

## ABSTRACT

Title of dissertation: THE MINIMUM LABELING SPANNING TREE  
PROBLEM AND SOME VARIANTS

Yupei Xiong, Doctor of Philosophy, 2005

Dissertation directed by: Dr. Bruce Golden  
Robert H. Smith School of Business

Minimum Labeling Spanning Tree Problem and Some Variants

by

Yupei Xiong

Dissertation submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
2005

Advisory Committee:

Dr. Bruce Golden, Chair/Advisor

Dr. Edward Wasil

Dr. Denny Gulick

Dr. Jeffrey Cooper

Dr. Hani Mahmassani

© Copyright by

Yupei Xiong

2005

This dissertation is dedicated to my parents and my beloved girlfriend Qian Li.

## TABLE OF CONTENTS

List of Figures	vii
1 Introduction	1
1.1 Overview . . . . .	1
1.2 The Minimum Labeling Spanning Tree (MLST) Problem . . . . .	1
1.3 The Label-Constrained Minimum Spanning Tree (LC-MST) Problem	3
1.4 The Colorful Traveling Salesman Problem (CTSP) . . . . .	3
2 Literature Review	5
3 Worst-case Behavior of the MVCA Heuristic for the Minimum Labeling Spanning Tree Problem	8
3.1 Introduction . . . . .	8
3.2 A Perverse-Case Family of Graphs . . . . .	9
3.3 Worst-case Bound of MVCA . . . . .	10
3.4 A Worst-case Family of Graphs . . . . .	15
3.5 Concluding Remarks . . . . .	17
4 A One-Parameter Genetic Algorithm for the Minimum Labeling Spanning Tree Problem	19
4.1 Introduction . . . . .	19
4.2 Genetic Algorithm . . . . .	20

4.2.1	Encoding . . . . .	21
4.2.2	Crossover . . . . .	22
4.2.3	Mutation . . . . .	24
4.2.4	Building each generation . . . . .	24
4.2.5	Running time analysis . . . . .	26
4.3	Computational Results . . . . .	27
4.3.1	Group I . . . . .	29
4.3.2	Group II . . . . .	30
4.3.3	Group III . . . . .	30
4.3.4	Summary . . . . .	31
4.4	Concluding Remarks . . . . .	32
5	Improved Heuristics for the Minimum Labeling Spanning Tree Problem	38
5.1	Introduction . . . . .	38
5.2	Three Modified Versions of MVCA . . . . .	39
5.3	A Modified Genetic Algorithm . . . . .	40
5.4	Computational Results . . . . .	41
5.5	Conclusions . . . . .	42
6	The Label-Constrained Minimum Spanning Tree Problem (LC-MST)	48
6.1	Introduction . . . . .	48

6.2	NP-completeness proof . . . . .	49
6.3	Local search methods . . . . .	50
6.3.1	Encoding . . . . .	51
6.3.2	Local search 1 (LS1) . . . . .	52
6.3.3	Local search 2 (LS2) . . . . .	52
6.3.4	Running time analysis . . . . .	53
6.4	Genetic algorithm . . . . .	54
6.4.1	Crossover . . . . .	54
6.4.2	Mutation . . . . .	55
6.4.3	Building each generation . . . . .	55
6.5	Computational Results . . . . .	56
6.5.1	Small cases . . . . .	56
6.5.2	Large Cases . . . . .	57
7	The Colorful Traveling Salesman Problem (CTSP)	62
7.1	Introduction . . . . .	62
7.2	CTSP is NP-hard . . . . .	63
7.3	Maximum Path Extension Algorithm . . . . .	64
7.3.1	How to extend a partial tour? . . . . .	65
7.3.2	Maximum Path Extension Algorithm . . . . .	67

7.3.3	Running time analysis . . . . .	69
7.4	Genetic Algorithm . . . . .	70
7.5	Computational results . . . . .	72
7.6	Concluding Remarks . . . . .	73
8	Conclusion	75
A	Overview	76
A.1	MVCA heuristic . . . . .	76
A.2	Group II graphs for the MLST problem . . . . .	77
A.3	A small sample graph for LC-MST problem . . . . .	78
A.4	A special family of graphs for LC-MST problem . . . . .	80
A.5	IP Formulation for the LC-MST Problem . . . . .	81
A.5.1	Notation and Variables . . . . .	81
A.5.2	IP Formulation . . . . .	82
A.6	Backtrack Search . . . . .	84
A.6.1	Backtrack Search for the MLST problem . . . . .	84
A.6.2	Backtrack Search for the LC-MST problem . . . . .	86
A.6.3	Backtrack Search for the CTSP . . . . .	86
A.7	Source Code . . . . .	88
	Bibliography	275



## LIST OF FIGURES

1.1	A spanning tree in a graph . . . . .	2
1.2	A tour or a Hamiltonian cycle in a graph . . . . .	4
2.1	A sample graph and its optimal solution for the MLST problem . . .	6
2.2	How MVCA works . . . . .	7
3.1	Perverse-case graph for $n = 4$ . . . . .	10
3.2	Perverse-case graph for $n = 6$ . . . . .	11
3.3	Worst-case graph for $b = 3$ . . . . .	16
3.4	Worst-case graph for $b = 4$ . . . . .	18
4.1	Encoding feasible solutions: $\{1, 2\}$ and $\{3, 4\}$ are two feasible solutions	22
4.2	An example of crossover . . . . .	23
4.3	An example of mutation . . . . .	25
4.4	Build generations with $p=4$ . . . . .	26
4.5	The general frequency of label $c$ is 6. There are two cycle edges (dashed lines). So, the net frequency of label $c$ is 4. . . . .	28
7.1	A small example of CTSP: Tour $h$ contains three different labels $b, c, f$ ; tour $g$ contains two different labels $b, e$ . So tour $g$ is better. . .	65
7.2	The original partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_{k+1} \rightarrow v_{j+1} \rightsquigarrow v_k$ . . . . .	67

7.3	The original partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_k \rightsquigarrow v_{j+1} \rightarrow v_{k+1}$ , $v_{k+1}$ is at the end. . . . .	68
7.4	The original partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is $v_{k+1} \rightarrow v_j \rightsquigarrow v_1 \rightarrow v_{j+1} \rightsquigarrow v_k$ . . . . .	69
7.5	The original partial tour is $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is $v_{k+1} \rightarrow v_j \rightsquigarrow v_1 \rightarrow v_k \rightsquigarrow v_{j+1}$ . . . . .	70
7.6	Subgraph induced by $\{b, e\}$ . It contains a Hamiltonian cycle $6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 6$ . The tour has two labels. . . . .	71
7.7	Subgraph induced by $\{b, d\}$ . It does not contain a Hamiltonian cycle. We have to add edge $(3, 6)$ to form a Hamiltonian cycle $6 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 6$ . The tour contains three labels. . . . .	72
A.1	Some minimum spanning trees in a graph with $n = 10$ . . . . .	80
A.2	An example where local optimal $\neq$ global optimal . . . . .	81
A.3	A sample graph for the recursive algorithm . . . . .	87

# Chapter 1

## Introduction

### 1.1 Overview

The focus of my dissertation research involves combinatorial optimization. This is a key area in operations research and computer science. It includes specific, well-known problems such as the traveling salesman problem, the bin packing problem, sequencing and scheduling problems, location and network design problems, etc. Each of these problems has a wide variety of real-world applications. In addition, most of these problems are inherently difficult to solve.

My specific dissertation topic is the minimum labeling spanning tree (MLST) problem and some variants, including the label-constrained minimum spanning tree (LC-MST) problem and the colorful traveling salesman problem (CTSP).

### 1.2 The Minimum Labeling Spanning Tree (MLST) Problem

Given a graph  $G = (V, E)$ , a spanning tree of  $G$  is a connected subgraph of  $G$  which has no cycles and spans all the nodes of  $G$ . If  $G$  contains  $n$  nodes, then a spanning tree of  $G$  has  $n - 1$  edges. Figure 1.1 shows an example of a spanning tree in a graph. The highlighted edges construct a spanning tree in the graph.

In the minimum labeling spanning tree (MLST) problem, we are given a

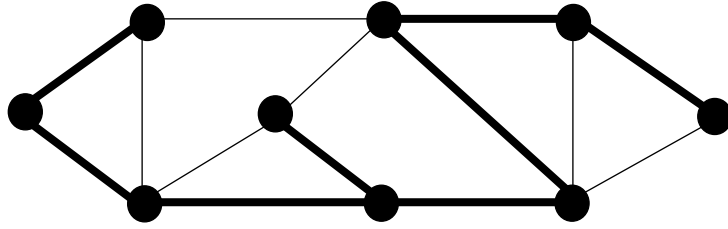


Figure 1.1: A spanning tree in a graph

labeled graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels (or colors). Each edge in  $E$  is colored by a label in  $L$ . Different edges may have the same label. The purpose of the MLST problem is to find a spanning tree of  $G$  which contains the smallest number of distinct labels.

In contrast to the traditional minimum spanning tree (MST) problem, the MLST problem focuses on the uniqueness of a graph instead of the cost of a graph. The MLST problem is motivated by applications in communications network design. For example, in communications networks, there are many different types of links, such as fiber optic, cable, microwave, telephone line, and so on. A communication node may communicate with different nodes by choosing different types of communication links. Given a set of nodes, the problem we are interested in is to find a spanning tree that uses as few types of communications links as possible. This spanning tree will reduce the construction cost and complexity of the network.

### 1.3 The Label-Constrained Minimum Spanning Tree (LC-MST) Problem

In the Label-Constrained Minimum Spanning Tree (LC-MST) problem, we are given an undirected graph  $G = (V, E, L)$  and a positive integer  $K$ , where  $V, E, L$  are the same as defined in the MLST problem. Each edge in  $E$  has both a label and a cost. The purpose of the LC-MST problem is to find a minimum-cost spanning tree of  $G$  with at most  $K$  distinct labels.

The LC-MST problem is a variant of the MLST problem. It has a wide degree of applicability. It is practical because it takes into account both the number of link types and the link costs or lengths. Given an upper bound on the number of link types, the LC-MST problem seeks a least-cost solution.

### 1.4 The Colorful Traveling Salesman Problem (CTSP)

Given a graph  $G = (V, E)$ , a tour or a Hamiltonian cycle of  $G$  is a connected subgraph of  $G$  which constructs a cycle and spans all the nodes of  $G$ . If  $G$  contains  $n$  nodes, then a tour or a Hamiltonian cycle of  $G$  has  $n$  edges. Figure 1.2 shows an example. The highlighted edges construct a Hamiltonian cycle in the graph.

In the Colorful Traveling Salesman Problem (CTSP), we are given an undirected labeled graph  $G = (V, E, L)$ , where  $V, E, L$  are the same as defined in the MLST problem. The purpose of the CTSP is to find a Hamiltonian cycle of  $G$  with the minimum number of distinct labels.

The CTSP is another variant of the MLST problem. It can be applied to a

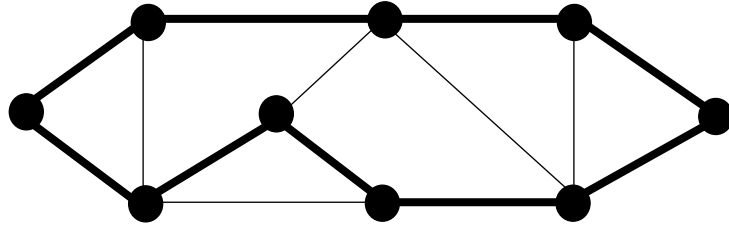


Figure 1.2: A tour or a Hamiltonian cycle in a graph

transportation system. Consider the following hypothetical problem. Suppose we want to visit a set of cities in a tour. Each road is controlled by a transportation company. Each company charges \$D to use all of its roads. The goal is then to choose roads so as to minimize the number of transportation companies that must be paid. Each company can be viewed as a label and the CTSP seeks the tour with the fewest number of labels.

## Chapter 2

### Literature Review

In the minimum labeling spanning tree (MLST) problem, we are given an undirected labeled graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels. The goal is to find a spanning tree of  $G$  such that it contains the smallest number of distinct labels.

The MLST problem was first introduced by Chang and Leu [3] in 1997. They proved that the MLST problem is NP-hard and they proposed two heuristics. Their computational experiments showed that the maximum vertex covering algorithm (MVCA) was a very effective heuristic. This version of MVCA begins with an empty graph. Then it adds labels by spanning as many unvisited nodes as possible until all the nodes are visited. But this version of MVCA has a fatal mistake. Sometimes, it does not yield a connected graph after all nodes are visited and thus fails. In 1998, Krumke and Wirth [11] proposed a modification to MVCA. This is a correct version of MVCA. This version of MVCA begins with an empty graph. It adds labels by reducing the number of connected components by as many as possible until only one connected component is left, i.e., a connected subgraph is obtained. The details of this version of MVCA are described in Appendix A.1. Figure 2.1 shows an input graph and its optimal MLST solution. Figure 2.2 shows how MVCA works on this graph. In the MVCA procedure, we first add label 1. Then we add label 3. At this

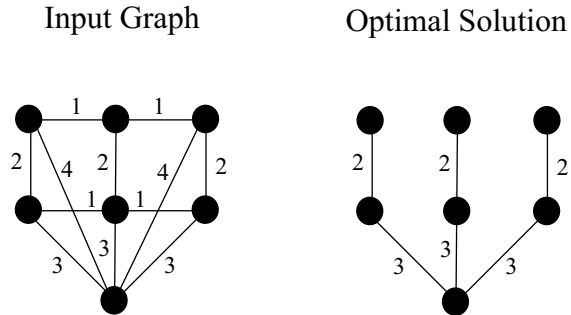


Figure 2.1: A sample graph and its optimal solution for the MLST problem

time, all the nodes of the graph are visited, but the subgraph is still disconnected. Finally, we add label 2 to get only one connected component. The MVCA solution is  $\{1, 2, 3\}$ , which is worse than the optimal solution  $\{2, 3\}$  in Figure 2.1.

Another feature of MVCA heuristic is its worst-case behavior. Krumke and Wirth [11] proved that MVCA can yield a solution no greater than  $2 \ln n + 1$  times optimal, where  $n$  is the number of nodes. In 2002, Wan, Chen, and Xu [14] obtained a better bound. They showed that MVCA can yield a solution no greater than  $1 + \ln(n - 1)$  times optimal. In 2004, we improved the performance guarantee to  $H_b$  for any graph with label frequency bounded by  $b$  (i.e., no label occurs more than  $b$  times in  $G$ ), where  $H_b = \sum_{i=1}^b \frac{1}{i}$  is the  $b$ th harmonic number. Moreover, we presented a worst-case family for which the MVCA solution is exactly  $H_b$  times the optimal solution. Our work finalized the research into the worst-case behavior of the MVCA heuristic. More details are presented in Chapter 3.

In 2003, Brüggemann, Monnot, and Woeginger [2] used a different approach; they applied local search techniques based on the concept of  $j$ -switch neighborhoods to a restricted version of the MLST problem. In addition, they proved a number of



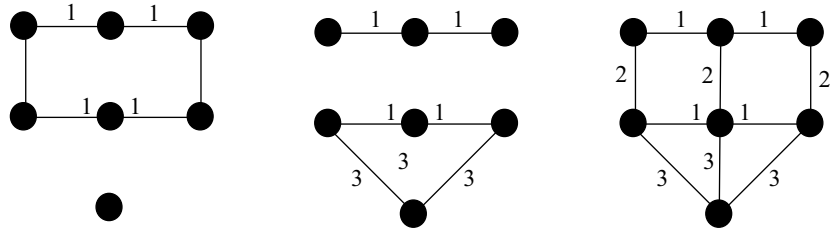


Figure 2.2: How MVCA works

complexity results and showed that if each label appears at most twice in the input graph, the MLST problem is solvable in polynomial time. In 2005, we presented an efficient genetic algorithm (GA) to solve the MLST problem. The GA outperforms MVCA in most cases. More details are presented in Chapter 4.

## Chapter 3

# Worst-case Behavior of the MVCA Heuristic for the Minimum Labeling Spanning Tree Problem

### 3.1 Introduction

In the minimum labeling spanning tree (MLST) problem, we are given a graph with labeled edges as input. We can think of each label as a unique color. The goal is to find a spanning tree with the minimum number of labels. The problem was first introduced by Chang and Leu [3], motivated by applications in communications network design.

The MVCA bounds obtained in [11, 14] are not tight bounds. In fact, these bounds can never be attained. In this chapter, we obtain a better bound, namely  $H_b$ , for any graph with label frequency bounded by  $b$ , where  $H_b = \sum_{i=1}^b \frac{1}{i}$  is the  $b$ th harmonic number. Next, for each  $b$ , we construct a worst-case example. In this example, the MVCA solution is exactly  $H_b$  times the optimal solution. Therefore, we claim that this bound is a tight result. Since  $H_b < 1 + \ln b$  is a well known result and one can always assume that  $b \leq n - 1$  (since otherwise the subgraph induced by the labels of maximum frequency contains a cycle and one can safely remove edges from the cycle without changing the problem), the tight bound  $H_b$  obtained in this paper is, therefore, an improvement on the previously known performance bound of

$1 + \ln(n - 1)$ .

