INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
MULTICASTING IN GENERALIZED CLASS NETWORKS

by

Carl Michael Lane II

A thesis submitted in partial fulfillment of the requirements
for the degree of

Master of Science

in

Computer Science

Department of Computer Science

University of Nevada, Las Vegas

August 1996
The Thesis of Carl Michael Lane II for the degree of Master of Science in Computer Science is approved.

Chairperson, Kia Makki, Ph.D.

Examiner Committee Member, John Minor, Ph.D.

Examiner Committee Member, Evangelos Yfantis, Ph.D.

Graduate Faculty Representative, Woosoon Yim, Ph.D.

Dean of the Graduate College, Ronald Smith, Ph.D.

University of Nevada, Las Vegas

August 1996
ABSTRACT

A problem that we are experiencing in communication networks is the ability to efficiently multicast information to a group of network sites. Multicasting is defined as point to multipoint communications. Within a network, for example, we may want to send a message from one network site to five other network sites. This is an example of a multicasting operation.

We are seeing an increased use of applications which blend not only data, but also voice, and video. Due to this increase in usage, we are experiencing a need to conserve network bandwidth and to avoid loading communication networks excessively. To do this, a method must be used to transport the necessary information to its destination sites in an efficient manner.

This is where our research comes in. By incorporating multicasting techniques within message transmissions in networks, we can drastically reduce the necessary bandwidth and resource overhead associated with moving messages throughout the network.
# TABLE OF CONTENTS

ABSTRACT ................................................................................................................................iii

LIST OF FIGURES ...................................................................................................................vi

ACKNOWLEDGMENTS ........................................................................................................vii

CHAPTER 1 INTRODUCTION
  1.1 Definitions ..................................................................................................................2
  1.2 Thesis Objectives ......................................................................................................3
  1.3 Applications of Multicasting ...................................................................................4
  1.4 Organization of the Thesis .......................................................................................5

CHAPTER 2 RESEARCH CONTEXT
  2.1 Classical Multicast Tree Algorithms .......................................................................7
    2.1.1 Takahashi and Matsuyama .....................................................................8
    2.1.1.1 Example of Takahashi and Matsuyama Algorithm ..............9
    2.1.2 Kou, Markowsky, and Berman ...........................................................11
    2.1.2.1 Example of Kou, Markowsky, and Berman Algorithm... 12
  2.2 Multicast Path Finding Algorithms .......................................................................15
    2.2.1 Dynamic Programming Multicast Path Finding Algorithm ..............15
    2.2.2 RST Heuristic Multicast Path Finding Algorithm ..............................16
    2.2.3 RMV Heuristic Multicast Path Finding Algorithm ...........................17
  2.3 Spatial Coherence in Real Time Multicasting .....................................................18
    2.3.1 End to End Delay ....................................................................................19
    2.3.2 Delay Jitter ..............................................................................................19
  2.4 Reliable Multicasting .............................................................................................20
  2.5 Spanning Tree Algorithm for Constructing Multicast Trees Within a Network
     of Generalized Classes .............................................................................................21
  2.6 Makki’s Algorithm for Optimal Multicast Tree Construction ........................23
  2.7 Summary ..................................................................................................................25

CHAPTER 3 A NEW ALGORITHM FOR MULTICASTING WITHIN A NETWORK
   OF GENERALIZED CLASSES
  3.1 Introduction ............................................................................................................27
  3.2 Introduction to Generalized Class Networks.....................................................28
  3.3 Integration of Makki’s Algorithm .......................................................................28
  3.4 New Algorithm ......................................................................................................29
  3.5 Summary ..................................................................................................................31

CHAPTER 4 IMPLEMENTATION
  4.1 Implementation of Spanning Tree Algorithm ...................................................32
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.1 Example of Spanning Tree Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>4.2 Implementation of New Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Example of Modified Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>CHAPTER 5 CONCLUSION</td>
<td></td>
</tr>
<tr>
<td>5.1 Summary</td>
<td>44</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>45</td>
</tr>
<tr>
<td>APPENDIX I: Code Attachment for Implementation of Spanning Tree Algorithm</td>
<td>46</td>
</tr>
<tr>
<td>APPENDIX II: Code Attachment for Implementation of New Algorithm</td>
<td>56</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>75</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: The Network N ........................................................................................................... 10
Figure 2: The Arbitrary Vertex v .............................................................................................. 10
Figure 3: After Connecting the Second Multicast Vertex ..................................................... 10
Figure 4: The Multicast Tree .................................................................................................... 11
Figure 5: The Network N ........................................................................................................... 13
Figure 6: The Network N1 ........................................................................................................ 13
Figure 7: The Minimum Spanning Tree T1 of N1 .................................................................. 14
Figure 8: The Network Ns ........................................................................................................ 14
Figure 9: The Network N .......................................................................................................... 34
Figure 10: The Minimum Spanning Tree ................................................................................ 35
Figure 11: The Final Communications Network After Trimming the Extraneous Nodes .... 35
Figure 12: The Network N ........................................................................................................ 39
Figure 13: Forests Resulting When Using Circles as the Destination Sites ....................... 40
Figure 14: Multicast Tree Resulting When Using Circles as the Destination Sites .......... 40
Figure 15: Forests Resulting When Using Squares as the Destination Sites ...................... 41
Figure 16: Multicast Tree Resulting When Using Squares as the Destination Sites ........ 41
Figure 17: Multicast Tree Resulting When Using Triangles as the Destination Sites ......... 42
Figure 18: The Best Cost Multicast Tree ................................................................................ 42
Figure 19: The Final Network .................................................................................................. 43
ACKNOWLEDGMENTS

The work done on this thesis would not have been possible without the help of certain individuals. I'd like to take the time here to thank those individuals for their contributions to my graduate studies.

My advisor, Dr. Kia Makki, was a driving force throughout the course of my graduate studies. He is what I would call the ideal professor. Without his guidance and patience, my graduate work would not have been possible. Dr. Makki was not only there to assist me out of a feeling of obligation, but he gave the feeling that he really did care about me and also all of his other students. I spent many late nights with Dr. Makki discussing new ideas and criticizing the work that I was working on. Throughout the whole process, Dr. Makki always made himself available to me and I am deeply indebted to him for his help, level of commitment, and caring attitude.

I would also like to thank my family. Specifically, my wife, parents, and my uncle David. My family was always their to support me and to give me that extra motivation when times seemed at their worst. On more than one occasion, my family gave me the support to make it through a rough spot that I encountered within the course of my graduate work.
I would also like to thank the members of my thesis committee for their comments and taking the time to serve on my committee. On my committee were Dr. John Minor, Dr. Evangelos Yfantis, and Dr. Woosoon Yim.
CHAPTER 1

INTRODUCTION

An important problem in communication networks is to be able to efficiently multicast information to a group of network sites. For example, if we have twenty workstations connected together in a communications network, we may want to send a message from one of these workstations to five of the other ones.

With the increased use of applications which blend not only data, but also voice, and video, we see an emergence of the need to conserve network bandwidth and to avoid loading communication networks excessively. To do this, a method must be used to transport the necessary information to its destination sites in an efficient manner.

Our research deals with the problem of accomplishing this task. If the objective is to send information from one network site to all of the remaining sites in the communications network, then what we have is a broadcasting situation. In this case, we can use either Kruskal or Prim’s algorithms to construct a minimum spanning tree across the network. This would give us an efficient means of broadcasting information. If we want to send the message to a group of these sites, however, instead of broadcasting it to all of them, then we are now dealing with a point to multipoint or a multicasting problem. In this situation, by using minimum spanning tree algorithms, we will not be accomplishing
the multicasting in an efficient manner. When faced with this situation, what we now must do is construct a multicast tree. This multicast tree will be an efficient usage of the available communication pathways within the network to transport the information to its destination sites.

### 1.1 Definitions

A **minimum spanning tree** is a connected graph with no cycles. In a graph with N vertices, the minimum spanning tree for the graph would be the set of edges which connects all N vertices of the graph in such a way that no other set of edges could connect all N vertices of the graph at a lesser cost.

A **multicast vertex** is a member of a set of vertices called the multicast vertices. The multicast vertices are the set of vertices which have been designated as the destination vertices in the network.

A **multicast tree** is a set of communication links in a network with no cycles. In a communications network with V total communication sites and S sites which need to receive a message, the multicast tree would be the set of communication links which would establish a connection between the S sites in such a way that no other set of communication links could establish that connection with a lesser cost. The sites in the set
V-S may also be used in constructing the multicast tree while the cost of a communication link may be based on various factors, such as network traffic loads or transmission time.

A communications network consisting of generalized classes of sites is a communications network in which the sites may be of different types or classes. For instance, in a network of five communications sites, there may be up to five different classes within the network. Site one may be labeled as belonging to Class A, site two may be labeled as belonging to Class B, site three may be labeled as belonging to Class C, site four may be labeled as belonging to site D, and site five may be labeled as belonging to Class E.

1.2 Thesis Objectives

The objective of this thesis is to design a method by which we can solve the problem of efficiently multicasting information in a network of generalized classes. In brief, the three major topics discussed in the thesis will be:

- Evaluation of existing multicasting algorithms and discussion of multicasting topics and concerns.

- Modification of algorithm for efficient multicasting within a network of generalized classes.
• Implementation of an existing algorithm and also implementation of the modified algorithm.

