DeCoNROM

Delay-Constrained Non-Rearrangeable Online Multicast Routing Heuristic (DeCoNROM) is a simple non-rearrangeable heuristic. DeCoNROM is very similar to the unconstrained non-rearrangeable online multicast routing heuristic of Waxman, GREEDY heuristic [7]. Major difference of those two heuristics is that DeCoNROM is a delay-constrained heuristic and it uses DMBDCLC as underlying delay-constrained least-cost path heuristics.

DeCoNROM heuristic handles join/leave requests in a simple, fast and efficient manner. In case of a join request, new multicast group member node is added to the multicast tree using the delay-constrained least-cost path between the tree and the node. This path is computed using DMBDCLC, which can calculate delay-constrained least-cost paths between a tree and a node with a time complexity of $O(n\log(n))$. In case of a leave request, the member node leaving the multicast group is pruned only if it is a leaf node; otherwise it becomes a relay node. Pruning algorithm firstly

deletes the member node from the tree and it continues to delete its ancestors iteratively until it meets a node with more than one child or it meets a member node or a source node. So, worst-case time complexity of DeCoNROM for a request is only $O(n\log(n))$. DeCoNROM is directly affected by the performance of DMBDCLC. Performance of DMBDCLC and DeCoNROM are evaluated using simulations. Results of those simulations are presented in Chapter 4.

3.4. Heuristic CoBaCROM

Although there is a vast amount of literature on multicast routing, online multicast routing is a relatively unexplored area of research. Online multicast routing heuristics are classified as rearrangeable and non-rearrangeable heuristics. Rearrangeable heuristics allow reconfiguration of the multicast tree, when degradation of the tree exceeds some limit. Non-rearrangeable heuristics do not allow reconfiguration. In the literature, very few rearrangeable heuristic is proposed for online multicast routing problem. In this section, the Contract Based Delay-Constrained Rearrangeable Online Multicast Routing Heuristic (CoBaCROM) is proposed and explained. CoBaCROM is compared with the existing heuristics in Chapter 4.

3.4.1. Existing Approaches and Their Deficiencies

In Chapter 2, some of the important heuristics from online multicast routing literature are shortly explained. Two impressive heuristics from rearrangeable online multicast routing literature are unconstrained heuristic, ARIES [31] and delay-constrained heuristic, CRCDM [42]. In ARIES and its delay-constrained counterpart CRCDM, mechanisms for triggering rearrangement similarly base on regions. Those regions, called *m-regions*, are assumed to represent degraded portions of a multicast tree. An example of m-regions is shown in Figure 3.8. M-regions are maximally connected regions and they are created by marked nodes (m-nodes). A node becomes an m-node in two cases. Firstly, if a member node is a leaf node and a leave request is made for this member node, the member node is pruned until another member node, source node or a node with more than two childs is encountered. If pruning is stopped at a node,

which is neither a member nor source node, this node becomes an m-node. Secondly, if the member node to be removed is an intermediate node, it is simply becomes an m-node and remains in the multicast tree in order to serve existing multicast members as a relay node. Each m-node must belong to an m-region. Figure 3.10 and Figure 3.11 show how nodes become m-node and how they constitute m-regions. Internal nodes of an m-region are either relay nodes or m-nodes and leaf nodes of this region are either member nodes or source node. If degradation in an m-region exceeds some limit, it is simply removed from the multicast tree. Removal of an m-region and resultant subtrees are shown in Figure 3.9. Those subtrees are reconnected to construct a better tree and this procedure is called rearrangement.

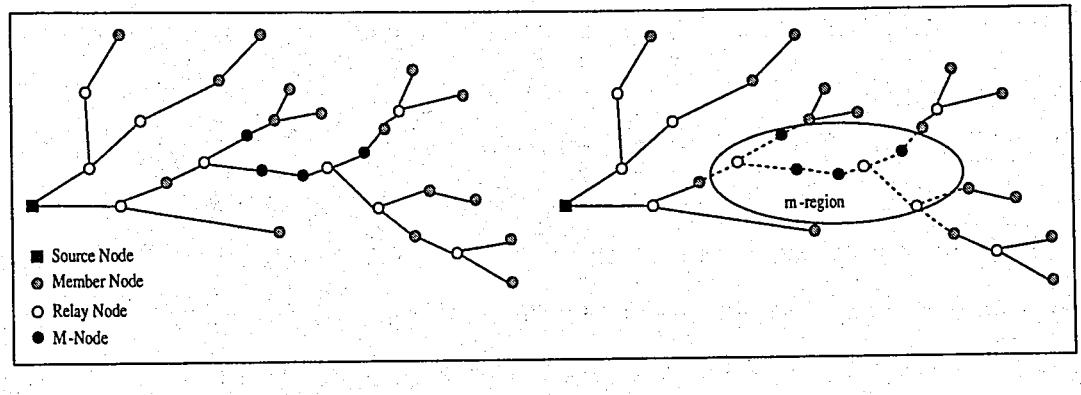

Figure 3.8. An example of an m-region in ARIES and CRCDM

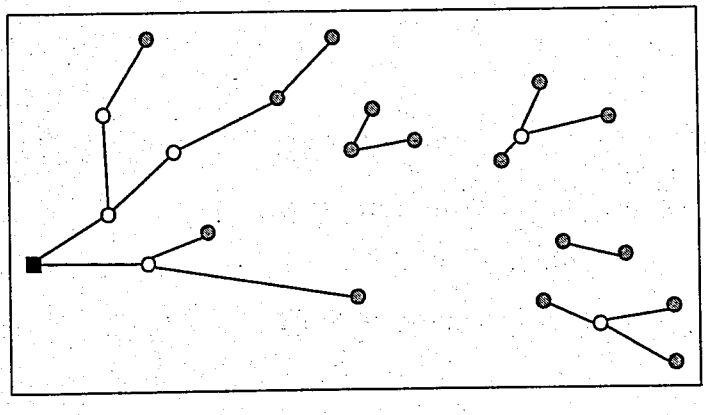

Figure 3.9. Resulting subtrees after the removal of m-region in ARIES and CRCDM

Let $\Re$ be an m-region, Θ be the number of m-nodes in $\Re$, and Ψ be the number of multicast group members serviced by those m-nodes in $\Re$. In ARIES, $\Re$ is removed and resulting subtrees are reconnected, when Θ exceeds a predefined threshold. In

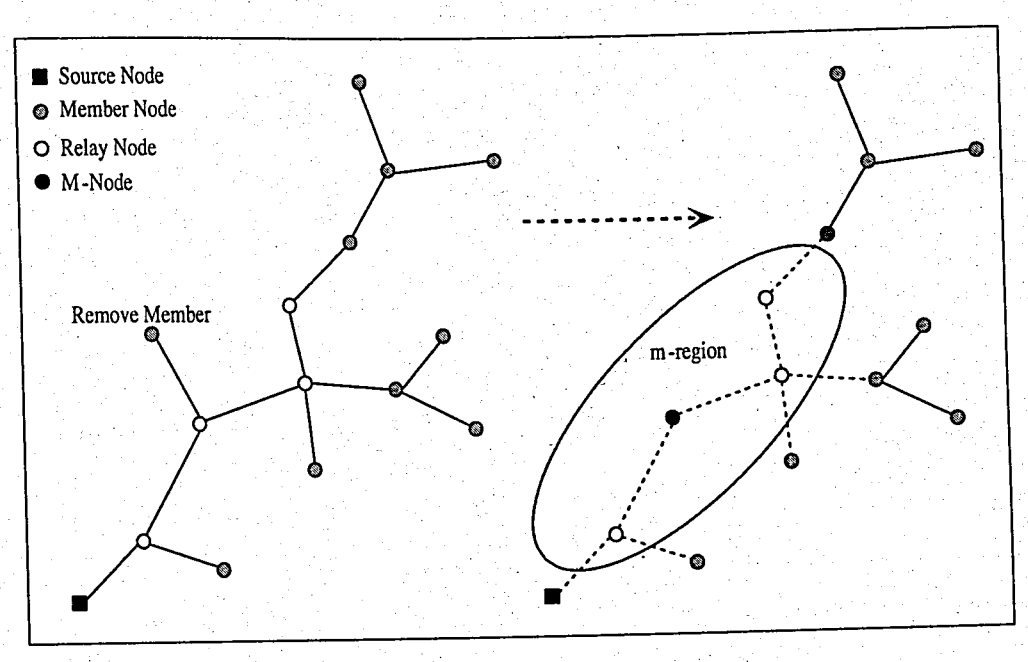

Figure 3.10. Removal of a member and resulting m-region in ARIES and CRCDM

[31], it has been shown that this rearrangement scheme of ARIES outperforms another unconstrained rearrangeable online multicast routing heuristic, EBA [7, 8]. In CRCDM, rearrangement is done when the ratio $\frac{\Psi}{\Theta}$ decreases below a predefined threshold. It is claimed that this ratio represents the usefulness of the m-region and usefulness of the region decreases as this ratio decreases. So, it is required to rearrange the tree by removing that region. In [42], it has been shown that trees computed by CRCDM are better than the trees computed by the delay-constrained non-rearrangeable online multicast routing heuristic, LRA [39].

It seems that both ARIES and CRCDM are very successful. However, definition of an m-region in both CRCDM and ARIES covers a maximally connected region and this region covers nodes and edges, which are actually not affected by the leave requests. In heuristics ARIES and CRCDM, removal of a member node results in a new m-region or expansion of an existing one. Construction or expansion of the m-region continues until a member node or the source node is encountered. This is depicted in Figure 3.10 and Figure 3.11. Resulting m-region also covers the part of the tree, which is nothing to do with the removed member nodes. This part should not be included in the m-region and should not be removed from the tree during the rearrangement phase. However, previous heuristics do not distinguish this part

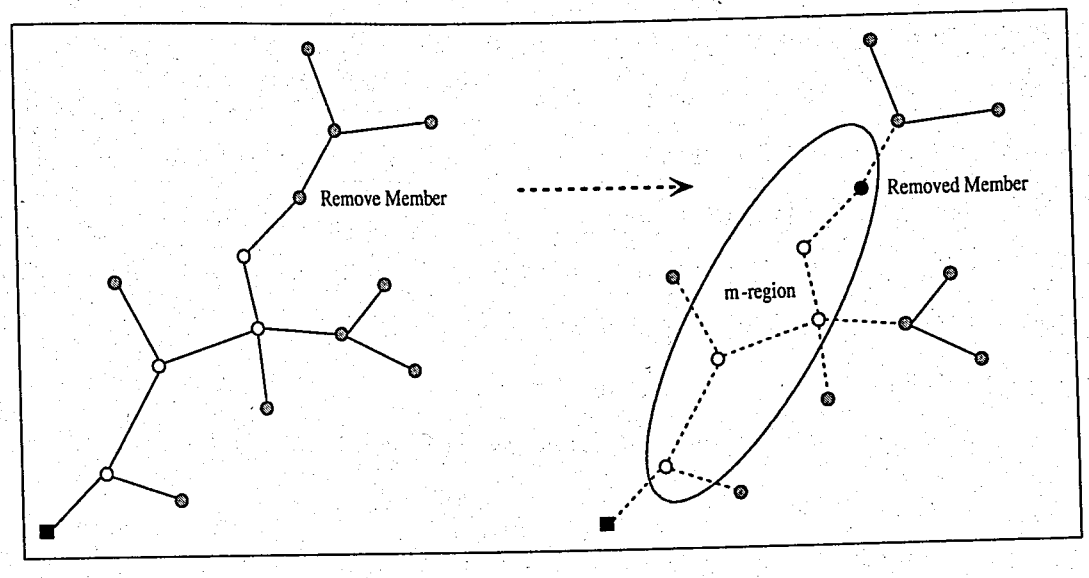

Figure 3.11. Removal of a member and resulting m-region in ARIES and CRCDM

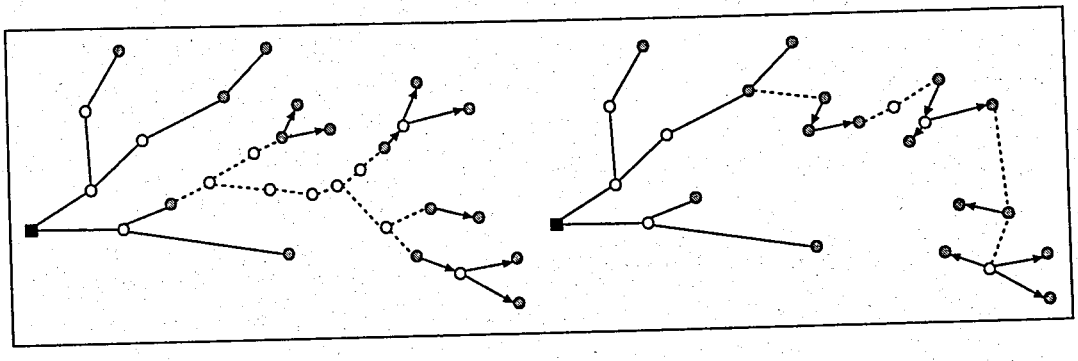

Figure 3.12. Tree before rearrangement and tree after rearrangement

from the rest of the m-region. Removal of this maximally connected region results in unnecessary and excess amount of reconfigurations on the multicast tree and so it may result in better trees than that of some non-rearrangeable or rearrangeable heuristics. Since it is explicit that making more reconfigurations will probably result in a better tree. This means that the criteria employed in triggering the rearrangement process for ARIES and CRCDM are disputable. Moreover, excessive number of reconfigurations result in excessive differences between consecutive trees and real-time multicast sessions cannot tolerate large changes in the multicast tree after each rearrangement, because this might cause unacceptable disruptions in the packet flow. There should be other metrics capable of determining maximally influenced portion of the tree and capable of triggering rearrangement of that portion only, as the membership of a multicast group changes.

Additionally, both ARIES and CRCDM work better on networks with symmetric links. However, links are usually asymmetric in real networks. In a network with asymmetric links, both heuristics may fail because change in the direction of links in a subtree may result in a more costly tree after rearrangement of the multicast tree. As shown in Figure 3.12, direction of some links in some subtrees can change during the rearrangement procedure, and this may result in more expensive trees because of the extra cost coming from reverse links and rearrangement may not result in a better tree. A heuristic for asymmetric network should take all this into account.

In the next section, the proposed delay-constrained rearrangeable online multicast routing heuristic, CoBaCROM, is presented. CoBaCROM defines a better metric for the initiation of rearrangement and it can handle asymmetric links successfully. CoBaCROM's performance is evaluated in Chapter 4.

3.4.2. Proposed Heuristic

In the previous section, two important rearrangeable online multicast routing heuristics are evaluated in terms of their deficiencies. Both heuristics fail in networks with asymmetric links, and their rearrangement decisions depend on disputable metrics. CoBaCROM proposes a new decision making scheme in order to decide when to rearrange the multicast tree and which parts will be rearranged. This scheme deals with individual member nodes instead of regions in order to make rearrangement decisions. So, each member can even set its individual threshold for the rearrangement decision. CoBaCROM rearranges the tree by removing edges and nodes, which are maximally affected by membership changes. CoBaCROM also can handle asymmetric links successfully. As in CRCDM, only leave request will contribute to the rearrangement decision, because previous studies on the unconstrained version of the on-line multicast routing problem in [5] and [53] have shown that when new member nodes are added to a multicast tree through a minimum cost path, good performance can be obtained. In the delay-constrained case, delay-constrained least-cost paths can be used to extend this result as in CRCDM. So, it is convenient to assume that damage on the multicast tree is especially the result of leave requests.

3.4.2.1. Contract Entity of a Group Member. CoBaCROM defines an entity called *contract* for each member node in the multicast tree. The *contract* keeps two price values, $InitialPrice$ and $CurrentPrice$. $InitialPrice$ is the price of adding a node to the multicast group as a member and it is set during a join request. $InitialPrice$ can be regarded as the contribution of a member node to the cost of multicast tree at the time it is added to the multicast group. Current Price can be defined as the current contribution of the member node to the cost of the multicast tree. $InitialPrice$ and $CurrentPrice$ of a member node are equal just after the addition of that member node to the multicast tree and unlike $InitialPrice$, $CurrentPrice$ changes as membership of the group changes.

The last variable that a *contract* contains is $JointNodeID$. $JointNodeID$ is the identity of the node from which the member node is connected to the tree. It is first set at the time the member is added to multicast tree and its value changes as membership of the group changes. $JointNodeID$ of a member node helps defining other member nodes, which are affected in case of a leave request concerning the member. This is shown in Figure 3.13. During the leave request, those member nodes' *contract* entities are then updated accordingly as explained in the following sections.

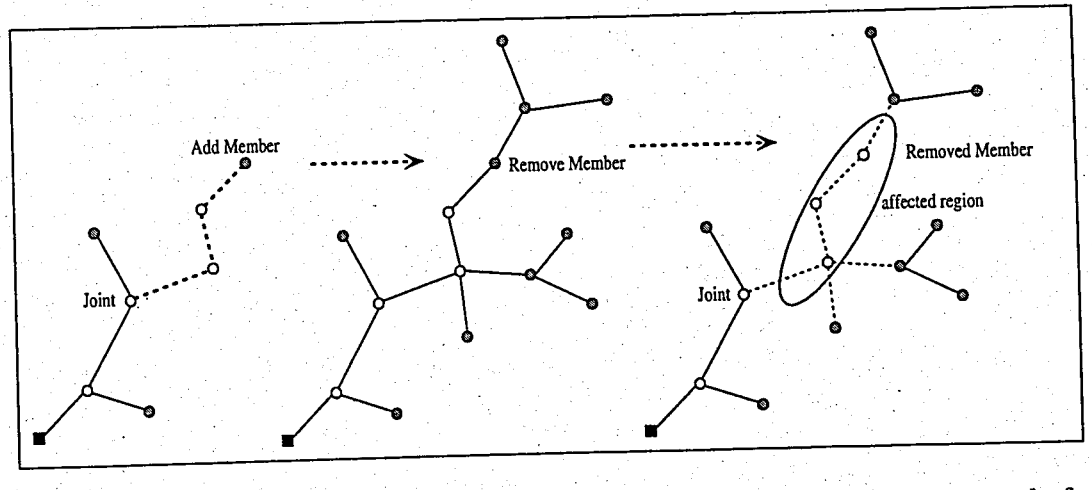

Figure 3.13. Addition of a member to multicast tree, appearance of the tree before removal of the member, and region affected by the removal

3.4.2.2. Join Requests. As a result of a join request, a member node is added to multicast tree using a delay-constrained least-cost path. Problem of finding delay-constrained least-cost paths is NP-complete [13]. However, there are fast and efficient heuristics to compute delay-constrained least-cost paths. Any of those heuristics can be used for the addition of new member node to existing multicast tree. However, the chosen heuristic should be good enough to be used in real-time communications.

In order to connect a new member node to the existing tree, unicast paths from each node in the tree to the new member node can be found and most appropriate path can be used for the connection. However, this may be costly. An alternative method is to use a subset of tree nodes as in CRCDM instead of all of them, and to find paths from those nodes to the new member node. However, in the implementation of CoBaCROM, DMBDCLC is used. DMBDCLC can find delay-constrained paths between a tree and a node within a time complexity of $O(n\log(n))$. DMBDCLC is chosen because of its low time complexity and its successful results in numerous simulations. The new member node sets *Initial Price* and *CurrentPrice* variables of its *contract* to the cost of its delay-constrained path between the tree and itself. *JointNodeID* variable of the *contract* is set to the ID of the tree node to which the new member is connected through this path. This is shown in Figure 3.14.