### 3.2 A Perverse-Case Family of Graphs

In this section, we discuss a perverse-case family of graphs. This family of graphs gives the MVCA heuristic some trouble. In Figure 3.1, the graph contains 4 rays and 3 concentric squares. An optimal solution is  $\{1, 2, 3, 4, a\}$  which contains 5 labels; the worst-case MVCA solution is  $\{a, b, c, 1, 2, 3, 4\}$  which contains 7 labels. So the worst-case ratio is  $\frac{7}{5}$ . Generally, for each graph in this family, it has  $n^2$  nodes, where  $n \geq 3$ . It contains  $n - 1$  concentric polygons with  $n$  sides and  $n$  rays. Each polygon is colored by a label and each ray is colored by a label. So there are a total of  $2n - 1$  labels. By observation, we know that an optimal MLST solution for this graph is to select  $n$  ray labels first and one polygon label next. The optimal solution contains  $n + 1$  labels. If we perform the MVCA heuristic, we find that each label can reduce the number of components by the same amount. So we have to select a random one. In the worst case, we might select  $n - 1$  polygon labels first and have to select  $n$  ray labels second, to make the graph connected. Thus, the worst MVCA solution contains all the  $2n - 1$  labels. Therefore, the worst-case ratio for this graph is  $\frac{2n-1}{n+1}$ , which approaches 2 as  $n$  approaches infinity. Figure 3.2 gives another example with  $n = 6$ . The optimal solution is  $\{1, 2, 3, 4, 5, 6, a\}$  with  $n + 1 = 7$  labels. The worst-case MVCA solution contains all the numbered and lettered labels with a total of  $2n - 1 = 11$  labels. Thus the worst-case ratio is  $\frac{11}{6}$ .

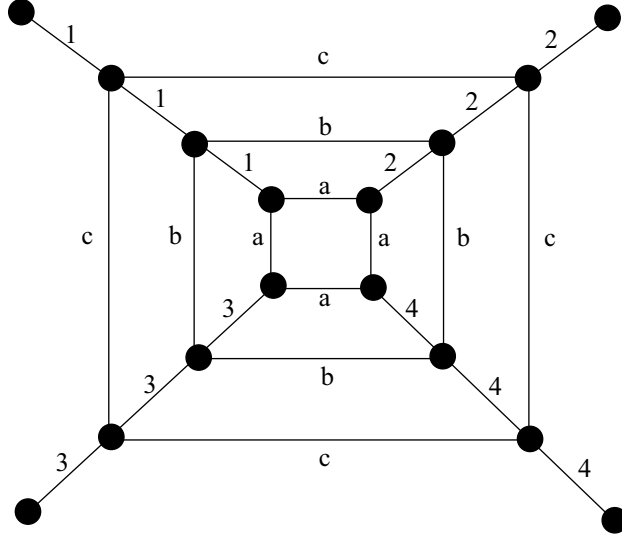


Figure 3.1: Perverse-case graph for  $n = 4$

### 3.3 Worst-case Bound of MVCA

In this section (much of this material has appeared in [15]), we will show that the worst-case bound of MVCA is  $H_b$ , where  $b$  is a bound on the frequency of the labels, and  $H_b = \sum_{i=1}^b \frac{1}{i}$  is the  $b$ th harmonic number. First, we prove a lemma.

**Lemma 1.** *Given  $c_1 > c_2 > \dots > c_k \geq 1$ , let us consider the linear program below:*

$$\text{maximize } F = \sum_{i=1}^k x_i \quad (3.1)$$

$$\text{subject to } \sum_{i=1}^{k-1} c_i x_i \geq S > 0 \quad (3.2)$$

$$\sum_{i=1}^k c_i x_i = T \geq S \quad (3.3)$$

$$\text{where } x_i \geq 0 \text{ for all } i = 1, \dots, k. \quad (3.4)$$

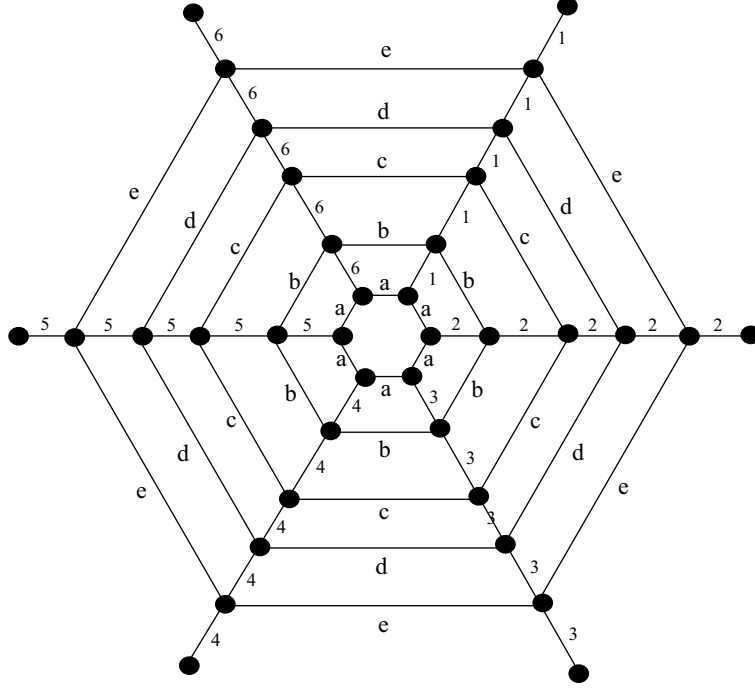


Figure 3.2: Perverse-case graph for  $n = 6$

In any optimal solution, we have

$$\sum_{i=1}^{k-1} c_i x_i = S. \quad (3.5)$$

*Proof.* By contradiction, assume that there is an optimal solution  $x_i$  satisfying  $T \geq \sum_{i=1}^{k-1} c_i x_i > S$ . Then there exists  $x_m > 0$ , where  $1 \leq m \leq k - 1$ . Let  $\varepsilon = \min\{x_m, \frac{\sum_{i=1}^{k-1} c_i x_i - S}{c_m}\} > 0$ . Now, if we decrease  $x_m$  by  $\varepsilon$ , this reduces the left-hand side of (3.2) and (3.3) by  $\varepsilon c_m$ . We can increase  $x_k$  by  $\frac{\varepsilon c_m}{c_k}$  to compensate in (3.3). Since  $c_m > c_k$ ,  $F$  increases by  $\frac{\varepsilon c_m}{c_k} - \varepsilon = \varepsilon(\frac{c_m}{c_k} - 1) > 0$ , which is impossible. Thus,  $\sum_{i=1}^{k-1} c_i x_i = S$  and  $x_k = \frac{T-S}{c_k}$ .  $\square$

In the MLST problem, we are given a graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels. Suppose the label that appears most frequently appears  $b$  times. Let  $OPT$  be an optimal solution with

$|OPT| = v$ . Let the MVCA solution be  $C = \{c_1, \dots, c_t\}$ . Remember we begin with  $n$  components. The addition of each label  $c_i$ , in the sequence determined by the MVCA heuristic, can reduce the number of components by at least one (otherwise the algorithm would not add this label) and by at most  $b$ . Suppose  $C_i \subseteq C$  is the set of all the labels such that at the time when they are added they reduce the number of components by  $i$ ,  $1 \leq i \leq b$ . Then we have  $C = C_1 \cup \dots \cup C_b$ . Let  $|C_i| = x_i$ . This yields  $|C| = x_1 + \dots + x_b$  and

$$bx_b + (b-1)x_{b-1} + \dots + 2x_2 + x_1 = n - 1. \quad (3.6)$$

We will seek to maximize  $|C|$  and compare it to  $v$  in order to determine the worst-case ratio. According to the MVCA steps, we first add all the edges with labels in  $C_b$  to an empty graph. Since  $C_b \cup OPT$  is a feasible solution, for each label  $c \in OPT - C_b$ ,  $c$  can reduce the number of components by at most  $b-1$ . So, all the labels in  $OPT - C_b$  can reduce the number of components by at most  $(b-1)v$ . Since  $C_b \cup OPT$  contains more than enough labels to reduce the number of components to one, we have  $bx_b + (b-1)v \geq n - 1$ . This implies  $(n-1) - (b-1)v \leq bx_b \leq n - 1$ .

Second, we add all the edges with labels in  $C_{b-1}$  to the subgraph that consists of edges with labels from  $C_b$ . Since  $C_b \cup C_{b-1} \cup OPT$  is a feasible solution, for each label  $c \in OPT - (C_b \cup C_{b-1})$ ,  $c$  can reduce the number of components by at most  $b-2$ . We, therefore, have  $bx_b + (b-1)x_{b-1} + (b-2)v \geq n - 1$ . This implies  $(n-1) - (b-2)v \leq bx_b + (b-1)x_{b-1} \leq n - 1$ .

Similarly, after we add all the edges with labels in  $C_{b-i}$ , we will obtain  $(n-1) - (b-i-1)v \leq bx_b + \dots + (b-i)x_{b-i} \leq n - 1$ . Finally, after we add all the edges

with labels in  $C_2$ , we will get  $(n-1) - v \leq bx_b + \dots + 2x_2 \leq n-1$ . The above logic generates the following constraints:

$$(n-1) - (b-1)v \leq bx_b \leq n-1$$

$$(n-1) - (b-2)v \leq bx_b + (b-1)x_{b-1} \leq n-1$$

$\vdots$

$$(n-1) - v \leq bx_b + \dots + 2x_2 \leq n-1.$$

Since  $b=1$  is a trivial case and in this case we have  $|C| = |OPT| = v = n-1$ , we can always assume  $b \geq 2$ . Then, we get  $(n-1) - v > 0$  and  $(n-1) - bv \leq 0$ . Thus, there exists an  $i \in \{1, \dots, b-1\}$  such that  $(n-1) - (b-i)v > 0$  but  $(n-1) - (b-i+1)v \leq 0$  or  $(n-1) - (b-i)v \leq v$ . We define  $F_b = x_b$ ,  $F_{b-1} = x_b + x_{b-1}$ ,  $\dots$ ,  $F_2 = x_b + \dots + x_2$ ,  $F_1 = x_b + \dots + x_1$ .

By Lemma 1, the optimal solution to the linear program

$$\text{maximize } F_1$$

$$\text{subject to } bx_b + \dots + x_1 = n-1 \tag{3.7}$$

$$bx_b + \dots + 2x_2 \geq (n-1) - v \tag{3.8}$$

$$\text{where } x_i \geq 0 \text{ for all } i = 1, \dots, b,$$

has the left-hand side equal to the right-hand side in (3.8). This yields  $x_1 = v$  from

(3.7). Similarly, the optimal solution to the linear program

$$\text{maximize } F_2$$

$$\text{subject to } bx_b + \cdots + 2x_2 = (n-1) - v \quad (3.9)$$

$$bx_b + \cdots + 3x_3 \geq (n-1) - 2v \quad (3.10)$$

where  $x_i \geq 0$  for all  $i = 1, \dots, b$ ,

has the left-hand side equal to the right-hand side in (3.10). This yields  $x_2 = \frac{v}{2}$  from (3.9).

We continue in this way until we obtain  $x_{b-i} = \frac{v}{b-i}$ . So we get

$$x_1 = v, x_2 = \frac{v}{2}, \dots, x_{b-i} = \frac{v}{b-i}. \quad (3.11)$$

From (3.6), we have  $bx_b + \cdots + x_1 = n-1$ . Substituting from (3.11) for  $x_1, x_2, \dots, x_{b-i}$ , we obtain

$$bx_b + \cdots + (b-i+1)x_{b-i+1} + v + \cdots + v = n-1 \text{ or}$$

$$bx_b + \cdots + (b-i+1)x_{b-i+1} + (b-i)v = n-1.$$

This implies

$$bx_b + \cdots + (b-i+1)x_{b-i+1} = (n-1) - (b-i)v \leq v.$$

Therefore, we have  $jx_j \leq v$  or  $x_j \leq \frac{v}{j}$  for all  $b-i+1 \leq j \leq b$ . This gives  $F_1 \leq v + \frac{v}{2} + \cdots + \frac{v}{b} = H_b v < v(1 + \ln b)$ , where  $H_b = \sum_{i=1}^b \frac{1}{i}$  is the  $b$ th harmonic number. Therefore, the worst-case ratio of MVCA is  $r = \frac{|C|}{v} = \frac{F_1}{v} \leq H_b$  and the following theorem results.

**Theorem 1.** *Given a graph  $G = (V, E, L)$  and suppose the label that appears most frequently appears  $b$  times. Then, the worst-case ratio of MVCA is  $H_b$ .*



### 3.4 A Worst-case Family of Graphs

In this section, we build a worst-case family of graphs such that MVCA attains its worst-case ratio of  $H_b$ . Given  $b$ ,  $b \geq 2$ , the maximum frequency of labels, we construct the graph as follows. Let  $|V| = n = b \cdot b! + 1$ . Then  $b \cdot b! = n - 1$  and  $n - 1$  is divisible by  $k$  for  $1 \leq k \leq b$ . With nodes  $V = \{1, 2, \dots, n\}$ , we divide  $V$  into  $b!$  non-disjunctive groups, where each group contains  $b + 1$  nodes. These groups of nodes are

$$\begin{aligned}
 V_1 &= \{1, 2, \dots, b + 1\} \\
 V_2 &= \{b + 1, b + 2, \dots, 2b + 1\} \\
 &\vdots \\
 V_i &= \{(i - 1)b + 1, (i - 1)b + 2, \dots, ib + 1\} \\
 &\vdots \\
 V_{b!} &= \{(b! - 1)b + 1, (b! - 1)b + 2, \dots, b! \cdot b + 1 = n\}.
 \end{aligned}$$

In each  $V_i$ , we label the edges of consecutive nodes  $((i - 1)b + 1, (i - 1)b + 2), \dots, (ib, ib + 1)$  with one label. This yields  $b!$  labels. These labels constitute an optimal solution  $OPT$ .

Next, we build label sets  $L_b, L_{b-1}, \dots, L_2$ . Each label in  $L_j$  can reduce the number of components by  $j$ . To build  $L_k$  (for  $k \in \{2, \dots, b\}$ ), we select the edge  $((i - 1)b + 1, (i - 1)b + 1 + k)$  in each  $V_i$ . There are  $b!$  such edges. The first  $k$  of these receive one label. The next  $k$  receive a second label, and so on. We need  $\frac{b!}{k}$  labels. So  $|L_k| = \frac{b!}{k}$ . The worst-case graph is now complete. An example

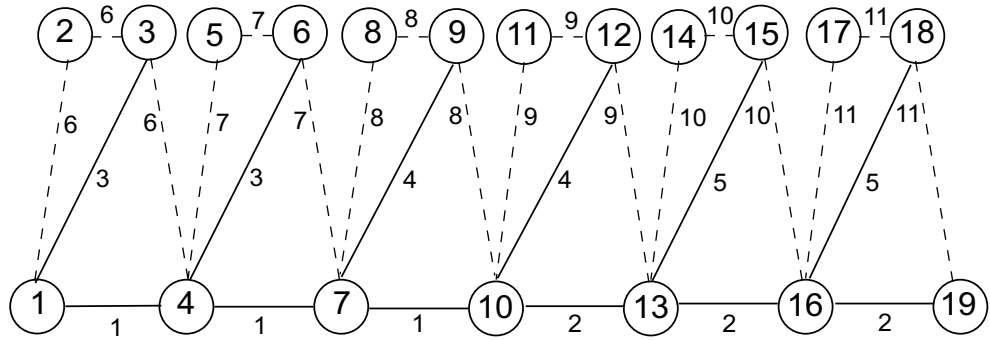


Figure 3.3: Worst-case graph for  $b = 3$

for  $b = 3$  is shown in Figure 3.3, where the nodes are numbered and the edge numbers represent labels. In this figure, we have  $6 = 3!$  groups:  $V_1 = \{1, 2, 3, 4\}$ ,  $V_2 = \{4, 5, 6, 7\}$ ,  $V_3 = \{7, 8, 9, 10\}$ ,  $V_4 = \{10, 11, 12, 13\}$ ,  $V_5 = \{13, 14, 15, 16\}$ , and  $V_6 = \{16, 17, 18, 19\}$ . The optimal solution is  $\{6, 7, 8, 9, 10, 11\}$ , indicated by the dashed edges in Figure 3.3; the MVCA solution contains all 11 labels.

The MVCA heuristic has us first select labels that reduce the number of components by  $b$ . Labels in  $L_b \cup OPT$  are ready for use. We may select all the labels in  $L_b$ . This selection causes each label in  $OPT$  to reduce the number of components by  $b - 1$ . Next, we select labels each of which reduces the number of components by  $b - 1$ . Labels in  $L_{b-1} \cup OPT$  are ready for use. We may select all the labels in  $L_{b-1}$ . This selection causes each label in  $OPT$  to reduce the number of components by  $b - 2$ . We continue this procedure and select labels in  $L_k$  ( $2 \leq k \leq b$ ) at each step. Finally, we must select all the labels in  $OPT$ . Thus, the MVCA solution is

given by

$$MVCA = L_b \cup L_{b-1} \cup \dots \cup L_2 \cup OPT.$$

Thus, we have

$$\begin{aligned} |MVCA| &= \frac{b!}{b} + \frac{b!}{b-1} + \dots + \frac{b!}{2} + b! \\ &= \left(1 + \frac{1}{2} + \dots + \frac{1}{b}\right) \cdot b! \\ &= H_b \cdot |OPT|. \end{aligned}$$

For the example in Figure 3.3,  $b = 3$ ,  $n = 19$ , and the worst-case ratio is  $\frac{|MVCA|}{|OPT|} = 1 + \frac{1}{2} + \frac{1}{3} = H_3$ . Figure 3.4 shows the worst-case example for  $b = 4$ . In this example,  $n = 4 \cdot 4! + 1 = 97$ , and the worst-case ratio is  $\frac{|MVCA|}{|OPT|} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = H_4$ .

### 3.5 Concluding Remarks

Given a labeled graph  $G$ , a label  $c$  is a *cut label* if the removal of all the edges with label  $c$  disconnects the graph. From Figure 3.3, we observe that labels 6, 7, 8, 9, 10, and 11 are cut labels. A smart implementation of MVCA would add the cut labels first, since these labels must be in any solution and they are easy to identify. Such an implementation would find the optimal solution in Figure 3.3. But, if we add edges  $(2, 4)$ ,  $(5, 7)$ ,  $(8, 10)$ ,  $(11, 13)$ ,  $(14, 16)$ , and  $(17, 19)$  with labels  $a, b, c, d, e$ , and  $f$ , respectively, then there are no cut labels in the new graph. The worst-case bound is the same as before.