1.3 Applications of Multicasting

There are many applications for the multicast tree problem in both industry and academia. One such example could be drawn with the campus computer system. For instance, let's imagine that the system administrators are shutting down the facilities which handle the faculty accounts for routine backups. In this case, the system administrators would want to send a message to all currently logged on faculty members asking them to please log off. A multicast tree would provide an efficient means of getting the message to all logged on faculty members while keeping the network traffic to a minimum.

In a network of generalized classes, the multicast tree may be applied for different purposes. Let's assume that we have a communications network in a university which consists of three different classes of sites. Let's call the first class a Mathematics site, the second class an Engineering site, and the third class a Business site. Now say that there is a need to send a message to at least one site of each class in the communications network. For instance, the President of the university may have declared tomorrow a holiday. In this case, the President assumes that if one Business site receives the message, then that site will relay the information to all members of its department. In this case, the multicast tree
will be very useful in getting the information to its destination while once again keeping network traffic down to a minimum.

1.4 Organization of the Thesis

The remaining chapters of this thesis are organized as follows:

In Chapter 2, we will give some background on related work which has been in this area and also on the research context. We will present two classical multicast tree construction algorithms. The first of these two algorithms was done by Takahashi and Matsuyama[28]. The second of the algorithms was written by Kou, Markowsky, and Berman[20]. We will then present some multicast path finding algorithms. Algorithms shown will be one utilizing the dynamic programming technique[7], one using the Reverse Spanning Tree technique[7] and another using the Reverse Multicast Vertex technique[7]. Both the RST and the RMV techniques[7] utilize heuristic methods in order to construct the multicast trees. I will then discuss problems which we see within multicasting such as end to end delay and delay jitter[23]. Also shown will be the topic of reliable multicasting by utilizing dualcasting[1] techniques. A spanning tree algorithm for constructing multicast trees within a network of generalized classes will then be presented. Finally, an algorithm by Makki for Optimal Multicast Tree Construction[22] will then be presented.
In Chapter 3, I will present a modification to an existing algorithm[22] for the construction of multicast trees within networks consisting of generalized classes. I will discuss the algorithm and explain its design.

In Chapter 4, the implementation of both the spanning tree algorithm and the modified algorithm for constructing multicast trees in networks of generalized classes will be discussed. In it will be a description of the data structures used in both implementations and the logic behind each implementation.

Chapter 5 will conclude our research work in this area and suggest some area for future work to be done.
In this chapter, we will present work which has already been done in this area in order to provide some background information leading up to the current. This information will also provide insight into the research context that we will be dealing with. We will first present two classical multicast tree algorithms. We will then present some multicast tree path finding algorithms and then discuss certain issues that arise in multicasting such as spatial coherence and reliability in multicasting[23].

2.1 Classical Multicast Tree Algorithms

We will now present two approximation algorithms. These algorithms are used to construct multicast trees. The first algorithm I will discuss was done by Takahashi and Matsuyama[28]. It was devised in 1980 and has an execution time of \( O(|S| |V|^2) \).

The second algorithm I will discuss was done by Kou, Markowsky, and Berman[20]. It was devised in 1981 and also has an execution time of \( O(|S| |V|^2) \).
Both of these algorithms perform relatively equally. We can see, however, that in a worst case scenario, these algorithms perform no better than the exact algorithms which have an execution time of $O(|V|^3)$. For example, let’s say that in a network we are working with, almost all of the vertices have been chosen as multicast vertices. In such a case, the number of multicast vertices approaches the number of vertices in the network. By using the two above algorithms where $S$ is the number of multicast vertices in the network and $V$ is the total number of vertices in the network, we see that we achieve a running time of $O(|V|^3)$ which is equivalent to, but no better than the exact solutions.

### 2.1.1 Takahashi and Matsuyama

Takahashi and Matsuyama’s algorithm[28] is based on Dijkstra’s shortest path algorithm for the multicast problem in networks. It basically works as follows:

- Choose an arbitrary vertex from the set of multicast vertices.

- Do the following $|S| - 1$ times.

  - Choose an arbitrary multicast vertex that has not already been chosen.
Run Dijkstra’s shortest path algorithm in order to find the set of edges to connect this multicast vertex to the set of already chosen multicast vertices such that no other set of edges can connect the multicast vertex at a lesser cost.

Dijkstra’s shortest path algorithm has a running time of $O(|V|^2)$. This represents the $O(|V^2|)$ portion of the total $O(|S||V^2|)$ time needed for the algorithm to run. The $S$ represents the number of times which the algorithm needs to be run. We can then see how the execution time can approach $O(|V|^3)$ when the number of multicast vertices reaches the total number of vertices in the network.

2.1.1.1 Example of Takahashi and Matsuyama Algorithm

Given the network $N$ in Figure 1, the first thing that we do is choose an arbitrary multicast vertex $v$ in Figure 2. Once $v$ is chosen, we then choose another arbitrary multicast vertex and run Dijkstra’s shortest path algorithm to connect the two vertices in Figure 3. We then choose the final multicast vertex and find the shortest path between it and the existing chosen multicast vertices. This results in the multicast tree in Figure 4.
Figure 1: The network N

Figure 2: The arbitrary vertex v

Figure 3: After connecting the second multicast vertex
2.1.2 Kou, Markowsky, and Berman

Kou, Markowsky, and Berman’s algorithm[20] is also based on Dijkstra’s shortest path algorithm for the multicast problem in networks. It basically works as follows:

- Given a network $N$, construct a network $N_1$. $N_1$ will be comprised of only the multicast vertices and one edge for each set of edges and vertices in $N$ which will construct the shortest path connecting the two multicast vertices. The cost of the one edge in $N_1$ will be equal to the sum of the costs of the equivalent edges used in $N$. In order to determine the shortest

Figure 4: The multicast tree
path between the two multicast vertices, we must run Dijkstra's shortest path algorithm.

- Now find a minimum spanning tree $T_1$ of $N_1$.

- Expand the tree $T_1$ by replacing each edge in $T_1$ by the set of edges and vertices which it represents in network $N$. This new network will be called $N_s$.

- Find a minimum spanning tree $T_S$ from $N_s$.

- Now delete all leaf vertices and their associated edges from $T_S$ which are not multicast vertices. The resulting network is the multicast tree.

Once again, Dijkstra's shortest path algorithm is the mitigating factor. Since it has a running time of $O(|V|^2)$ and because it represents the $O(|V|)$ portion of the total $O(|S| |V^2|)$, we are in the same dilemma that Takahashi and Matsuyama's algorithm produces. The time needed for the algorithm to run approaches $O(|V|^3)$ when the number of multicast vertices reaches the total number of vertices in the network.

**2.1.2.1 Example of Kou, Markowsky, and Berman Algorithm**
Given the network $N$ in Figure 5, we first construct the network $N_1$ which we see in figure 6. In figure 7, we construct the minimum spanning tree of $N_1$ and call it $T_1$.

Finally, in figure 8 we expand $T_1$ into $N_s$. In this case, there is no need to construct the minimum spanning tree $T_S$ or the trimmed tree $T_H$. This is because the network $N_S$ does not have to be trimmed and no minimum spanning tree can be derived from this which will construct a network with a lesser edge cost than what is currently present. In this particular case, the network $N_S$ happens to also be the final multicast tree.

Figure 5: The network $N$

Figure 6: The network $N_1$
Figure 7: The minimum spanning tree $T_1$ of $N_1$

Figure 8: The network $N_s$
2.2 Multicast Path Finding Algorithms

In this section, three multicast path finding algorithms will be discussed. The first will be an algorithm based on the dynamic programming technique[7]. The second algorithm will be the RST algorithm[7], and the third algorithm will be the RMV algorithm[7]. The last two of the three algorithms are heuristic based algorithms for finding multicast trees.

2.2.1 Dynamic Programming Multicast Path Finding Algorithm

This multicast path finding algorithm is based on dynamic programming techniques. The algorithm basically works as follows:

First, the shortest path between all pairs of nodes in the network is computed. These pairs of points are treated as multicast paths. All further multicast paths are created from the union of two already existing multicasting paths with each path being a non-empty disjoint subset of the entire network.

In this algorithm, we can see that the majority of the time complexity resides in the computation of the vertex sets which will be used for the further construction of the multicast trees. This is the same problem seen with both of the multicast tree algorithms already presented and it can increase time complexity time to a factor of $O(|V|^3)$ when
combined with the additional work that has to be done in order to implement this algorithm.

2.2.2 RST Heuristic Multicast Path Finding Algorithm

This algorithm applies the spanning tree algorithm in order to implement the construction of the multicast tree. It does this in a reverse edge direction and this is why the algorithm is called the Reverse Spanning Tree algorithm[7]. If there is a need to calculate a multicast tree from a node v to a set of destination vertices S, the algorithm would function as follows:

- Use the spanning tree algorithm to find a path P from v to an arbitrary node in S and let's call this arbitrary node s. The resulting nodes in the path P let's call p.

- Now, for each node in S - v, do the following: Use the spanning tree algorithm which traverses edges in reverse direction to find the shortest path between the current node to any multicast node in S. Add the edges in this path to the set of edges in P and update p.