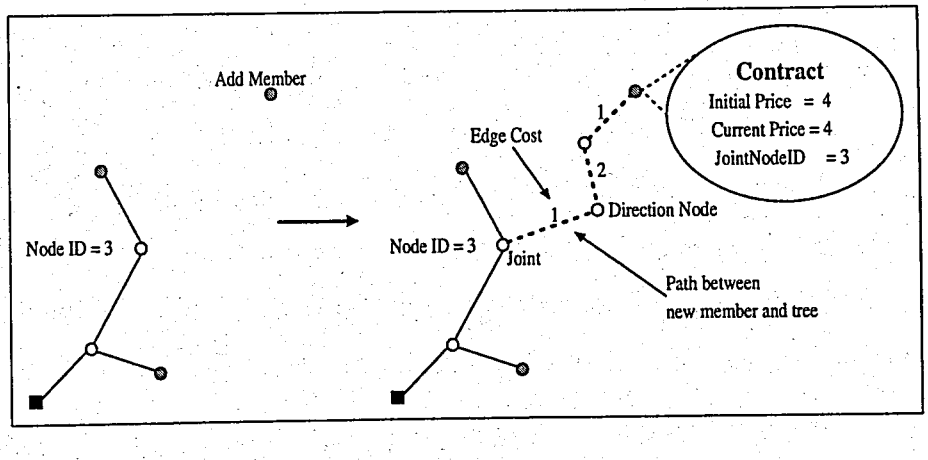

Figure 3.14. Addition of a new member node and *contract* entity of the new member

If the new member node is already in the multicast tree, it is marked as member. Then, *InitialPrice* and *CurrentPrice* variables of its *contract* are set to zero. Since

contribution of the new member node to the cost of the tree is zero. This is shown in Figure 3.15.

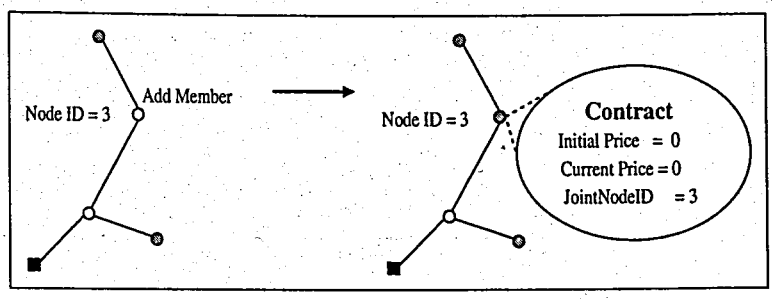

Figure 3.15. Addition of a new member and *contract* entity of the member

3.4.2.3. Leave Requests. The member node to be removed may be a leaf node or an intermediate node. Each case requires different actions to be taken during the removal of the member node. For both cases, $DirectionNode$ should be found. $DirectionNode$ is the first node after joint node of the member node on the path from the joint node to the member node in the tree. Joint node of a member node is referred by $JointNodeID$ of the member. A $DirectionNode$ is shown in Figure 3.14.

If a member node is a leaf node, a pruning algorithm is run. Pruning algorithm first deletes the member node from the tree and it continues to delete its ancestors iteratively until it meets a node with more than one child, or a member node, or a source node. Pseudo-code of the pruning algorithm is shown in Figure 3.17 and a demonstration of the algorithm is shown in Figure 3.16. Pruning algorithm returns ID of the node at which pruning is stopped. At this point, removal of the member node from the multicast group and the multicast tree is accomplished successfully if this ID is equal to $JointNodeID$ of the member node. This means that none of other member nodes will be affected by the removal of this member node. But there will be an exception. This exception will be explained at the end of this subsection.

If the ID returned by the pruning algorithm and the $JointNodeID$ are not equal, there should be other members, which will be influenced by the removal of the member node. This case is shown in Figure 3.18. Member nodes, which are connected to the

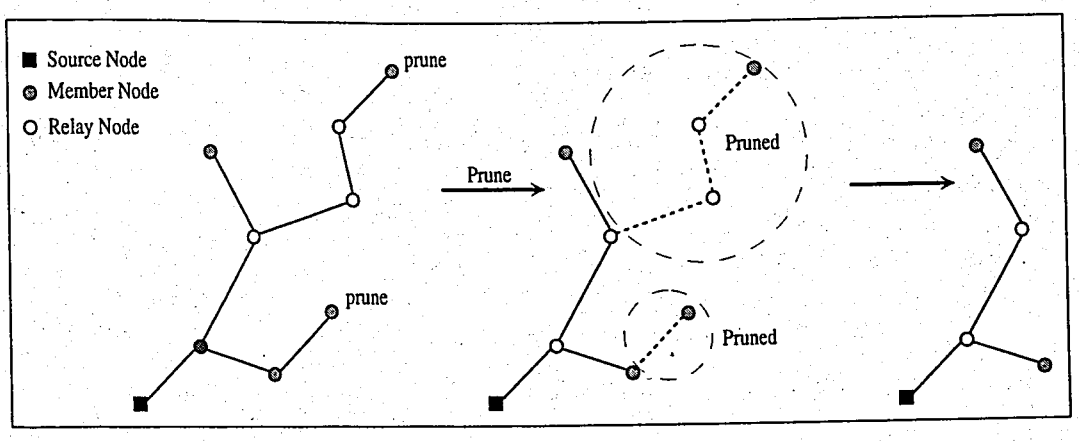

Figure 3.16. An example of pruning procedure

```
function Prune(int nodeID,Tree tree){
    tree.Nodes[nodeID].IsMember = false;
    if(tree.Nodes[ID].childs.size() >0) return ID;
    int ID=nodeID;
    while(tree.Nodes[ID].childs.size() <2 && !tree.Nodes[ID].IsMember && !tree.Nodes[ID].IsSource){
            int parentID = tree.Nodes[ID].parentID;
            tree.deleteNode(ID),
            ID = parentID;
    }
    return ID;
}
```

Figure 3.17. Pseudo-code for pruning algorithm

multicast tree using the nodes on the path between the removed member node and its joint node, are affected by the remove request. The *contract* entities of those members should be updated to reflect their more realistic contribution to the cost of the current multicast tree. Figure 3.19 explicitly shows an example of the dramatic change in the contribution of a member node to the cost of the multicast tree in case of a leave request.

SharePrices algorithm is used to update *contract* entities of members, which are influenced by the leave request. This algorithm starts from the joint node of the removed member and goes toward the direction of the *DirectionNode* until it visits all member nodes in the affected region. This algorithm makes member nodes in the affected region share the cost of edges, by which they connect to the joint node or

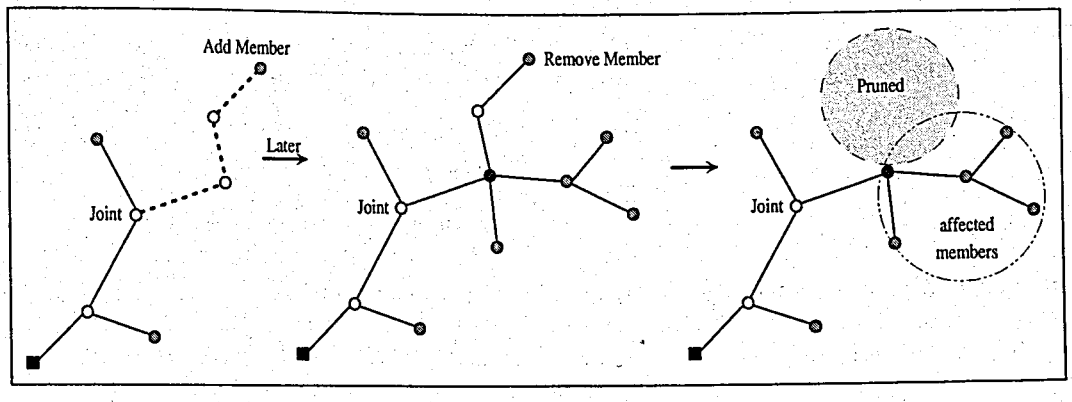

Figure 3.18. Addition of a member node, removal of the same node and member nodes affected by this remove request

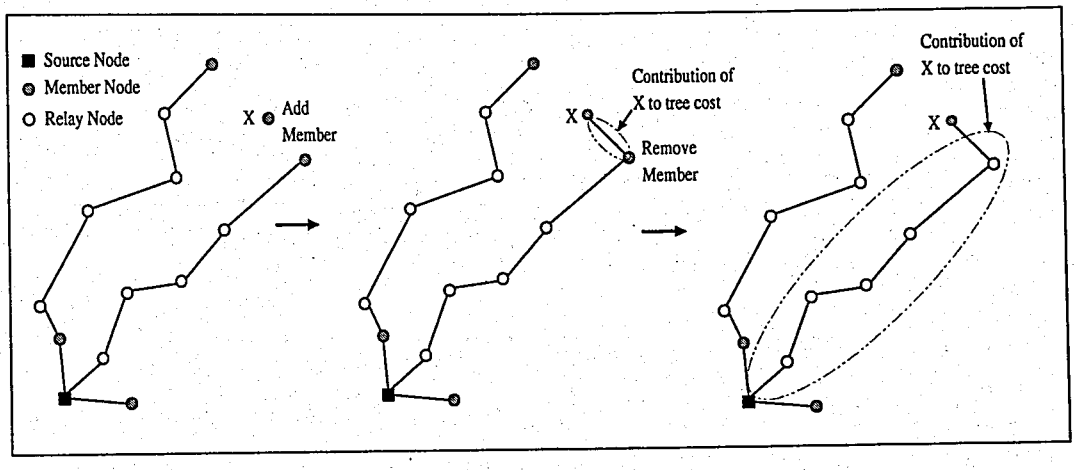

Figure 3.19. Dramatic change in the contribution of a member node to cost of the tree

another member node in the affected region. So, members in the affected region share the contribution of the region to the cost of the multicast tree. Figure 3.20 shows the pseudo-code for *SharePrices* algorithm.

In the pseudo-code, *parentEdge* object represents the edge by which a node is connected to its parent node, so to the tree. Figure 3.21 shows the removal of a member node and result of *SharePrices* algorithm. The member nodes in the region uses *SharePrices* algorithm to share cost of edges in the affected region as in Figure 3.21. The members using those edges share the cost of them. Any cost sharing strategy can be used. However, *SharePrices* algorithm is used as the cost sharing strategy in CoBaCROM. Algorithm starts from the joint node of a removed member and goes

```
function SharePrices(int JointNodeID,int DirectionNodeID) {
    SharePrice(JointNodeID, DirectionNodeID , false);
}
function SharePrice(int nodeID,int directionID, bool addStartPrice) {
    for(int i=0;i<tree.Nodes[nodeID].childs.size();i++){
        if(directionID!=-1 && directionID != i)  continue;
        int childID = tree.Nodes[directionID].childs[i];
        double initialprice = 0
        if(addStartPrice)   initialprice = tree.Nodes[nodeID].Contract.CurrentPrice / tree.Nodes[nodeID].child s.size();
        tree.Nodes[childID].Contract.CurrentPrice =  initialprice + tree.Nodes[childID].parentEdge.cost;
        if(tree.Nodes[nodeID].childs.size() > 1||tree.Nodes[nodeID].IsMember || tree.Nodes[nodeID].IsSource){
            tree.Nodes[childID].Contract.JointN odeID = nodeID;
        }else{
            tree.Nodes[childID].Contract.JointNodeID = tree.Nodes[nodeID].Contract.JointNodeID
        }
        if(!tree.Nodes[childID].IsMember)  {
            SharePrice(childID,-1, true);
            tree.Nodes[childID].Contract.clear();
        }else{
            SharePrice(childID,-1, false);
        }
    }
}
```

Figure 3.20. Pseudo-code for *SharePrices* algorithm

toward the direction of its *DirectionNode*. Algorithm uses a variable to keep the sum of edge costs on the direction. At joint node, this sum is set to zero. As it visits other edges in the direction of *DirectionNode*, it adds the cost of the edges to the sum. In a node with more than one child, this sum is divided to the number of child nodes and old sum variable is not used any more. The result of this division is assigned to new variables; each is a variable to keep the sum of edge costs at the direction of each child node. If the algorithm meets a member node, value of the corresponding sum variable is set as the value of *CurrentPrice* of this member node and *JointNodeID* of the member is set to the identity of the member's nearest ancestor node, which is a member node or a source node or a node with more than two childs. Then, the value of this sum variable is set to zero and the algorithm goes similarly toward the same direction until it reaches a leaf node. So, the cost of edges in an affected region is shared by the member nodes in that region. After the *SharePrices* algorithm, *CurrentPrice* values of the member nodes in the affected region reflect their more realistic contributions to the cost of the multicast tree. *JointNodeID* values of those member nodes are also updated accordingly by *SharePrices* procedure. Other approaches to share prices

are also plausible, but this approach is used in the implementation of CoBaCROM. Essence of the *SharePrices* procedure is to compute more realistic quantities for the contribution of member nodes to the cost of the multicast tree after a leave request.

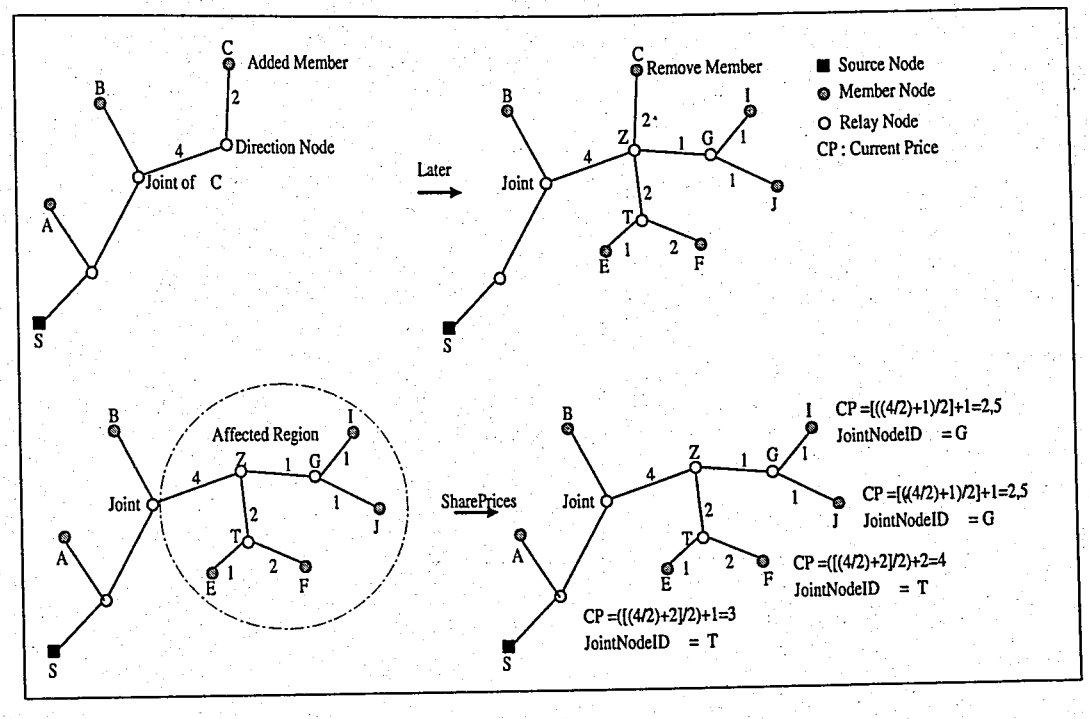

Figure 3.21. Removal of a member node and result of *SharePrices* algorithm

It was previously noted that there is an exception in which there will be members affected by a leave request, even if the removed member node is pruned until its joint node. That is the case that the removed member was sharing cost of some edges with other member nodes. If this member is completely removed from the tree by pruning it until its joint node is reached, the shared cost must be re-shared among the other member nodes. The portion of the cost shared by the removed member node becomes an excess cost. This excess cost is distributed among those members sharing edge costs with the member node. This exceptional case is shown in Figure 3.22.

If a leave request concerns a member node, which is an intermediate node instead of a leaf node, the pruning algorithm does not prune any node. This member node becomes a relay node. If *CurrentPrice* of the member node is non-zero, *CurrentPrice* values of member nodes in the region affected by this leave request is updated using the joint node and *DirectionNode* of the leaving member node. This update is done using

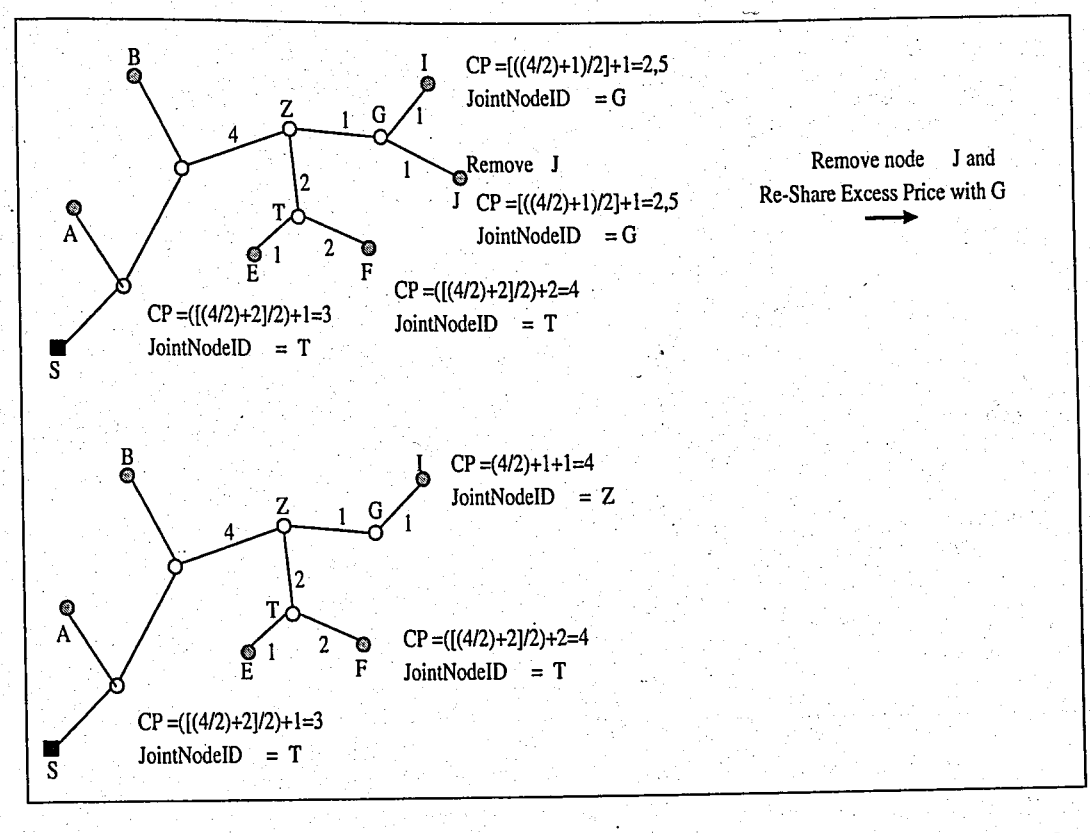

Figure 3.22. Removal of a leaf member node, which shares some cost with other members

SharePrices algorithm just like explained in the previous subsection. *SharePrices* algorithm changes *CurrentPrice*, not *InitialPrice*. This is shown in Figure 3.23.

If *CurrentPrice* of the leaving member is zero, nothing is done except the node becomes a relay node. This means that this member node was on the tree before becoming a member and its contribution to the cost of the tree is zero. So, it is safe to do nothing except making it a relay node. This case is shown in Figure 3.24.