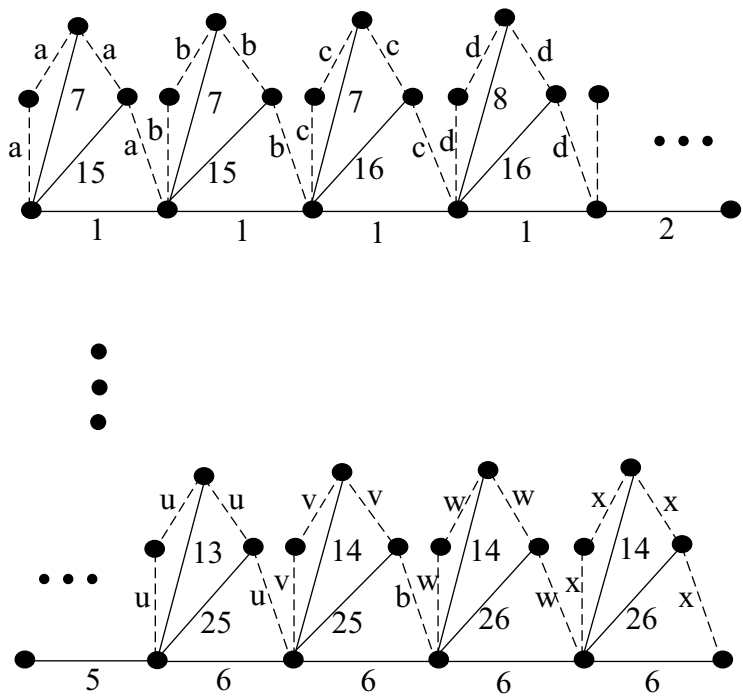


Figure 3.4: Worst-case graph for  $b = 4$

## Chapter 4

# A One-Parameter Genetic Algorithm for the Minimum Labeling Spanning Tree Problem

### 4.1 Introduction

In the minimum labeling spanning tree (MLST) problem, we are given an undirected graph with labeled edges as input. Each edge has a single label and different edges can have the same label. We can think of each label as a unique color. The goal is to find a spanning tree with the minimum number of labels. The problem was first introduced by Chang and Leu [3], motivated by applications in communications network design. In this chapter (much of this material has appeared in [16]), we describe an effective genetic algorithm (GA) to address the MLST problem.

Since the MVCA heuristic is the most popular and best studied MLST heuristic in the literature, we will use it as a benchmark. MVCA begins with the nodes of  $G$  and an empty set of edges as an initial graph. At each iteration, one label is selected and the edges with this label are added to the graph. The goal is to reduce the number of connected components by as many as possible. The procedure continues until the graph is connected. A more detailed description is provided in Chapter 3.

The MVCA heuristic has to check all the labels before it selects a specific label

which reduces the number of components by the largest amount. It makes sense, but it takes time. MVCA can, and often does, select unnecessary labels. The local-1 search (due to Brüggenmann, Monnot, and Woeginger [2]) can remove unnecessary labels one by one, but it does not guarantee an optimal solution. In our genetic algorithm, we apply ideas from MVCA in the crossover operation, without checking all the labels at each step. We apply local-1 search in the mutation operation in order to remove unnecessary labels. Based upon our computational results, the genetic algorithm beats MVCA in most cases.

In recent years, a wide variety of network design problems has been addressed using genetic algorithms. For example, see Chou, Premkumar, and Chu [4], Palmer and Kershbaum [12], and Golden, Raghavan, and Stanojević [9].

## 4.2 Genetic Algorithm

During the last three decades, there has been a growing interest in algorithms that rely on analogies to natural processes. The emergence of massively parallel computers, and faster computers in general, has made these algorithms of practical interest. The genetic algorithm (GA) is one of the best known algorithms in this class. It is based on the principle of evolution, operations such as crossover and mutation, and the concept of fitness. In the MLST problem, fitness is the number of distinct labels in the candidate solution. After some number of generations, the algorithm converges and the best individual, we hope, represents a near-optimal solution.

In the MLST problem, we are given a graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels. Let  $|V| = n$ ,  $|E| = m$ , and  $|L| = \ell$ . An individual (or a chromosome) in a population is a feasible solution, which is defined as a subset  $C$  of  $L$  such that all the edges with labels in  $C$  construct a connected subgraph of  $G$  and span all the nodes in  $G$ . Each label in  $C$  can be viewed as a gene. A random individual can be generated by adding random labels to an empty set until a feasible solution emerges. So, it is not difficult to build a population of individuals.

After we get the initial population, we can apply crossover and mutation operations in order to build one generation from the last. The details of the GA are discussed next.

#### 4.2.1 Encoding

Ultimately, a solution to the MLST problem is a spanning tree. However, it is somewhat inconvenient to encode a spanning tree and much easier to think in terms of a feasible solution (as defined above). We, therefore, encode a feasible solution (i.e., a list of labels). Two simple observations follow.

Observation 1. If  $C$  is a feasible solution and  $G_C$  is the subgraph induced by  $C$ , then any spanning tree of  $G_C$  has at most  $|C|$  labels.

Observation 2. If  $C$  is an optimal solution and  $G_C$  is the subgraph induced by  $C$ , then any spanning tree of  $G_C$  is a minimum labeling spanning tree of  $G$ .

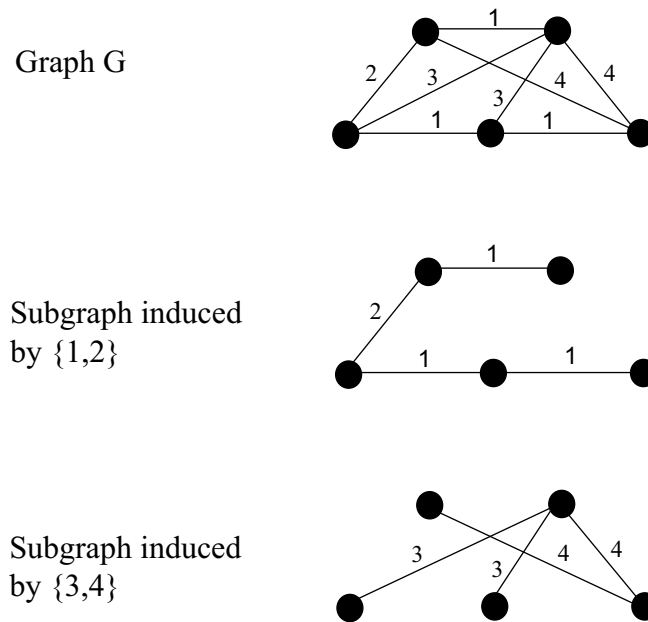


Figure 4.1: Encoding feasible solutions:  $\{1, 2\}$  and  $\{3, 4\}$  are two feasible solutions

From the above observations, to solve the MLST problem, we seek a feasible solution with the least number of labels. Then, we want an arbitrary spanning tree from the subgraph induced by this feasible solution. Thus, encoding a feasible solution enables us to solve the MLST problem. An illustration of encoding is provided in Figure 4.1.

#### 4.2.2 Crossover

Crossover builds one offspring from two parents. It begins by forming the union of the two parents, then sorts the labels in the union in descending order of their frequencies. The operator adds labels in their sorted order to the initially empty offspring until the offspring represents a feasible solution. The following sketch summarizes the operator. Figure 4.2 presents an example of its application.



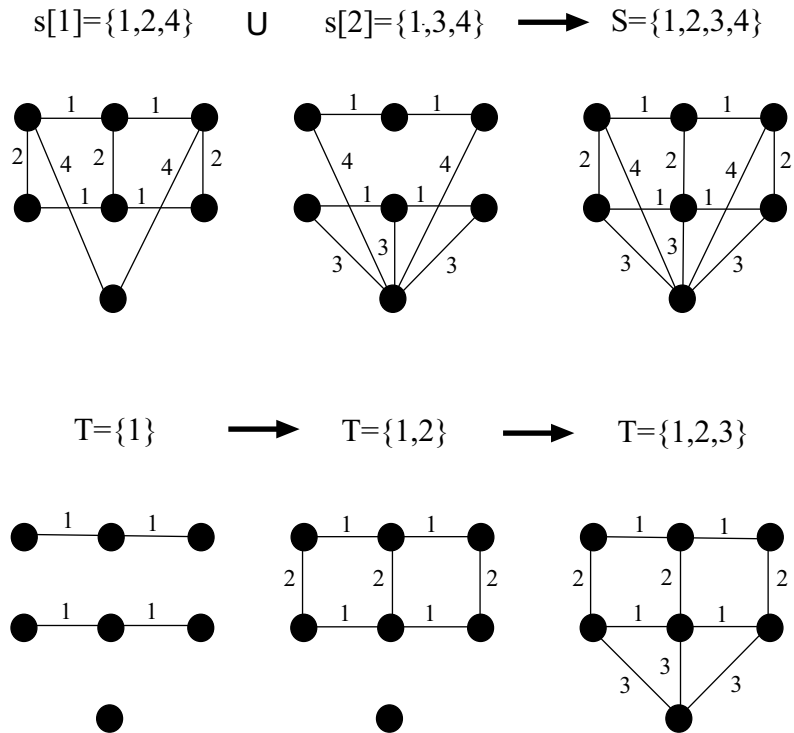


Figure 4.2: An example of crossover

### Crossover( $s[1],s[2]$ )

1. Let  $S = s[1] \cup s[2]$  and  $T$  be an empty set.
2. Sort  $S$  in decreasing order of the frequency of labels in  $G$ .
3. Add labels of  $S$ , from the first to the last, to  $T$  until  $T$  represents a feasible solution.
4. Output  $T$ .

### 4.2.3 Mutation

Given a solution  $S$ , a new solution  $T = \text{mutation}(S)$  can be built using our mutation operation. First, a new label is added to  $S$ . Next, labels are removed (i.e., the associated edges), from the least frequently occurring label to the most, as long as  $S$  remains feasible. A more detailed description of the mutation operation follows. An example is presented in Figure 4.3.

In Figure 4.3, after adding a label, we remove label 4 first. Next, we try to remove label 3, but cannot without violating feasibility. Likewise, label 2 cannot be removed. Finally, label 1 is removed to obtain the mutation  $T$ .

#### **Mutation( $S$ )**

1. Randomly select  $c$  not in  $S$  and let  $T = S \cup c$ .
2. Sort  $T$  in decreasing order of the frequency of labels in  $G$ .
3. From the last label of the above list to the first, try to remove one label from  $T$  and keep  $T$  as a feasible solution.
4. Repeat 3 until no labels can be removed.
5. Output  $T$ .

### 4.2.4 Building each generation

In our genetic algorithm, we use only one parameter: the population size. In each generation, the probability of crossover is 100% and the probability of mutation

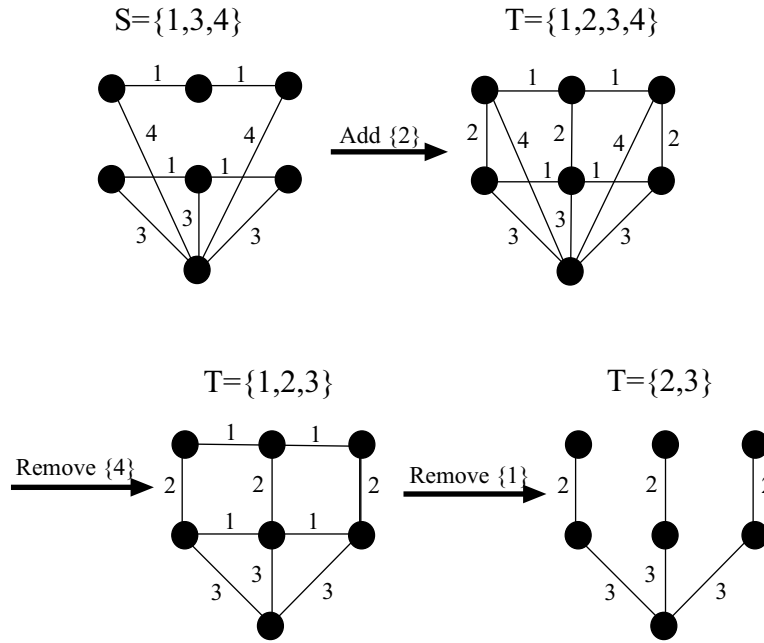


Figure 4.3: An example of mutation

is 100%. There is no biased random selection from one generation to the next and there is no fine-tuning of numerous parameters required in order to determine an effective combination of parameter values. The genetic algorithm is designed to be simple and easy to replicate.

In many applications of metaheuristics (e.g., tabu search, genetic algorithms, ant colony optimization, etc.) to combinatorial optimization, performance is highly dependent on and sensitive to the parameter values. If a metaheuristic performs well with only a single parameter, then, in our view, the procedure is especially well-suited to solve the problem under study.

Suppose the initial generation consists of  $p$  individuals. These individuals are denoted by  $s[0], s[1], \dots, s[p-1]$ . We can build generation  $k$  from generation  $k-1$  as follows, where  $1 \leq k < p$ . In all, there are  $p$  generations (i.e.,  $0, 1, \dots, p-1$ ). Note

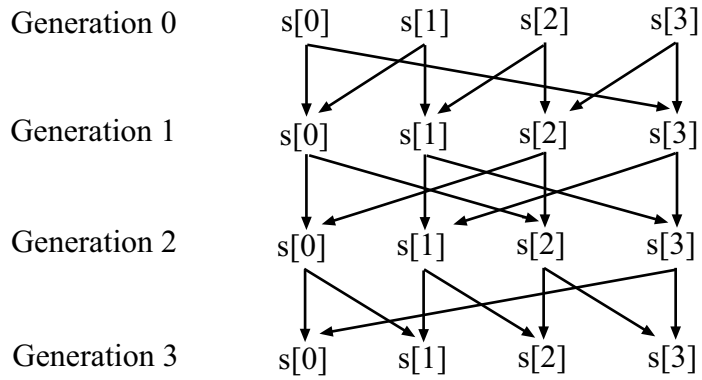


Figure 4.4: Build generations with  $p=4$

that the number of generations is equal to the initial population. An example of generation building (with  $p=4$ ) is shown in Figure 4.4.

1. **For** each  $j$  from 0 to  $p-1$ , do
2.  $t[j] = \text{crossover}(s[j], s[(j+k) \bmod p])$
3.  $t[j] = \text{mutation}(t[j])$
4.  $s[j] = \text{the better solution of } s[j] \text{ and } t[j]$
5. **End For**

#### 4.2.5 Running time analysis

Let us begin with an analysis of the crossover operation. Given a subset  $C$  of  $L$ , we can use depth-first search (DFS) to determine whether the subgraph of  $G$  restricted to  $V$  and edges with labels from  $C$  is connected. The running time of DFS is  $O(m + n)$ . In each crossover operation, a feasible solution has  $O(\ell)$  labels. Merge sort is used to combine two feasible solutions and this requires  $O(\ell)$  computations.

In Step 3, we can add at most  $O(\ell)$  labels and we use DFS after each addition to determine whether a feasible solution has been obtained. Thus, the worst-case running time of a crossover operation is  $O(\ell(m+n))$ .

Similarly, the worst-case running time of a mutation operation is  $O(\ell(m+n))$ . Each generation requires  $p$  crossovers and  $p$  mutations, and there are  $p$  generations in all. Therefore, the worst-case running time of GA is  $O(p^2\ell(m+n))$ . A running time analysis of MVCA is provided by Krumke and Wirth [11].

### 4.3 Computational Results

The one-parameter GA and the MVCA heuristic were compared on 78 cases of the MLST problem. This section describes these comparisons. Throughout,  $d = \frac{m}{\binom{n}{2}} = \frac{2m}{n(n-1)}$  is the density of the graph.

The frequency of labels plays an important role in the GA. But for a label  $c$ , we may have two different definitions of its frequency. One definition is the number of appearances of  $c$  in  $G$ . This is called the general frequency. The other definition is the number of appearances in the subgraph induced by  $c$  after removing cycle edges. This is called the net frequency. An example is shown in Figure 4.5. So, we have two GAs to report on. GA1 is described in Section 4.2. GA1 represents the GA using general frequency and GA2 is a minor variant which uses the concept of net frequency. In other words, the concept of net frequency is based on the fact that a minimum labeling spanning tree will contain no cycles and GA2 tries to take advantage of this fact.

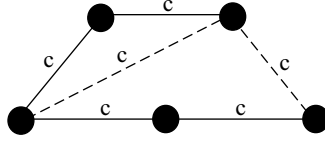


Figure 4.5: The general frequency of label  $c$  is 6. There are two cycle edges (dashed lines). So, the net frequency of label  $c$  is 4.

We use OPT to represent the optimal MLST solution. We obtain the optimal solution via backtrack search. Given  $0 < k \leq \ell$ , backtrack search tries all possible subsets of  $L$  with  $k$  labels and determines whether a feasible solution exists. We can sort all the labels by their frequencies. If the sum of the frequencies of the  $k$  labels is less than  $n - 1$ , we have an infeasible solution. We increase  $k$  until a feasible solution is found or a running time limit is exceeded. See A.6.1 for details.

The running time of backtrack search grows exponentially, but if the problem size is small or the optimal solution is small, the running time is reasonable. In our experiments, the optimal solution is reported unless a single instance requires more than 20 minutes of CPU time. In such a case, we report NF (Not Found).

We use  $n, \ell, d$  as input. For each input combination, we construct 20 random graphs and we apply MVCA, GA1, GA2, and OPT to each one. Each procedure is run on each instance once only. The output is the average number of labels for each solution procedure. In the GAs, the population size is set to 20 if  $n, \ell \leq 100$  and 30 if  $n > 100$ . We divide all the computational tests into three groups as described below.