This algorithm has a worst case time complexity better than the previous algorithms. Its running time complexity is O(|S||e|log|e|) where e is the number of edges in
the network and S is the set of destination nodes in the multicast tree. We can see that in
the worst case where the network is very dense and the number of nodes in S approaches
the total number of nodes in the network, that we can reach a time complexity of
$O(|V|^2 \log |V|)$.

2.2.3 RMV Heuristic Multicast Path Finding Algorithm

The RMV algorithm[7] is designed to run in networks where all sites may not have
multicast capability. Faced with this new constraint, we now assume that that a multicast
path will be comprised of two parts. The first part will be the shortest path between the
source node and some multicast node within the network from which all the other
destination nodes can be reached. The other part of this multicast path will be the
multicast tree from this intermediary multicast node to all the destination nodes. This leads
to the reason for the naming of the algorithm as the RMV algorithm[7] since the algorithm
first tries to find the reachable multicast vertices. The algorithm can be outlined as follows:

- Find the set of all reachable multicast vertices from s. Let's call this
  RMVS.

- For each multicast vertex in RMVS find all the shortest paths from the
  vertex to the destination nodes in the multicast tree.
• Select a multicast vertex from RMVS where the cost of the shortest path from the source node \( v \) to the vertex and the cost of the shortest paths from the vertex to \( S \) is a minimum.

• Use the RST algorithm\[7\] to find the multicast path from the selected multicast vertex from RMVS to \( S \).

Once this is done, you would have the multicast tree from \( s \) to \( S \). The algorithm has a time complexity \( O(|S||RMVS||V|^2) \). We can see from this that as the size of \( S \) increases and the number of multicast vertices increase within the network, that the algorithm has a very poor time complexity.

2.3 Spatial Coherence in Real Time Multicasting

In high speed networks, performance requirements are high. Most real time applications can be seen to initiate periodic data transfers, bounded delay transfers, and are also to withstand transfer failures while still ensure reliable delivery of transmission information. Pokam and Michel[23] defined a communications model based on a connection oriented mixed packet switching and circuit switching network. In this model, when a connection is requested, a parameter is sent defining the level for the quality of the connection. An admission control function then decides whether the connection can be given or not. The connection will be granted if there is enough available bandwidth,
computing power, buffer space, and if it will not degrade the service being provided to current connections.

2.3.1 End To End Delay

In critical real time applications, data must be transferred within a specific time bound. If the data is not sent and received within this bound, then by the time that it arrives at its destination, it is already outdated. In order to specify the maximum time that can pass when a transmission of a packet is started to when the packet is received, we define an end to end delay bound. This delay bound is passed as a parameter when requesting a multicast or point to multipoint connection. The parameter must then be adhered to as the delay bound in the resulting multicast tree.

2.3.2 Delay Jitter

The communications network is a constantly changing environment. As traffic increases or slows down within the network, we see the end to end delay of transmitted packets rise and then fall. This constantly changing variation in the end to end delay is what we call the delay jitter. We do not want a node to experience a long idle time and then to be flooded unexpectedly with a stream of transmission packets. In order to smooth out the variance in the end to end delays, we define something called a delay jitter bound.
This bound defines the maximum delay variation on transmitted packets of information and helps to eliminate spurts of transmission.

2.4 Reliable Multicasting

In a paper by Aggarwal and Raghav[1], the discussion of reliable multicasting is entered into. In networks, transmitted data packets should always reach their destination. If the conditions of the network degrade such as the noise level escalating or the traffic reaching upper limits, the network should be able to handle these situations. One way to build reliability in networks is to build redundancy in the transmission paths. This is exactly what is proposed by Aggarwal and Raghav. They suggest that what you do in an instance where a multicasting operation is to take place is to construct not one but two multicasting trees. They call this philosophy dualcast and it is implemented as follows:

- The minimum cost from every node to every other node in the network is found using the Floyd Warshall algorithm basing the costs on edge weight.

- From the results of step 1, we construct the first multicast tree $S_1$ using the Takahashi and Matsuyama[28] method.

- The network is then transformed into another network using the following definition.
• If an edge is not part of the first multicast tree then it remains the same. If an edge is part of the multicast tree, however, then a multiplication factor is applied to the edge.

• Steps 1 and 2 are then reiterated on the new network to produce the second multicasting tree.

There are always tradeoffs when dealing with situations such as these. In this particular case, we are sacrificing network traffic by transmitting information along two multicasting trees rather than one. We are, however, practically guaranteeing that the transmitted information will reach its destination.

The algorithm may be even more appealing if the method of constructing the multicasting tree may be altered. As we have seen in the previous works, whenever we attempt to find to shortest paths between all pairs of nodes in the network, we approach running times of \(O(|V|^3)\). There is no difference in this case.

2.5 Spanning Tree Algorithm for Constructing Multicast Trees Within a Network of Generalized Classes
The following algorithm solves the situation of finding a multicasting tree to link nodes of different classes within a communications network. The algorithm is based upon the spanning tree algorithm and it works as follows:

Given a network N with V sites, E communication pathways, and C classes of network sites, we want to construct a minimal communications pathway which will allow transfer of information between at least C network sites. That is, at least one network site of each class must be included in the resulting communications network.

- We first construct a minimum spanning tree among the communication sites in the network N using the P communications pathways as edges. This minimum spanning tree may be constructed using either Prim or Kruskal's algorithms.

- Now we must trim the resulting network. We do this from the network sites at the outer edges and work our way in. That is, for every network site which has only one communications pathway leading to it, we determine if there is another network site of the same class as itself already in the communications network. If we do determine that there is a duplicate network site, then we remove the network site that we are currently inspecting and also the communications pathway which was leading to it. Repeat this iterative process until we are no longer able to trim any
network sites. Once this has been completed, we have finished implementing the algorithm.

The result of this algorithm meets the requirements which were specified at the beginning of this section. This algorithm is not, however, efficient. It performs very poorly in certain cases. One such case is where the network sites are lined up in a chain fashion with a communications path linking both ends of the chain.

2.6 Makki's Algorithm For Optimal Multicast Tree Construction

The following algorithm was designed by Makki[22] and constructs an optimal multicast tree in a more efficient manner than we have previously seen. The algorithm works as follows:

Given a network N with V sites, E communication pathways, and S destination sites, we want to construct a minimal communications pathway which will allow transfer of information between the S destination sites using any of the V-S sites as relay points and also using any of the E communication pathways.

- Create a virtual site v and create e virtual communication pathways all of weight 0. Connect these pathways from the virtual site v to each destination site S.
• Now we use Dijkstra’s shortest path algorithm with the virtual site v as the starting point.

• Once we have completed the implementation of the shortest path algorithm, we are left with a new network topology. Every network site in the set V - S has now been grouped with one network site in the set of S destination sites. The grouping is based on the fact that this network site has no set of communication pathways which can link it to another network site in the set of destination sites S at a lesser cost.

• Now among the resulting groups of network sites, we will attempt to link the groups together via the remaining communication pathways which have not yet been marked as chosen. The basis for the selection of the communication pathways is as follows: Choose the communication pathway which connects the two groups of network sites such that the communications path from the one network site in the first group of network sites which is also of the group of destination sites S to the other network site in the second group of network sites which is also of the group of destination sites S is a minimum. Do this for pairs of groups of network sites.
• We now trim off all communications pathways and network sites which are not in a communications path leading from a destination site to another destination site.

• Construct a minimum spanning tree among the network to eliminate any cycles in the communications network.

This algorithm produces an efficient set of communication pathways by which we can transfer information between the $S$ destination sites in the communication network. Since it does not attempt to construct the shortest paths between all pairs of nodes in the communications network, it has a lesser running time than what we have seen so far. The main time factor in this algorithm is the running time of Dijkstra’s shortest path algorithm and leads to an execution time of $O(|E| + |V| \log |V|)$. In the worst case of a very dense network we approach a running time of $O(|E| + |V| \log |V|)$. This is one order of magnitude less than any algorithm which we have reviewed thus far.

2.7 Summary

In most of the previous work done in multicasting, we see that execution time is not very efficient in the worst case scenarios. The algorithm which stands out above the rest is the one designed by Makki[22] which has a worst case execution time of $O(|E| + |V| \log |V|)$. Because of this, our modified algorithm utilizes Makki’s algorithm[22] as
its core for the construction of the multicast trees. Also, we have seen one algorithm which refers to the problem of the multicast tree in networks of generalized classes. This spanning tree algorithm, however, can encounter situations where the network topology will cause it to construct very poor multicasting trees. We will see that the modified algorithm which will be presented has both an efficient running time and also will not encounter the problems that the spanning tree algorithm encounters.
CHAPTER 3

A MODIFIED ALGORITHM FOR MULTICASTING WITHIN A NETWORK OF GENERALIZED CLASSES

This chapter will present a modified algorithm which provides an efficient means of constructing a multicast tree within a network of generalized classes. It is an extremely efficient algorithm and does not encounter the problems that the spanning tree algorithm can result in. First will be an introduction discussing how this modified algorithm came to be. We will then give an overview on networks of generalized classes in general. Discussion will then focus on how the new algorithm integrates Makki’s algorithm[22] for Optimal Multicast Tree Construction. Finally, the modified algorithm will then be presented and then it will be followed up by a summary of the chapter.