3.4.2.4. Rearrangement. The *contract* entity of each member helps the member node make decisions on rearrangement. During the addition phase of a member, the new member selects the least-cost path to the tree among the other alternative paths so that the cost of the tree will increase minimum with the addition of this new member. However, this selection may be much more costly in the future because of a leave request. In Figure 3.19, such a dramatic change in the contribution to the cost of the

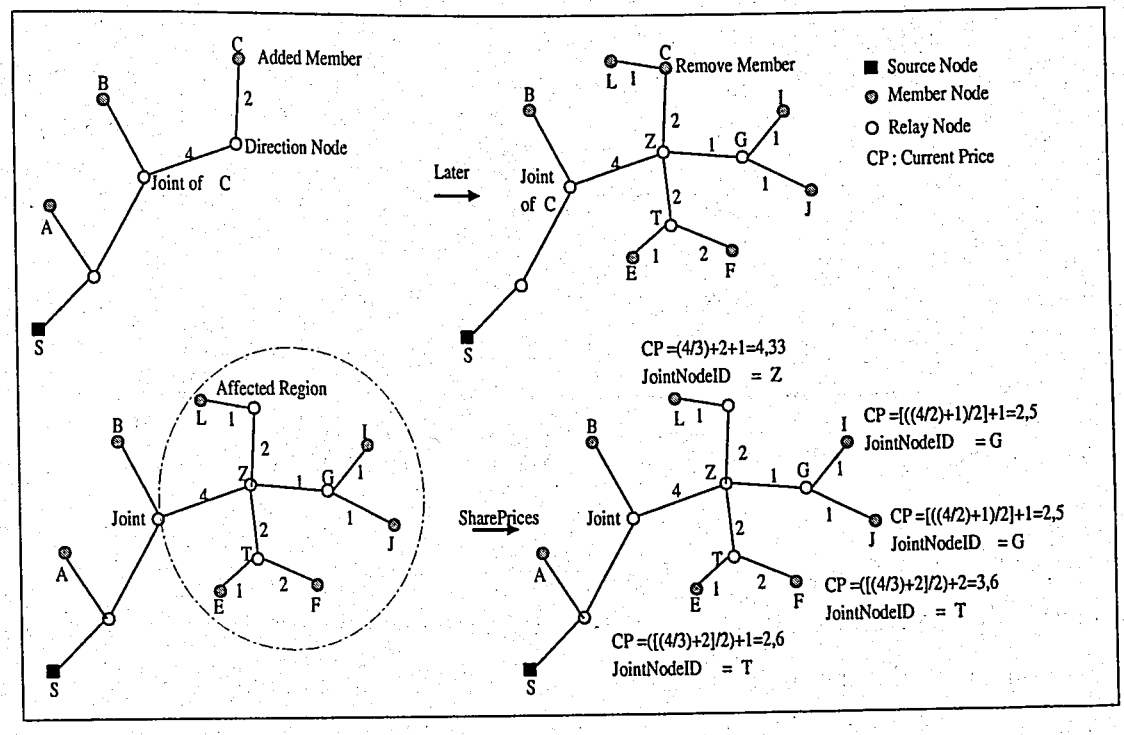

Figure 3.23. Removal of a non-leaf member node and result of *SharePrices* algorithm

tree is shown. In the figure, a new member selects to connect to the tree through a path, but the path becomes more costly after a leave request.

In order to make healthy decisions on rearrangement, a member should remember under which conditions it has selected the path by which it is currently connected to the rest of the tree. Current price of this connection over that path is also important in rearrangement decision. If the conditions making a memeber node select its current path become invalid because of the changing membership, the member node should revise its current connection. The *contract* entity keeps this data and rearrangement decisions are done using *InitialPrice* and *CurrentPrice* variables. Percentage increase in the price of keeping a member node in the tree using the path by which it is initially added to the multicast tree plays an important role in making rearrangement decision. Calculation of this percentage increase is calculated as shown in Equation 3.11. Calculation of the percentage increase resembles the calculation of inflation so it is called as *inflation*.

$$Inflation = \frac{CurrentPrice - InitialPrice}{InitialPrice} \times 100 \qquad (3.11)$$

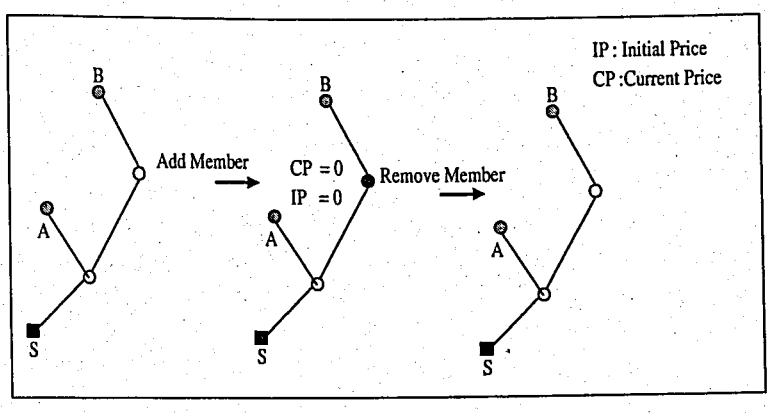

Figure 3.24. Removal of a member whose *CurrentPrice* and *InitialPrice* are zero

A threshold, which is called $InflationTolerance$ (τ), is used to decide when to rearrange a member node's connection to the tree. The τ is the maximally tolerable inflation for a member to stay in its current on-tree location using its current path in the tree. Each member can define its τ or a global τ can be used for the whole session as in our simulations. If $inflation$ calculated for a member node exceeds τ, this member node is rearranged and some of the other member nodes may also be affected from this rearrangement.

After a decision is made on the rearrangement of a member node, a region called r-region is defined. An r-region is defined as a region containing the path from the joint node of the member node to that member node in the tree. This region also contains paths to subtrees branching from the nodes in this path, except the member node and the joint node. Leaf nodes of each branching subtree are member nodes. Those subtrees are at the leaves of the r-region. Edges and relay nodes in the r-region are then removed and resultant subtrees are connected to the source subtree using delay-constrained least-cost paths. Figure 3.25 shows how a member node in rearrangement defines its r-region and resultant subtrees after removal of edges and relay nodes of that r-region.

Any heuristic can be used to find delay-constrained paths. However, DMBDCLC is used in CoBaCROM. DMBDCLC can find delay-constrained paths between a source subtree and another subtree with a time complexity of $O(n\log(n))$. DMBDCLC is chosen for its low time complexity and its successful results in numerous simulations.

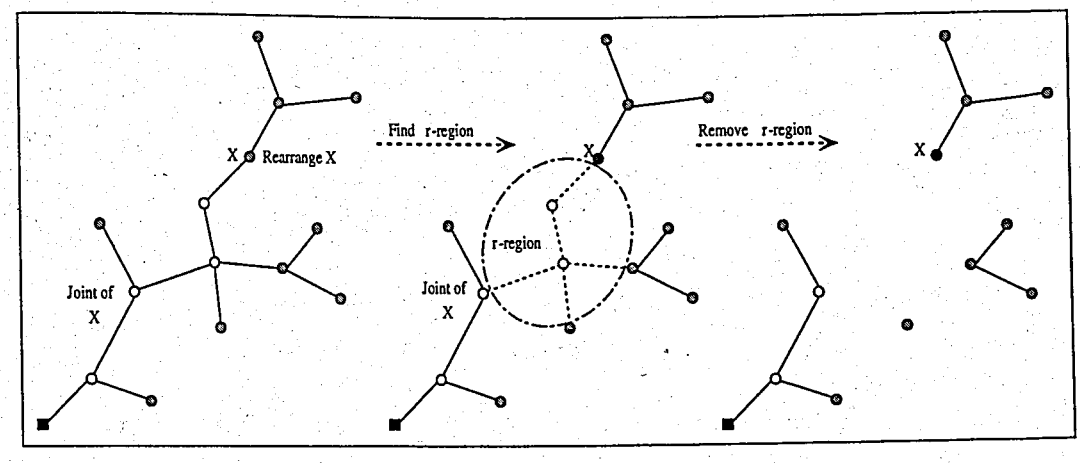

Figure 3.25. How a member node in rearrangement defines an r-region and resultant subtrees

There is an exception for rearrangement if a member node's *InitialPrice* is zero. If its *CurrentPrice* is also zero, there is no need to compute $inflation$ for rearrangement decision and rearrangement is not initiated. If its *CurrentPrice* is greater than zero, rearrangement is initiated without computing $inflation$. For this case, r-region also contains paths to the subtrees branching from the member node deciding rearrangement. This is shown in Figure 3.26.

There can be different approaches on how to connect subtrees to the source subtree. Let *MaxCrossDelay* be the maximum of end-to-end delay between any two nodes in a subtree. A subtree with greater *MaxCrossDelay* is connected to source subtree first. The way that subtrees are connected to the source subtree and rearrangement is accomplished is explained in the following paragraph.

Let a subtree T_1 be connected to source subtree T_S using a path P. First node in P is node X, which is a node in T_S and last node of P is node Y, which is a node in T_1. Other nodes in P are neither in T_S nor in T_1. After connecting T_S and T_1 using P, source subtree becomes $T_S \cup P \cup T_1$. Let node Z be the root node of T_1. If node Y is different from node Z, then *SharePrices* algorithm is run starting from node X through direction of the path P. In case that node Y is different from node Z, some links are reversed during the merging of the two subtrees and it may be much more costly for some members in T_1 to connect the tree via their current paths.

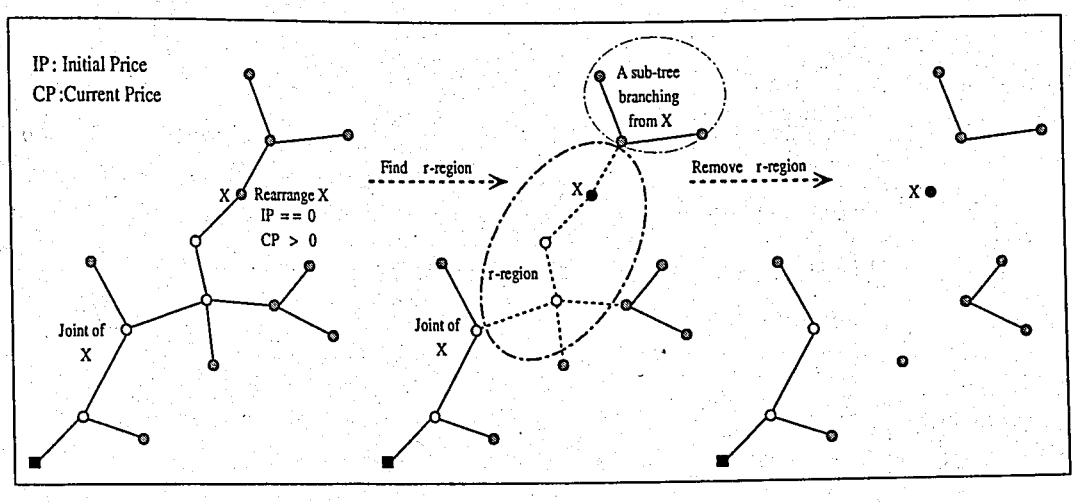

Figure 3.26. How a member node in rearrangement defines an r-region and resultant subtrees in case its *InitialPrice* is zero and its *CurrentPrice* greater than zero

So, *SharePrices* algorithm also helps CoBaCROM work in networks with asymmetric links by making CoBaCROM take reversed links into consideration. After connecting T_S and T_1 using P, *InitialPrice* of node Y is set to its *CurrentPrice*. In Figure 3.27, a scenario is shown to illustrate how *InitialPrice* and *CurrentPrice* values change and how rearrangement is initiated and accomplished. Nodes in Figure 3.27 are labelled according to *InitialPrice* : *CurrentPrice* format and numbers above the edges in the figure show costs of the edges. Rearrangement procedure is triggered, when $inflation$ of a member is more than τ, which is less than 200 per cent in the scenario of Figure 3.27.

Lemma 3.4.1: Worst-case complexity of rearrangement operation is $O(n^2 \log(n))$ (if DMBDCLC is used as delay-constrained least-cost path heuristic).

Proof: In the rearrangement operation, after removing an r-region, resulting subtrees are connected to the source subtree. Rearrangement operation is dominated by connecting these subtrees to source subtree to construct better multicast trees. In the worst-case scenario, there will be $n-2$ one-node subtrees and a source subtree consisting only of the source node. If DMBDCLC is used to connect a subtree to the source subtree, the worst-case time complexity of connecting a subtree to source subtree is $O(n \log(n))$. So, for $n-2$ subtrees, the total worst-case time complexity becomes $O((n-2) \cdot n \log(n)) = O(n^2 \log(n))$. ■

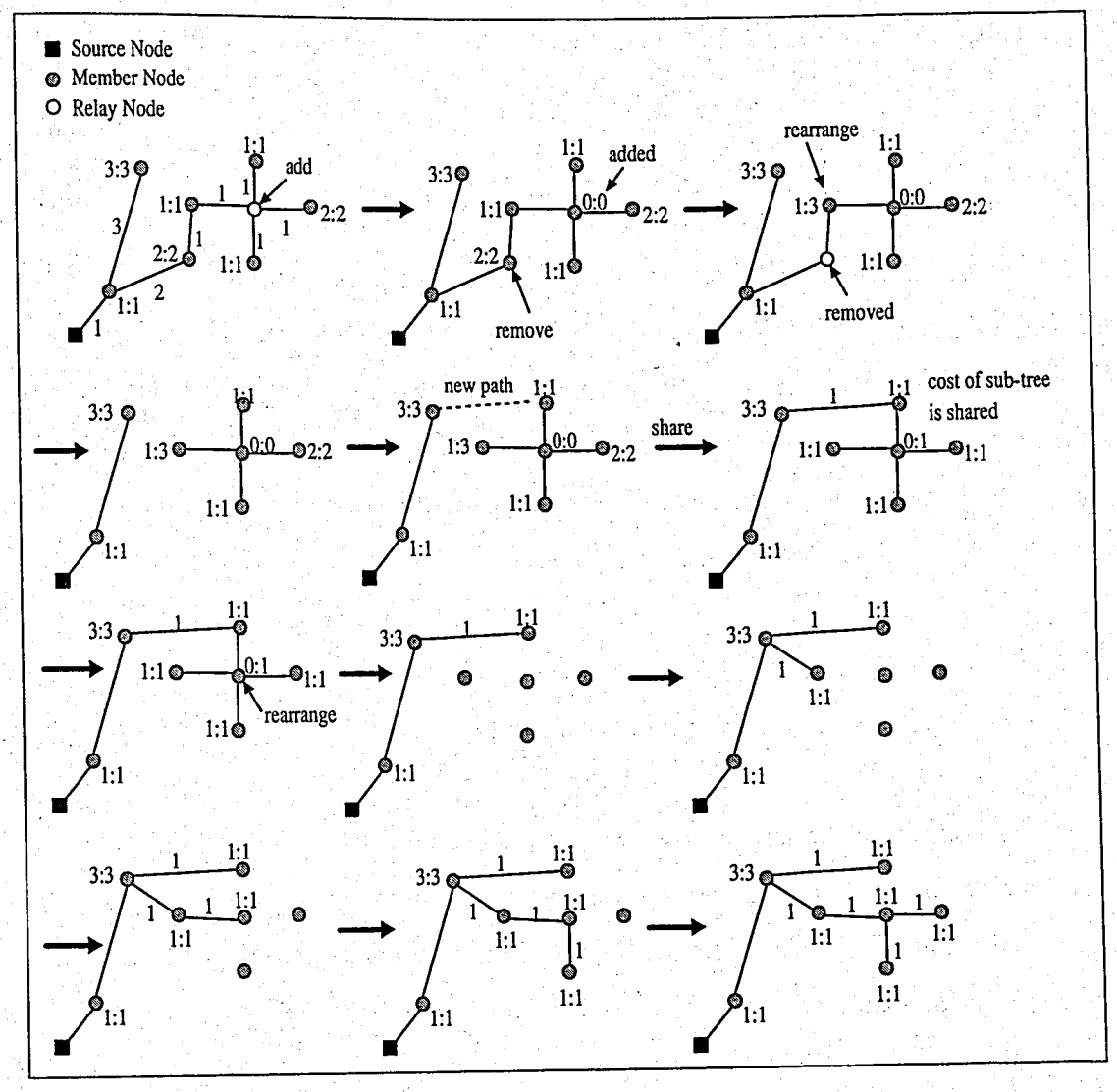

Figure 3.27. How *InitialPrice* and *CurrentPrice* values change and how rearrangement is initiated and accomplished

Lemma 3.4.2: Worst-case complexity of CoBaCROM for a request is $O(n^2 \log(n))$ (if DMBDCLC is used as the delay-constrained least-cost path heuristic).

Proof: A request can either be a join request or a leave request. If it is a join request, complexity of adding a new member to the multicast tree is $O(n \log(n))$. In the worst-case scenario of a leave request, rearrangement is triggered and the leave request procedure is dominated by the rearrangement operation. If DMBDCLC is used, worst-case time complexity of rearrangement operation is $O(n^2 \log(n))$ by lemma 3.4.1. So, the total worst-case time complexity of CoBaCROM for a request is $O(n^2 \log(n))$. ■

4. PERFORMANCE ANALYSIS OF THE PROPOSED HEURISTICS

In this chapter, we present the results of our simulations, which are conducted to analyze the performance of the proposed heuristics; DMBDCLC, DeCoNROM and CoBaCROM. In order to compare our heuristics with different heuritics from multicast routing literature, a network simulator is implemented in C++ using well-known random network and random request generation models. Our simulator is run under Windows XP Professional OS using a PC with Intel Pentium 4 1.2 GHz CPU and 256 MB RAM.

4.1. Performance Metrics

In order to analyze the performance of our heuristics, we used the metrics described below. Other heuristics in the literature uses similar metrics to evaluate their performance. However, some additional metrics are also used to measure the performance of our heuristics. Those additional metrics are the number of rearrangements, the gain per rearrangement, the tree change per rearrangement and the number of subtrees per rearrangement.

i. **Cost Competitiveness (CC):** This metric evaluates how well a given heuristic performs in terms of the cost of the multicast tree it produced with respect to the optimal algorithm. However, the problem of finding optimal delay-constrained least-cost trees is NP-complete. So, it is not practical to use the optimal algorithm for large networks for the comparison of the trees produced by different heuristics. For the basis of comparison, BSMA [36] is used in our simulations. BSMA is an iterative heuristic proposed to find delay-constrained least-cost multicast trees in polynomial time. BSMA is the best polynomial time heuristic, in terms of tree cost, among other heuristics proposed for the problem [38]. That is why we choose to use BSMA for the comparison. For an online multicast session with join/leave

request set R, CC is calculated with the formula in Equation 4.1. T_i is the tree computed by a heuristic after i^{th} request and T_i^{BSMA} is the tree computed by the heuristic BSMA after i^{th} request. $C(T)$ represents the cost of a tree, T. A heuristic with a low CC value indicates that it is able to construct multicast trees whose cost is not much larger than the cost of a multicast tree constructed by the optimal algorithm. Since CC is computed with respect to BSMA and BSMA produces good multicast trees in terms of cost [36, 38].

$$CC = \frac{\sum_{i=1}^{|R|} \left[\frac{C(T_i)}{C(T_i^{BSMA})} \times 100 \right]}{|R|} \tag{4.1}$$

ii. **CPU Time:** This metric measures the average CPU time required for a heuristic to construct a multicast tree after a join/leave request in an online multicasting session and it is measured in seconds. CPU time is computed using the formula in Equation 4.2. $Time(T_i)$ represents the CPU time required to compute a tree by a heuristic after i^{th} request of R.

$$CPUTime = \frac{\sum_{i=1}^{|R|} [Time(T_i)]}{|R|} \tag{4.2}$$

iii. **Change per Request (CPRQ):** Each request result in a difference in the tree. This difference can be measured as the difference in edges between two consecutive multicast trees. Each new edge in the tree means a new connection. If a request results in too much difference between consecutive trees, it may be hard to handle the real-time communications properly. CPRQ is computed using the formula in Equation 4.3. $(T_i \cup T_{i-1})$ represents the union of edges in two consecutive trees, the tree after $i-1^{th}$ request and the tree after i^{th} request. $(T_i \cap T_{i-1})$ represents common edges in the two consecutive trees. $|T_i|$ represents the number of edges in T_i.

$$CPRQ = \frac{\sum_{i=1}^{|R|} \left[\frac{(T_i \cup T_{i-1}) - (T_i \cap T_{i-1})}{|T_{i-1}|} \times 100 \right]}{|R|} \tag{4.3}$$

iv. **Number of Rearrangements (NR):** Rearrangement procedure is triggered

depending on the cumulative damage of leave requests on the tree. Although they result in better trees in terms of cost, rearrangements are costly in terms of CPU time and they may cause great differences between consecutive trees. So, NR in a session for an online multicast routing heuristic is a good metric to measure its performance.

v. **Gain per Rearrangement (GPR):** As mentioned before, rearrangements may be costly, so it should be worth making rearrangements. Gain of a rearrangement is a good metric to measure how successful a heuristic on making rearrangement decisions. In a non-rearrangeable online multicast routing heuristic, leave requests are accomplished by the pruning algorithm, which prunes a member node if it is a leaf node and does nothing if it is a non-leaf node. In rearrangeable heuristics, after running the pruning algorithm for a leave request, rearrangement decision is made and rearrangement is performed. GPR is computed using the formula in Equation 4.4. Let i^{th} request be a leave request for which a rearrangement is made and let T_i represent the tree after i^{th} request. Then, T_{i-1}^{pruned} represents the tree after running pruning algorithm on T_{i-1} according to the i^{th} request. How successful the rearrangements are in a session is measured by this metric. Shortly, GPR is the average percentage achievement of a rearrangement in terms of multicast tree cost for a session.

$$GPR = \frac{\sum_{i=1}^{|R|} \left[\frac{C(T_i) - C(T_{i-1}^{pruned})}{C(T_{i-1}^{pruned})} \times 100 \right]}{NR} \tag{4.4}$$

vi. **Change per Rearrangement (CPR):** This metric is similar to GPR, but it measures the percentage difference between pruned tree and rearranged tree in terms of edges.

$$CPR = \frac{\sum_{i=1}^{|R|} \left[\frac{(T_i \cup T_{i-1}^{pruned}) - (T_i \cap T_{i-1}^{pruned})}{|T_{i-1}^{pruned}|} \times 100 \right]}{NR} \tag{4.5}$$

vii. **Number of Subtrees per Rearrangement (NSPR):** This metric measures the average number of subtrees per rearrangement produced by a heuristic's re-

arrangement procedure during a session. Average number of subtrees to be connected during a rearrangement procedure may be an important metric to evaluate the performance of the heuristic in terms of time complexity and change between consecutive trees.