	OPT	GA1	GA2	MVCA
$n = \ell = 20, d = 0.8$	2.40	2.40	2.40	2.50
$n = \ell = 20, d = 0.5$	3.20	3.25	3.25	3.40
$n = \ell = 20, d = 0.2$	6.85	6.85	6.85	7.00
$n = \ell = 30, d = 0.8$	2.45	2.45	2.45	2.65
$n = \ell = 30, d = 0.5$	3.70	3.70	3.70	3.80
$n = \ell = 30, d = 0.2$	7.25	7.30	7.30	7.70
$n = \ell = 40, d = 0.8$	2.95	2.95	2.95	2.95
$n = \ell = 40, d = 0.5$	3.85	3.95	3.95	3.95
$n = \ell = 40, d = 0.2$	7.20	7.20	7.25	7.85
$n = \ell = 50, d = 0.8$	3.00	3.00	3.00	3.00
$n = \ell = 50, d = 0.5$	4.00	4.05	4.05	4.20
$n = \ell = 50, d = 0.2$	7.45	7.65	7.65	8.25

Table 4.1: Computational results for Group I

#### 4.3.1 Group I

Group I consists of small graphs, with  $n = \ell \leq 50$ . There are three density levels: high( $d = 0.8$ ), medium( $d = 0.5$ ), and low ( $d = 0.2$ ). We can generate optimal solutions here without difficulty. The average number of labels is reported for 12 input combinations (or cases) in Table 4.1. GA1 beats MVCA in 9 of these cases and ties MVCA in 3 cases. GA1 beats GA2 in one case and ties GA2 in 11 cases.

### 4.3.2 Group II

Group II consists of a specially constructed family of graphs. In addition, the graphs are larger than the Group I graphs. For each member in this family, each label appears exactly  $b$  times and we know an optimal solution has  $\lceil \frac{n-1}{b} \rceil$  labels (see Appendix A for details). This family is constructed in order to challenge the label selection strategy of MVCA. At each step, many labels are available for selection since each one reduces the number of components equally. Thus, MVCA has a greater chance of selecting the labels incorrectly. The results for 21 input combinations of  $n$ ,  $\ell$ , and  $b$  are presented in Table 4.2. The optimal solution value is also included. GA1 beats MVCA in all of these cases (often by a wide margin). GA1 beats GA2 in 2 cases and GA2 beats GA1 in 2 cases. There are 17 ties. Therefore, both GAs clearly outperform MVCA for Group II graphs.

### 4.3.3 Group III

Group III consists of graphs with a range of sizes and different values of  $\ell$  for each  $n$ . In particular,  $\ell$  is set to  $0.25n$ ,  $0.5n$ ,  $n$ , and  $1.25n$  (approximately). In Tables 4.3 through 4.6, results from 48 input combinations (cases) are presented. GA1 beats MVCA in 30 cases. MVCA beats GA1 in 3 cases and there are 15 ties. GA1 beats GA2 in 4 cases and GA2 beats GA1 in 5 cases. There are 39 ties.

To build a Group I or Group III graph, we are given  $n$ ,  $\ell$ , and  $d$ . So, the number of edges should be  $m = \frac{dn(n-1)}{2}$ . We begin with a graph that consists of the  $n$  nodes only. Next, we randomly generate  $m$  distinct edges. For each edge  $e$ ,



we label it with a random label from  $L$ . We check to see if the graph is connected. If yes, we stop. If not, we continue to generate random edges with random labels and add them until the graph is connected. Thus, the density of the graph might actually be a little higher than  $d$ . This is most likely to happen when  $n < 10$  and  $d = 0.2$  and is unlikely to happen when  $n \geq 20$ .

#### 4.3.4 Summary

In total, we tested 78 input combinations (there are 3 duplicated combinations when  $n = \ell = 50$  in Table 4.1 and Table 4.3). GA1 beats MVCA in 58 cases. MVCA beats GA1 in 3 cases and there are 17 ties. GA2 beats MVCA in 57 cases. MVCA beats GA2 in 3 cases and there are 18 ties. In Table 1 and Tables 3 to 6, we applied backtrack search successfully in 740 instances (there are 60 duplicated instances when  $n = \ell = 50$  in these tables). GA1 obtains an optimal solution in 701 or 94.73% of the instances. GA2 obtains an optimal solution in 704 or 95.14% of the instances. Although one might have predicted superior performance from GA2, the computational results indicate otherwise. In fact, GA1 and GA2 are essentially the same in quality of performance, but GA2 is slightly more time consuming. GA2 requires  $O(\ell(m + n))$  additional computations to find the net frequency for each label. From the point of view of running time, GA1 is slightly slower than MVCA. This is not a major issue, however, since the longest running time for GA1 to solve a single MLST problem instance is under 7 seconds of CPU time on a Pentium 4 PC with 1.80 GHz and 256 MB RAM, while GA2 takes at most 8 seconds. Thus, the GAs are very fast, they contain a single parameter, and they typically outperform

MVCA. Based on these results, we recommend GA1 as a fast and effective solution procedure for the MLST problem. We point out that if  $p = 20$  for all  $n$  and  $\ell$ , then GA1 and GA2 perform only slightly worse for  $n > 100$ . Similarly, if  $p = 30$  for all  $n$  and  $\ell$ , GA1 and GA2 perform slightly better for  $n, \ell \leq 100$ .

As mentioned, in 20 of the 78 cases, MVCA ties or beats GA1. We ran one final experiment in which we included the MVCA solution in the initial population and ran GA1. In five of the cases, a slight improvement resulted.

#### 4.4 Concluding Remarks

In this chapter, we present a one-parameter genetic algorithm for the minimum labeling spanning tree problem which is an NP-hard problem. The GA is simple, fast, and effective. We compare the GA with the MVCA, the most popular MLST heuristic in the literature. Based on extensive computational tests, the GA clearly outperforms MVCA. This is a nice example of the ability of GAs to successfully solve difficult NP-hard problems. A final point of interest is as follows: for the worst-case family of graphs (presented by Xiong, Golden, and Wasil [15]), mentioned in Section 4.1, MVCA can perform poorly, but GA1 obtains the optimal solution in each case.

	Optimal	GA1	GA2	MVCA
$n = \ell = 50, b = 4$	13	13.00	13.00	14.25
$n = \ell = 50, b = 5$	10	10.00	10.00	11.55
$n = \ell = 50, b = 10$	5	5.90	5.90	6.30
$n = \ell = 50, b = 15$	4	4.00	4.00	4.65
$n = \ell = 50, b = 20$	3	3.00	3.00	3.60
$n = \ell = 75, b = 4$	19	19.00	19.00	21.30
$n = \ell = 75, b = 7$	11	11.40	11.40	12.90
$n = \ell = 75, b = 15$	5	6.00	6.00	6.65
$n = \ell = 75, b = 25$	3	4.00	4.00	4.50
$n = \ell = 100, b = 4$	25	25.20	25.20	28.95
$n = \ell = 100, b = 10$	10	11.30	11.25	12.60
$n = \ell = 100, b = 20$	5	6.75	6.75	7.00
$n = \ell = 100, b = 30$	4	5.00	4.95	5.05
$n = \ell = 150, b = 4$	38	38.00	38.00	42.65
$n = \ell = 150, b = 15$	10	12.10	12.15	13.10
$n = \ell = 150, b = 30$	5	7.00	7.00	7.30
$n = \ell = 150, b = 45$	4	5.00	5.00	5.10
$n = \ell = 200, b = 4$	50	50.95	50.95	57.60
$n = \ell = 200, b = 20$	10	13.10	13.10	13.75
$n = \ell = 200, b = 40$	5	7.45	7.50	7.90
$n = \ell = 200, b = 60$	4	5.40	5.45	5.55

Table 4.2: Computational results for Group II

	OPT	GA1	GA2	MVCA
$n = 50, \ell = 12, d = 0.8$	1.10	1.10	1.10	1.10
$n = 50, \ell = 12, d = 0.5$	1.95	1.95	1.95	1.95
$n = 50, \ell = 12, d = 0.2$	3.20	3.30	3.30	3.45
$n = 50, \ell = 25, d = 0.8$	2.00	2.00	2.00	2.00
$n = 50, \ell = 25, d = 0.5$	2.95	2.95	2.95	2.95
$n = 50, \ell = 25, d = 0.2$	5.00	5.10	5.10	5.45
$n = 50, \ell = 50, d = 0.8$	3.00	3.00	3.00	3.00
$n = 50, \ell = 50, d = 0.5$	4.00	4.05	4.05	4.20
$n = 50, \ell = 50, d = 0.2$	7.45	7.65	7.65	8.25
$n = 50, \ell = 62, d = 0.8$	3.00	3.10	3.10	3.40
$n = 50, \ell = 62, d = 0.5$	4.15	4.45	4.50	4.75
$n = 50, \ell = 62, d = 0.2$	NF	8.55	8.55	9.20

Table 4.3: Computational results for Group III with  $n = 50$

	OPT	GA1	GA2	MVCA
$n = 100, \ell = 25, d = 0.8$	1.45	1.50	1.45	1.45
$n = 100, \ell = 25, d = 0.5$	2.00	2.00	2.00	2.00
$n = 100, \ell = 25, d = 0.2$	3.95	3.95	3.95	4.00
$n = 100, \ell = 50, d = 0.8$	2.00	2.00	2.00	2.00
$n = 100, \ell = 50, d = 0.5$	3.00	3.00	3.00	3.05
$n = 100, \ell = 50, d = 0.2$	NF	5.95	5.95	6.15
$n = 100, \ell = 100, d = 0.8$	3.00	3.20	3.15	3.60
$n = 100, \ell = 100, d = 0.5$	NF	4.95	4.95	4.90
$n = 100, \ell = 100, d = 0.2$	NF	8.90	8.90	9.40
$n = 100, \ell = 125, d = 0.8$	3.95	4.00	4.00	4.05
$n = 100, \ell = 125, d = 0.5$	NF	5.40	5.40	5.65
$n = 100, \ell = 125, d = 0.2$	NF	10.15	10.15	10.95

Table 4.4: Computational results for Group III with  $n = 100$

	OPT	GA1	GA2	MVCA
$n = 150, \ell = 37, d = 0.8$	1.85	1.85	1.85	1.85
$n = 150, \ell = 37, d = 0.5$	2.00	2.00	2.00	2.00
$n = 150, \ell = 37, d = 0.2$	4.00	4.00	4.00	4.10
$n = 150, \ell = 75, d = 0.8$	2.00	2.10	2.10	2.30
$n = 150, \ell = 75, d = 0.5$	3.00	3.05	3.00	3.35
$n = 150, \ell = 75, d = 0.2$	NF	6.30	6.30	6.60
$n = 150, \ell = 150, d = 0.8$	NF	3.95	3.95	3.95
$n = 150, \ell = 150, d = 0.5$	NF	5.05	5.05	5.10
$n = 150, \ell = 150, d = 0.2$	NF	9.80	9.80	10.35
$n = 150, \ell = 187, d = 0.8$	NF	4.05	4.10	4.35
$n = 150, \ell = 187, d = 0.5$	NF	6.00	6.00	6.00
$n = 150, \ell = 187, d = 0.2$	NF	11.45	11.45	11.85

Table 4.5: Computational results for Group III with  $n = 150$

	OPT	GA1	GA2	MVCA
$n = 200, \ell = 50, d = 0.8$	2.00	2.00	2.00	2.00
$n = 200, \ell = 50, d = 0.5$	2.00	2.00	2.00	2.00
$n = 200, \ell = 50, d = 0.2$	4.00	4.00	4.00	4.30
$n = 200, \ell = 100, d = 0.8$	2.20	2.35	2.25	2.65
$n = 200, \ell = 100, d = 0.5$	3.00	3.30	3.30	3.75
$n = 200, \ell = 100, d = 0.2$	NF	6.80	6.80	6.90
$n = 200, \ell = 200, d = 0.8$	NF	4.00	4.00	4.00
$n = 200, \ell = 200, d = 0.5$	NF	5.75	5.85	5.80
$n = 200, \ell = 200, d = 0.2$	NF	10.85	10.85	10.90
$n = 200, \ell = 250, d = 0.8$	NF	4.80	4.85	4.75
$n = 200, \ell = 250, d = 0.5$	NF	6.50	6.50	6.50
$n = 200, \ell = 250, d = 0.2$	NF	12.25	12.15	12.55

Table 4.6: Computational results for Group III with  $n = 200$

## Chapter 5

### Improved Heuristics for the Minimum Labeling Spanning Tree

#### Problem

#### 5.1 Introduction

In Chapter 3, we proved that MVCA can yield a solution no greater than  $H_b$  times optimal for any graph with label frequency bounded by  $b$  (i.e., no label occurs more than  $b$  times in  $G$ ), where  $H_b = \sum_{i=1}^b \frac{1}{i}$  is the  $b^{\text{th}}$  harmonic number. In Chapter 4, we presented an efficient genetic algorithm (GA) to solve the MLST problem. The GA outperforms MVCA in most cases.

Subsequently, Voss et al. [13] applied their *pilot* method (originally developed by Duin and Voss [5]) to the MLST problem. Their pilot method is a greedy heuristic with a limited look-ahead capability. It generates high-quality solutions to the MLST problem, but running times are much longer than those of GA.

In this chapter, we implement the pilot method of Voss et al. [13], along with faster “short-cut” versions. We also present a modified version of GA. This modified GA is competitive with the best of the pilot methods in terms of solution quality and nearly comparable to the fastest of the pilot methods in terms of running time.



## 5.2 Three Modified Versions of MVCA

The MVCA heuristic is a greedy algorithm. It begins with an empty label set  $C$  and a subgraph consisting of the  $n$  nodes of  $G$  (with no edges). It adds labels to  $C$ , one by one (as well as the associated edges) until the subgraph induced by  $C$  is connected. At each step, the label added is the one that results in the least number of connected components (after that label and the associated edges are added).

Our modified versions of MVCA focus on the first label added. Voss et al. [13] modify MVCA in that they try each label at the first step and then apply MVCA in a greedy fashion from there on. They then select the best of the  $|L| = \ell$  resulting solutions. We implement their pilot method in MVCA1.

If the number of labels is large, MVCA1 can be quite time-consuming. MVCA2 is designed to remedy this. In MVCA2, we try the most promising 10% of the labels at the first step, follow this with straightforward MVCA, and then select the best of the  $\frac{\ell}{10}$  resulting solutions. MVCA3 is similar to MVCA2, except that in MVCA3 we try the most promising 30% of the labels at the first step. The three versions are summarized below.

MVCA1: We try each label in  $L$  as the first or pilot label. Then, we run MVCA to determine the remaining labels. We need to call MVCA  $\ell$  times and we obtain  $\ell$  solutions. We output the best solution. The MVCA heuristic is fast, but for large  $\ell$ , we expect that MVCA1 will be very slow.

MVCA2: Here, we sort all of the labels by their frequencies in  $G$ , from highest to lowest. Then, we select each of the top 10% of the labels, with respect to

frequency, to serve as the pilot label. Then, we run MVCA to determine the remaining labels. Thus, we call MVCA  $\ell/10$  times. Compared with MVCA1, MVCA2 can potentially reduce running time by about 90%. Since a higher frequency label may not always be the best place to start, MVCA2 may not perform as well as MVCA1.

MVCA3: We select each of the top 30% of the labels, with respect to frequency, to serve as the pilot label. Then, we run MVCA to determine the remaining labels. We call MVCA  $3\ell/10$  times. We expect that MVCA3 will be between MVCA1 and MVCA2 with respect to accuracy and running time.

### 5.3 A Modified Genetic Algorithm

In Xiong, Golden, and Wasil [16], a GA is presented with a simple and fast crossover operator. Given two parents  $S_1$  and  $S_2$ , where each is a set of labels, crossover begins by taking the union. Next, we sort the labels in the union from highest frequency to lowest frequency. We add these labels to an empty label set  $C$ , one at a time (as well as the associated edges), until the subgraph induced by  $C$  is connected. See chapter 4 for details.

In our modified GA (denoted MGA), we alter the crossover operation in a simple and powerful way. As before, we take the union of the parents (i.e.,  $S = S_1 \cup S_2$ ). Next, we apply MVCA to the subgraph of  $G$  with label set  $S$  (where  $S \subseteq L$ ), node set  $V$ , and the edge set  $E'$  (where  $E' \subseteq E$ ) associated with  $S$ . The new crossover operation takes more time than the old one. We can, therefore, reduce

the number of generations. In our computational experiments, the population size is 40 and the number of generations is 20.

## 5.4 Computational Results

In Tables 5.1 to 5.4, we present the computational results for MVCA, GA, MGA, MVCA1, MVCA2, and MVCA3 for graphs with 50, 100, 150, and 200 nodes. In these four tables,  $n$  is the number of nodes,  $\ell$  is the number of labels, and  $d$  is the density of the graph ( $d = 1$  implies  $\binom{n}{2}$  edges). Each row is an average over 20 instances.

In Table 5.5, the relative performance of the six heuristics is presented. In the right-most column, the row total gives the number of times each heuristic has outperformed the others. The overall ranking (from best to worst) with respect to solution quality is MVCA1, MGA, MVCA3, MVCA2, GA, MVCA.

Next, we focus on running times. All the instances are run on a Pentium 4 PC at 1.80 GHz with 256 MB RAM. Running times for the 12 most demanding cases are presented in Table 5.6. These are the cases on which at least one heuristic required more than about two seconds of running time. We observe that MGA is nearly comparable to MVCA1 in solution quality, but is much faster. In addition, MGA easily outperforms MVCA2 in solution quality, but is slower. MGA also outperforms MVCA3 in solution quality. We point out that MVCA3 has been designed to be as computationally demanding as MGA. For example, on the largest problem tested, both MGA and MVCA3 require about 35 seconds of running time. Thus, of the six

heuristics, MGA offers the best compromise with respect to accuracy and running time.