3.1 Introduction

We started this research by deciding to implement some algorithms in multicasting. One of the algorithms implemented was an algorithm by Makki for construction of multicast trees within generalized class networks. In the midst of this implementation, we found that the algorithm was very difficult to implement and had a very high resource overhead. We then looked into ways of overcoming these issues and this led to the
modified algorithm. The modified algorithm was much easier to implement and also had much less of an overhead than the original algorithm.

3.2 Overview Of Generalized Class Networks

A common occurrence that we now see in networks is the presence of generalized class networks. These types of networks show themselves all across industry. In a generalized class network, the individual network nodes are of different classes. The word ‘class’ can span a variety of definitions. It could mean that the different nodes are functionally different such as one node being a payroll site and another node being a human resources site. It could also mean that the nodes are physically different, such as one node being an NT Server site while another node is a Novell Server site. Regardless of the specific definition of the word class, the need remains the same. This requirement is that we need to connect at least one node from each of the different classes together. This is where the new algorithm comes into play. It accomplishes this task in an efficient manner and provides an optimal solution every time.

3.3 Integration of Makki’s Algorithm

The key factor behind the efficiency of this modified algorithm is due to the fact that it uses an efficient and already proven algorithm at its core. Makki’s algorithm[22] for Optimal Multicast Tree Construction is extremely efficient and has a worst case execution
time of $O(|E| + |V| \log |V|)$. This algorithm has been integrated into my new algorithm as the method by which all the multicast trees are constructed. Since the majority of the execution time for this new algorithm will be spent in the construction of the multicast trees, it is very important that we use an efficient algorithm for multicast tree construction and Makki’s algorithm[22] provides this for us.

3.4 Modified Algorithm

Given a network $N$ with $V$ sites, $E$ communication pathways, and $C$ classes of network sites, we want to construct a minimal communications pathway which will allow transfer of information between at least $C$ network sites. That is, at least one network site of each class must be included in the resulting communications network. When we construct the multicast trees in this modified algorithm, we do not follow Makki’s algorithm to the letter. After we connect the different forests together, we do not create a minimum spanning tree on the resulting multicast tree. This guarantees that every vertex in the network will be connected. For details on constructing the multicast trees, see Makki’s Optimal Multicast Tree Construction Algorithm[22].

- We first implement the multicast tree algorithm a total of $C$ times. For each iteration of the algorithm, we mark a different set of network sites as the multicast nodes. In the first iteration, the multicast nodes will be the network sites of class 1. In the second iteration of the algorithm, the
network sites of class 2 will be the multicast nodes, etc. As each multicast
tree is constructed, we will store information identifying the cost of
constructing this particular multicast tree.

- After constructing the C multicast trees, we will now choose the one which
  serves our purpose best. The criteria for this selection is as follows: Choose
  the resulting multicast tree where the sum of the costs of all communication
  pathways used is less than any other sum of communication pathways for
  any other multicast tree. This particular multicast tree can now be
  reconstructed and will be the one which we use for the remainder of the
  algorithm.

- Now we must trim the resulting network. We do this from the network
  sites at the outer edges and work our way in. That is, for every network
  site which has only one communications pathway leading to it, we
determine if there is another network site of the same class as itself already
in the communications network. If we do determine that there is a duplicate
network site, then we remove the network site that we are currently
inspecting and also the communications pathway which was leading to it.
Repeat this iterative process until we are no longer able to trim any
network sites. Once this has been done, what we have is the final multicast
tree and the algorithm is complete.
3.5 Summary

After running this algorithm, what we now have is a multicast tree which connects at least one of each of the different classes in the network together. This tree is the optimal tree out of all possible trees which could have been constructed in the network. It was also constructed within a worst case bound of $O(|C|(|E|+|V|\log|V|))$. When compared to the execution time of the spanning tree algorithm we see that the spanning tree algorithm has a faster execution time, but it does not guarantee an optimal multicast tree. In fact, there are instances where the spanning tree algorithm can create multicast trees for the generalized class problem with an extremely high cost compared to the possible optimal multicast tree for the network.
CHAPTER 4
IMPLEMENTATION

We have implemented both the spanning tree algorithm and the new algorithm for multicast tree construction in a network of generalized classes. What will follow will be a description of how each algorithm was implemented. By this I mean data structures used and logic flow. Also, we will show an example case for each algorithm. Immediately following in Appendix I and Appendix II will be the actual code attachments for the implementation of these two algorithms.

4.1 Implementation of Spanning Tree Algorithm

In this algorithm we make use of a structure called Edge which contains information about edge communications pathways in the network. The structure contains the cost of the pathway, the identification of both network sites connected to each end, and also whether it has been chosen as a member of the final new network or not.

We also have an array called Vertex Status which we will use in constructing the minimum spanning tree. In it we will track the current status of each network site as we
are constructing the minimum spanning tree. The possible values in the structure will be UNINITIALIZED, UNUSED, LEAF, or NON_LEAF.

Two other arrays used is the Vertex Class and the Vertex Degree arrays. In them we keep the class of every network site in the communications network and also the current number of communication pathways currently leading to the network site.

Finally, we have a Class Counter array. This is a counter of how many network sites of each class are currently including in the network. This structure is used extensively in the trimming of the communication networks in the final step of the algorithm.

Now that we have defined the structures to be used I will quickly outline the routines in the algorithm.

• Capture user input as to the topology of the network.

• Construct the minimum spanning tree.

• Trim the extraneous network sites.

• Display the result network.
4.1.1 Example of Spanning Tree Algorithm

Given the network N which we can see in figure 9, we construct a minimum spanning tree on the network which results in figure 10. This minimum spanning tree is constructed using one of the spanning tree algorithms such as Kruskal’s or Prim’s. We then must trim off leaf nodes iteratively until no other leaf nodes can be trimmed from the network. After trimming extraneous nodes off of figure 10, we result with the network shown in figure 11.

Figure 9: The network N
Figure 10: The minimum spanning tree

Figure 11: The final communications network after trimming the extraneous nodes
4.2 Implementation of Modified Algorithm

As is usually the case, with a more efficient solution, we also complicate the implementation of the algorithm. Therefore, in addition to having the structure called Edge, we now also have two additional structures called Vertice and Forest Link.

The Vertice structure has information such as its multicast parent in the communications network, its direct parent in the communications network, a label status which is used in implementation of Dijkstra's shortest path algorithm which will hold the values, UNINITIALIZE, TEMPORARY LABEL, OR PERMANENT LABEL, and the cost of this network site's communications path back to its multicast parent.

The forest link structure holds information such as the identification of the multicast network sites in each forest that is linked with this link, the sum of the communications pathway leading from both sites just mentioned, the identification label of the communications pathway itself which is being used to establish the link, and finally a variable to let us know whether this forest link has been chosen to be a part of our final communications network.
Now I will give a brief outline of the new algorithm since we have defined the structures to be used.

- First capture user input as to the topology of the communications network.

- Now iterate through a loop for the number of classes entered in step 1. In each loop, we construct a multicast tree. At the end of each loop, we check to see if this multicast tree is a better tree than any of the previous ones. At the end of step 2, we know which class produces the multicast tree with the least cost sum of communication pathways.

- We now reconstruct the multicast tree with the least cost.

- Now trim any duplicate network sites.

- Display the resulting network.

4.2.1 Example of Modified Algorithm

Given the network N which we see in figure 12, we will start the construction of the multicast trees. Since there are three different classes in this network we will construct a total of three different multicast trees. The three different classes in the network are...
circles, triangles, and squares. What will follow will be a series of figures depicting the construction of the three different multicast trees. There will be two figures for the first two classes and only one for the third class when constructing each multicast tree. The first figure will be the result of constructing the forests from the original network using a modified form of Makki’s Optimal Multicast Construction Algorithm[22]. The second figure will be the result of grouping together the forests which result from the first figure. This will result in a total of five figures depicting the construction of the three different multicast trees. The first multicast tree we will construct will use the circle class. Figure 13 shows the resulting two forests. In figure 14 we see the connection of these two forests into the multicast tree for the circle class. Next we will construct the multicast tree for the square class. Figure 15 shows the two resulting forests resulting in using the square class as the multicast vertices and figure 16 shows the resulting multicast tree when joining these two forests together. We then use the triangle class as the multicast vertices and since there is only one triangle, there is only one forest resulting and we do not have to do any forest linking. The multicast tree for the triangle class is seen in figure 17.

After constructing the three multicast trees, we can see that the cost of the first multicast tree using circles as the destination sites, has a total cost of 14. The second multicast tree using squares as the destination sites has a total cost of 12. The third multicast tree using triangles as the destination sites also has a total cost of 12. As a matter of fact, the second and third multicast trees which were constructed are identical. Since we have two multicast trees with equal cost, we will choose the second one using squares as the destination sites for the remainder of the example.
Figure 18 shows the multicast tree which we have decided is the best cost tree to use. Finally, in figure 19, we see the resulting network after the implementation of the final step of the algorithm. This step I'm referring to is the trimming of all redundant nodes in the communications network. Figure 19 is a representation of an efficient network which would provide communication between at least one of each of the three different classes of network sites which were a part of the original communications network. The total cost of this resulting network is 3.

Figure 12: The network N
Figure 13: Forests resulting when using circles as the destination sites.

Figure 14: The multicast tree resulting from using circles as the destination sites.
Figure 15: Forests resulting from using squares as the destination sites.