4.2. Simulation Model

Simulation model used for the evaluation of the heuristics is adopted from the literature. This model was first proposed by Waxman [7] and is frequently used after Waxman for the evaluation of online multicast routing heuristics [5, 31, 36, 38, 42, 53]. In the next subsections, model is described in detail.

4.2.1. Random Graph Model

Random graph generation model is adopted from the Waxman's model in literature [7]. In the model, nodes are placed randomly in a rectangular coordinate grid by generating uniformly distributed values for their x and y coordinates. In our simulations, rectangular grid is chosen as $20000km \times 20000km$ and Takuji Nishimura's pseudo-random number generator [54] is used to generate uniformly distributed pseudo-random numbers. By generating a random spanning tree connecting all nodes in the graph, connectivity of the random graph is ensured. That is done iteratively choosing two random nodes, one in the spanning tree and other not in the spanning tree, and then an edge is generated to connect the two nodes. Other edges in the graph are generated by examining each pair of nodes, $[u, v]$, in the graph as a candidate for the generation of a new edge. A random number, $rand$, is generated for the node pair $[u, v]$, so that $0 \leq rand \leq 1$. The edges, (u, v) and (v, u), are included in the graph if $rand$ is less than $Pr(u, v)$, which is defined as in Equation 4.6 by Waxman.

$$Pr(u, v) = \beta \times e^{\frac{-d(u,v)}{\alpha \cdot L}} \tag{4.6}$$

In Equation 4.6, $d(u, v)$ is the Euclidian distance between two nodes u and v, α and β are tunable parameters, and L is the maximum possible distance between two nodes

in the graph. An increase in α results in an increase in the number of connections between far nodes and an increase in β results in an increase in the degree of each node.

This method of random graph construction produces graphs, which model some aspects of real networks. For values of 0.25 and 0.2 for α and β, generated graphs roughly resemble the graphs of the major nodes in the Internet [5]. So, those values are used in our simulations for α and β. However, there is one problem related to this model. As the number of nodes increases, the number of edges from each node, so the degree of each node tends to increase. A modification to the model is proposed in [5] to solve this problem and this new version of the model is used in our simulations. Modified definition of $Pr(u,v)$ is shown in Equation 4.7.

$$Pr(u,v) = \left[\frac{K \cdot \overline{\varepsilon}}{n}\right] \times \beta \times e^{\frac{-d(u,v)}{\alpha \cdot L}} \tag{4.7}$$

In Equation 4.7, $\overline{\varepsilon}$ is the mean degree of a node and n is the number of nodes in the graph. In order to ensure that mean degree of each node remains constant, another scale related to the mean distance between two random nodes is introduced. This factor is K and its value is 25 for $\alpha = 0.25$ and $\beta = 0.2$ [5]. In our simulations, $\overline{\varepsilon}$ is taken as four. Additionally, minimum and maximum degrees of a node is bounded as two and eight respectively to resemble real networks.

For the edges, 155 Mbps dual links are used. Propagation delay of a link is computed using the length of the link in $20000km \times 20000km$ rectangular grid. Cost of a link is a real number and randomly chosen between one and ten in our simulations. So, cost of a link (u,v) may be 10 times greater than the cost of reverse link (v,u) if links are chosen to be asymmetric. If links are chosen to be symmetric, costs of (u,v) and (v,u) are the same. Our simulations are run for asymmetric links.

4.2.2. Random Request Model

In order to generate join/leave requests for a session in the simulations, another model from the literature is used. This model was also proposed by Waxman [7] and it is employed in the online multicasting literature frequently [5, 31, 36, 38, 42, 53]. In the model, probability of a join-request is given by Equation 4.8. In the equation, n is the network size and m is the number of multicast group members as well as γ is a constant in the range (0,1). The value of γ determines the equilibrium point at which the probability of a join or leave request is equally likely.

$$Pr(join) = \frac{\gamma \times (n-m)}{\gamma \times (n-m) + (1-\gamma) \times m} \tag{4.8}$$

In Equation 4.8, when $\gamma = \frac{m}{n}$, the probability becomes 0.5. For our simulations, $\gamma = \frac{m}{n}$ and the probability of a join or leave request is equally likely. For each simulation, 100 random join/leave requests are generated.

Let Δ_{i-1} be the delay bound after $i-1^{th}$ request and i^{th} request be a join request for node X. If node X does not have a least-delay path bounded by Δ_{i-1}, delay bound is re-negotiated for the new member. Delay bound becomes $\Delta_i = scale \times LD(X)$, where *scale* is a scaling factor and $LD(X)$ is delay of the least-delay path between source node and node X in the graph. $LD(X)$ is computed using the Dijkstra algorithm with delay as the metric. During the simulation design phase of this work, different scale factors, in the range of (1.25-5.25), are used for scale factor selection. Simulations for different scale factors give similar results in terms of comparison of our heuristics with others. However, for low scale factors, it may not be possible to add the new member to the multicast tree for any of the heuristics because of the fact that the current structure of the multicast tree may not let the existence of the least-delay path in the tree. Similarly, for high values of scaling factor, delay bounds become so loose that the problem turns into unconstrained. So, for our simulations, the scaling factor is set as two.

4.3. Heuristics for Comparison

For the comparison of the proposed heuristics, several heuristics from the literature are used. Those heuristics are; LC, LD, CDKS, BSMA, DDCLCMR and CRCDM. For the performance evaluation of DMBDCLC, BSMA heuristic is used. Multicast trees produced by BSMA are compared with that of FBSMA, which uses DMBDCLC as the underlying delay-constrained unicast routing heuristic. Implementations of BSMA and FBSMA are exactly the same except DMBDCLC is used instead of a k-shortest path based delay-constrained least-cost path heuristic. BSMA is chosen, because of the fact that BSMA was shown to be the best heuristic in terms of tree cost among other heuristics proposed for delay-constrained offline multicast routing problem [38].

A non-rearrangeable online multicast routing heuristic, DDCLCMR [41], is used for the performance evaluation of DeCoNROM and a rearrangeable online multicast routing heuristic, CRCDM [42], is used for the performance evaluation of CoBaCROM. CRCDM's parameters are set accordingly so that performance of CRCDM in terms of the tree cost will be the best. CRCDM's κ parameter for join requests is set to the tree size. CRCDM's minimum region utilization threshold ρ for rearrangement decision making is set to infinity so that CRCDM will try to rearrange the tree after almost each leave request. So, these parameters result in the best possible performance of CRCDM, as explained in [42]. Performance of CoBaCROM with different parameters is compared with the CRCDM, which is parameterized for the best performance. Original implementation of CRCDM uses preferred-link based delay-constrained least-cost routing heuristic [43]. However, CRCDM is implemented using DMBDCLC, because CoBaCROM is implemented using DMBDCLC. CRCDM and CoBaCROM will not be comparable, if they do not use the same underlying unicast routing heuristic. Moreover, it is shown that the worst-case time complexity of the heuristic used in the original implementation of CRCDM goes exponential [55]. So, CRCDM would not be suitable for real-time communication if it was implemented as in [42].

Two shortest path-based unconstrained heuristics are also used for the comparison. Those heuristics are LC and LD, which are the same as NAIVE approach of

Doar and Leslie with cost metric and delay metric respectively [5]. LC calculates multicast trees by combining least cost paths from source node to each of multicast group members. Similarly, LD calculates multicast trees by combining least delay paths from source node to each of multicast group members. Both LC and LD use Dijkstra's shortest path algorithm. CDKS heuristic is one of the constrained shortest path based multicast routing heuristics in the literature. It is a combination of LC and LD. In CDKS, paths of LD replace paths of LC, which are violating the delay constraint. CDKS is also used in our simulations for comparisons.

4.4. Simulation Results

In order to compare performance of our heuristics in sparse and dense multicast sessions, four groups of multicast sessions are used. Those groups are classified according to initial member size of sessions with respect to network size. Member size does not change much during a session because join and leave requests are equally likely. In those four groups, initial member sizes are 10, 20, 40 and 60 per cent of the network size.

4.4.1. Simulation Results for DMBDCLC

Performance of DMBDCLC as an underlying unicast routing heuristic in delay-constrained multicast routing is evaluated using simulations on randomly generated networks. An online multicast session is simulated for each network. A multicast session has 100 join/leave requests. After each request a new delay-constrained multicast tree is computed for the new set of multicast members using BSMA and FBSMA. BSMA uses an embedded k-shortest path based delay-constrained least-cost path heuristic as underlying delay-constrained unicast routing heuristic. Multicast trees produced by BSMA are compared with that of FBSMA, which uses DMBDCLC as underlying delay-constrained unicast routing heuristic. Complexity of k-shortest path based heuristic is $O(kn^3)$ and that of DMBDCLC is $O(n\log(n))$. BSMA heuristic is run for different k values of its k-shortest path algorithm for better comparison. Those k values are 100 and 1000. Total 800 different networks are simulated, 400 for

50-node networks and 400 for 100-node networks. Simulation results are presented and evaluated in this section. In our simulations, it is shown that DMBDCLC is a fast and efficient heuristic by figuring out that FBSMA can compute the same quality multicast trees much faster than BSMA without trading off tree cost for speed.

4.4.1.1. Cost Competitiveness. Average CC of BSMA and FBSMA heuristics for 50-node networks are shown in Figure 4.1 and Figure 4.2 for different k values. In the figures, CC for 400 50-node networks is depicted. Average CC of FBSMA looks slightly better in Figure 4.1 than average CC of FBSMA in Figure 4.2. Because, in Figure 4.2, k parameter of BSMA is 1000, whereas it is 100 in Figure 4.1. As k increases, result of BSMA is expected to approach the optimum. So, cost of the trees computed by BSMA heuristic will be less, if k increases. That is why the average CC of FBSMA looks slightly better in Figure 4.1 than the average CC of FBSMA in Figure 4.2. However, the difference between the average CC values of FBSMA is negligible. For k is 100, average CC changes between 99.996 and 100.014 and for k is 1000, average CC changes between 100.015 and 100.037. This means that 100 is a good value for k parameter of k-shortest path algorithm used in BSMA heuristic, because BSMA with this k value produces results similar to the case that k is 1000. This k value may be enough to find most of the constrained paths in 50-node networks. As a result of our simulations on 400 different 50-node networks, it is explicitly shown in the figures that FBSMA can successfully produce multicast trees for sparse and dense multicast groups. In terms of tree cost, those multicast trees produced by FBSMA are as good as the multicast trees produced by BSMA heuristic with different k values, 100 and 1000. FBSMA is used to evaluate the performance of DMBDCLC heuristic. Simulations show that performance of DMBDCLC in terms of tree cost is as good as the performance of k-shortest path algorithm based delay-constrained least-cost path heuristic used in BSMA for 50-node networks. Difference between performances of the two heuristics in terms of CC are negligible.

Similar simulations are run for 100-node networks to measure the performance of FBSMA, so the performance of DMBDCLC, in 100-node networks. Result of simula-

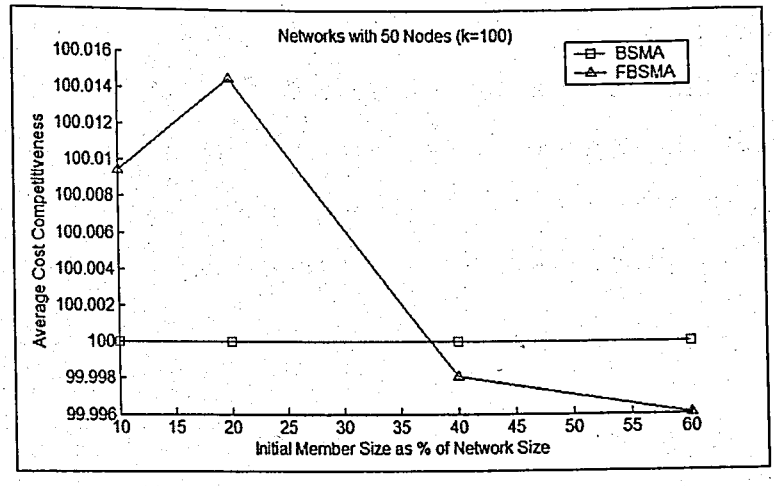

Figure 4.1. Average CC for 50-node networks (k=100)

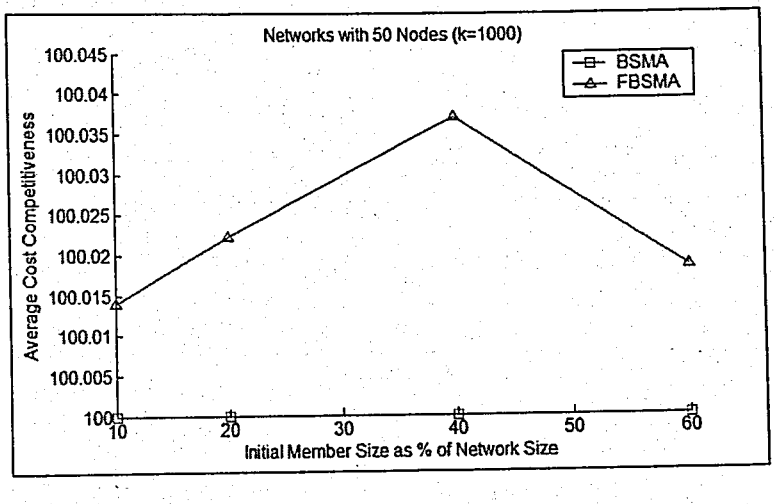

Figure 4.2. Average CC for 50-node networks (k=1000)

tions on 100-node networks is very similar to that of simulations on 50-node networks. Average cost competitiveness of BSMA and FBSMA heuristics for 100-node networks are shown in Figure 4.3 and Figure 4.4 for different k values, 100 and 1000 respectively. In those figures, average cost competitiveness for 400 100-node networks is depicted. As shown in Figure 4.3 and Figure 4.4, the difference between average cost competitiveness of FBSMA for different k values is also negligible for 100-node networks. For k is 100, average cost competitiveness of FBSMA changes between 99.97 and 100.04 and for k is 1000, average cost competitiveness changes between 99.99 and 100.054. As a result of simulations on 400 different 100-node networks, it is shown in the figures that FBSMA can successfully produce multicast trees for sparse and dense multicast

groups. Those multicast trees produced by FBSMA are as good as multicast trees produced by BSMA heuristic. In this set of simulations, FBSMA is used to evaluate the performance of DMBDCLC heuristic in 100-node networks. Simulations show that DMBDCLC's performance in terms of tree cost is as good as the performance of k-shortest path algorithm based delay-constrained least-cost path heuristic used in BSMA for 100-node networks.

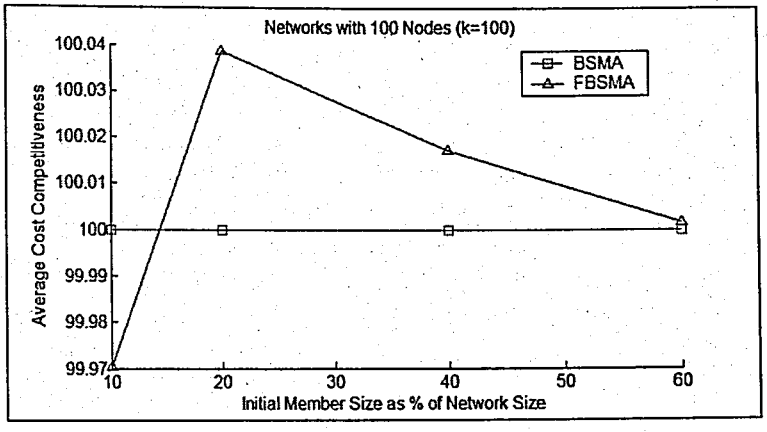

Figure 4.3. Average CC vs. member size for 100-node networks (k=100)

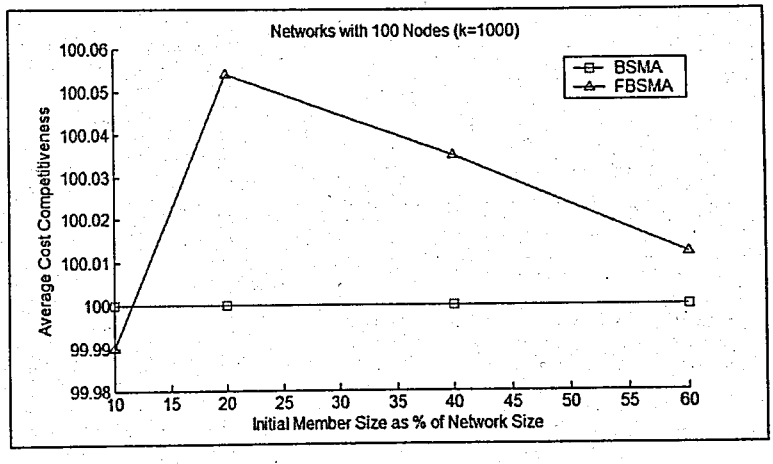

Figure 4.4. Average CC vs. member size for 100-node networks (k=1000)

4.4.1.2. CPU Time. In terms of average CC, it is previously shown that FBSMA is as good as BSMA heuristic for different k values. Moreover, FBSMA is much faster than BSMA as its name implies. The worst-case time complexity of FBSMA is only $O([n\log(n)]^2)$, whereas worst-case time complexity of BSMA is $O(KSTP(n)\times n\log(n))$

where $KSTP(n)$ is the time complexity of k-shortest path algorithm used in the BSMA, that is $O(kn^3)$ [37].