	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3
$n = 50, \ell = 12, d = 0.8$	1.10	1.10	1.10	1.10	1.10	1.10
$n = 50, \ell = 12, d = 0.5$	1.95	1.95	1.95	1.95	1.95	1.95
$n = 50, \ell = 12, d = 0.2$	3.45	3.30	3.20	3.20	3.45	3.35
$n = 50, \ell = 25, d = 0.8$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 50, \ell = 25, d = 0.5$	2.95	2.95	2.95	2.95	2.95	2.95
$n = 50, \ell = 25, d = 0.2$	5.45	5.10	5.00	5.10	5.45	5.25
$n = 50, \ell = 50, d = 0.8$	3.00	3.00	3.00	3.00	3.00	3.00
$n = 50, \ell = 50, d = 0.5$	4.20	4.05	4.00	4.00	4.00	4.00
$n = 50, \ell = 50, d = 0.2$	8.25	7.65	7.55	7.60	8.00	7.70
$n = 50, \ell = 62, d = 0.8$	3.40	3.10	3.00	3.00	3.05	3.05
$n = 50, \ell = 62, d = 0.5$	4.75	4.45	4.15	4.20	4.50	4.30
$n = 50, \ell = 62, d = 0.2$	9.20	8.55	8.40	8.60	9.00	8.70

Table 5.1: Computational results for graphs with  $n = 50$  where each entry is the average number of labels over 20 instances.

## 5.5 Conclusions

In this chapter, we have presented three modified versions of MVCA and a modified GA (denoted MGA). These four new procedures have been tested over a wide range of minimum labeling spanning tree problems and compared against

	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3
$n = 100, \ell = 25, d = 0.8$	1.45	1.50	1.45	1.45	1.45	1.45
$n = 100, \ell = 25, d = 0.5$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 100, \ell = 25, d = 0.2$	4.00	3.95	3.95	3.95	3.95	3.95
$n = 100, \ell = 50, d = 0.8$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 100, \ell = 50, d = 0.5$	3.05	3.00	3.00	3.00	3.00	3.00
$n = 100, \ell = 50, d = 0.2$	6.15	5.95	5.85	5.85	6.05	5.95
$n = 100, \ell = 100, d = 0.8$	3.60	3.20	3.05	3.00	3.15	3.10
$n = 100, \ell = 100, d = 0.5$	4.90	4.95	4.80	4.65	4.80	4.75
$n = 100, \ell = 100, d = 0.2$	9.40	8.90	8.60	8.75	8.90	8.75
$n = 100, \ell = 125, d = 0.8$	4.05	4.00	3.95	3.95	3.95	3.95
$n = 100, \ell = 125, d = 0.5$	5.65	5.40	5.10	5.10	5.25	5.15
$n = 100, \ell = 125, d = 0.2$	10.95	10.15	9.85	9.95	10.30	10.00

Table 5.2: Computational results for graphs with  $n = 100$  where each entry is the average number of labels over 20 instances.

MVCA and GA. All four of these modified procedures generated better results than MVCA and GA, but were computationally more burdensome. MGA and MVCA1 were similar with respect to solution quality. However, MGA was much faster on average than MVCA1.

	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3
$n = 150, \ell = 37, d = 0.8$	1.85	1.85	1.85	1.85	1.85	1.85
$n = 150, \ell = 37, d = 0.5$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 150, \ell = 37, d = 0.2$	4.10	4.00	4.00	4.00	4.10	4.00
$n = 150, \ell = 75, d = 0.8$	2.30	2.10	2.00	2.00	2.00	2.00
$n = 150, \ell = 75, d = 0.5$	3.35	3.05	3.00	3.00	3.00	3.00
$n = 150, \ell = 75, d = 0.2$	6.60	6.30	6.00	6.00	6.30	6.05
$n = 150, \ell = 150, d = 0.8$	3.95	3.95	3.95	3.85	3.85	3.85
$n = 150, \ell = 150, d = 0.5$	5.10	5.05	5.00	5.00	5.00	5.00
$n = 150, \ell = 150, d = 0.2$	10.35	9.80	9.50	9.65	9.80	9.75
$n = 150, \ell = 187, d = 0.8$	4.35	4.05	4.00	4.00	4.00	4.00
$n = 150, \ell = 187, d = 0.5$	6.00	6.00	6.00	5.85	5.85	5.85
$n = 150, \ell = 187, d = 0.2$	11.85	11.45	10.85	10.95	11.20	11.05

Table 5.3: Computational results for graphs with  $n = 150$  where each entry is the average number of labels over 20 instances.

	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3
$n = 200, \ell = 50, d = 0.8$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 200, \ell = 50, d = 0.5$	2.00	2.00	2.00	2.00	2.00	2.00
$n = 200, \ell = 50, d = 0.2$	4.30	4.00	4.00	4.00	4.30	4.00
$n = 200, \ell = 100, d = 0.8$	2.65	2.35	2.30	2.20	2.30	2.25
$n = 200, \ell = 100, d = 0.5$	3.75	3.30	3.20	3.00	3.35	3.15
$n = 200, \ell = 100, d = 0.2$	6.90	6.80	6.50	6.55	6.80	6.65
$n = 200, \ell = 200, d = 0.8$	4.00	4.00	4.00	4.00	4.00	4.00
$n = 200, \ell = 200, d = 0.5$	5.80	5.75	5.25	5.05	5.15	5.05
$n = 200, \ell = 200, d = 0.2$	10.90	10.85	10.35	10.20	10.45	10.30
$n = 200, \ell = 250, d = 0.8$	4.75	4.80	4.50	4.00	4.15	4.05
$n = 200, \ell = 250, d = 0.5$	6.50	6.50	6.10	6.00	6.00	6.00
$n = 200, \ell = 250, d = 0.2$	12.55	12.25	11.75	11.75	12.05	12.00

Table 5.4: Computational results for graphs with  $n = 200$  where each entry is the average number of labels over 20 instances.

	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3	Row total
MVCA	-	3	0	0	0	0	3
GA	30	-	0	1	9	4	44
MGA	33	30	-	10	20	16	109
MVCA1	35	30	10	-	24	20	119
MVCA2	31	20	5	0	-	0	56
MVCA3	34	27	8	0	23	-	92

Table 5.5: Summary of computational results with respect to accuracy for six heuristics on 48 cases. The entry  $(i, j)$  represents the number of cases heuristic  $i$  generates a solution that is better than the solution generated by heuristic  $j$ .



	MVCA	GA	MGA	MVCA1	MVCA2	MVCA3
$n = 100, \ell = 125, d = 0.2$	0.05	1.80	7.50	8.25	0.80	2.30
$n = 150, \ell = 150, d = 0.5$	0.10	1.85	4.90	11.85	1.15	3.45
$n = 150, \ell = 150, d = 0.2$	0.15	3.45	13.55	21.95	2.15	6.35
$n = 150, \ell = 187, d = 0.5$	0.15	2.20	6.70	21.70	2.00	6.15
$n = 150, \ell = 187, d = 0.2$	0.20	3.95	17.55	39.35	3.60	11.20
$n = 200, \ell = 100, d = 0.2$	0.15	3.75	11.40	11.25	1.15	3.35
$n = 200, \ell = 200, d = 0.8$	0.25	2.45	5.80	26.70	2.70	8.00
$n = 200, \ell = 200, d = 0.5$	0.25	3.45	10.15	38.65	3.90	10.15
$n = 200, \ell = 200, d = 0.2$	0.35	6.20	26.65	68.25	6.85	20.35
$n = 200, \ell = 250, d = 0.8$	0.30	3.05	7.55	52.25	5.25	15.35
$n = 200, \ell = 250, d = 0.5$	0.30	3.95	12.60	69.90	6.80	20.35
$n = 200, \ell = 250, d = 0.2$	0.50	6.90	33.15	124.35	12.10	35.80
Average running time	0.23	3.58	13.13	41.20	4.04	11.90

Table 5.6: Running times for 12 demanding cases (in seconds).

## Chapter 6

### The Label-Constrained Minimum Spanning Tree Problem (LC-MST)

#### 6.1 Introduction

Computing a minimum-weight spanning tree (MST) is one of the fundamental and classic problems in graph theory. Given an undirected graph  $G$  with a nonnegative weight (or distance) on each edge, the MST of  $G$  is the spanning tree of  $G$  with the minimum total edge weight among all possible spanning trees [7]. This problem and its related problems,  $k$  smallest spanning tree [6], edge update of minimum spanning tree [17], minimum diameter spanning tree [10], most and least uniform spanning trees [7, 8] and so on, have been intensely studied. MST problems have applications in many areas, including network design, VLSI, and geometric optimization.

In this chapter, the label-constrained minimum spanning tree (LC-MST) problem is defined and studied. The LC-MST is somewhat more realistic than the MLST problem. For example, in communications networks, there are many different types of communication media, such as fiber optic, cable, microwave, telephone line, and so on. A communication node may communicate with other nodes via different types of communication media. There is a weight between any two nodes. Given a set of communications network nodes, the problem we are interested in is to find a spanning tree that has a minimum total weight which uses at most  $K$  distinct

types of communication media. This is a more realistic situation, because we are still more concerned with the total weight, but we only allow a restricted number of communication media. This problem can be formulated as a graph problem as follows.

**Problem**(LC-MST problem). Given an undirected labeled graph  $G = (V, E, L)$  and a weight function  $w(e)$  for all  $e \in E$ , and a positive integer  $K$ , find a spanning tree  $T$  of  $G$  such that the total weight  $\sum_{e \in T} w(e)$  is minimized and the number of distinct labels of  $T$  is  $|L_T| \leq K$ .

It is shown that the LC-MST problem is NP-complete. Two local search methods are applied to solve the LC-MST problem. A genetic algorithm is also presented. Experimental results indicate that the two local search methods can obtain optimal or near-optimal solutions, while the genetic algorithm can get a comparable result, but it is much faster.

## 6.2 NP-completeness proof

**Theorem 2.** *The Label-Constrained Minimum Spanning Tree (LC-MST) problem is NP-hard.*

*Proof.* We first show that LC-MST belongs to NP. Given an instance of the problem, we consider an arbitrary spanning tree  $T$ . The verification algorithm checks that  $T$  contains at most  $K$  labels and its total cost is less than or equal to a given positive

number  $W$ . This process can certainly be done in polynomial time.

To prove that LC-MST is NP-hard, we show that  $\text{MLST} \leq_P \text{LC-MST}$ , which means that MLST is *polynomial-time reducible* to LC-MST. Here MLST is the minimum labeling spanning tree problem, which is known to be NP-complete. Let  $G = (V, E, L)$  be an instance of MLST. It has a minimum labeling spanning tree  $T$  with  $K$  labels. We construct an instance of LC-MST as follows. We extend  $G$  to the complete graph  $G' = (V, E', L')$ , where  $E'$  contains  $E$  and  $L'$  contains  $L$ . For each  $e \in E$ , we define its weight  $w(e)$  to be 0. For each  $e \notin E$ , we define its weight  $w(e)$  to be 1. For each  $e \notin E$ , we also define its label to be different from each other.

We now show that graph  $G$  has a minimum spanning tree with  $K$  labels if and only if graph  $G'$  has a minimum-weight spanning tree with  $K$  labels. Suppose that graph  $G$  has a minimum labeling spanning tree  $T$  with  $K$  labels. Then, each edge in  $T$  has weight 0 in  $G'$  and, thus, has total weight 0. This tree  $T$  is the minimum-weight spanning tree in  $G'$ . Conversely, suppose that graph  $G'$  has a minimum-weight spanning tree  $T$  with  $K$  labels and total weight 0, then all the edges in  $T$  are contained in  $E$ . Thus,  $T$  is also a spanning tree in  $G$  with  $K$  labels.

Therefore, LC-MST is NP-hard. □

### 6.3 Local search methods

In this section, two methods of local search are introduced. Given a labeled graph  $G$  with a weight for each edge, it has a minimum labeling spanning tree  $T_1$ , and it also has a minimum-weight spanning tree  $T_2$ . Suppose  $T_1$  has  $n_1$  distinct

labels and  $T_2$  has  $n_2$  distinct labels. In the LC-MST problem, a positive number  $K$  is given. If  $K < n_1$ , then the LC-MST problem has no solution. If  $K > n_2$ , then the optimal solution of the LC-MST problem is  $T_2$ , which can be solved in polynomial time. So, in this paper, we always assume that  $n_1 \leq K \leq n_2$ .

### 6.3.1 Encoding

The solution of the LC-MST problem is a spanning tree. Each spanning tree has a label set. Conversely, given any label set  $A \in 2^L$ , we can get a subgraph  $G_A$  of  $G$  induced by  $A$ . We always assume that  $G_A$  contains all the nodes of  $G$ . If the subgraph  $G_A$  is connected and spans all the nodes, a minimum-weight spanning tree can be found by Prim's algorithm in polynomial time. Then, we can get the total weight of the spanning tree. In the subgraph  $G_A$ , the minimum-weight spanning tree may not be unique, but the total weight is unique. Thus, we can build a map  $f : 2^L \rightarrow \mathbb{R}$  as follows. For any  $A \in 2^L$ , if  $G_A$  is connected, then  $f(A)$  is the total weight of a minimum-weight spanning tree of  $G_A$ ; if  $G_A$  is not connected, then  $f(A) = \infty$ . Let  $L_K \subset 2^L$  represent the collection of all label sets with  $K$  labels. We restrict  $f$  to  $L_K$ . Then the goal of the LC-MST problem is to find  $A^* \in L_K$ , such that  $f(A^*) = \min_{A \in L_K} f(A)$ . By this well-defined map  $f$ , it is sufficient to consider a label set  $A \in L_K$  as a solution of the LC-MST problem.

### 6.3.2 Local search 1 (LS1)

In local search 1 (LS1), we begin with an arbitrary label set  $A \in L_K$ . Suppose  $A = \{a_1, a_2, \dots, a_K\}$ . Then, we begin a Replacing Loop (RL) as follows. We first replace  $a_1$  by some label  $b_1 \in L$ . We check all labels in  $L - A$ . If we can make an improvement, then we find  $b_1$  such that  $f(A - \{a_1\} + \{b_1\}) = \min_{b \in L} f(A - \{a_1\} + \{b\})$ . Then we set  $A := A - \{a_1\} + \{b_1\}$ . Otherwise, we let  $b_1 = a_1$ . Next, we seek to replace  $a_2$  by some label  $b_2 \in L$ . We check all labels in  $L - A$  and, if we can make an improvement, we obtain  $f(A - \{a_2\} + \{b_2\}) = \min_{b \in L} f(A - \{a_2\} + \{b\})$  and set  $A := A - \{a_2\} + \{b_2\}$ . Otherwise, we let  $b_2 = a_2$ . We continue this replacement process until we replace  $a_k$  by some suitable  $b_k$  (possibly,  $b_k = a_k$ ). After one RL, we improve the label set to  $A = \{b_1, b_2, \dots, b_K\}$ . We continue RL until no improvement can be made between two consecutive label sets with respect to RL.

### 6.3.3 Local search 2 (LS2)

In local search 2 (LS2), we begin with an arbitrary label set  $A \in L_K$ . Suppose  $A = \{a_1, a_2, \dots, a_K\}$ . Then we begin a Replacing Loop (RL) as follows. For each  $b \in L - A$ , we first set  $A := A + \{b\}$  and let  $a_{K+1} = b$ . So  $A$  has  $K + 1$  labels. To keep  $K$  labels, we have to remove one label from  $A$ . We check all the labels  $a_i (1 \leq i \leq K + 1)$  and find  $a_j$  such that  $f(A - \{a_j\}) = \min_{1 \leq i \leq K+1} f(A - \{a_i\})$ . Then we set  $A = A - \{a_j\}$ . After one RL, we make some improvement for the label set  $A$ . We continue RL until no improvement can be made between two consecutive

label sets with respect to RL.

By simple observation, if we compare the two methods of local search, we have the following result.

**Theorem 3.** *A label set  $A$  cannot be improved by LS1 if and only if  $A$  cannot be improved by LS2.*

*Proof.* Suppose a label set  $A$  cannot be improved by LS1. Then, for any label  $a \in A$  and  $b \notin A$ , we know  $f(A - \{a\} + \{b\}) \geq f(A)$ . If we apply LS2 to the label set  $A$ , we first add some label  $b \notin A$ , and then remove some label  $a \in A$ , we get the label set  $A + \{b\} - \{a\} = A - \{a\} + \{b\}$ . We still get  $f(A + \{b\} - \{a\}) = f(A - \{a\} + \{b\}) \geq f(A)$ . So the label  $A$  cannot be improved by LS2. Conversely, if a label set  $A$  cannot be improved by LS2, from the above proof, because of the set equation  $A + \{b\} - \{a\} = A - \{a\} + \{b\}$ , we also know that  $A$  cannot be improved by LS1. □

#### 6.3.4 Running time analysis

In the LC-MST problem, we are given a labeled graph  $G = (V, E, L)$  and a positive integer  $K$ . Let  $|V| = n$ ,  $|E| = m$ , and  $|L| = \ell$ . Let us only consider complete graphs. So  $m = O(n^2)$ . For each  $A \in 2^L$ , we use Prim's algorithm to find a minimum-weight spanning tree in the subgraph  $G_A$ . Prim's algorithm requires  $O(m + n \lg n) = O(n^2)$  running time. So  $f(A)$  requires  $O(n^2)$  running time. Both LS1 and LS2 run  $f(A)$  for at most  $K|L| = K\ell$  times in each RL. Thus, the running time of each RL in LS1 and LS2 is  $O(K\ell n^2)$ .

## 6.4 Genetic algorithm

From the computational results, the two local search methods can obtain optimal or near-optimal solutions. But they are very slow. In this section, we introduce a genetic algorithm (GA) to solve the LC-MST problem. This GA can obtain comparable solutions, but it is much faster than LS1 and LS2. The encoding of GA is the same as in local search. Each chromosome is a label set in  $L_K$ . Each label is a gene.

### 6.4.1 Crossover

Given two parent chromosomes  $P$  and  $Q$ , one child  $C$  is created by the crossover operation. First, we set  $R = P \cup Q$  and we get a subgraph  $G_R$  induced by  $R$ . We also set  $C = \phi$ . Second, we apply Prim's algorithm in the subgraph  $G_R$ . In Prim's algorithm, we begin with a random node  $v \in V$ . Then, we grow a tree by adding the least-weight edge each time until the tree spans all the nodes in  $G_R$ . In this procedure, each time we add the least-weight edge  $e$ ,  $e$  has a label  $c$ . If  $c \notin C$ , we set  $C = C \cup \{c\}$ . So when a tree is growing, the label set  $C$  is also expanding. When  $|C|$  reaches  $K$ , we stop Prim's algorithm. Finally we get the output child  $C$ . The crossover operation is very fast. The running time of crossover is the same as that of Prim's algorithm.