Figure 16: The multicast tree resulting from using squares as the destination sites.
Figure 17: The multicast tree resulting from using triangles as the destination sites

Figure 18: The best cost multicast tree
Figure 19: The final network
CHAPTER 5

CONCLUSION

We now conclude by briefly revisiting the major topics of this thesis and then providing some directions for future research.

5.1 Summary

Previous work in the area of multicasting have produced a variety of algorithms for the construction of these multicasting trees. We have seen, however, that the majority of them have an extremely poor execution time in the worst case scenario. The most efficient one was the Optimal Multicast Tree Construction algorithm by Makki. Due to this, the decision was made to base the new algorithm for multicasting within a network of generalized classes on it. We also have seen that the spanning tree algorithm for networks of generalized classes also has an extremely efficient execution time. The difference, however, is in the fact that the spanning tree algorithm can produce extremely poor results under certain conditions where the new algorithm will not.
In conclusion, what we have shown is a modified algorithm which can be used to find an efficient means of multicasting information in a network of generalized classes. This method will produce an optimal multicasting tree every time. It uses Makki's Optimal Multicast Tree Construction algorithm as the heart of its implementation. Using this algorithm which has an execution time of $O(|C|(|E|+|V|\log|V|))$ in the worst case produces an efficient means of solving the generalized class problem in networks.

5.2 Future Work

Within this area, there are a variety of areas which can be further researched. For instance, a topic of research could be the integration of this algorithm into specific areas of teleconferencing. By this I mean communication between subgroups within a teleconferencing session. In such a case, each targeted subgroup member would become a class to be connected by the multicast tree. By doing this we can have teleconferencing sessions within teleconferencing sessions. Work also needs to be done in the area of multicasting within mobile communication environments where the network topology goes through frequent changes. Also in order to effectively utilize network bandwidth and resources, more efficient multicast routing algorithms need to be designed for both wireless and wired communications.
APPENDIX I

CODE ATTACHMENT FOR IMPLEMENTATION OF SPANNING TREE ALGORITHM

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define FALSE 0
#define TRUE 1

// used for entry limits
#define MAX_CLASSES 10
#define MAX_EDGES 100
#define MAX_VERTICES MAX_EDGES + 1

// used to show vertice status
#define UNINITIALIZED 0
#define UNUSED 1
#define LEAF 2
#define NON_LEAF 3

struct edge
{
    int edge_cost;
    int vertice_a_num;
    int vertice_b_num;
    int chosen;
};

typedef struct edge EDGE;

struct SEARCH_EDGE
{
    int edge_cost;
    int edge_index;
};

struct SEARCH_EDGE least_cost_edge;  // used in MST construction
struct SEARCH_EDGE max_cost_edge;    // used to trim the MST
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
EDGE edge_info[MAX_EDGES];  //array of all edges and
int connected_vertex_info[MAX_VERTICES];  //UNINITIALIZED,UNUSED,LEAF,or NON_LEAF
int connected_vertex_class[MAX_VERTICES];  //the class of each vertex
int connected_vertex_degree[MAX_VERTICES];  //the degree of each vertex
int class_counter[MAX_CLASSES];  //number of vertices of
//each class

void initialize_arrays_counter( void );  //initialize to 0
int enter_parameters( void );  //take in the network
void calculate_mst( int );  //calculate the MST
void find_initial_edge( int );  //called from calculate_mst
void trim_leaves( void );  //prune unnecessary vertices
void display_result( int );  //display the resulting
//edges

void main()
{
  int num_of_edges;

  //get the network information
  num_of_edges = enter_parameters();

  //if at least one edge was entered by the user
  if ( num_of_edges > 0 )
  {
    //calculate the minimum spanning tree
    calculate_mst( num_of_edges );

    //trim vertices
    trim_leaves();

    //display resulting network
    display_result( num_of_edges );
  }
  else
  {
    clrscr();
    printf("At least one edge must be entered!");
  }
}
//initialize all arrays to 0
void initialize_arrays()
{
    int counter;

    for( counter = 0; counter < MAX_CLASSES; ++counter )
    {
        class_counter[counter] = 0;
    }

    for( counter = 0; counter < MAX_VERTICES; ++counter )
    {
        connected_vertice_info[counter] = 0;
        connected_vertice_class[counter] = 0;
        connected_vertice_degree[counter] = 0;
    }
}

//allow the user to input the network's information
int enter_parameters()
{
    char user_entry[20];
    int counter;

    clrscr();
    for( counter = 0; counter < MAX_EDGES; ++counter )
    {
        printf ( "Enter cost of edge #\d \(0 to Quit)\n\n":counter );
        gets( user_entry );
        if ( atoi(user_entry) == 0 )
        {
            break;
        }
        edge_info[counter].edge_cost = atoi( user_entry );
        printf ( "Enter vertice number of one vertice connected to edge #\d\n\(0-%d\):",counter,
        MAX_VERTICES - 1 ) ;
        gets( user_entry );
        edge_info[counter].vertice_a_num = atoi( user_entry 
        );
        if( connected_vertice_info[edge_info[counter].vertice_a_num] == 
        UNINITIALIZED )
        {
            connected_vertice_info[edge_info[counter].vertice_a_num] = 
            UNUSED;
        }
    }
}
printf ( "Enter class of vertice #%d\n(0-%d):", 
    edge_info[counter].vertice_a_num, 
MAX_CLASSES - 1 );
gets( user_entry );

connected_vertex_class[edge_info[counter].vertice_a_num] = 
    atoi( user_entry );
}

printf ( "Enter vertice number of other vertice 
connected to edge #%d\n(0-%d):",counter, 
MAX_VERTICES - 1 );
gets( user_entry );
    edge_info[counter].vertice_b_num = atoi( user_entry );
    if( 
        connected_vertex_info[edge_info[counter].vertice_b_num] == 
        UNINITIALIZED )
    {
        connected_vertex_info[edge_info[counter].vertice_b_num] = 
            UNUSED;
        printf ( "Enter class of vertice #%d\n(0-%d):", 
            edge_info[counter].vertice_b_num, 
MAX_CLASSES - 1 );
        gets( user_entry );
        connected_vertex_class[edge_info[counter].vertice_b_num] = 
            atoi( user_entry );
    }
    printf( "\n" );
return counter;
}

//calculate the MST using Kruskal's algorithm
void calculate_mst( int num_of_edges )
{
    int counter,loop,found_one = FALSE;

    find_initial_edge( num_of_edges );

    //check out every edge, discarding the ones creating cycles
    for( loop = 0; loop < num_of_edges - 1; ++loop )
    {
        least_cost_edge.edge_cost = 0;
        for( counter = 0; counter < num_of_edges; ++counter
        )
        {
            if ( ( !least_cost_edge.edge_cost ||
                least_cost_edge.edge_cost > 
                edge_info[counter].edge_cost )

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
&& (  
  connected_vertex_info[edge_info[counter].vertex_a_num]
  == UNUSED || 
  connected_vertex_info[edge_info[counter].vertex_b_num] == UNUSED )
&&
!(connected_vertex_info[edge_info[counter].vertex_a_num] == UNUSED 
  && 
  connected_vertex_info[edge_info[counter].vertex_b_num] == UNUSED ) )
{
  least_cost_edge.edge_cost = 
  edge_info[counter].edge_cost;
  least_cost_edge.edge_index = counter;
  found_one = TRUE;
}
}
if( found_one )
{
  //increment the degrees of the two vertices  
  //associated with //this edge
  ++connected_vertex_degree[
  edge_info[least_cost_edge.edge_index].vertex_a_num];
  ++connected_vertex_degree[
  edge_info[least_cost_edge.edge_index].vertex_b_num];

  //update class counter information and vertice
  //status info //for both vertices associated with
  the edge
  switch(connected_vertex_info[
  edge_info[least_cost_edge.edge_index].vertex_a_num] )
  {
    case UNUSED:
      ++class_counter[connected_vertex_class[
        edge_info[
        least_cost_edge.edge_index].vertex_a_num]];
      connected_vertex_info[
        edge_info[least_cost_edge.edge_index].vertex_a_num] = LEAF;
      break;
    case LEAF:

connected_vertex_info[edge_info[
least_cost_edge.edge_index].vertex_a_num] =
    NON_LEAF;
break;
}
switch( connected_vertex_info[edge_info[
    least_cost_edge.edge_index].vertex_b_num] )
{
    case UNUSED:
        ++class_counter[connected_vertex_class[
            edge_info[
least_cost_edge.edge_index].vertex_b_num]];
        connected_vertex_info[edge_info[
least_cost_edge.edge_index].vertex_b_num] = LEAF;
        break;
    case LEAF:
        connected_vertex_info[edge_info[
least_cost_edge.edge_index].vertex_b_num] =
            NON_LEAF;
        break;
}
found_one = FALSE;
edge_info[least_cost_edge.edge_index].chosen =
    TRUE;
}