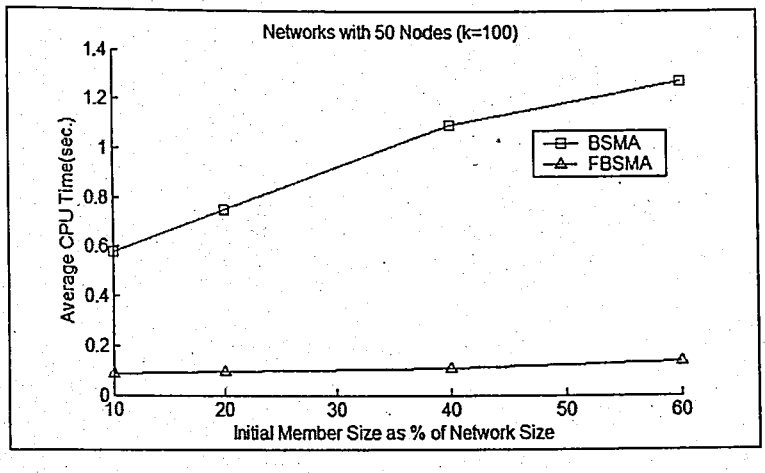

Figure 4.5. Average CPU time vs. member size for 50-node networks (k=100)

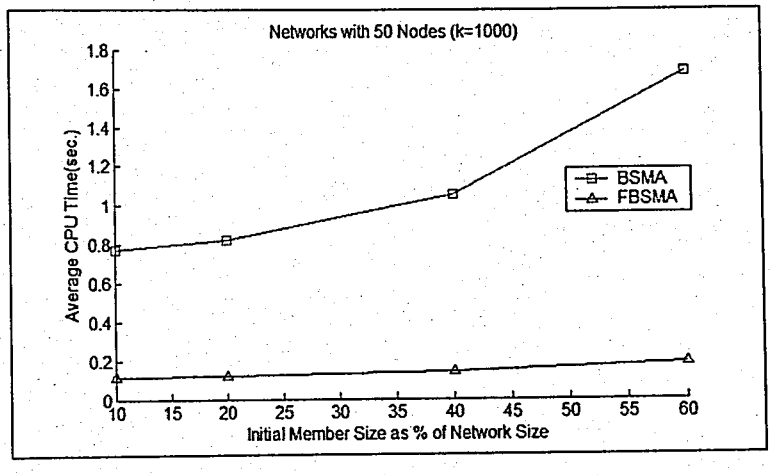

Figure 4.6. Average CPU time vs. member size for 50-node networks (k=1000)

Average CPU time of BSMA and FBSMA heuristics for 50-node networks and 100-node networks is shown in figures from Figure 4.5 through Figure 4.8 for different k values. In the figures, it is shown that CPU time of BSMA increases dramatically with increasing number of nodes in the network or multicast members in the multicast group. However, CPU time of FBSMA is minor with respect to that of BSMA and does not increase much with increasing number of nodes or multicast group members. This means that FBSMA can compute multicast trees as good as that of BSMA much faster. High time complexity of BSMA is its bottleneck. This bottleneck is relaxed

without trading off the tree cost for speed by using DMBDCLC in FBSMA instead of the k-shortest path algorithm.

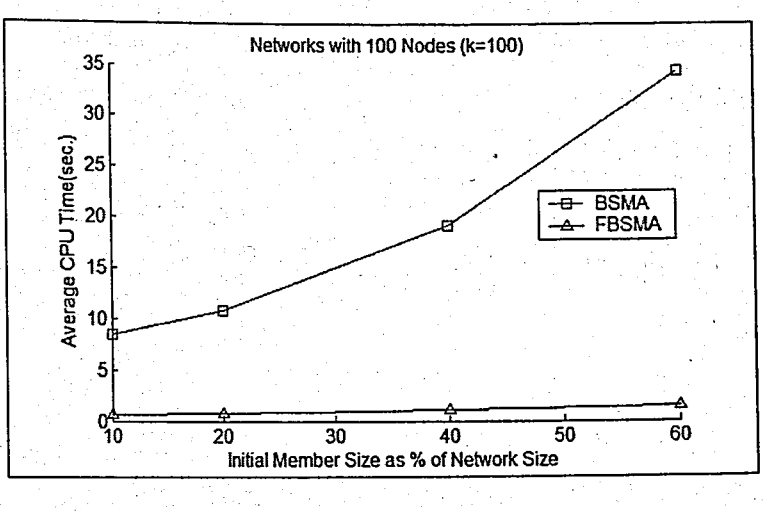

Figure 4.7. Average CPU time vs. member size for 100-node networks (k=100)

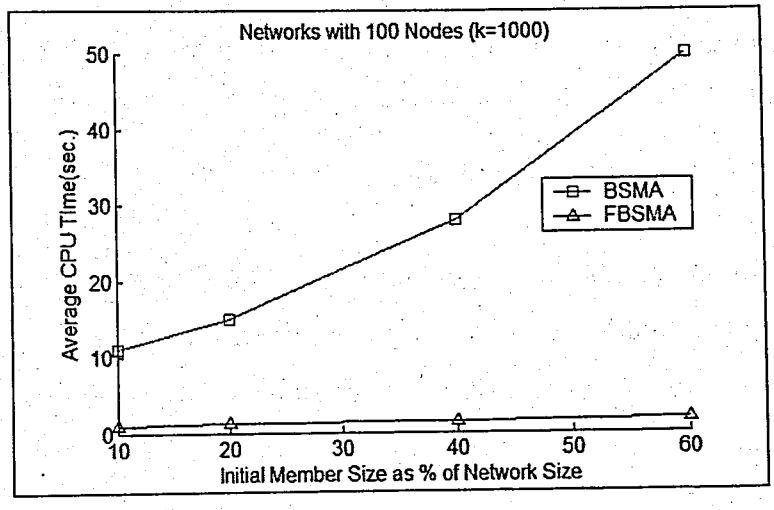

Figure 4.8. Average CPU time vs. member size for 100-node networks (k=1000)

4.4.2. Simulation Results for DeCoNROM and CoBaCROM

We compared our heuristics, CoBaCROM and DeCoNROM, with the heuristics from the literature. These heuristics are CRCDM, DDCLCMR and BSMA. Parameter k of k-shortest path algorithm in BSMA is chosen as 100 for the simulations and initial trees for online heuristics are computed by BSMA. Two shortest path-based unconstrained heuristics are also used for the comparison. These heuristics are LC and

LD, which are the same as NAIVE approach of Doar and Leslie with cost metric and delay metric respectively [5]. LC calculates multicast trees by combining least cost paths from source node to each of multicast group members. Similarly, LD calculates multicast trees by combining the least delay paths from source node to each of the multicast group members. Both LC and LD use Dijkstra's shortest path algorithm. CDKS heuristic is one of the constrained shortest path based multicast routing heuristics in the literature. It is a combination of LC and LD. In CDKS, paths of LD replace paths of LC, which are violating the delay constraint. CDKS is also used in our simulations for comparisons.

CRCDM is a parameterized heuristic. In the simulations, parameters of CRCDM are set for the best possible performance of CRCDM, as explained in [42]. CoBaCROM is tested for different values of τ; those are ranging from 0 to 1000 by 50. However, simulation results only for 6 out those 20 values of τ are illustrated because of the space limitations; those values for τ are 0, 50, 100, 200, 400 and 800. Those different τ values are shown as subscripts, CoBaCROM_{τ}, within the figure legends in this section. For example CoBaCROM_{100} is used to express that CoBaCROM is run for τ value 100.

Total 1200 different networks are simulated: 600 for 50-node networks and 600 for 100-node networks. An online multicast session is simulated for each network. A multicast session has 100 join/leave requests. After each request a new delay-constrained multicast tree is computed for the new set of multicast members by offline and online heuristics.

4.4.2.1. Cost Competitiveness. In Figure 4.9 and Figure 4.10, average CC versus multicast member size is shown for the heuristics, LC, LD, CDKS, DDCLCMR, DeCoNROM, CRCDM and CoBaCROM_0. Figure 4.9 and Figure 4.10 show the results of simulations on 50- node and 100-node networks respectively. CC of the trees produced by LC is less than CC of the trees produced by LD and CDKS. However, LC does not guarantee constrained multicast trees. CC of CDKS is slightly worse than that of LC, but CDKS guarantees constrained multicast trees. The worst performance in terms of

CC belongs to LD on the average. Performance of DDCLCMR is between that of LD and CDKS on the average. It is much worse in sparse multicast groups and becomes better as multicast groups get denser. Performance of DeCoNROM in terms of CC is much better than that of LC, LD, CDKS and DDCLCMR heuristics. It is almost independent of multicast group size or networks size and stable around 113 per cent both for 50-node and 100-node networks.

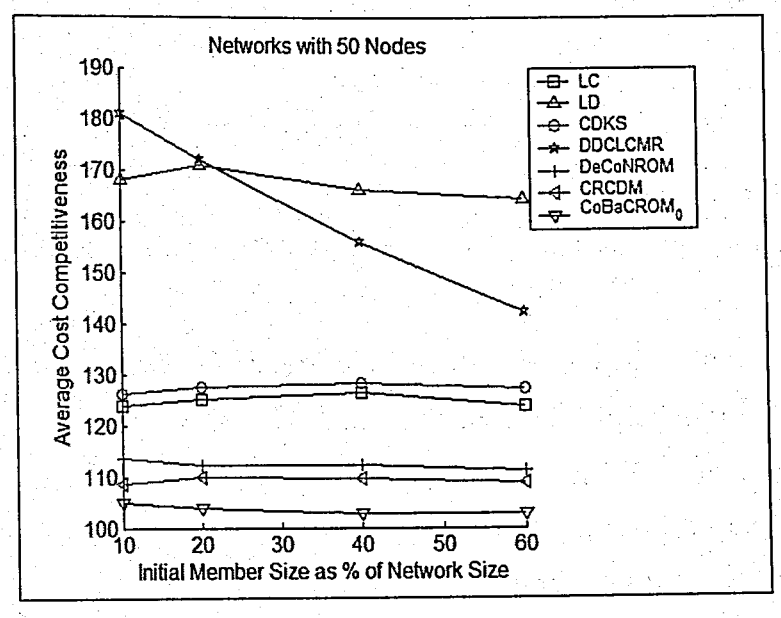

Figure 4.9. Average CC vs. member size for 50-node networks

Performance of rearrangeable heuristic, CRCDM, in terms of CC is slightly better than that of the non-rearrangeable heuristic, DeCoNROM, in Figure 4.9 and Figure 4.10. Parameters of CRCDM in our simulations are set for the best performance in terms of CC. It is almost independent of the multicast group size or networks size and stable around 110 per cent both for 50-node and 100-node networks. CoBaCROM_0 in figure legends refers to CoBaCROM heuristic with τ value of 0. This value results in the best performance of CoBaCROM in terms of CC. Performance of CoBaCROM in terms of CC is much better than other heuristics. It is almost independent of the multicast group size or the networks size and stable around 103 per cent.

In Figure 4.11 and Figure 4.12, average CC versus multicast member size is shown for the CoBaCROM heuristic with different τ values: 0, 50, 100, 200, 400 and

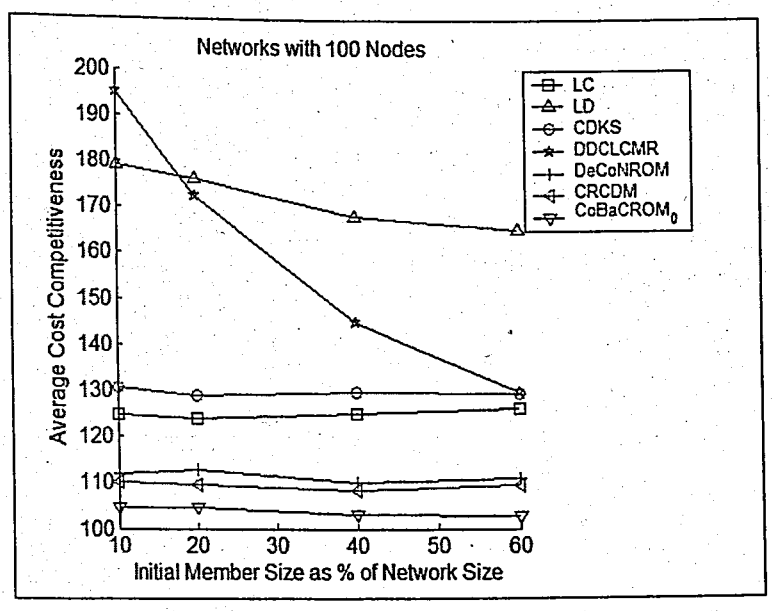

Figure 4.10. Average CC vs. member size for 100-node networks

800. Those figures show simulation results on 50-node and 100-node networks respectively. Simulation results of DeCoNROM and CRCDM heuristics are also included in the figures for comparison. As τ parameter of CoBaCROM increases, number of rearrangements decreases and performance of CoBaCROM in terms of CC approaches to performance of non-rearrangeable heuristic DeCoNROM. CoBaCROM is much better than CRCDM for the most of the τ values. Performance of CRCDM is almost equal to that of CoBaCROM, on the average, both for 50-node and 100-node networks when τ is 400. The best performance of CoBaCROM is attained in terms of CC, when τ is 0. This is shown in figure legends as CoBaCROM_0. In the simulations CRCDM's parameters are set accordingly so that the best performance of CRCDM is attained. Our simulations show that the performance of CRCDM, which is parameterized for the best CC performance, is almost equivalent to the performance of CoBaCROM_{400} on the average.

In Figure 4.13 and Figure 4.14, CC versus delay constraint is shown for the heuristics: DeCoNROM, CRCDM and CoBaCROM with different τ values. As delay bound relaxes, heuristics's performance in terms of CC increases. Slopes of the curves in the figures show that CoBaCROM utilizes relaxing delay bounds better than DeCoNROM and CRCDM both for 50-node and 100-node networks.

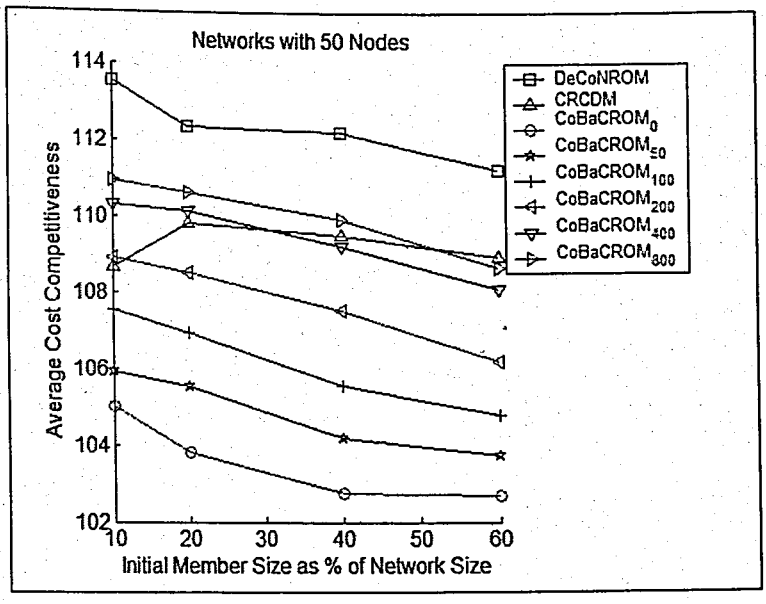

Figure 4.11. Average CC vs. member size for 50-node networks for different τ values

4.4.2.2. CPU Time. In Figure 4.15 and Figure 4.16, average CPU time versus multicast member size is shown for DDCLCMR and DeCoNROM. Although CC performance of DDCLCMR is very bad, its CPU time is very low. DeCoNROM is much better than DDCLCMR in terms of CC. CPU time of DeCoNROM is higher than that of DDCLCMR and decreases as the multicast groups get denser. However, it is less than 0.2 milliseconds for 50-node networks and it is less than 2.3 milliseconds for 100-node networks on the average.

In Figure 4.17 and Figure 4.18, average CPU time versus multicast member size is shown for the heuristics: DeCoNROM, CRCDM and CoBaCROM with different τ values. Those figures show simulation results on 50-node and 100-node networks respectively. The least CPU time belongs to DeCoNROM. This is normal because it is a non-rearrangeable heuristic. CPU time of DeCoNROM consistently decreases, as multicast groups get denser. The highest CPU time belongs to CoBaCROM_0. CPU time of CoBaCROM approaches to that of DeCoNROM, as value of τ increases. CPU time of CRCDM is the second highest one. CPU time of CRCDM is, on the average, 2.5 times and 1.8 times greater than that of CoBaCROM_{400} in 50-node and 100-node networks respectively. It is previously shown that performance of CRCDM is almost equivalent to the performance of CoBaCROM_{400}.

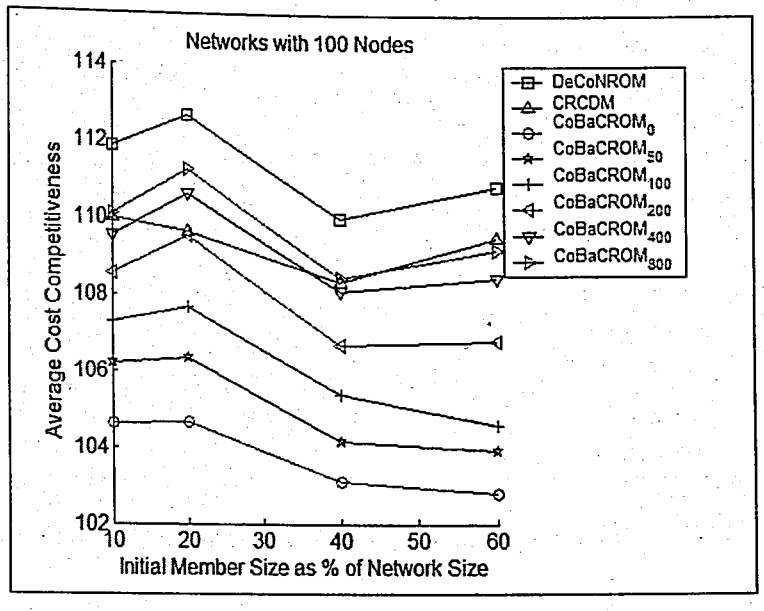

Figure 4.12. Average CC vs. member size for 100-node networks for different τ values

4.4.2.3. Change per Request. Change of edges in a multicast tree per request is an important performance criterion, especially for real-time communications. In Figure 4.19 and Figure 4.20, the average CPRQ versus multicast member size is shown for the heuristics, DDCLCMR and DeCoNROM. CPRQ decreases for both heuristic, as multicast groups get denser. CPRQ of DeCoNROM is always lower than that of DDCLCMR, and they approach each other, as the multicast groups get denser. If a request results in too much difference between consecutive trees, it may be hard to handle the real-time communications properly. So, in terms of CPRQ, DeCoNROM is more convenient than DDCLCMR for real-time online multicast sessions. Both DDCLCMR and DeCoNROM are non-rearrangeable online multicast routing heuristics. So, their CPRQ values are expected to be lower than that of rearrangeable online multicast routing heuristics.

In Figure 4.21 and Figure 4.22, the average CPRQ versus multicast member size is shown for the heuristics: DeCoNROM, CRCDM and CoBaCROM with different τ values. The highest CPRQ belongs to CRCDM and the lowest CPRQ belongs to DeCoNROM. CPRQ of CoBaCROM approaches that of DeCoNROM, as τ increases. CPRQ of the heuristics decreases, as multicast groups get denser. CPRQ of CRCDM is 1.5 times and 2.1 times greater than that of CoBaCROM_0 and CoBaCROM_{400} respec-

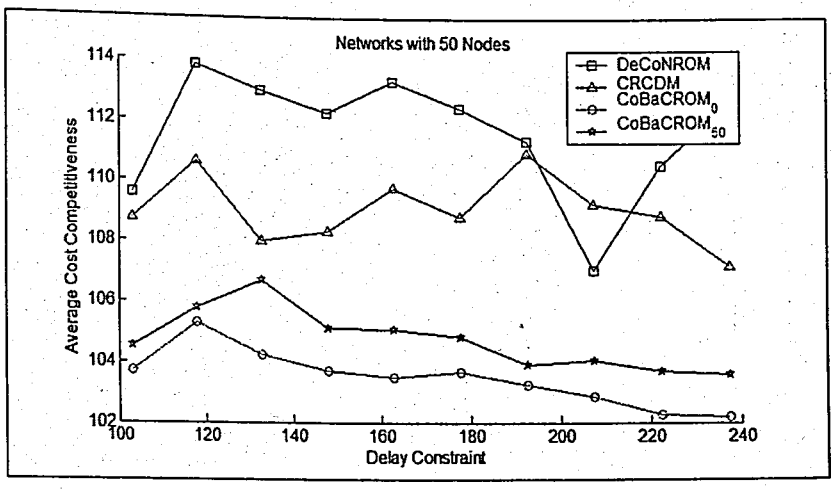

Figure 4.13. Average CC vs. delay constraint for 50-node networks

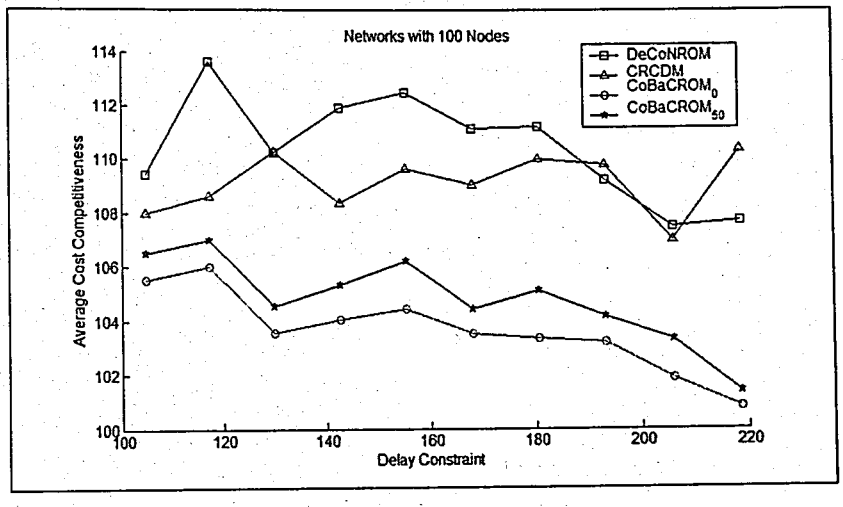

Figure 4.14. Average CC vs. delay constraint for 100-node networks

tively both for 50-node and 100-node networks. This means that CoBaCROM is much more convenient for real-time online multicast sessions in terms of CPRQ. CPRQ of CRCDM is dramatically high, especially for sparse multicast groups. For the multicast sessions whose member size is about 10 per cent of the network size, 12 per cent and 6.8 per cent of multicast tree links change for a request on 50-node and 100-node networks respectively, when CRCDM is used as online multicast routing heuristic. This may be beyond the acceptable for most of the real-time multicast routing applications.