To improve efficiency, we apply the Queen-bee crossover. In each generation, we find the best chromosome  $QB$  and set it as the queen-bee chromosome. Then, we only do crossover between  $QB$  and other chromosomes.



### 6.4.2 Mutation

Given one chromosome  $P$ , the mutation operation creates a new chromosome  $Q$ . Suppose  $P = \{p_1, p_2, \dots, p_K\}$ . First, we randomly select  $b \in L - P$ . Second, we check each  $p_i (1 \leq i \leq K)$  to see if the replacement of  $p_i$  by  $b$  obtains a better solution. If it does for some  $p_i$ , then we set  $Q = P - \{p_i\} + \{b\}$ . We always take the first  $p_i$  which can make an improvement. If no  $p_i$  works, then we set  $Q = P$ . The mutation operation may run Prim's algorithm for at most  $K$  times. It is slower than the crossover operation.

### 6.4.3 Building each generation

Suppose the population size is  $p$  (We set  $p = 20$  in the experiments.).  $p$  chromosomes are randomly created as the initial generation. We set the mutation rate to be 20% and the crossover rate to be 70%. To build the next generation, we first find the queen-bee chromosome  $QB$ . Then, we perform the mutation operation for  $QB$  with probability 20%. Next, we do the crossover operation between  $QB$  and each of  $p - 1$  other chromosomes with probability 70%.  $QB$  is passed forward to the next generation. For every other chromosome  $P$ , if it has a chance to do crossover with  $QB$  and obtain a better child  $C$ , then  $P$  is replaced by  $C$  in the next generation; if it does not have a chance to do crossover or it creates a worse child after crossover, then  $P$  is passed forward to the next generation. So, in each generation, if we have

$p$  chromosomes, then after the crossover and mutation operations, we still have  $p$  chromosomes. The generation building procedure stops if the best chromosome from three consecutive generations does not change. In the last generation, we make a final improvement. We find the queen-bee chromosome  $QB$  and execute one RL of LS1 on  $QB$ .

## 6.5 Computational Results

### 6.5.1 Small cases

In this section, we report on computational results for small case instances. For all the instances, we use Integer Programming (IP) to obtain the optimal solution. (Work on the IP was done by Si Chen.) See Appendix A.5 for a more detailed description of the IP solution. Tables 6.1-6.4 show the results and the instance solved in Table 6.1 is given in Appendix A.3. Table 6.1 tests all possible  $K$ 's ranging from the number of labels in the MLST to the number of labels in the MST. Tables 6.2-6.4 test for  $K$ 's ranging from the number of labels in the MLST up to the number such that the IP can obtain an optimal solution in a reasonable amount of time. For each instance, we give solutions of IP, LS1, LS2, and GA. We also compute the gap of each heuristic from the IP (optimal) solution. The gaps are fairly small. We also find that the three heuristics have bigger gaps when  $K$  is very small. But when  $K$  increases, the three heuristics work very well. They reach the optimal solution or the gap is less than 1%. For LS1 and LS2, we make five runs for each input and report the best one. They are very fast in small cases. On average, they need 1

second for each input. GA is faster and it requires less than 0.5 second for each input.

$K$	IP	Time(sec)	LS1	Gap(%)	LS2	Gap(%)	GA	Gap(%)
2	6491.33	3.30	6491.35	0.00	6491.35	0.00	6491.35	0.00
3	5013.51	7.70	5013.51	0.00	5013.51	0.00	5013.51	0.00
4	4534.67	81.20	4534.67	0.00	4534.67	0.00	4534.67	0.00
5	4142.57	277.56	4142.57	0.00	4142.57	0.00	4142.57	0.00
6	3846.49	216.00	3846.50	0.00	3846.50	0.00	3846.50	0.00
7	3598.03	97.97	3598.05	0.00	3598.05	0.00	3598.05	0.00
8	3436.56	65.56	3436.57	0.00	3436.57	0.00	3436.57	0.00
9	3281.04	74.36	3281.05	0.00	3281.05	0.00	3281.05	0.00
10	3152.03	30.34	3152.05	0.00	3152.05	0.00	3152.05	0.00
11	3033.99	9.77	3034.01	0.00	3034.01	0.00	3034.01	0.00

Table 6.1: Computational results for a graph with  $n = \ell = 20$

### 6.5.2 Large Cases

In this section, we report on computational results for large cases. We only consider  $K = 20$  and  $K = 40$ . For each instance, we run LS1 and LS2 5 times and output the best results. Table 6.5 gives the results. Table 6.6 gives the running time for each instance. From the two tables, we can see that GA is much faster than LS1 and LS2. The solution quality of GA is not as good as that of LS1 and LS2, but the gap between GA and the best-obtained solution is still under 1%. In

$K$	IP	Time(sec)	LS1	Gap(%)	LS2	Gap(%)	GA	Gap(%)
3	7901.81	20.83	7901.81	0.00	7901.81	0.00	7901.81	0.00
4	6431.58	38.89	6431.58	0.00	6431.58	0.00	6431.58	0.00
5	5597.36	30.47	5597.36	0.00	5597.36	0.00	5597.36	0.00
6	5106.94	174.20	5106.94	0.00	5106.94	0.00	5106.94	0.00
7	4751.00	464.30	4751.00	0.00	4773.04	0.46	4751.00	0.00
8	4473.11	1229.48	4473.11	0.00	4473.11	0.00	4473.11	0.00
9	4196.71	579.63	4196.71	0.00	4196.71	0.00	4196.71	0.00
10	3980.99	931.83	4001.72	0.52	4001.72	0.52	4001.72	0.52
11	3827.23	279.73	3827.23	0.00	3827.23	0.00	3853.12	0.68
12	3702.08	297.36	3702.08	0.00	3710.58	0.23	3710.58	0.23
13	3585.42	90.78	3585.42	0.00	3585.42	0.00	3585.42	0.00

Table 6.2: Computational results for a graph with  $n = \ell = 30$

in addition to solving problems with  $n = 100, 150,$  and  $200,$  we also test a very large case when  $n = \ell = 500.$  The GA solution is of high quality and it takes much less time than LS1 or LS2. From Table 6.6, we find that the average running time of LS1 and LS2 is about 10 times the average running time of GA. Because the results of LS1 and LS2 strongly depend on the initial label set, one run of each will probably obtain a worse solution than GA. Table 6.7 gives the results of a small example with  $n = \ell = 50$  and  $K = 3.$  For LS1, only the fourth and the fifth run obtain a good solution; for LS2, only the fifth run obtains a good solution. The gap between the best and the worst solutions is about 5.76%.

$K$	IP	Time(sec)	LS1	Gap(%)	LS2	Gap(%)	GA	Gap(%)
3	11578.61	9651	11578.64	0.00	11578.64	0.00	12260.52	5.89
4	9265.42	5305	9424.53	1.72	9502.18	2.56	9502.18	2.56
5	8091.45	5841	8091.48	0.00	8091.48	0.00	8151.99	0.75
6	7167.27	1844	7167.30	0.00	7167.30	2.53	7332.11	2.30
7	6653.23	2904	6661.71	0.13	6653.27	0.00	6653.27	0.00
8	6221.63	10744	6221.67	0.00	6221.67	0.00	6255.24	0.54
9	5833.39	16487	5833.42	0.00	5833.42	0.00	5888.17	0.94
10	5547.08	8830	5547.11	0.00	5547.11	0.00	5547.11	0.00
11	5315.92	13641	5315.95	0.00	5315.95	0.00	5315.95	0.00
12	5164.14	109284	5164.17	0.00	5164.17	0.00	5164.17	0.00

Table 6.3: Computational results for a graph with  $n = \ell = 40$

$K$	IP	Time(sec)	LS1	Gap(%)	LS2	Gap(%)	GA	Gap(%)
3	14857.09	3285	14857.11	0.00	14857.11	0.00	15315.30	3.08
4	12040.89	5894	12040.91	0.00	12040.91	0.00	12040.91	0.00
5	10183.95	2750	10183.95	0.00	10183.95	0.00	10183.95	0.00
6	9343.69	11820	9343.69	0.00	9343.69	0.00	9343.69	0.00
7	8594.36	74138	8594.36	0.00	8594.36	0.00	8594.36	0.00
8	7965.52	303389	7965.55	0.00	7965.55	0.00	7965.55	0.00

Table 6.4: Computational results for a graph with  $n = \ell = 50$

	LS1	LS2	GA	GA Gap(%)
$n = 100, \ell = 50, K = 20$	8308.68	8308.68	8366.80	0.70
$n = 100, \ell = 100, K = 20$	10055.85	10055.85	10153.92	0.98
$n = 100, \ell = 100, K = 40$	7344.72	7335.61	7344.72	0.12
$n = 150, \ell = 75, K = 20$	11882.62	11846.80	11917.01	0.59
$n = 150, \ell = 75, K = 40$	9046.71	9046.71	9055.67	0.10
$n = 150, \ell = 150, K = 20$	15427.54	15398.42	15512.27	0.74
$n = 150, \ell = 150, K = 40$	10618.58	10627.36	10640.83	0.21
$n = 200, \ell = 100, K = 20$	14365.95	14365.95	14365.95	0.00
$n = 200, \ell = 100, K = 40$	10970.94	10970.94	10970.94	0.00
$n = 200, \ell = 200, K = 20$	18951.05	18959.37	19133.50	0.96
$n = 200, \ell = 200, K = 40$	12931.46	12941.85	12980.52	0.38
$n = 500, \ell = 500, K = 40$	34320.00	34138.61	34551.08	1.21

Table 6.5: Computational results for large cases

	LS1	LS2	GA
$n = 100, \ell = 50, K = 20$	9	9	1
$n = 100, \ell = 100, K = 20$	14	17	2
$n = 100, \ell = 100, K = 40$	27	29	3
$n = 150, \ell = 75, K = 20$	30	37	3
$n = 150, \ell = 75, K = 40$	55	60	7
$n = 150, \ell = 150, K = 20$	65	55	5
$n = 150, \ell = 150, K = 40$	139	144	12
$n = 200, \ell = 100, K = 20$	71	64	9
$n = 200, \ell = 100, K = 40$	181	226	20
$n = 200, \ell = 200, K = 20$	120	161	14
$n = 200, \ell = 200, K = 40$	344	332	35
$n = 500, \ell = 500, K = 40$	9063	7440	672

Table 6.6: Running time (in seconds)

Run	LS1	LS2
1	15315.29	15315.29
2	15712.38	15712.38
3	15712.38	15712.38
4	14857.11	15712.38
5	14857.11	14857.11

Table 6.7: Computational results for a graph with  $n = \ell = 50$  and  $K = 3$

## Chapter 7

### The Colorful Traveling Salesman Problem (CTSP)

#### 7.1 Introduction

In the colorful TSP (CTSP), we are given an undirected complete graph with labeled edges as input. Each edge has a single label and different edges can have the same label. We can think of each label as a unique color. The goal is to find a Hamiltonian cycle or tour with the minimum number of labels. The problem is motivated by the following hypothetical situation. An individual wants to visit  $n$  cities without repetition and return to his city of origin. Suppose that all pairs of cities are directly connected by railroad or bus and that there are  $\ell$  transportation companies. Each company controls a subset of the railroad and bus lines (edges) connecting the cities and each company charges the same flat fee for using its lines. We can draw the lines owned by Company 1 in red, the lines owned by Company 2 in blue, and so on. The objective is to construct a Hamiltonian tour that uses the smallest number of colors.

In this chapter, we first prove that the CTSP is NP-hard. Then, we introduce the Maximum Path Extension Algorithm (MPEA) to solve the CTSP. MPEA is a greedy algorithm and it borrows ideas from MVCA. We also introduce a genetic algorithm (GA) to solve CTSP. This GA combines the idea of MPEA and the genetic algorithm (GA) of Chapter 4 and it generates better computational results than



MPEA.

## 7.2 CTSP is NP-hard

**Problem:** Given a complete graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels. Each edge in  $E$  is assigned a label in  $L$ . The goal is to find a tour, or a Hamiltonian cycle, which contains the minimum number of distinct labels.

**Theorem 4.** *The Colorful TSP (CTSP) is NP-hard.*

*Proof.* We first show that CTSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of  $n$  vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the number of labels, and checks whether the sum is at most  $k$ . This process can certainly be done in polynomial time.

To prove that CTSP is NP-hard, we show that  $\text{HAM-CYCLE} \leq_P \text{CTSP}$ . Here HAM-CYCLE is the Hamiltonian-cycle problem, which is known to be NP-complete. Let  $G = (V, E)$  be an instance of HAM-CYCLE. We construct an instance of CTSP as follows. We form the complete graph  $G' = (V, E')$ , where  $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$ , and we define the labels as follows. All the edges in  $E$  have the same label  $c$ . Each edge in  $E' - E$  has a unique (other) label. The instance of CTSP is easily formed in polynomial time.

We now show that graph  $G$  has a Hamiltonian cycle if and only if graph  $G'$  has a tour with only one label. Suppose that graph  $G$  has a Hamiltonian cycle  $h$ . Each edge in  $h$  belongs to  $E$  and thus has label  $c$  in  $G'$ . Thus,  $h$  is a tour in  $G'$  with only one label  $c$ . Conversely, suppose that graph  $G'$  has a tour  $h'$  with only one label. Since each edge in  $E' - E$  has a unique label other than  $c$ , if  $h'$  has at least 2 edges, then all the edges in  $h'$  must be in  $E$  and thus  $h'$  is a Hamiltonian cycle in graph  $G$ . If  $h'$  has only one edge, then  $G$  has only 2 nodes. So we must have  $E = E'$  and  $h'$  is also a Hamiltonian cycle in graph  $G$ .

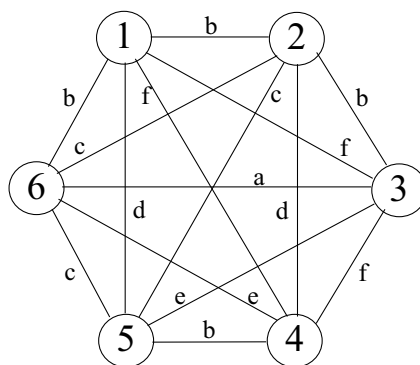
Therefore, CTSP is NP-complete. □

Figure 7.1 shows a small example of CTSP. In this example, the complete graph is  $G = (V, E, L)$ , where  $V = \{1, 2, 3, 4, 5, 6\}$  and  $L = \{a, b, c, d, e, f\}$ . It gives two tours  $g$  and  $h$ . Tour  $g$  contains three labels and tour  $h$  contains two labels. So tour  $h$  is a better solution. Note that we use the terms node and vertex interchangeably.

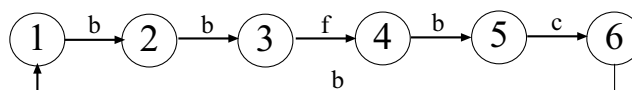
### 7.3 Maximum Path Extension Algorithm

In this section, we introduce a heuristic algorithm to solve CTSP. This algorithm is called *Maximum Path Extension Algorithm* (MPEA). The basic idea of MPEA is to visit as many vertices as possible while maintaining the number of labels in the current partial tour.

$G = (V, E, L)$



Tour g



Tour h

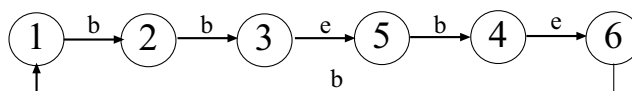


Figure 7.1: A small example of CTSP: Tour  $h$  contains three different labels  $b, c, f$ ; tour  $g$  contains two different labels  $b, e$ . So tour  $g$  is better.

### 7.3.1 How to extend a partial tour?

In a complete labeled graph  $G = (V, E, L)$ , suppose we have a partial tour  $h : v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ . Let  $C_h$  be the set of labels in the partial tour  $h$ . We want to extend the partial tour  $h$  to  $h'$  by one more vertex  $v_{k+1}$  such that  $C_{h'} = C_h$ , where  $C_{h'}$  is the set of labels in  $h'$ . The trivial case is: we can find an unvisited vertex  $v_{k+1}$ , such that the label of the edge  $(v_k, v_{k+1})$  or the label of the edge  $(v_1, v_{k+1})$  belongs to  $C_h$ . Then we directly insert  $v_{k+1}$  after  $v_k$  or prior to  $v_1$ . The following four nontrivial cases can also make this extension possible.

Case 1: If we can find some unvisited vertex  $v_{k+1}$  and some  $v_j \in h$ , such that the labels of the edges  $(v_j, v_{k+1})$  and  $(v_{j+1}, v_{k+1})$  belong to  $C_h$ , then we can insert

the vertex  $v_{k+1}$  into the tour  $h$  without increasing  $|C_h|$ . Figure 7.2 shows the insertion.

Case 2: If we can find some unvisited vertex  $v_{k+1}$  and some  $v_j \in h$ , such that the labels of the edges  $(v_{k+1}, v_{j+1})$  and  $(v_j, v_k)$  belong to  $C_h$ , then we can insert  $v_{k+1}$  into the tour  $h$  without increasing  $|C_h|$  and  $v_{k+1}$  will be one end vertex of the new tour. Figure 7.3 shows the insertion.

Case 3: If we can find some unvisited vertex  $v_{k+1}$  and some  $v_j \in h$ , such that the labels of the edges  $(v_{k+1}, v_j)$  and  $(v_1, v_{j+1})$  belong to  $C_h$ , then we can insert  $v_{k+1}$  into the tour  $h$  without increasing  $|C_h|$  and  $v_{k+1}$  will be one end vertex of the new tour. Figure 7.4 shows the insertion.