//find the minimum edge in the network, this will be the
initial edge in the MST
void find_initial_edge( int num_of_edges )
{
    //find the least cost edge
    least_cost_edge.edge_cost = 0;
    for( int counter = 0; counter < num_of_edges; ++counter 
    )
    {
        if ( !least_cost_edge.edge_cost || 
            least_cost_edge.edge_cost > edge_info[counter].edge_cost )
            { 
                least_cost_edge.edge_cost = edge_info[counter].edge_cost;
                least_cost_edge.edge_index = counter;
            } //update necessary vertice information
++connected_vertex_degree[edge_info[least_cost_edge.edge_index].vertex_a_num];
++connected_vertex_degree[edge_info[least_cost_edge.edge_index].vertex_b_num];

connected_vertex_info[edge_info[least_cost_edge.edge_index].vertex_a_num] = LEAF;
++class_counter[connected_vertex_class[edge_info[least_cost_edge.edge_index].vertex_a_num]];  
connected_vertex_info[edge_info[least_cost_edge.edge_index].vertex_b_num] = LEAF;
++class_counter[connected_vertex_class[edge_info[least_cost_edge.edge_index].vertex_b_num]];  
edge_info[least_cost_edge.edge_index].chosen = TRUE;
}

//trim unnecessary vertices
void trim_leaves()
{
    int found_one = TRUE;
    int all_pruned = FALSE;

    //keep looping through all the classes until nothing is found
    //through one iteration
    while( !all_pruned )
    {
        all_pruned = TRUE;
        for( int class_loop = 0; class_loop<MAX_CLASSES; ++class_loop )
        {
            found_one = FALSE;
            max_cost_edge.edge_cost = 0;

            //look through all vertices for the LEAF of the current class where
            //the class count on that vertex type is greater than 1
            //and it has the greatest cost edge connecting it to the MST
            for( int trim_search = 0; trim_search < MAX_VERTICES; ++trim_search )
            {
                //found a vertex
                if( connected_vertex_info[trim_search] == LEAF &&
                    connected_vertex_class[trim_search] ==
                    class_loop
                    &

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
class_counter[class_loop] > 1 )
{
    //find the edge associated with this
    vertice
    //and see if the edge connecting it is
    greater
    for( int edge_search = 0; edge_search <
        MAX_EDGES;
    {
        if( (edge_info[edge_search].vertex_a_num ==
            trim_search
            ||
            edge_info[edge_search].vertex_b_num ==
            trim_search )
            && edge_info[edge_search].chosen
            && ( max_cost_edge.edge_cost ==
                0 ||
                max_cost_edge.edge_cost <
                edge_info[edge_search].edge_cost ) )
        {
            max_cost_edge.edge_cost =
            edge_info[edge_search].edge_cost;
            max_cost_edge.edge_index =
            edge_search;
            found_one = TRUE;
            all_pruned = FALSE;
        }
    }
}

//if a LEAF was is to be pruned, update all its
associated
if( found_one )
{
    //decrement the degree of the LEAF and the
    other vertice
    --connected_vertex_degree[edge_info[
        max_cost_edge.edge_index].vertex_a_num];
    --connected_vertex_degree[edge_info[
        max_cost_edge.edge_index].vertex_b_num];

    //find which edge it was, mark one vertice
    UNUSED and
    //the other vertice as a leaf
    and - 1 from that class
    edge_info[max_cost_edge.edge_index].chosen =
    FALSE;
if( connected_vertex_info[edge_info[
    max_cost_edge.edge_index].vertex_b_num] == LEAF )
    {
        --class_counter[connected_vertex_class[
            edge_info[
                max_cost_edge.edge_index].vertex_b_num]];}

connected_vertex_info[edge_info[
    max_cost_edge.edge_index].vertex_b_num] = UNUSED;

if( connected_vertex_degree[edge_info[
    max_cost_edge.edge_index].vertex_a_num] == 1 )
    {
        connected_vertex_info[edge_info[
            max_cost_edge.edge_index].vertex_a_num] = LEAF;
    }
else
    {
        --class_counter[connected_vertex_class[
            edge_info[max_cost_edge.edge_index].vertex_a_num]];}

connected_vertex_info[edge_info[max_cost_edge.edge_index].vertex_a_num] = UNUSED;

if( connected_vertex_degree[edge_info[
    max_cost_edge.edge_index].vertex_b_num] == 1 )
    {
        connected_vertex_info[edge_info[
            max_cost_edge.edge_index].vertex_b_num] = LEAF;
    }
}

//display the edges which were chosen
void display_result( int num_of_edges )
{
    printf("\n\nEdges Chosen:\n");
    for ( int counter = 0; counter < num_of_edges; ++counter)
    {
        if( edge_info[counter].chosen == TRUE )
            {
                printf("Edge %d, with weight %d, connecting
vertice %d, class %d with vertice
%d, class %d\n",
counter, edge_info[counter].edge_cost, edge_info[counter].vertex_a_num,
  connected_vertex_class[edge_info[counter].vertex_a_num],
  edge_info[counter].vertex_b_num,
  connected_vertex_class[edge_info[counter].vertex_b_num]);
}
APPENDIX II

CODE ATTACHMENT FOR IMPLEMENTATION OF NEW ALGORITHM

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <fstream.h>

#define FALSE 0
#define TRUE 1
#define INFINITY 9999

// used for entry limits
#define MAX_CLASSES 10
#define MAX_EDGES 100
#define MAX_VERTICES MAX_EDGES + 1

// used to show vertice status
#define UNINITIALIZED 0
#define TEMP_LABEL 1
#define UNUSED 1
#define PERM_LABEL 2
#define LEAF 2
#define NON_LEAF 3

int num_classes = 0; // the number of classes in the network
int num_vertices = 0; // the number of vertices in the network
int num_multicast_vertices = 0; // the number of multicast vertices in this network
int num_of_edges = 0; // the number of edges in this network
int num_of_links = 0; // the number of links joining the forests
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
int best_weight = 0; // least cost
weight of a
// multicast tree
int best_class = 0; // the class
which created
multicast tree

struct edge
    // necessary
    // an
    information of
    // necessary
edge
{
    int edge_cost;
    int vertice_a_num;
    int vertice_b_num;
    int chosen;
};
typedef struct edge EDGE;

struct vertice
    // necessary
vertice info
{
    int weight;
    int label_status;
    int parent;
    int multicast_parent;
};
typedef struct vertice VERTICE;

struct forest_link
    // necessary to
join forests
{
    int vertice_a;
    int vertice_b;
    int distance;
    int edge_num;
    int chosen;
};
typedef forest_link FOREST_LINK;

struct SEARCH_EDGE
{
    int edge_cost;
    int edge_index;
};
struct SEARCH_EDGE least_cost_edge; // used in MST
construction
struct SEARCH_EDGE max_cost_edge; // used to trim the
    // resulting
MST

struct SEARCH_VERTICE
    // used in
shortest path
    // algorithm
{

int vertex_cost;
int vertex_index;
;
struct SEARCH_VERTICE least_cost_vertex;

EDGE edge_info[MAX_EDGES]; //array of all edges and info
VERTICE vertex_info[MAX_VERTICES]; //array of all edges and info
FOREST_LINK links[MAX_EDGES]; //used to join forests
int connected_vertex_info[MAX_VERTICES]; //UNINITIALIZED, UNUSED, LEAF, or NON_LEAF
int connected_vertex_class[MAX_VERTICES]; //the class of each vertex
int connected_vertex_degree[MAX_VERTICES]; //the degree of each vertex
int class_counter[MAX_CLASSES]; //number of vertices of each class

void initialize_arrays( void ); //initialize variables
int enter_parameters( void ); //take in the network
void calculate_mst( int ); //calculate the MST
void find_initial_edge( int ); //called from calculate_mst
void trim_leaves( void ); //prune unnecessary vertices
void display_result( int ); //display the resulting //edges
void build_forests( int ); //use Dijkstra's shortest path algorithm
void label_multicast_vertices( int ); //called from build_forests
void find_min_labeled_vertex( void ); //called from build_forests
void add_labels( int ); //called from build_forests
void relabel( int, int, int ); //called from build_forests
void join_forests( void ); //join the forests together
void check_for_new_link( int ); //called from join_forests

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
void expand_mst( void );  //expand the
MST to
//individual edges
void calc_tree_weight( int );  //calculate
this tree's    //total
weight
void build_multicast_tree( int ); //build a
multicast tree
//for a class
void bookkeeping( void ); //get vertex's
degrees and    //set class
counters

void main()
{
    //get the network information
    num_of_edges = enter_parameters();

    //if at least one edge was entered by the user
    if ( num_of_edges > 0 )
    {
        for( int class_loop = 0; class_loop<num_classes;
++class_loop )
        {
            build_multicast_tree( class_loop );
        }

        //recalculate the tree resulting from the class
        //creating the //least total edge weight
        build_multicast_tree( best_class );

        //expand the MST to the individual edges
        expand_mst();

        //calculate the degrees of each vertex and count
        elements //in each class
        bookkeeping();

        //trim_vertices
        trim_leaves();

        //display resulting network
        display_result( num_of_edges );
    }

    //error, must have at least one edge
    else
    {
        //clrscr();
        printf("At least one edge must be entered!");
    }
//initialize all arrays to 0
void initialize_arrays()
{
    int counter;

    for( counter = 0; counter < MAX_VERTICES; ++counter )
    {
        connected_vertex_info[counter] = 0;
        connected_vertex_degree[counter] = 0;
    }
}

//allow the user to input the network's information
int enter_parameters()
{
    char user_entry[20];
    int counter;