4.4.2.4. Number of Rearrangements. As the NR for a heuristic increases, performance of the heuristic in terms of CC is expected to increase. However, increasing NR re-

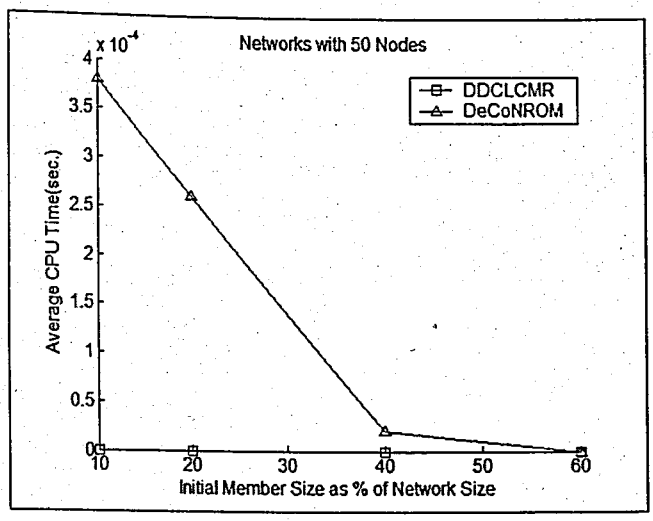

Figure 4.15. Average CPU time vs. member size for 50-node networks

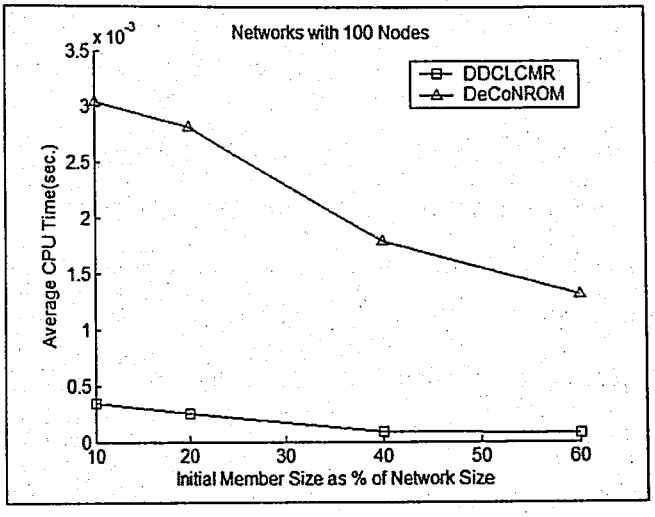

Figure 4.16. Average CPU time vs. member size for 100-node networks

sults in inefficiencies in terms of CPU time and it may be unacceptable for real-time communications. In Figure 4.23 and Figure 4.24, the average NR versus the multicast member size is shown. In our simulations, CRCDM is parameterized for the maximum performance in term of CC, so for the maximum number of rearrangements. Hence, NR of CRCDM is the highest in the figures. CoBaCROM$_0$ is also parameterized for the maximum performance in term of CC, so for the maximum number of rearrangements. However, NR of CoBaCROM$_0$ is much less than NR of CRCDM. NR of CoBaCROM heuristic decreases, as τ increases. CoBaCROM can produce multicast trees better than CRCDM in terms of CC, CPU time and CPRQ, despite its low NR.

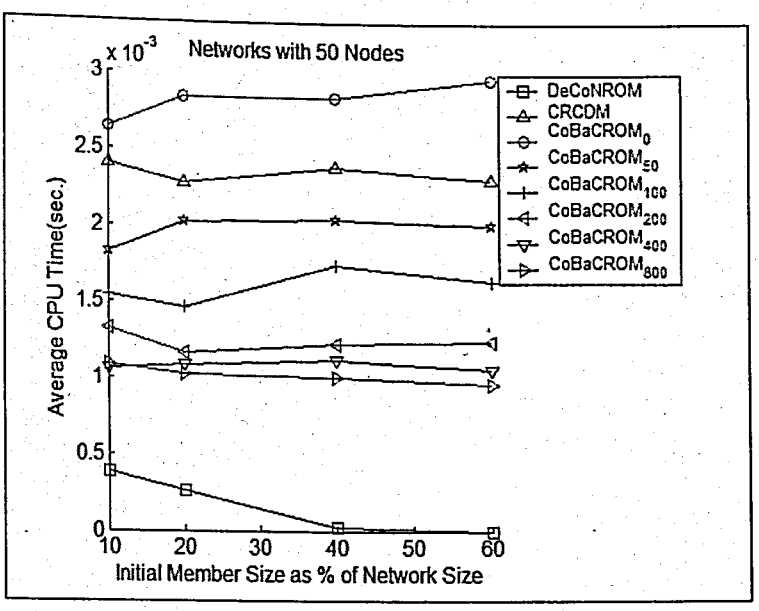

Figure 4.17. Average CPU time vs. member size for 50-node networks for different τ values

4.4.2.5. Gain per Rearrangement. GPR metric measures the average gain of a rearrangement in terms of tree cost through out a multicast session. In Figure 4.25 and Figure 4.26, the average GPR versus the multicast member size is shown. The worst GPR value belongs to CRCDM in the figures. GPR of CoBaCROM$_0$ approximately doubles that of CRCDM. GPR of CoBaCROM usually increases, as τ increases. Because increasing τ results in a decrease in the number of rearrangements. So, more degradation accumulates on multicast trees. Therefore rearrangements may result in more gain. Low GPR values of CRCDM and high GPR values of CoBaCROM indicate that rearrangement mechanism of CoBaCROM is much more successful than that of CRCDM. CoBaCROM makes rearrangements on the right portion of the multicast trees at the right time. So, rearrangements of CoBaCROM result in more gain.

4.4.2.6. Change per Rearrangement. CPR metric measures the average change in the edges of the multicast tree after a rearrangement throughout a multicast session. A rearrangement mechanism for real-time online multicast routing heuristics should not only result in upgraded multicast trees but also keep existing links in the multicast tree unchanged as much as possible. If rearrangements result in too much edge difference on multicast trees, it may be hard to handle the real-time communications properly. In

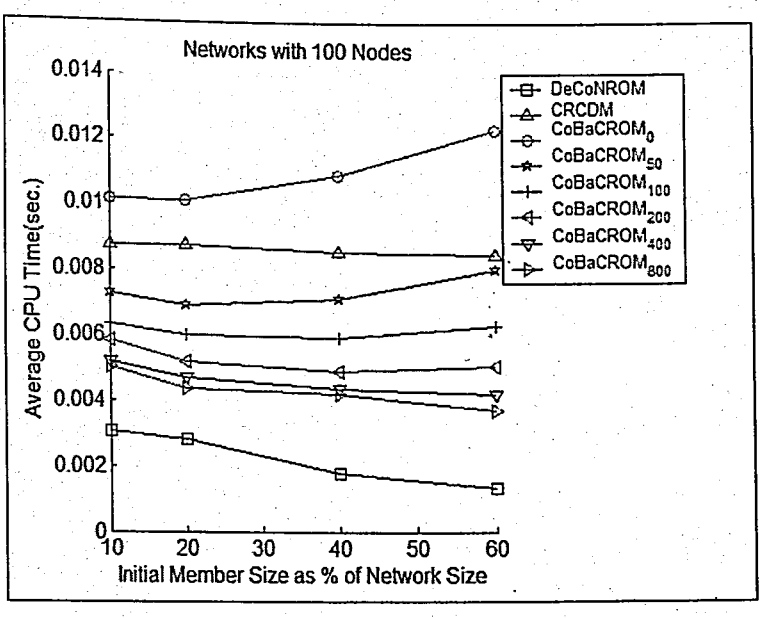

Figure 4.18. Average CPU time vs. member size for 100-node networks for different τ values

Figure 4.27 and Figure 4.28, average CPR versus multicast member size is shown. It is explicitly shown in the figures that CPR is highest for CRCDM. There is a huge gap between CPR of CRCDM and CPR of CoBaCROM. Moreover, increasing τ value does not change CPR much for CoBaCROM. CoBaCROM has a good nature of making less change on multicast trees, while resulting in much gain. Those results are consistent for 50-node and 100-node networks.

4.4.2.7. Number of Subtrees per Rearrangement. CPU time consumed by the rearrangeable online multicast routing heuristics depend on the number of rearrangements and computational complexity of a rearrangement. Computational complexity of a rearrangement depends on the number of subtrees connected during the rearrangement. NSPR metric measures the average number of subtrees produced during a rearrangement. In Figure 4.29 and Figure 4.30, average NSPR versus multicast member size is shown for 50-node and 100-node networks. NSPR of CoBaCROM$_0$ is more than that of CRCDM. This explains why CPU time of CoBaCROM$_0$ is more than that of CRCDM, even though NR of CoBaCROM$_0$ is much less than that of CRCDM. However, for other values of τ, CPU time of CoBaCROM is less than that of CRCDM in Figure 4.4.21 and Figure 4.4.22, because NR decreases significantly with increasing τ.

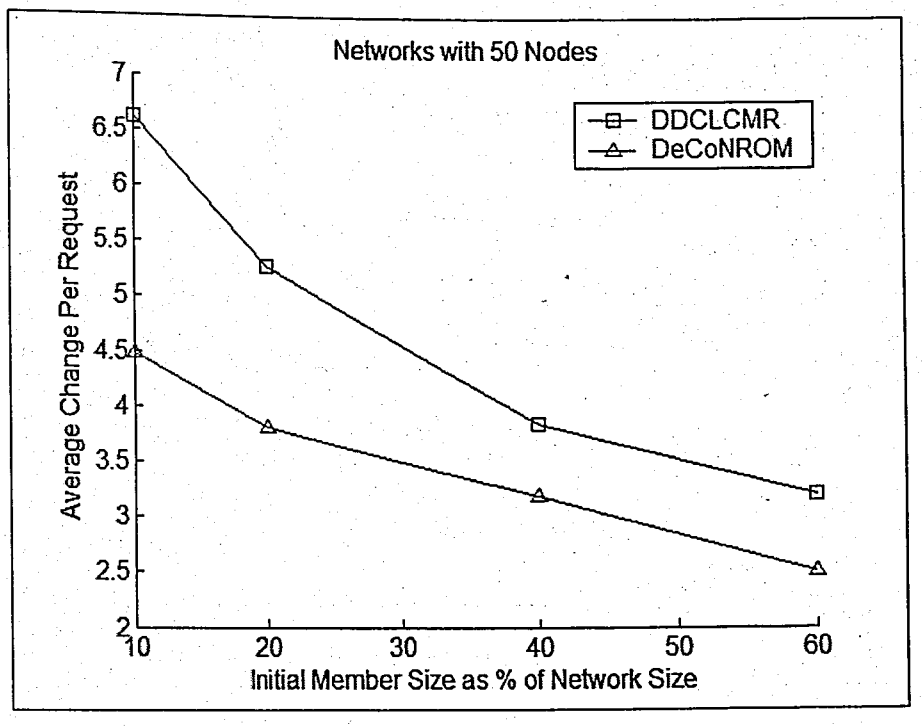

Figure 4.19. Average CPRQ vs. member size for 50-node networks

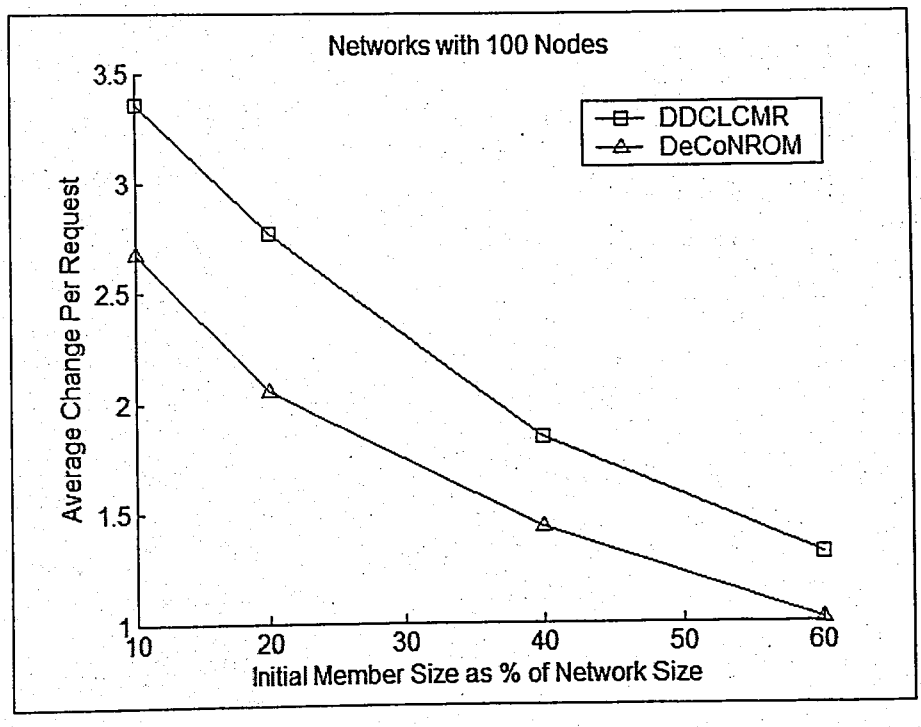

Figure 4.20. Average CPRQ vs. member size for 100-node networks

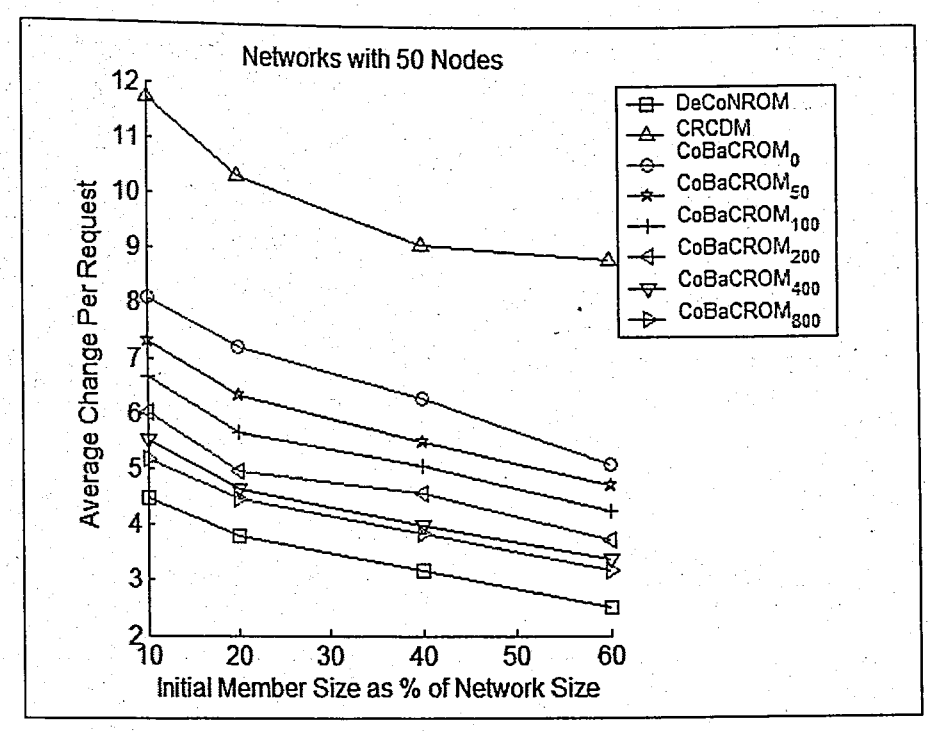

Figure 4.21. Average CPRQ vs. member size for 50-node networks for different τ values

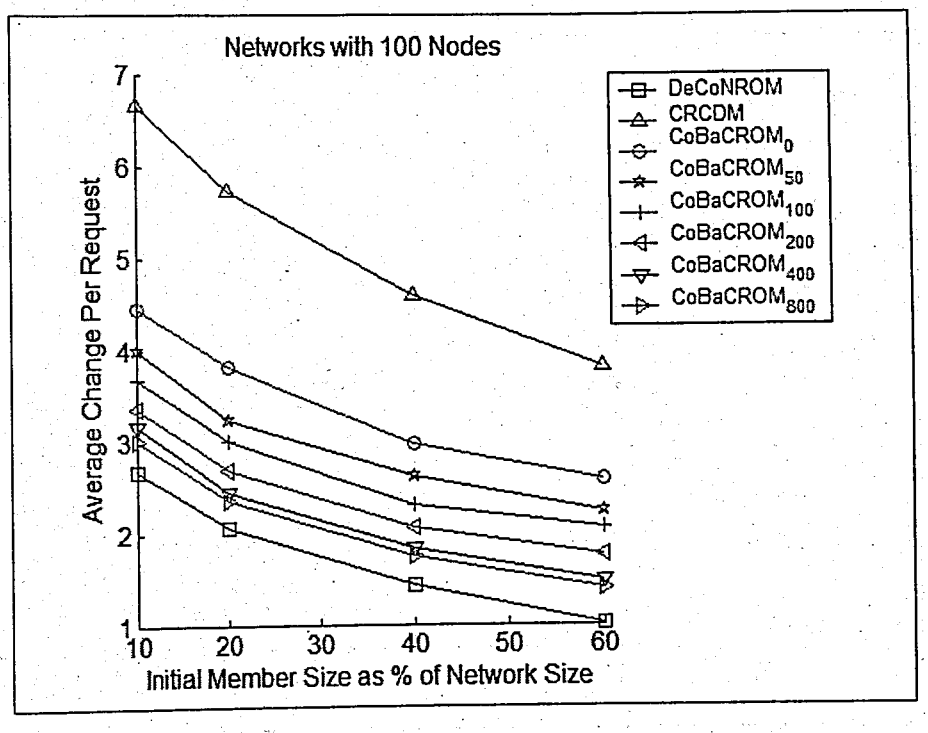

Figure 4.22. Average CPRQ vs. member size for 100-node networks for different τ values