Case 4: If the label of  $(v_1, v_k)$  belongs to  $C_h$ , and we can find some unvisited vertex  $v_{k+1}$  and some  $v_j$  in  $h$ , such that the label  $(v_j, v_{k+1})$  belongs to  $C_h$ , then we can insert  $v_{k+1}$  into  $h$  without increasing  $|C_h|$ .  $v_{k+1}$  and  $v_{j+1}$  will be the two end vertices of the new tour. Figure 7.5 shows the insertion.

If any unvisited vertex cannot satisfy any of the above trivial or nontrivial cases, then we extend the partial tour  $h$  by inserting a random unvisited vertex  $v_{k+1}$  to the end  $v_k$  of the tour  $h$  and add the new label of the edge  $(v_k, v_{k+1})$  to  $C_h$ . We can also select an unvisited vertex  $v_{k+1}$  with the highest frequency in the graph. This makes sense because this selection gives more opportunities to succeed in the extension of this path.

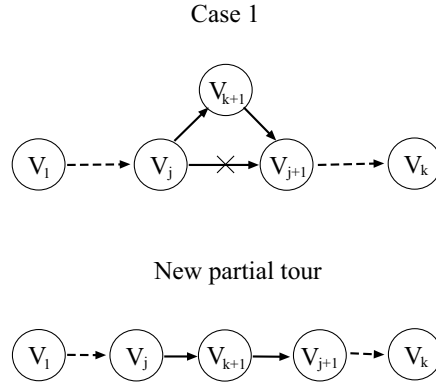


Figure 7.2: The original partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_{k+1} \rightarrow v_{j+1} \rightsquigarrow v_k$

### 7.3.2 Maximum Path Extension Algorithm

Suppose the input is a complete labeled graph  $G = (V, E, L)$  with  $|V| = n$  vertices. We want to output a Hamiltonian cycle  $h$ . Suppose  $C$  is the set of labels in  $h$ . The detailed description of MPEA is as follows.

Step 1: Sort all the labels in  $G$  according to their frequencies.

Step 2: Randomly select  $v_1 \in V$ . Then find  $v_2 \in V$  such that the label  $c_{12}$  of the edge  $(v_1, v_2)$  has the highest frequency.

Step 3: Let  $h = \{v_1, v_2\}$  and  $C = \{c_{12}\}$ .

Step 4: Add unvisited vertices into  $h$  according to the rules in Section 7.3.1 until  $h$  contains all the  $n$  vertices.

Step 5: Suppose  $h = \{v_1, \dots, v_n\}$  is an ordered sequence of vertices, and let label  $c_{1n}$  denote the label of the edge  $(v_1, v_n)$ . If  $c_{1n}$  is not in  $C$ , then add it to  $C$ .

Step 6: Output  $h$ .

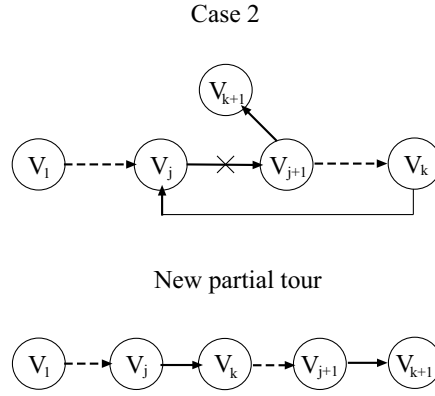


Figure 7.3: The original partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_k \rightsquigarrow v_{j+1} \rightarrow v_{k+1}$ ,  $v_{k+1}$  is at the end.

Now we consider the small example in Figure 7.1 and show how MPEA works for this example.

In this example,  $V = \{1, 2, 3, 4, 5, 6\}$  and  $L = \{b, c, f, d, e, a\}$ , where  $L$  is sorted in decreasing order based on label frequency. We begin with vertex 1 and we select 2 as the second vertex because the label of the edge  $(1, 2)$  is  $b$ , which has the highest frequency. So  $h : 1 \rightarrow 2$  and  $C = \{b\}$ . By the rules in Section 7.3.1,  $h$  can be extended to  $h : 6 \rightarrow 1 \rightarrow 2 \rightarrow 3$  with  $C$  unchanged. Next, we select 4 at random and find edge  $(3, 4)$  with label  $f$ , which has the highest frequency. So, we obtain  $h : 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$  and  $C = \{b, f\}$ . Next, we extend  $h$  again according to the rules in Section 7.3.1 and get  $h : 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ . Now  $h$  visits all the vertices in  $G$ . In the end, we check the label of the edge  $(5, 6)$ . It is  $c$ , which is not in  $C$ . We add it into  $C$  and  $C = \{b, f, c\}$ . We, therefore, obtain a tour  $g$  with 3 labels, as shown in Figure 7.1.

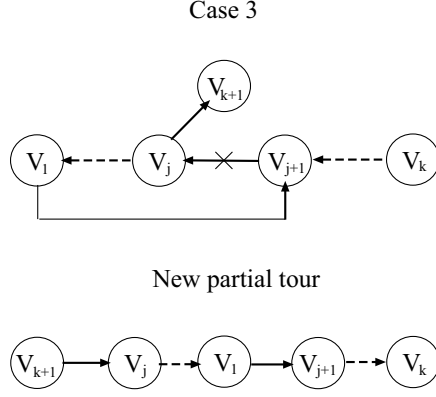


Figure 7.4: The original partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is  $v_{k+1} \rightarrow v_j \rightsquigarrow v_1 \rightarrow v_{j+1} \rightsquigarrow v_k$

### 7.3.3 Running time analysis

Given a complete labeled graph  $G = (V, E, L)$ , let  $|V| = n$ ,  $|L| = \ell$ , then,  $|E| = \frac{n(n-1)}{2} = O(n^2)$ . Usually  $\ell$  and  $n$  are of the same order of magnitude. We assume  $O(\ell) = O(n)$ . In step 1, we use quicksort and it requires  $O(\ell \ln \ell)$  running time. Step 2 requires  $O(n)$  running time and Step 3 requires  $O(1)$  running time. Step 4 is the main part of MPEA. In each loop, suppose  $h$  contains  $k$  vertices  $v_1, \dots, v_k$ , so there are  $n - k$  unvisited vertices. For each unvisited vertex  $u$ , we need to check the label of edges  $(u, v_i)$  for each  $1 \leq i \leq k$  and determine whether we can extend  $h$  to  $u$  without changing  $C$ . So, it requires  $O((n - k)k)$  running time. If we succeed in extending  $h$  to  $u$ , then the re-ordering operation requires  $O(k)$  running time; if we fail to extend  $h$  to all unvisited vertices, then we select the unvisited vertex with the highest frequency and it requires  $O(n - k)$  running time. The loop will be repeated at most  $n$  times and  $k$  goes from 2 to  $n$ . Thus the total running time of Step 4 is  $O(n^3)$ . Step 5 requires  $O(1)$  running time. Therefore, the total running time MPEA

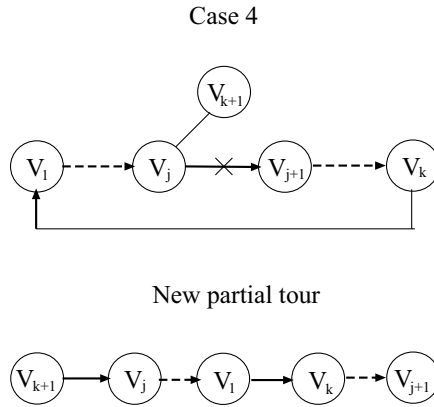


Figure 7.5: The original partial tour is  $v_1 \rightsquigarrow v_j \rightarrow v_{j+1} \rightsquigarrow v_k$ ; the new partial tour is  $v_{k+1} \rightarrow v_j \rightsquigarrow v_1 \rightarrow v_k \rightsquigarrow v_{j+1}$

is  $O(n^3)$ .

## 7.4 Genetic Algorithm

In this section, we introduce a genetic algorithm (GA) to solve the CTSP. We use the same procedure of MPEA, but we begin with a selective label set  $C$  which contains more than one label. The subgraph  $H$  induced by  $C$  must be connected and span all the vertices of  $G$ . Based on this label set  $C$ , extending a partial tour in  $H$  is more likely to succeed. We are trying to find a Hamiltonian cycle  $h$  containing approximately  $|C|$  labels. So  $|C|$  should be as small as possible. If MPEA can find a Hamiltonian cycle in the subgraph induced by  $C$ , then we find a tour with at most  $|C|$  labels. If MPEA fails to find a Hamiltonian cycle, then we continue MPEA by adding labels not in the induced subgraph and find a Hamiltonian cycle.



Subgraph induced by  $\{b,e\}$

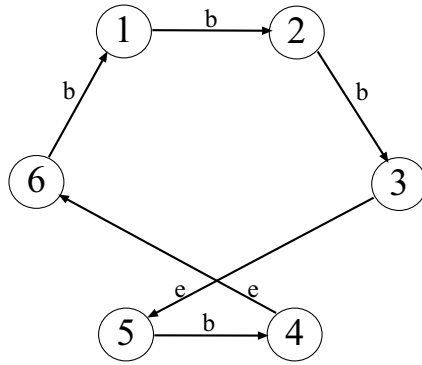


Figure 7.6: Subgraph induced by  $\{b,e\}$ . It contains a Hamiltonian cycle  $6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 6$ . The tour has two labels.

This will probably increase the number of labels in the Hamiltonian cycle. The MLST problem serves as a useful starting point. As a result, we apply the GA from Chapter 4, followed by MPEA.

For the small example in Figure 7.1, if we apply GA, we can find two optimal subgraphs. One is induced by the label set  $\{b,e\}$  and this subgraph contains a Hamiltonian cycle. We can use MPEA to find this cycle, as shown in Figure 7.6. So this tour contains two labels, which is the optimal solution. The other is induced by the label set  $\{b,d\}$ . But this subgraph only contains a Hamiltonian path. After we connect the head and the tail of the path, a new label is created, as shown in Figure 7.7. So, we can find a Hamiltonian cycle with three labels by MPEA.

Subgraph induced by  $\{b,d\}$

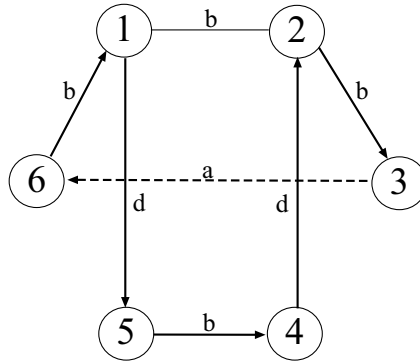


Figure 7.7: Subgraph induced by  $\{b, d\}$ . It does not contain a Hamiltonian cycle. We have to add edge  $(3, 6)$  to form a Hamiltonian cycle  $6 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 6$ . The tour contains three labels.

## 7.5 Computational results

In this section, we give the computational results of MPEA and GA. For each input  $n$  and  $\ell$ , we randomly generated 10 sample graphs. For each instance of graph, we ran MPEA 1000 times and found the best result. We ran GA only once and found the best result in the final generation. Finally, we output the average value of the 10 sample graphs for each input. Table 7.1 shows the computational results. In this table, combinatorial inputs of  $n$  and  $\ell$  are presented in column 1. The average values of MPEA results are shown in column 2 and average running times for MPEA are displayed in column 3. The average values of GA results and average running times for GA are presented in columns 4 and 5. For the 17 combinatorial inputs of this table, GA beats MPEA in 9 cases, MPEA beats GA in 5 cases, and they tie in 3 case. But the GA is a little bit slower than the MPEA. These results were run on

a Pentium 4 PC with 1.80 GHz and 256 MB RAM.

## 7.6 Concluding Remarks

In this chapter, we introduce the colorful traveling salesman problem(CTSP). We show that the CTSP is NP-hard. We present a heuristic algorithm MPEA and a genetic algorithm (GA) to solve CTSP. The MPEA is fast, but the GA works better than MPEA and its running time is reasonable. This is another nice example of the ability of GAs to successfully solve difficult NP-hard problems.

	MPEA	Avg time(sec)	GA	Avg time(sec)
$n = 50, \ell = 25$	2.3	0.2	2.4	0.3
$n = 50, \ell = 50$	4.3	0.2	4.2	0.4
$n = 50, \ell = 75$	5.4	0.2	5.7	0.5
$n = 50, \ell = 100$	6.3	0.2	6.8	0.6
$n = 100, \ell = 50$	3.0	1.1	3.0	0.9
$n = 100, \ell = 75$	4.0	1.3	4.1	1.2
$n = 100, \ell = 100$	5.3	1.3	5.1	1.5
$n = 100, \ell = 125$	6.0	1.4	6.9	1.7
$n = 100, \ell = 150$	7.0	1.5	6.9	1.7
$n = 150, \ell = 75$	3.2	3.7	3.0	5.1
$n = 150, \ell = 100$	4.3	3.9	4.1	6.2
$n = 150, \ell = 150$	5.8	4.2	5.5	7.6
$n = 150, \ell = 200$	7.3	4.5	7.3	8.9
$n = 200, \ell = 100$	3.4	8.1	3.4	10.1
$n = 200, \ell = 150$	5.2	9.4	4.9	13.0
$n = 200, \ell = 200$	6.7	10.1	6.2	14.7
$n = 200, \ell = 250$	8.1	10.8	7.4	17.2

Table 7.1: Computational results of MPEA and GA

## Chapter 8

### Conclusion

In this dissertation, we discussed three topics: the MLST problem, the LC-MST problem, and the CTSP. Each of these topics is based on the notion of an edge labeled graph.

The MLST problem was known to be NP-hard. We proved in this dissertation that the LC-MST and the CTSP are also NP-hard. We also analyzed the worst-case performance of the MVCA heuristic for the MLST.

Furthermore, we developed heuristic and genetic algorithms for each of the three problems. Optimal solutions were generated for small instances of each problem class using backtrack search or integer programming.

Based on extensive computational experiments, the genetic algorithms always perform well with respect to solution quality and running time. In addition, they are relatively straightforward, with a small number of parameters.

Overall, this work confirms that genetic algorithms can be very effective in handling difficult combinatorial optimization problems. We expect to see many more successful applications of genetic algorithms to NP-hard problems of the sort studied here.

## Appendix A

### Overview

#### A.1 MVCA heuristic

**Input:** A graph  $G = (V, E, L)$ , where  $V$  is the set of nodes,  $E$  is the set of edges, and  $L$  is the set of labels.

1. Let  $C \leftarrow \emptyset$  be the set of used labels.
2. **repeat**
3.   Let  $H$  be the subgraph of  $G$  restricted to  $V$  and edges with labels from  $C$ .
4.   **for all**  $i \in L - C$  **do**
5.     Determine the number of connected components when inserting all edges with label  $i$  in  $H$ .
6.   **end for**
7.   Choose label  $i$  with the smallest resulting number of components and do:  
     $C \leftarrow C \cup \{i\}$ .
8. **until**  $H$  is connected.

Labels	Edges
1	(8,2) (2,6) (3,8)
2	(7,2) (5,3) (3,2)
3	(7,9) (6,8) (10,6)
4	(5,4) (2,4) (9,8)
<b>5</b>	<b>(10,9) (9,3) (3,4)</b>
<b>6</b>	<b>(4,6) (6,7) (7,8)</b>
7	(10,4) (2,10) (6,1)
<b>8</b>	<b>(2,5) (5,1) (1,10)</b>
9	(7,1) (5,7) (9,5)
10	(10,7) (4,8) (3,10)

Table A.1: Group II graph with  $n = \ell = 10$  and  $b = 3$

## A.2 Group II graphs for the MLST problem

Given inputs  $n = \ell$  and  $b$ , we can build a Group II graph as follows. Suppose the graph is  $G = (V, E, L)$ , where  $V = \{1, 2, \dots, n\}$ . First, we randomly select  $opt = \lceil \frac{n-1}{b} \rceil$  different labels from  $L$ . Second, for each of these labels, we assign it to  $b$  edges for a total of  $b \cdot opt$  edges. We must ensure that the  $b \cdot opt$  edges construct a connected subgraph of  $G$  and span all nodes in  $G$ . Thus, the  $\lceil \frac{n-1}{b} \rceil$  labels construct an optimal solution, since at least  $b \cdot opt$  edges are required for a feasible solution. Third, for each of the other labels in  $L$ , we assign it to  $b$  randomly selected edges. We now have the graph. An example is shown in Table A.1 with  $n = \ell = 10$  and  $b = 3$ . In this example, the optimal solution is  $\{5, 6, 8\}$ . It is indicated in bold.