    //get the number of classes and vertices in the network
    printf ( "Enter the number of classes in this network.
(1 to %d)\n ": MAX_CLASSES );
    gets( user_entry );
    if ( atoi( user_entry ) == 0 )
    {
        return 0;
    }
    else
    {
        num_classes = atoi( user_entry );
    }
    printf ( "Enter the number of vertices in this network.
(1 to %d)\n ": MAX_VERTICES );
    gets( user_entry );
    if ( atoi( user_entry ) == 0 )
    {
        return 0;
    }
    else
    {
        num_vertices = atoi( user_entry );
    }

    //now get the edge and vertice info info
    for( counter = 0; counter < MAX_EDGES; ++counter )
    {
        printf ( "Enter cost of edge #\d (0 to Quit)\n ": counter );
        gets( user_entry );
        if ( atoi( user_entry ) == 0 )
        {
            
        }
    }
}
break;
}
edge_info[counter].edge_cost = atoi( user_entry );
printf( "Enter vertice number of one vertice connected to edge #%d\n(0-%d)\n",counter,
num_vertices - 1 );
gets( user_entry );
edge_info[counter].vertice_a_num = atoi( user_entry ) ;
if {
connected_vertex_info[edge_info[counter].vertice_a_num] == UNINITIALIZED )
{
connected_vertex_info[edge_info[counter].vertice_a_num] = UNUSED;
printf ( "Enter class of vertice #%d\n(0-%d)\n:\n",edge_info[counter].vertice_a_num, num_classes - 1 );
gets( user_entry );
connected_vertex_class[edge_info[counter].vertice_a_num] = atoi( user_entry );
}
printf ( "Enter vertice number of other vertice connected to edge #%d\n(0-%d)\n":,counter,
num_vertices - 1 );
gets( user_entry );
edge_info[counter].vertice_b_num = atoi( user_entry ) ;
if ( connected_vertex_info[edge_info[counter].vertice_b_num] == UNINITIALIZED )
{
connected_vertex_info[edge_info[counter].vertice_b_num] = UNUSED;
printf ( "Enter class of vertice #%d\n(0-%d)\n:\n",edge_info[counter].vertice_b_num, num_classes - 1 );
gets( user_entry );
connected_vertex_class[edge_info[counter].vertice_b_num] = atoi( user_entry );
}
printf( "\n" );
return counter;

void build_multicast_tree( int class_loop )
{  
    //initialize buffers  
    initialize_arrays();

    //comprise forests  
    build_forests( class_loop );

    //join the forests together  
    join_forests();

    //create the MST  
    calculate_mst( num_of_links );

    //calculate this tree's total weight  
    calc_tree_weight( class_loop );
}

//build the forests using Djikstra's shortest path algorithm  
void build_forests( int class_loop )  
{
    int done = 0;

    //find all initial Multicast vertices  
    label_multicast_vertices( class_loop );

    //loop until all vertices have been labeled  
    while( !done )
    {
        //find the minimum of all temp_labeled vertices  
        find_min_labeled_vertex();

        //all vertices were labeled  
        if( least_cost_vertex.vertex_index == -1 )
        {
            done = TRUE;  
            continue;
        }

        //mark the least_cost vertex's label status to permanent  
        vertice_info[least_cost_vertex.vertex_index].label_status =
            PERM_LABEL;

        //find any children of this vertex which are not perm labeled  
        //and label them temp  
        add_labels( least_cost_vertex.vertex_index );
    }

    //label initial multicast vertices  
    void label_multicast_vertices( int class_loop )
num_multicast_vertices = 0;
for( int vertice_loop = 0; vertice_loop < num_vertices;
  ++vertice_loop )
{
  if( connected_vertice_class[vertice_loop] ==
    class_loop )
  {
    ++num_multicast_vertices;
    vertice_info[vertice_loop].label_status =
    TEMP_LABEL;
    vertice_info[vertice_loop].weight = 0;
    vertice_info[vertice_loop].parent = -1;
    vertice_info[vertice_loop].multicast_parent =
    vertice_loop;
  }
  else
  {
    vertice_info[vertice_loop].label_status =
    UNINITIALIZED;
    vertice_info[vertice_loop].parent = 0;
    vertice_info[vertice_loop].multicast_parent = 0;
    vertice_info[vertice_loop].weight = INFINITY;
  }
}

//find the minimum of all the temp labeled vertices
void find_min_labeled_vertice()
{
  least_cost_vertice.vertex_cost = INFINITY;
  least_cost_vertice.vertex_index = -1;
  for( int vertice_loop = 0; vertice_loop < num_vertices;
       ++vertice_loop )
  {
    if( vertice_info[vertice_loop].weight <=
        least_cost_vertice.vertex_cost
        && vertice_info[vertice_loop].label_status ==
        TEMP_LABEL )
    {
      least_cost_vertice.vertex_cost =
      vertice_info[vertice_loop].weight;
      least_cost_vertice.vertex_index = vertice_loop;
    }
  }
}

//add temp labels and recompute weights if necessary
void add_labels( int the_vertice )
{
  //loop through all edges finding those adjacent to the
  vertice just
  //perm_labelled
for( int counter = 0; counter < num_of_edges; ++counter )
{
  if( edge_info[counter].vertex_a_num == the_vertex \\
    ||
    edge_info[counter].vertex_b_num == the_vertex )
  {
    if( edge_info[counter].vertex_a_num == the_vertex )
    {
      if( vertice_info[the_vertex].weight + \\
        edge_info[counter].edge_cost< \\
        vertice_info[edge_info[counter].vertex_b_num].weight )
        relabel(
          vertice_info[the_vertex].weight + \\
          edge_info[counter].edge_cost,
          edge_info[counter].vertex_b_num,
          edge_info[counter].vertex_a_num );
    }
    else
    {
      if( vertice_info[the_vertex].weight + \\
        edge_info[counter].edge_cost< \\
        vertice_info[edge_info[counter].vertex_a_num].weight )
        relabel(
          vertice_info[the_vertex].weight + \\
          edge_info[counter].edge_cost,
          edge_info[counter].vertex_a_num,
          edge_info[counter].vertex_b_num );
    }
  }
}

//add a temp label and relabel the original path if necessary
void relabel( int new_cost, int the_vertex, int new_parent )
{
  vertice_info[the_vertex].weight = new_cost;
  vertice_info[the_vertex].parent = new_parent;
  vertice_info[the_vertex].label_status = TEMP_LABEL;
  vertice_info[the_vertex].multicast_parent = \\
  vertice_info[new_parent].multicast_parent;
//join the forests together
void join_forests()
{
    num_of_links = 0;
    for( int edge_counter = 0; edge_counter < num_of_edges;
        ++edge_counter )
    {
        check_for_new_link( edge_counter );
    }
}

//check if this edge creates a link which doesn't already exist
void check_for_new_link( int edge_number )
{
    int found_it = FALSE;
    //if this edge doesn't link two different forests, then return
    if ( vertice_info[edge_info[ edge_number].vertex_a_num].multicast_parent ==
        vertice_info[edge_info[ edge_number].vertex_b_num].multicast_parent )
    {
        return;
    }

    //loop through C(number of multicast vertices,2) times looking for
    //this link
    for( int link_counter = 0; link_counter < num_of_links;
        ++link_counter )
    {
        if ( ( links[link_counter].vertice_a ==
              vertice_info[edge_info[ edge_number].vertex_a_num].multicast_parent
              && links[link_counter].vertice_b ==
              vertice_info[edge_info[ edge_number].vertex_b_num].multicast_parent )
            || ( links[link_counter].vertice_a ==
                 vertice_info[edge_info[ edge_number].vertex_b_num].multicast_parent
                 && links[link_counter].vertice_b ==
                 vertice_info[edge_info[ edge_number].vertex_a_num].multicast_parent ) )
        {
            //if this edge constructs a shorter link, then
            use it //instead of the last one
            found_it = TRUE;
        }
if( ( edge_info[edge_number].edge_cost + 
    vertex_info[edge_info[
        edge_number].vertex_a_num].weight + 
    vertex_info[edge_info[
        edge_number].vertex_b_num].weight ) < 
    links[link_counter].distance ) 
{
    links[link_counter].distance = 
    edge_info[edge_number].edge_cost + 
    vertex_info[edge_info[
        edge_number].vertex_a_num].weight + 
    vertex_info[edge_info[
        edge_number].vertex_b_num].weight;
    links[link_counter].edge_num = edge_number;
}

//new link so add it to the list
if ( !found_it )
{
    links[num_of_links].edge_num = edge_number;
    links[num_of_links].vertex_a = 
    vertex_info[edge_info[
        edge_number].vertex_a_num].multicast_parent;
    links[num_of_links].vertex_b = 
    vertex_info[edge_info[
        edge_number].vertex_b_num].multicast_parent;
    links[num_of_links].distance = 
    vertex_info[edge_info[
        edge_number].vertex_a_num].weight + 
    vertex_info[edge_info[
        edge_number].vertex_b_num].weight + 
    edge_info[edge_number].edge_cost;
    ++num_of_links;
}