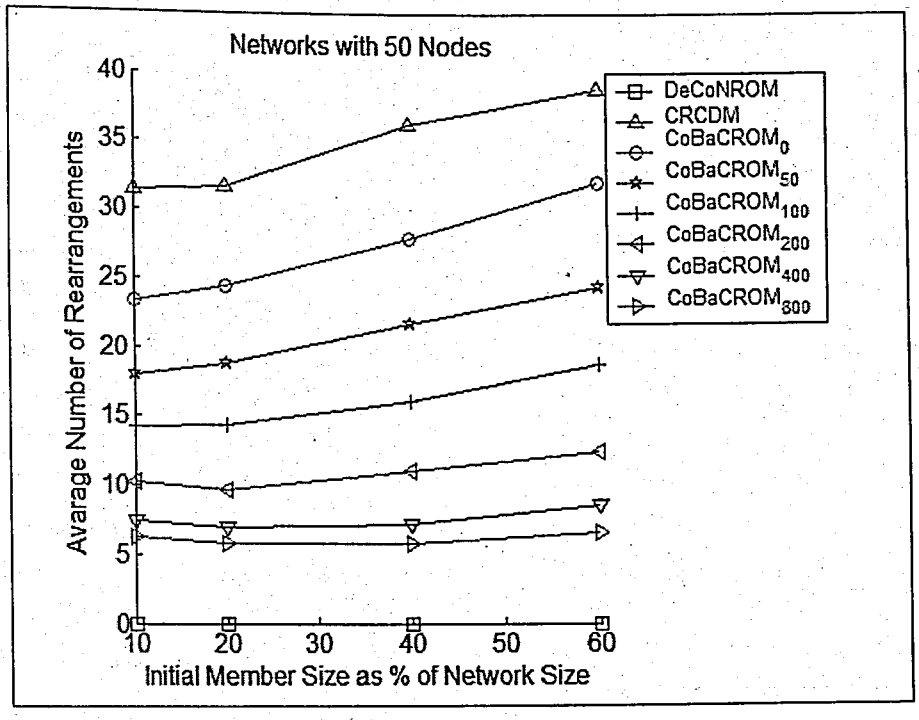

Figure 4.23. Average NR vs. member size for 50-node networks for different τ values

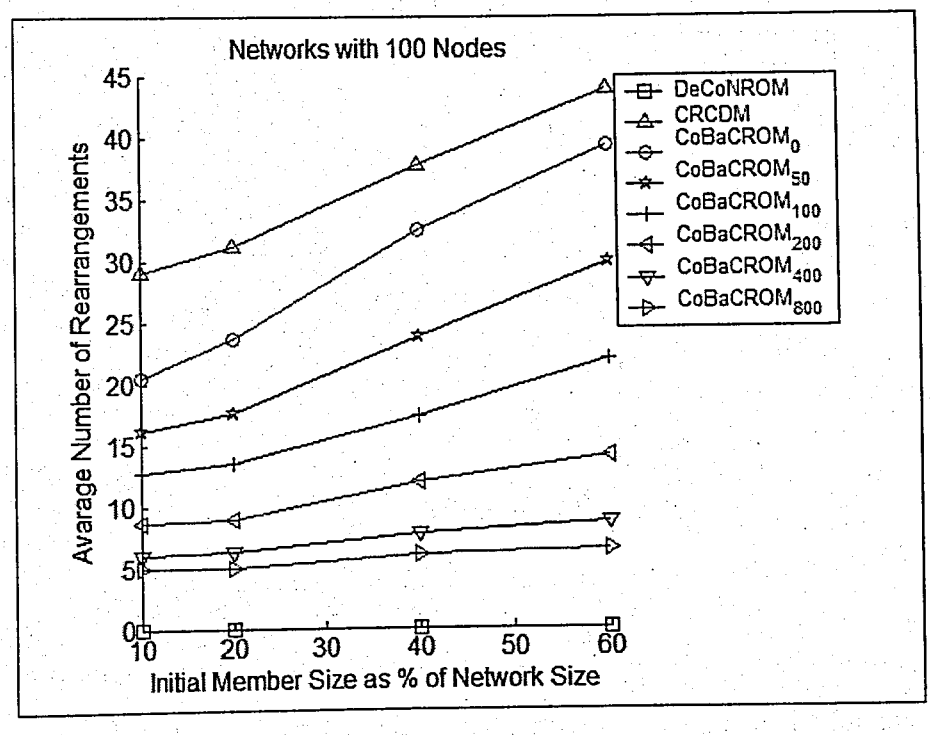

Figure 4.24. Average NR vs. member size for 100-node networks for different τ values

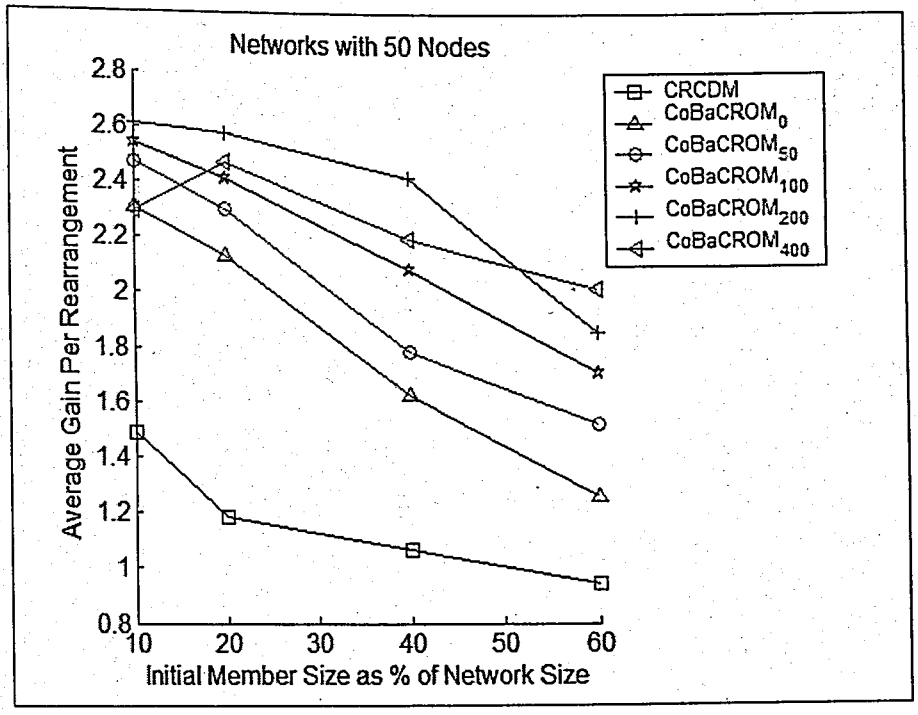

Figure 4.25. Average GPR vs. member size for 50-node networks for different τ values

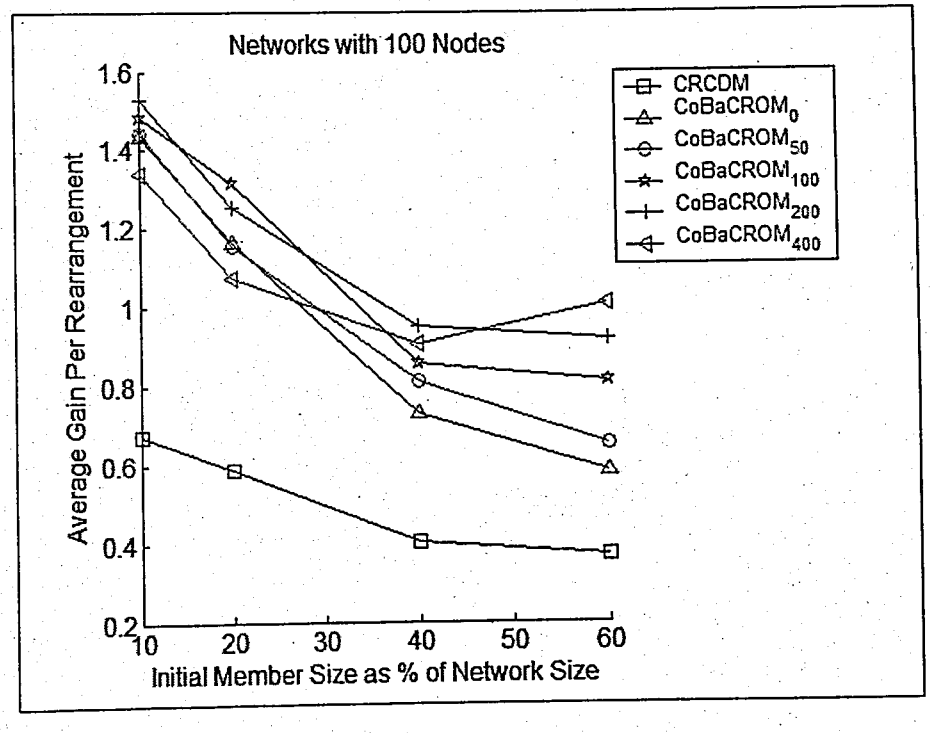

Figure 4.26. Average GPR vs. member size for 100-node networks for different τ values

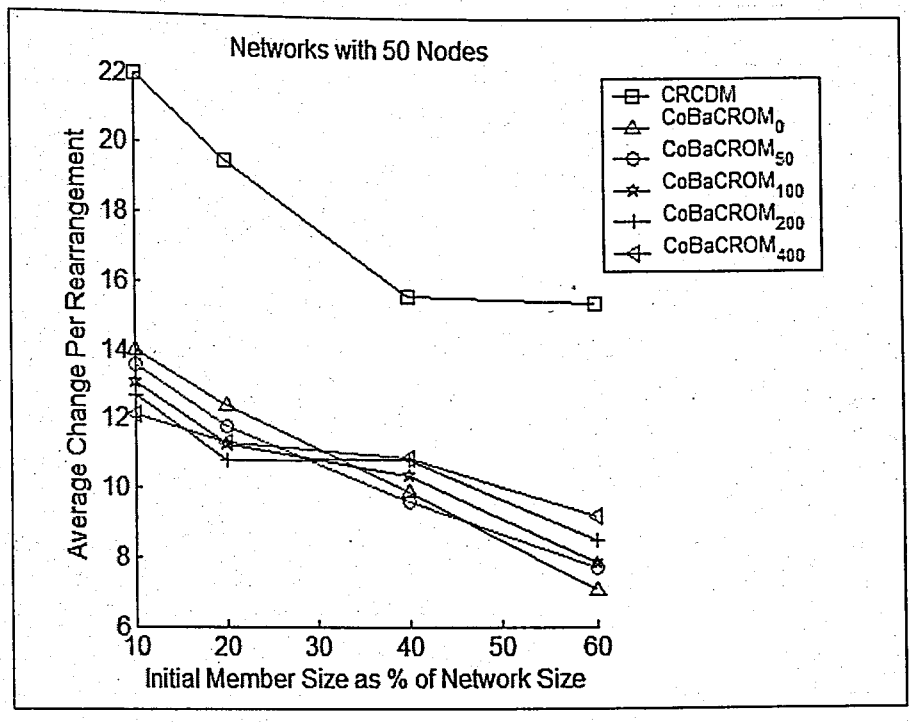

Figure 4.27. Average CPR vs. member size for 50-node networks for different τ values

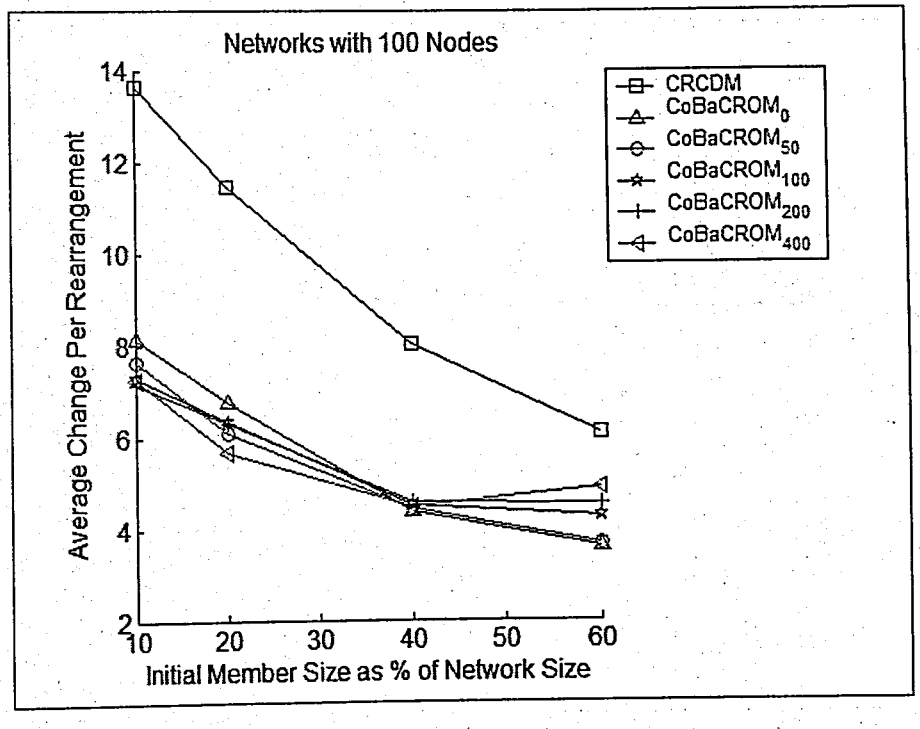

Figure 4.28. Average CPR vs. member size for 100-node networks for different τ values

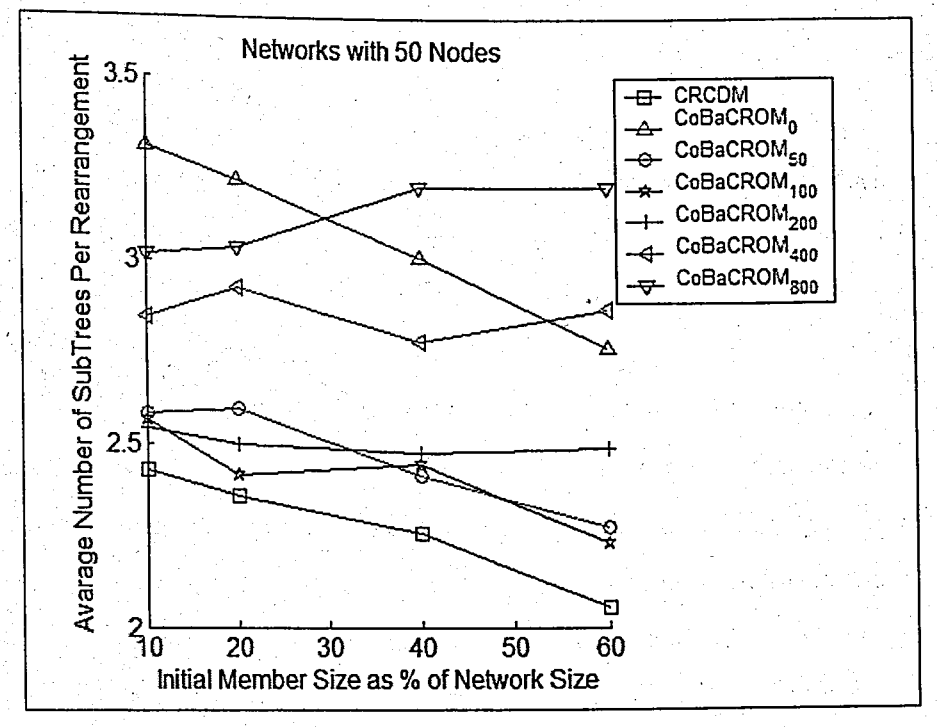

Figure 4.29. Average NSPR vs. member size for 50-node networks for different τ values

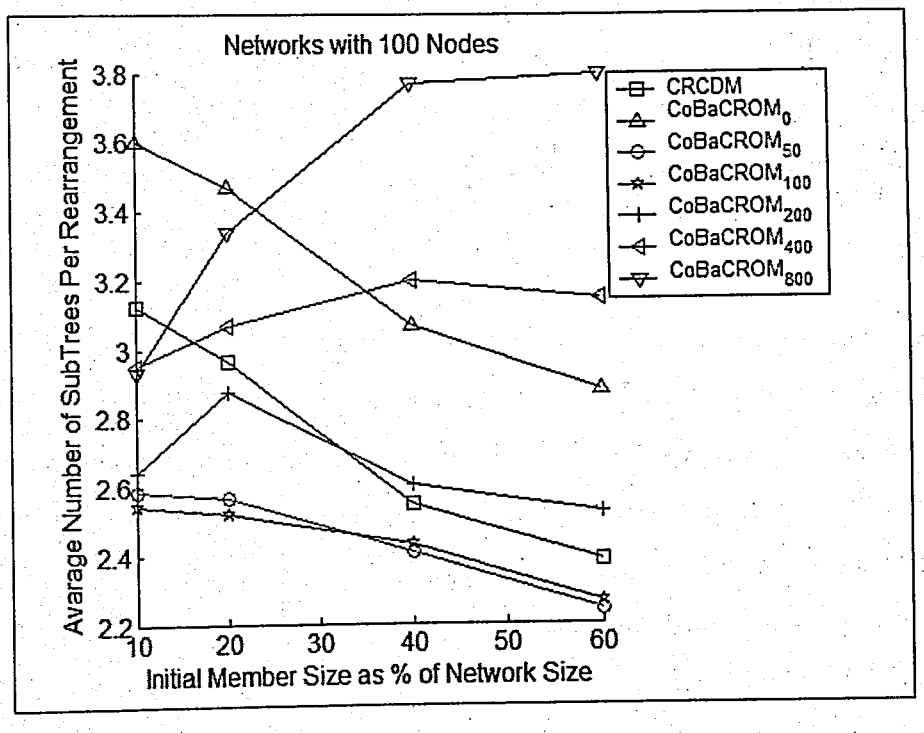

Figure 4.30. Average NSPR vs. member size for 100-node networks for different τ values

5. CONCLUSIONS

In this thesis, we proposed and evaluated CoBaCROM, a heuristic for delay-constrained rearrangeable online multicast routing. The heuristic bases on monitoring the accumulated damage to the multicast tree as members join and leave the multicast session. CoBaCROM introduces the *contract* concept of a multicast member. Using *contract* concept, CoBaCROM rearranges only the mostly damaged portion of the multicast tree and does not change the more valid portions. It can handle both symmetric and asymmetric links successfully by continuously updating the price of keeping a member node in the multicast tree using its current path. If the price exceeds acceptable limits for the member node or the session, rearrangement of the member starts. As its name implies, *contract* and τ parameter of CoBaCROM constitute a sort of agreement on the rearrangement decisions.

We compared the performance of our heuristic with offline multicast routing heuristics; BSMA, LD, LC, CDKS and online multicast routing heuristics; DDCLCMR, CRCDM and DeCoNROM, which is non-rearrangeable version of CoBaCROM in this thesis. Simulation results indicate that CoBaCROM offers the best balance among cost competitiveness, computational effort and change in the multicast tree during delay-constrained online multicast sessions.

In our simulations, CRCDM is parameterized for the maximum performance in term of cost competitiveness, so for the maximum number of rearrangements. Although rearrangements result in better trees in terms of cost, they are costly in terms of CPU time and they may cause great differences between consecutive trees. So, it should be worth making rearrangements. CRCDM makes more rearrangements than CoBaCROM and causes more difference between consecutive multicast trees. Gain per rearrangement of CoBaCROM is much more than that of CRCDM. This means that CoBaCROM makes rearrangements on the right portion of the multicast trees at the right time. So, rearrangements of CoBaCROM result in more gain. Thus, CoBaCROM's cost competitiveness is much better than that of CRCDM and

CoBaCROM can produce better trees using smaller number of rearrangements. Moreover, rearrangements of CoBaCROM result in less change between consecutive multicast trees. Therefore CoBaCROM is more convenient for real-time communications.

CoBaCROM is a parametrized heuristic and can be tuned for the best balance between cost competitiveness and computational effort. Our simulation results confirm that CoBaCROM suits the needs of real-time multicast routing applications. CoBaCROM is very flexible and generalized to be used in conjunction with any delay-constrained unicast routing heuristic. However, DMBDCLC is used in the CoBaCROM as the underlying unicast routing heuristic.

Another motivation of this thesis is the need for fast and efficient delay-constrained least-cost path heuristics to be used as an underlying unicast routing heuristic in delay-constrained multicast routing. DMBDCLC can find least-cost delay-constrained paths in a graph between two nodes or a tree and a node or two trees within a worst-case time complexity of $O(n\log(n))$. DMBDCLC is convenient for the applications, which require fast computation of delay-constrained paths without significantly trading off performance for speed. So, it can easily be used as the underlying delay-constrained least-cost path heuristic in online multicast routing. BSMA, which is one of the best polynomial-time heuristics for delay-constrained multicast routing problem, is used for the performance evaluation of DMBDCLC. A delay-constrained path heuristic basing on k-shortest path algorithm is used in BSMA. However, k-shortest path algorithms have high time complexities. Therefore the high time complexity of BSMA is its bottleneck. This bottleneck is relaxed by using DMBDCLC instead of a k-shortest path based heuristic without significantly compromising cost competitiveness. The modified heuristic is named as FBSMA. Our simulation results confirm that DMBDCLC is very convenient to be used in delay sensitive multicast routing applications as the underlying delay-constrained least-cost path heuristic. FBSMA using DMBDCLC can produce almost the same quality multicast trees much faster than the original BSMA heuristic. So, DMBDCLC can be used as a stand-alone unicast routing heuristic or an underlying unicast routing heuristic in delay-constrained multicast routing heuristics or protocols. Additionally, DeCoNROM and CoBaCROM heuristics, which are proposed

for delay-constrained online multicast routing problem in this thesis, use DMBDCLC as the underlying unicast routing heuristic. Admirable performance of those heuristics strengthen the performance analysis of DMBDCLC.

The heuristics proposed in this thesis are centralized heuristics. It requires that a node running any of those heuristics have the complete topology information. There are algorithms and protocols to deliver such information at each node. However, they are beyond the scope of this work and not cited in this thesis. Distributed versions of those heuristics are set aside as a future work.

APPENDIX A: PSEUDO-CODES FOR DMBDCLC

Pseudo-codes for DMBDCLC heuristic are demonstrated in this chapter. Those pseudo-codes belong to Dijkstra and Reverse-Dijkstra algorithms and modified versions of those algorithms. Dijkstra and Reverse-Dijkstra algorithms are modified as explained in Section 3.2.

```
function Dijkstra(Graph &G,int source, bool CostMetric){
    // Create Dijkstra object
    DKS dks(G, CostMetric);
    // Preprocess
    for(int i=0;i<G.GraphSize;i++){
      double value = infinity;
      if(i == source){ value= 0; }
      dks.Node[i].cost  = value;
      dks.Node[i].delay = value;
      dks.Node[i].predecessorNodeID = -1;
    }
    // Start Dijkstra
    while(dks.Heap.size() > 0){
      int u = dks.Heap.getCheapestNode;
      for(int i=0;i<G.Nodes[u].outEdges.size();i++){
         int v           = G.Nodes[u].outEdges[i].to;
         doube LinkCost  = G.Nodes[u].outEdges[i].cost;
         doube LinkDelay = G.Nodes[u].outEdges[i].delay;
         // RELAXATION
         if((CostMetric  &&
            dks.Node[v].cost > dks.Node[u].cost+LinkCost)||
            (!CostMetric  &&
            dks.Node[v].delay > dks.Node[u].delay+LinkDelay)){
              dks.Node[v].cost  = dks.Node[u].cost+LinkCost;
              dks.Node[v].delay = dks.Node[u].delay+LinkDelay;
              dks.Node[v].predecessorNodeID = u;
         }
      }
      // Update Heap
      dks.Heap.remove(u);
      dks.Heap.Update();
    }
    return dks;
}
```

Figure A.1. Pseudo-code for Dijkstra algorithm with cost and delay metrics

```
function ReverseDijkstra(Graph &G,int destination, bool CostMetric){
    // Create Reverse-Dijkstra object
    RevDKS revdks(G, CostMetric);
    // Preprocess
    for(int i=0;i<G.GraphSize;i++){
      double value = infinity;
      if(i == destination){ value= 0; }
      revdks.Node[i].cost  = value;
      revdks.Node[i].delay = value;
      revdks.Node[i].forwardNodeID = -1;
    }
    // Start Reverse-Dijkstra
    while(revdks.Heap.size() > 0){
     int v = revdks.Heap.getCheapestNode;
     for(int i=0;i<G.Nodes[v].inEdges.size();i++){
         int u                = G.Nodes[v].inEdges[i].from;
         doube LinkCost       = G.Nodes[v].inEdges[i].cost;
         doube LinkDelay      = G.Nodes[v].inEdges[i].delay;
         // RELAXATION
         if((CostMetric  &&
            revdks.Node[u].cost > revdks.Node[v].cost+LinkCost)||
           (!CostMetric  &&
            revdks.Node[u].delay > revdks.Node[v].delay+LinkDelay)){
              revdks.Node[u].cost = revdks.Node[v].cost+LinkCost;
              revdks.Node[u].delay = revdks.Node[v].delay+LinkDelay;
              revdks.Node[u].forwardNodeID = v;
         }
     }
     // Update Heap
     revdks.Heap.remove(v);
     revdks.Heap.Update();
    }
    return revdks;
}
```

Figure A.2. Pseudo-code for Reverse-Dijkstra algorithm with cost and delay metrics

```
function Dijkstra(Graph &G,Tree &SourceTree, bool CostMetric){
    // Create Dijkstra object
    DKS dks(G, CostMetric);
    // Preprocess
    for(int i=0;i<G.GraphSize;i++){
     dks.Node[i].cost  = infinity;
     dks.Node[i].delay = infinity;
     dks.Node[i].predecessorNodeID = -1;
    }
    // CHANGES EXPLAINED IN LEMMA 3.2.2
    for(int i=0;i< SourceTree.Nodes.size();i++){
     dks.Node[SourceTree.Nodes[i].nodeID].cost  = 0;
     dks.Node[SourceTree.Nodes[i].nodeID].delay = 0;
     dks.Node[SourceTree.Nodes[i].nodeID].predecessorNodeID = -1;
    }
    // Start Dijkstra
    while(revdks.Heap.size() > 0){
      int u = dks.Heap.getCheapestNode;
      for(int i=0;i<G.Nodes[u].outEdges.size();i++){
         int v                = G.Nodes[u].outEdges[i].to;
         doube LinkCost       = G.Nodes[u].outEdges[i].cost;
         doube LinkDelay      = G.Nodes[u].outEdges[i].delay;
         // RELAXATION
         if((CostMetric  &&
            dks.Node[v].cost > dks.Node[u].cost+LinkCost) ||
           (!CostMetric  &&
             dks.Node[v].delay > dks.Node[u].delay+LinkDelay)){
               dks.Node[v].cost  = dks.Node[u].cost+LinkCost;
               dks.Node[v].delay = dks.Node[u].delay+LinkDelay;
               dks.Node[v].predecessorNodeID = u;
         }
      }
      // Update Heap
      dks.Heap.remove(u);
      dks.Heap.Update();
    }
    return dks;
}
```

Figure A.3. Pseudo-code for Dijkstra algorithm modified by Lemma 3.2.2

```
function ReverseDijkstra(Graph &G,Tree &DestTree, bool CostMetric){
    // Create Rev-Dijkstra object
    RevDKS revdks(G, CostMetric);
    // Preprocess
    for(int i=0;i<G.GraphSize;i++){
     revdks.Node[i].cost  = infinity;
     revdks.Node[i].delay = infinity;
     revdks.Node[i].predecessorNodeID = -1;
    }
    // CHANGES EXPLAINED IN LEMMA 3.2.3
    for(int i=0;i< DestTree.Nodes.size();i++){
     revdks.Node[DestTree.Nodes[i].nodeID].cost  = 0;
     revdks.Node[DestTree.Nodes[i].nodeID].delay = 0;
     revdks.Node[DestTree.Nodes[i].nodeID].predecessorNodeID = -1;
    }
    // Start Reverse-Dijkstra
    while(revdks.Heap.size() > 0){
     int v = revdks.Heap.getCheapestNode;
     for(int i=0;i<G.Nodes[v].inEdges.size();i++){
          int u               = G.Nodes[v].inEdges[i].from;
         doube LinkCost       = G.Nodes[v].inEdges[i].cost;
         doube LinkDelay      = G.Nodes[v].inEdges[i].delay;
         // RELAXATION
         if((CostMetric  &&
            revdks.Node[u].cost > revdks.Node[v].cost+LinkCost) ||
           (!CostMetric  &&
            revdks.Node[u].delay > revdks.Node[v].delay+LinkDelay)){
              revdks.Node[u].cost = revdks.Node[v].cost+LinkCost;
              revdks.Node[u].delay = revdks.Node[v].delay+LinkDelay;
              revdks.Node[u].forwardNodeID = v;
         }
     }
     // Update Heap
     revdks.Heap.remove(v);
     revdks.Heap.Update();
    }
    return revdks;
}
```

Figure A.4. Pseudo-code for Reverse-Dijkstra algorithm modified by Lemma 3.2.3

```
function Dijkstra(Graph &G,Tree &SourceTree, bool CostMetric){
    // Create Dijkstra object
    DKS dks(G, CostMetric);
    // Preprocess
    for(int i=0;i<G.GraphSize;i++){
      dks.Node[i].cost  = infinity;
      dks.Node[i].delay = infinity;
      dks.Node[i].predecessorNodeID = -1;
    }
    // CHANGES EXPLAINED IN LEMMA 3.2.2
    for(int i=0;i< SourceTree.Nodes.size();i++){
      dks.Node[SourceTree.Nodes[i].nodeID].cost  = 0;
      dks.Node[SourceTree.Nodes[i].nodeID].delay = 0;
      dks.Node[SourceTree.Nodes[i].nodeID].predecessorNodeID=-1;
    }
    // Start Dijkstra
    while(revdks.Heap.size() > 0){
     int u = dks.Heap.getCheapestNode;
     for(int i=0;i<G.Nodes[u].outEdges.size();i++){
         int v                = G.Nodes[u].outEdges[i].to;
         doube LinkCost       = G.Nodes[u].outEdges[i].cost;
         doube LinkDelay      = G.Nodes[u].outEdges[i].delay;
         // RELAXATION
         double cost,delay;
         // CHANGES EXPLAINED IN LEMMA 3.2.3
         if(SourceTree.IsNodeInTree(u)){
            int tID = SourceTree.IndexOfNode(u);
            cost  = SourceTree.Nodes[tID].cost;
             delay = SourceTree.Nodes[tID].delay;
         }else{
            cost  = dks.Node[u].cost;
             delay = dks.Node[u].delay;
         }
         if((CostMetric  && dks.Node[v].cost > cost +LinkCost) ||
          (!CostMetric  && dks.Node[v].delay > delay+LinkDelay)){
              dks.Node[v].cost  = cost  + LinkCost;
              dks.Node[v].delay = delay + LinkDelay;
              dks.Node[v].predecessorNodeID = u;
          }
      }
      // Update Heap
      dks.Heap.remove(u);
      dks.Heap.Update();
     }
     return dks;
}
```

Figure A.5. Pseudo-code for Dijkstra algorithm modified by Lemma 3.2.3

REFERENCES

1. Hwang, F. and D. Richards, "Steiner Tree Problems", *Networks*, Vol. 25, pp. 55–89, 1992.

2. Winter, P. and D. Richards, "Steiner Problem in Networks: A Survey", *Networks*, Vol. 17, No. 2, pp. 129–167, 1987.

3. Miller, R. and J. Thatcher, *Complexity of Computer Computation*, Plenum Press, New York, 1972.

4. Prim, R., "Shortest Connection Networks and Some Generalizations", *The Bell Systems Technical Journal*, Vol. 36, No. 6, pp. 1389–1401, November 1957.

5. Doar, M. and I. Leslie, "How Bad is Naive Multicast Routing", *Proceedings of IEEE INFOCOM'93*, pp. 82–89, 1993.

6. Kou, L., G. Markowsky, and L. Berman, "A Fast Algorithm for Steiner Trees", *Acta Informatica*, Vol. 15, No. 2, pp. 141–145, 1981.

7. Waxman, B. M., "Routing Multipoint Connections", *IEEE Journal on Selected Areas in Communications*, Vol. 6, No. 9, December 1988.

8. Imase, M. and B. Waxman, "Dynamic Steiner Tree Problem", *SIAM Journal of Discrete Mathematics*, Vol. 4, No. 3, pp. 369–384, August 1991.

9. Bellman, R., *Dynamic Programming*, Princeton University Press, New Jersey, 1957.

10. Dossey, J., L. S. A. Otto, and C. Eynden, *Discrete Mathematics*, Harper Collins College Publishers, New York, 2nd edition, 1993.

11. Tanenbaum, A., *Computer Networks*, Printice-Hall International Inc., New Jersey,

4th edition, 2002.

12. Deering, S. and D. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs", *ACM Transactions on Computer Systems*, Vol. 8, No. 2, pp. 85–110, May 1990.

13. Garey, M. R. and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

14. Salama, H., *Multicast Routing for Real-Time Communication on High-Speed Networks*, Ph.D. Thesis, North Carolina State University, 1996.

15. Widyono, R., *The Design and Evaluation of Routing Algorithms for Real-Time Channels*, Technical Report ICSI TR-94-024, University of California at Berkeley, International Computer Science Institute, June 1994.

16. Sun, Q. and H. Langendoerfre, "Efficient Multicast Routing for Delay-Sensitive Applications", *Proceedings of the second Workshop on Protocols for Multimedia Systems*, pp. 452–458, October 1995.

17. Takahashi, H. and A. Matsuyama, "An Approximate Solution for the Steiner Problem in Graphs", *Mathematica Japonica*, Vol. 24, No. 6, pp. 573–577, February 1980.

18. Rayward-Smith, V., "The Computation of Nearly Minimal Steiner Trees in Graphs", *International Journal of Mathematical Education in Science and Technology*, Vol. 14, No. 1, pp. 15–23, January/February 1983.

19. Wall, D., "Selective Broadcast in Packet-Switched Networks", *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 239–258, February 1982.

20. Wall, D., *Mechanisms for Broadcast and Selective Broadcast*, Ph.D. Thesis, Stanford University, June 1980.

21. Doar, M., *Multicast in the Asynchronous Transfer Mode Environment*, Ph.D. Thesis, University of Cambridge, 1993.

22. Rayward-Smith, V. and A. Clare, "On Finding Steiner Vertices", *Networks*, Vol. 16, pp. 283–294, 1986.

23. Bauer, F., *Multicast Routing in Point-to-Point Networks Under Constraints*, Ph.D. Thesis, University of California, Santa Cruz, 1996.

24. Jiang, X., "Routing Broadband Multicast Streams", *Computer Communications*, Vol. 15, No. 1, pp. 45–51, January/February 1992.

25. Jiang, X., "Distributed Path Finding Algorithm for Stream Multicast", *Computer Communications*, Vol. 16, No. 12, pp. 767–775, December 1993.

26. Ramanathan, S., "An Algorithm for Multicast Tree Generation in Networks with Asymmetric Links", *Proceedings of IEEE INFOCOM' 96*, pp. 337–344, 1996.

27. Chow, C. H., "On Multicast Path Finding Algorithms", *Proceedings of IEEE INFOCOM' 91*, pp. 1274–1283, 1991.

28. Leung, Y. W. and T. S. Yum, "Efficient Algorithms for Multiple Destinations Routing", *Proceedings of the IEEE International Conference on Communications*, pp. 1311–1317, 1991.

29. Bauer, F. and A. Varma, "Distributed Algorithms for Multicast Path Setup in Data Networks", *IEEE/ACM Transactions on Networking*, Vol. 4, No. 2, pp. 181–191, April 1996.

30. Kadirire, J. and G. Knight, "Comparison of Dynamic Multicast Routing Algorithms for Wide-Area Packet Switched (Asynchronous Transfer Mode) Networks", *Proceedings of IEEE INFOCOM' 95*, pp. 212–219, 1995.

31. Bauer, F. and A. Varma, "ARIES: A Rearrangeable Inexpensive Edge Based On-

Line Steiner Algorithm", *IEEE Journal on Selected Areas in Communications*, Vol. 15, pp. 382–397, April 1997.

32. Kompella, V., J. Pasquale, and G. Polyzos, "Multicasting for Multimedia Applications", *Proceedings of IEEE INFOCOM' 92*, pp. 2078–2085, 1992.

33. Kompella, V., J. Pasquale, and G. Polyzos, "Multicast Routing for Multimedia Communication", *IEEE/ACM Transactions on Networking*, Vol. 1, No. 3, 1993.

34. Noronha, C. and F. Tobagi, "Optimum Routing of Multicast Streams", *Proceedings of IEEE INFOCOM' 94*, pp. 865–873, 1994.

35. Sun, Q. and H. Langendorfer, "An Efficient Delay-Constrained Multicast Routing Algorithm", *Journal of High Speed Networks*, Vol. 7, No. 1, pp. 43–55, August 1998.

36. Parsa, M., Q. Zhu, and J. J. Garcia-Luna-Aceves, "An Iterative Algorithm for Delay-Constrained Minimum-Cost Multicasting", *IEEE/ACM Transactions On Networking*, Vol. 6, No. 4, August 1998.

37. Lawler, E., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

38. Salama, H. F., D. S. Reeves, and Y. Viniotis, "Evaluation of Multicast Routing Algorithms for Real-Time Communications on High-Speed Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 15, No. 3, pp. 332–345, April 1997.

39. Hong, S., H. Lee, and B. H. Park, "An Efficient Multicast Routing Algorithm for Delay-Sensitive Applications with Dynamic Membership", *Proceedings of IEEE INFOCOM' 98*, pp. 1433–1440, 1998.

40. Baoxian, Z., L. Yue, and C. Changjia, "An Efficient Delay-Constrained Multicast Routing Algorithm", *Proceedings of International Conference on Communication Technologies (ICCT 2000)*, p. S07.2, 2000.

41. Zhengying, W., S. Bingxin, and Z. Ling, "A Delay-Constrained Least-Cost Multicast Routing Heuristic for Dynamic Multicast Groups", *Electronic Commerce Research*, Vol. 2, pp. 323–335, 2002.

42. Raghavan, S. and G. Manimaran, "A Rearrangeable Algorithm for the Construction of Delay-Constrained Dynamic Multicast. Trees", *IEEE/ACM Transactions on Networking*, Vol. 7, No. 4, pp. 514–529, August 1999.

43. Raghavan, S., G. Manimaran, and C. S. R. Murthy, "Preferred-Link Based Delay-Constrained Least Cost Routing in Wide Area Networks", *Computer Communications*, Vol. 21, No. 18, pp. 1655–1669, November 1998.

44. Jia, Z. and P. Varaiya, "Heuristic Methods for Delay Constrained Least Cost Routing Using k-Shortest-Paths", *Proceedings of IEEE INFOCOM' 01*, April 2001.

45. Mokbel, M. F., W. A. El-Haweet, and M. N. El-Derini, "A Delay-Constrained Shortest Path Algorithm for Multicast Routing in Multimedia Applications", *Proceedings of IEEE Middle East Workshop on Networking*, November 1999.

46. Handler, G. and I. Zang, "A Dual Algorithm for the Constrained Shortest Path Problem", *Networks*, Vol. 10, pp. 293–310, 1980.

47. Skiscim, C. C. and B. L. Golden, "Solving k-Shortest and Constrained Shortest Path Problems Efficiently", *Annals of Operations Research*, Vol. 20, No. 1–4, pp. 249–282, 1989.

48. Eppstein, D., "Finding the k Shortest Paths", *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pp. 154–165, November 1994.

49. Guo, L. and I. Matta, "Search Space Reduction in QoS Routing", *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 142–149, May 1999.

50. Juttner, A., B. Szviatovszki, I. Mecs, and Z. Rajko, "Lagrange Relaxation Based

Method for the QoS Routing Problem", *Proceedings of IEEE INFOCOM' 01*, April 2001.

51. Wang, Z. and J. Crowcroft, "Quality of Service Routing for Supporting Multimedia Applications", *IEEE Journal on Selected Areas in Communications*, Vol. 14, pp. 1228–1234, September 1996.

52. Weiss, M. A., *Data Structures and Algorithm in C++*, Addison Wesley Longman Inc., Florida, 2nd edition, February 1999.

53. Ramanathan, S., "Multicast Tree Generation in Networks with Asymmetric Links", *IEEE/ACM Transactions on Networking*, Vol. 4, pp. 558–568, August 1996.

54. Matsumoto, M. and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Vol. 8, No. 1, pp. 3–30, 1998.

55. Zhang, B. and C. Chen, "On the Complexity of Preferred Link Based Delay-Constrained Least Cost Routing in Wide Area Networks", *Computer Communications*, Vol. 26, No. 1029–1030, 2003.