//calculate the MST using Kruskal's algorithm
void calculate_mst( int num_of_edges )
{
    int counter, loop, found_one = FALSE;
    find_initial_edge( num_of_edges );
    //check out every edge, discarding the ones creating cycles
    for( loop = 0; loop < num_of_edges - 1; ++loop )
    {
        least_cost_edge.edge_cost = 0;
        for( counter = 0; counter < num_of_edges; ++counter
            }
if ( ( !least_cost_edge.edge_cost || least_cost_edge.edge_cost > links[counter].distance )
    && ( connected_vertex_info[links[counter].vertex_a] == UNUSED
        || connected_vertex_info[links[counter].vertex_b] == UNUSED )
    && !(connected_vertex_info[links[counter].vertex_a] == UNUSED
        && connected_vertex_info[links[counter].vertex_b] == UNUSED ) )
{
    least_cost_edge.edge_cost = links[counter].distance;
    least_cost_edge.edge_index = counter;
    found_one = TRUE;
}
if ( found_one )
{
    //update vertice status info for both vertices
    //associated with the edge
    switch( connected_vertex_info[links[least_cost_edge.edge_index].vertex_a] )
    {
        case UNUSED:
            connected_vertex_info[links[least_cost_edge.edge_index].vertex_a] = LEAF;
            break;
        case LEAF:
            connected_vertex_info[links[least_cost_edge.edge_index].vertex_a] = NON_LEAF;
            break;
    }
    switch( connected_vertex_info[links[least_cost_edge.edge_index].vertex_b] )
    {
        case UNUSED:
            connected_vertex_info[links[least_cost_edge.edge_index].vertex_b] = LEAF;
            break;
        case LEAF:
            connected_vertex_info[links[least_cost_edge.edge_index].vertex_b] = NON_LEAF;
            break;
    }
}
connected_vertex_info[links[least_cost_edge.edge_index].vertex_b] = NON_LEAF;
break;
}
found_one = FALSE;
links[least_cost_edge.edge_index].chosen = TRUE;
}
}

//find the minimum edge in the network, this will be the initial edge in the MST
void find_initial_edge( int num_of_edges )
{
    //find the least cost edge
    least_cost_edge.edge_cost = 0;
    for( int counter = 0; counter < num_of_edges; ++counter
    {
        if ( !least_cost_edge.edge_cost || least_cost_edge.edge_cost >
            links[counter].distance )
        {
            least_cost_edge.edge_cost =
            links[counter].distance;
            least_cost_edge.edge_index = counter;
        }
    }
    //update necessary vertex information
    connected_vertex_info[links[least_cost_edge.edge_index].vertex_a]
    = LEAF;
    connected_vertex_info[links[least_cost_edge.edge_index].vertex_b]
    = LEAF;
    links[least_cost_edge.edge_index].chosen = TRUE;
}

//calculate this tree's total weight
void calc_tree_weight( int the_class )
{
    int total_weight = 0;
    //for each pair of vertices, find the edge connecting the two then
    //add it to the total weight
    for( int counter = 0; counter < num_vertices; ++counter

for( int edge_loop = 0; edge_loop < num_of_edges; ++edge_loop )
{
    if( ( vertice_info[counter].parent ==
        edge_info[edge_loop].vertice_a_num
        &&
        counter == edge_info[edge_loop].vertice_b_num )
    || ( vertice_info[counter].parent ==
        edge_info[edge_loop].vertice_b_num &&
        counter == edge_info[edge_loop].vertice_a_num ) )
    {
        total_weight +=
        edge_info[edge_loop].edge_cost;
    }
}

//for each link between forests, add the links weight to the total weight
for( counter = 0; counter < num_of_links; ++counter )
{
    if ( links[counter].chosen == TRUE )
    {
        total_weight +=
        edge_info[links[counter].edge_num].edge_cost;
    }
}

//replace best weight found yet if this weight is less than it
if ( total_weight < best_weight || best_weight == 0 )
{
    best_weight = total_weight;
    best_class = the_class;
}

//expand the MST to its individual edges
void expand_mst()
{
    //for each pair of vertices, find the edge connecting the two then
    //mark it used
    for( int counter = 0; counter < num_vertices; ++counter )
    {
for( int edge_loop = 0; edge_loop < num_of_edges; ++edge_loop )
{
    if( ( vertice_info[counter].parent == 
            edge_info[edge_loop].vertice_a_num 
        &&
        counter == 
            edge_info[edge_loop].vertice_b_num )
        || ( vertice_info[counter].parent == 
            edge_info[edge_loop].vertice_b_num &&
        counter == 
            edge_info[edge_loop].vertice_a_num ))
    {
        edge_info[edge_loop].chosen = TRUE;
    }
}

//for each link between forests, add the links weight to the total weight
for( counter = 0; counter < num_of_links; ++counter )
{
    if ( links[counter].chosen == TRUE )
        
        edge_info[links[counter].edge_num].chosen = TRUE;
}

//get vertex's degrees and set class counters
void bookkeeping()
{
    //calculate the degree of each vertice
    for( int edge_loop = 0; edge_loop < num_of_edges; ++edge_loop )
    {
        if( edge_info[edge_loop].chosen == TRUE )
            
            ++connected_vertex_degree[
                edge_info[edge_loop].vertice_a_num];
            
            ++connected_vertex_degree[
                edge_info[edge_loop].vertice_b_num];
    }

    //relabel the network nodes as LEAF or NON_LEAF and count the number of elements in each class
    for( int vertice_loop = 0; vertice_loop < num_vertices; ++vertice_loop )
    {
        if( connected_vertex_degree[vertice_loop] > 1 )

{  
  connected_vertex_info[vertice_loop] = NON_LEAF;
}
else
{
  connected_vertex_info[vertice_loop] = LEAF;
}
++class_counter[connected_vertex_class[vertice_loop]];
}

//trim unnecessary vertices
void trim_leaves()
{
  int found_one = TRUE;
  int all_pruned = FALSE;

  //keep looping through all the classes until nothing is found
  //through one iteration
  while( !all_pruned )
  {
    all_pruned = TRUE;
    for ( int class_loop = 0; class_loop < num_classes; ++class_loop )
    {
      found_one = FALSE;
      max_cost_edge.edge_cost = 0;

      //look through all vertices for the LEAF of
      //the current class where the class count on
      //that vertice type is greater than 1
      //and it has the greatest cost edge connecting it to
      //the MST
      for( int trim_search = 0; trim_search < num_vertices; ++trim_search )
      {
        //found a vertice
        if(
        connected_vertex_info[trim_search] == LEAF
        &&
        connected_vertex_class[trim_search] ==
        class_loop &&
        class_counter[class_loop] > 1 )
        {
          //find the edge associated
          //with this vertice
          //and see if the edge
          //is greater
for( int edge_search = 0; 
    ++edge_search )
{
    if( ( edge_info[ 
         edge_search].vertex_a_num == 
        trim_search ) || edge_info[ 
         edge_search].vertex_b_num == 
        trim_search )
        &&
        edge_info[ edge_search].chosen == 
        TRUE
        && ( 
        max_cost_edge.edge_cost == 0
        < 
        edge_search].edge_cost ) )
    {
        max_cost_edge.edge_cost = 
        edge_info[ 
        edge_search].edge_cost;
        max_cost_edge.edge_index = 
        edge_search;
        found_one = TRUE;
        all_pruned = FALSE;
    }
}

// if a LEAF was is to be pruned, update all 
// its // associated information
if( found_one )
{
    // decrement the degree of the LEAF
    // vertex connected to the
    -- connected_vertex_degree[ 
        edge_info[ 
        max_cost_edge.edge_index].vertex_a_num];
    -- connected_vertex_degree[ 
        edge_info[ 
        max_cost_edge.edge_index].vertex_b_num];
    // find which edge it was, mark one
    vertice // UNUSED and the other vertice
    as a leaf and - 1 // from that class counter

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
edge_info[max_cost_edge.edge_index].chosen = FALSE;
    if( connected_vertex_info[
        max_cost_edge.edge_index].vertex_b_num] == 'LEAF')
    {
        --
        class_counter[connected_vertex_class[
            edge_info[
                max_cost_edge.edge_index].vertex_b_num]];
        connected_vertex_info[edge_info[
            max_cost_edge.edge_index].vertex_b_num] = UNUSED;
        if(connected_vertex_degree[edge_info[
            max_cost_edge.edge_index].vertex_a_num] == 1 )
        {
            connected_vertex_info[
                max_cost_edge.edge_index].vertex_a_num] = LEAF;
        }
    }
    else
    {
        --
        class_counter[connected_vertex_class[
            edge_info[
                max_cost_edge.edge_index].vertex_a_num]];
        connected_vertex_info[edge_info[
            max_cost_edge.edge_index].vertex_a_num] = UNUSED;
        if( connected_vertex_degree[
            max_cost_edge.edge_index].vertex_b_num] == 1 )
        {
            connected_vertex_info[
                max_cost_edge.edge_index].vertex_b_num] = LEAF;
        }
    }
}
/display the edges which were chosen
void display_result( int num_of_edges )
{
    printf("\n\nEdges Chosen:\n");
    for ( int counter = 0; counter < num_of_edges; ++counter)
    {
        if ( edge_info[counter].chosen == TRUE )
        {
            printf("Edge %d, with weight %d, connecting
            vertice %d, class %d with vertice
            %d, class %d\n",
            counter, edge_info[counter].edge_cost,
            edge_info[counter].vertex_a_num,
            connected_vertex_class[edge_info[counter].vertex_a_num],
            edge_info[counter].vertex_b_num,
            connected_vertex_class[edge_info[counter].vertex_b_num]);
        }
    }
}
BIBLIOGRAPHY