### A.3 A small sample graph for LC-MST problem

This is a complete graph with 20 vertices and 20 labels. The set of vertices is  $V = \{0, 1, 2, \dots, 18, 19\}$ . The set of labels is  $L = \{0, 1, 2, \dots, 18, 19\}$ . For each vertex, its  $x$ - and  $y$ - coordinates are randomly selected in  $[0, 999]$ . The label matrix of the graph is a symmetric matrix. Each entry is randomly selected from  $L$ . The computational results are shown in Table 6.1. Table A.2 gives the coordinates of the nodes in the graph. Table A.3 gives the label matrix of the graph.

vertex	1	2	3	4	5	6	7	8	9	10
x	41	334	169	478	962	705	281	961	995	827
y	467	500	724	358	464	145	827	491	942	436
vertex	11	12	13	14	15	16	17	18	19	20
x	391	902	292	421	718	447	771	869	667	35
y	604	153	382	716	895	726	538	912	299	894

Table A.2: Coordinates of vertices for the graph with  $n = \ell = 20$



–	4	12	3	14	14	5	2	12	14	9	8	5	3	18	18	20	4	2	10
4	–	19	17	16	11	3	9	7	1	3	5	9	7	6	11	10	11	11	7
12	19	–	2	14	9	10	4	5	15	17	1	7	17	12	9	5	20	7	4
3	17	2	–	18	19	19	3	10	2	14	16	20	19	5	11	18	7	14	7
14	16	14	18	–	2	6	5	13	11	10	18	14	18	13	7	11	2	17	16
14	11	9	19	2	–	8	16	15	12	13	11	11	2	5	7	11	8	12	8
5	3	10	19	6	8	–	18	18	8	14	4	6	10	10	19	2	9	3	7
2	9	4	3	5	16	18	–	7	11	14	9	1	12	3	16	11	20	5	18
12	7	5	10	13	15	18	7	–	9	4	16	2	3	11	12	17	15	1	17
14	1	15	2	11	12	8	11	9	–	2	9	20	9	5	2	15	14	20	19
9	3	17	14	10	13	14	14	4	2	–	19	1	9	8	8	7	14	9	4
8	5	1	16	18	11	4	9	16	9	19	–	8	2	11	18	14	15	10	17
5	9	7	20	14	11	6	1	2	20	1	8	–	16	12	1	10	20	17	19
3	7	17	19	18	2	10	12	3	9	9	2	16	–	4	5	9	5	10	10
18	6	12	5	13	5	10	3	11	5	8	11	12	4	–	3	16	6	14	4
18	11	9	11	7	7	19	16	12	2	8	18	1	5	3	–	4	8	15	4
20	10	5	18	11	11	2	11	17	15	9	14	10	9	16	4	–	9	1	19
4	11	20	7	2	8	9	20	15	14	14	15	20	5	6	8	9	–	19	1
2	11	7	14	17	12	3	5	1	20	9	10	17	10	14	15	1	19	–	17
10	7	4	7	16	8	7	18	17	19	4	17	19	10	4	4	19	1	17	–

Table A.3: The label matrix for the graph with  $n = \ell = 20$

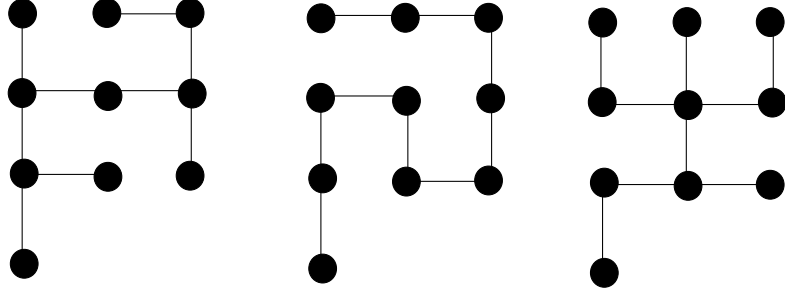


Figure A.1: Some minimum spanning trees in a graph with  $n = 10$

#### A.4 A special family of graphs for LC-MST problem

We construct a special family of graphs for which we know the optimal solution of the LC-MST problem. Given  $|V| = n$ ,  $|L| = \ell$ , and  $K$ , we construct the graph  $G$  as following. First, we set  $r = \lfloor \sqrt{n} \rfloor$  and  $s = \lfloor \frac{n}{r} \rfloor$ . Then we put  $rs$  nodes in an  $r$  by  $s$  array. Each node has coordinate  $(i, j)$ , where  $1 \leq i \leq s$  and  $1 \leq j \leq r$ . If  $n = rs$ , then the  $n$  nodes are set up; if  $n > rs$ , then we add a row with  $n - rs$  nodes. Each node has coordinate  $(s + 1, j)$ , where  $1 \leq j \leq n - rs$ . By easy observation, this graph has many minimum spanning trees. The total cost of each minimum spanning tree is  $n - 1$ . Figure A.1 shows some minimum spanning trees in a graph with 10 nodes. So we can randomly select a minimum spanning tree  $T$ . Next, we build a label set  $C$  with random  $K$  distinct labels from  $L$ . Then we color each edge in  $T$  with a random label in  $C$ . For other edges in  $G$ , we color them by random labels in  $L$ . For this special family of graphs, we may wonder whether the local optimal solution by LS1 or LS2 is always equal to the global optimal solution. The answer is NO. Figure A.2 shows the case such that the solution by LS1 or LS2 is not the global optimal solution.

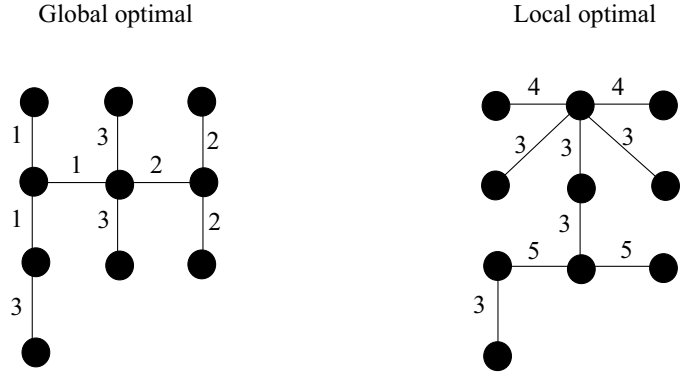


Figure A.2: An example where local optimal  $\neq$  global optimal

Next, we give the computational results of LS1, LS2, and GA for this special family of graphs. From the results, each of the three algorithms can reach the global optimal solutions. Table A.4 shows the results. In the table, we fixed  $K = 20$ .

## A.5 IP Formulation for the LC-MST Problem

This section was contributed by Si Chen (a fellow Ph.D. student). She provides an optimal approach by Integer Programming (IP) for the LC-MST problem. The details of the IP formulation are as follows.

### A.5.1 Notation and Variables

In the LC-MST problem, we assume that  $L$  is the set of all labels,  $E$  is the set of all edges,  $V$  is the set of all nodes.  $|V| = n$  is the total number of nodes.  $K$  is the maximum number of labels allowed in a solution. Let  $V = \{1, 2, \dots, n\}$ . For

	OPT	LS1	LS2	GA
$n = \ell = 50$	49	49	49	49
$n = \ell = 100$	99	99	99	99
$n = \ell = 120$	119	119	119	119
$n = \ell = 150$	149	149	149	149
$n = \ell = 180$	179	179	179	179
$n = \ell = 200$	199	199	199	199
$n = \ell = 250$	249	249	249	249
$n = \ell = 300$	299	299	299	299
$n = \ell = 350$	349	349	349	349
$n = \ell = 400$	399	399	399	399

Table A.4: Computation results for the special family of graphs with  $K = 20$

any  $1 \leq i, j \leq n$  and  $i \neq j$ , we define  $C_{ij}$  to be the cost of an edge; for any label  $k$  in  $L$ , we define the input  $Z_{ij}^k$  and the variable  $x_{ij}^k$  as follows.

$$Z_{ij}^k = \begin{cases} 1, & \text{if edge}(i,j) \text{ has label } k; \\ 0, & \text{otherwise.} \end{cases} \quad x_{ij}^k = \begin{cases} 1, & \text{if edge}(i,j) \text{ and label } k \text{ are used;} \\ 0, & \text{otherwise.} \end{cases}$$

#### A.5.2 IP Formulation

$$\begin{aligned} & \min \sum_{(i,j) \in E} C_{ij} \sum_{k \in K} x_{ij}^k \\ & \text{subject to } \sum_{(i,j) \in E} \sum_k x_{ij}^k = n - 1 \end{aligned} \tag{A.1}$$

$$\sum_{i:(i,j) \in E} f_{ij} - \sum_{l:(j,l) \in E} f_{jl} = 1 \quad \forall j \in V - \{1\} \quad (\text{A.2})$$

$$\sum_{i:(i,1) \in E} f_{i1} - \sum_{l:(1,l) \in E} f_{1l} = -(n-1) \quad (\text{A.3})$$

$$f_{ij} \leq (n-1) \cdot \sum_k x_{ij}^k \quad (\text{A.4})$$

$$f_{ji} \leq (n-1) \cdot \sum_k x_{ij}^k \quad (\text{A.5})$$

$$x_{ij}^k \leq Z_{ij}^k \quad (\text{A.6})$$

$$\sum_{(i,j) \in E} x_{ij}^k \leq (n-1) \cdot y_k \quad (\text{A.7})$$

$$\sum_k y_k \leq K \quad (\text{A.8})$$

$$x_{ij}^k, y_k \in B \quad (\text{A.9})$$

$$f_{ij} \geq 0. \quad (\text{A.10})$$

Constraint (A.1) ensures that the tree contains  $n-1$  edges. The flow variable  $f_{ij}$  is added to guarantee connectivity. It works as follows:

- Select one node (call it node 1) out of the set  $V$  and label it as the root node.  
Create a supply of  $n-1$  units of flow.
- For all other nodes except the root node, create a demand of 1 unit of flow.
- Send one unit of flow from the root to all other nodes to satisfy their demands.

Constraints (A.2) and (A.3) are the flow balance constraints and they state that the total net flow into a node other than the root should equal to 1. Constraints (A.4) and (A.5) ensure that if flow is sent on an edge, then this edge must be chosen. Constraint (A.6) says that an edge and color combination can be selected only if it

belongs to the input graph. Constraint(A.7) ensures that if an edge uses a label, this label must be selected. Constraint (A.8) restricts the number of labels that can be selected. The IP formulation is implemented using OPL Studio 3.7 on a Pentium III PC with 993MHZ and 512MB RAM.

## A.6 Backtrack Search

In this section, we describe the backtrack search method to get optimal solutions for the MLST problem, the LC-MST problem, and the CTSP. The basic idea of the backtrack search is to exhaust all the possible solutions in some order, while truncating most of the worse solutions by some constraints. The details for each problem are as follows.

### A.6.1 Backtrack Search for the MLST problem

From Chapters 3 and 4, we know that the MLST problem is NP-hard and the solution is a set of labels. The decision question of the MLST problem is: Given a positive integer  $k$ , can we find a feasible solution with  $k$  distinct labels? To answer this question, we need to check all possible combinations of  $k$  labels out of  $|L| = \ell$  labels. There are  $\binom{\ell}{k}$  possibilities. When  $k$  is increasing,  $\binom{\ell}{k}$  is increasing exponentially. (According to Sterling's formula, when  $k < \frac{\ell}{2}$ ,  $\binom{\ell}{k} = O(\frac{(\ell/e)^\ell}{(k/e)^k((\ell-k)/e)^{\ell-k}}) = O(\frac{\ell^\ell}{(k(\ell-k))^k(\ell-k)^{\ell-2k}}) \geq O(\frac{\ell^\ell}{(\ell/2)^{2k}(\ell-k)^{\ell-2k}}) \geq O(2^{2k}).)$

The backtrack search tries to reduce as many possibilities as possible. We know, for each label set  $C = \{c_1, c_2, \dots, c_k\}$  with  $k$  distinct labels, if it is a feasible solution, it must satisfy the inequality

$$\sum_{i=1}^k f(c_i) \geq n - 1 \quad (\text{A.11})$$

where  $f(c_i)$  is the frequency of the label  $c_i$  in the graph and  $n$  is the number of the nodes of the graph. The inequality is an constraint. In the backtrack search, we sort all the labels in  $L$  by their frequencies. We can assume that  $L = \{c_1, c_2, \dots, c_\ell\}$ , where  $f(c_1) \geq f(c_2) \geq \dots \geq f(c_\ell)$ . Suppose  $\{c_{i_1}, \dots, c_{i_k}\}$  is not feasible, where  $i_1 < \dots < i_k$ . Then we check inequality A.11 for this label set. If it does not satisfy A.11, then all the label sets with the form  $\{c_{i_1}, \dots, c_{i_{k-1}}, c_t\}$ , where  $t > i_k$ , are also not feasible. So we do not have to check the feasibilities of many possible label sets. With respect to the inequality A.11, the backtrack search can reduce many combinations of label sets.

The optimal algorithm of the MLST problem applies the backtrack search method with  $k$  ranging from 1 to the number of distinct labels in the first feasible solution. For small cases with  $n \leq 50$ , we can find optimal solutions for all instances in a reasonable amount of time. For large cases, we can find some optimal solutions, but not all, in a reasonable amount of time. The computational results are described in Chapter 4.

### A.6.2 Backtrack Search for the LC-MST problem

From Chapter 6, we know that the solution of the LC-MST problem is also determined by a label set. So, we can also use the same backtrack search method from Section A.6.1 with respect to the inequality A.11. But when the constraint  $K$  is much greater than the number of the labels in the MLST, the inequality A.11 does not help much to reduce the number of label sets examined. Therefore, the backtrack search does not work as well as IP in Section A.5. We were able to solve problems of size  $n = 25$  using the backtrack search method in a maximum of 2892 seconds of CPU time.

### A.6.3 Backtrack Search for the CTSP

The CTSP is similar to the MLST problem. Instead of a spanning tree, we try to find a tour or a Hamiltonian cycle of the graph with the smallest number of distinct labels. We can use essentially the same backtrack search method to find a feasible solution. But in the subgraph induced by a feasible solution, a Hamiltonian cycle may not exist. The Hamiltonian Cycle problem is also NP-hard.

To find a Hamiltonian cycle in a subgraph, we can use a recursive algorithm. In this recursive algorithm, we begin with an arbitrary node. Then, we track its neighbors and find an unvisited node as the next node. We continue this procedure until one of two cases happens. One case is that all the nodes in the subgraph are visited, i.e., a Hamiltonian path is obtained. There also exists an edge connecting the two end nodes of the Hamiltonian path. Then, we add this edge and get a



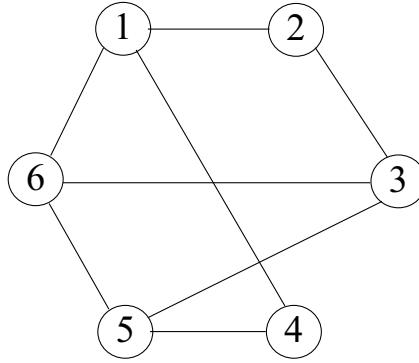


Figure A.3: A sample graph for the recursive algorithm

Hamiltonian cycle. The other case is that we do not finish the tour, since at some node  $v$  we find that all its neighbors are visited. In this case, we have to go back to its parent node  $w$  and find another unvisited child node  $u$  of  $w$ , i.e.,  $u$  and  $v$  are brothers. This is a recursive procedure and requires exponential running time. If we fail to find a tour in the end, then the label set is not a solution. Figure A.3 shows an example. In Figure A.3, we begin with node 1, then we get a partial tour  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4$ , which fails because all the neighbors of node 4 are visited. Then we go back to the previous node 5 and get a partial tour  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$ , which also fails because all the neighbors of 6 are visited. Then we go back to the node 5 and then back to 3 and get a Hamiltonian path  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 4$ . Now all the nodes are visited. The edge  $(1, 4)$  exists. So, finally we get a Hamiltonian tour  $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 1$ . If the edge  $(1, 4)$  does not exist, then this subgraph has no Hamiltonian tour. Thus, the corresponding label set is not a solution.

In the computational results, because the above recursive algorithm requires

much time, we only test the input  $n$  and  $\ell$  up to 25. In the worst case, 2892 seconds are required to work out an optimal solution for an instance with  $n = \ell = 25$ .

## A.7 Source Code

```

// BuildDGraph.cpp
// Build density labeled graph for the MLST problems.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VN 500
#define CN 500

class Graph {
public: int c[VN][VN];
int cn;
int vn;
public: void InitGraph(int vk, int ck, float d);
void PrintGraph();
void RecordToFile(FILE *fp);
private: bool IsConnected();
};

void Graph::InitGraph(int vk, int ck, float d) {
int i,j;
int k,total;

if (d==1) {
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
if (i==j) c[i][j]=cn;
else c[i][j]=c[j][i]=rand()%cn;
}
}
return;
}
cn=ck; vn=vk;
total=(int) ((vn/2)*(vn-1)+0.0)*d;
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=c[j][i]=cn;
}
}
k=0;
while (k<total) {
i=rand()%vn; j=rand()%vn;
if (i==j) continue;
if (c[i][j]<cn) continue;
c[i][j]=c[j][i]=rand()%cn;
k++;
}
}

```

```

}
while (!IsConnected()){
i=rand()%vn; j=rand()%vn;
if (i==j) continue;
if (c[i][j]<cn) continue;
c[i][j]=c[j][i]=rand()%cn;
//printf("%d-%d(%d)\n",i,j,c[i][j]);
}
}

void Graph::PrintGraph() {
int i,j;

for (i=0; i<vn; i++) {
printf("%d: ",i);
for (j=0; j<vn; j++) {
printf("%d->%d(%d) ",i,j,c[i][j]);
}
printf("\n");
}
}

void Graph::RecordToFile(FILE *fp) {
int i,j;

for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {
fprintf(fp, "%d ",c[i][j]);
}
fprintf(fp,"\n");
}
}

bool Graph::IsConnected() {
int i,j,k;
int temp[VN];
bool f[VN];

for (i=0; i<vn; i++) f[i]=false;
temp[0]=0; k=1;
while (k>0) {
i=temp[k-1]; f[i]=true; k--;
for (j=0; j<vn; j++) {
if (c[i][j]==cn) continue;
if (f[j]) continue;
temp[k]=j; k++;
}
}
}

```

```

}
for (i=0; i<vn; i++) {
if (!f[i]) return false;
}
return true;
}

void main() {
int cn,vn,sn;
float density;
char buf[20];
int i;
Graph G;
FILE *fp;

printf("Number of vertices: ");
scanf("%d",&vn);
printf("Number of colors: ");
scanf("%d",&cn);
printf("Density: ");
scanf("%f",&density);
printf("Number of samples: ");
scanf("%d",&sn);
if (density==1) sprintf(buf,"CGraph%d_%d.txt",vn,cn);
else if (density<0.45) sprintf(buf,"LDGraph%d_%d.txt",vn,cn);
else if (density>0.55) sprintf(buf,"HDGraph%d_%d.txt",vn,cn);
else sprintf(buf,"MDGraph%d_%d.txt",vn,cn);
fp=fopen(buf,"wt");
if (fp==NULL) {
printf("cannot open file %s\n",buf);
return;
}
fprintf(fp,"%d %d\n",vn,cn);
for (i=0; i<sn; i++) {
G.InitGraph(vn,cn,density);
//G.PrintGraph();
G.RecordToFile(fp);
}
fclose(fp);
}

```

```

// BuildBGraph.cpp
// Build graphs with bounded label frequency
// for the MLST problems.
// An optimal solution is created in the graph.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VN 500
#define CN 500

class Graph {
public: int c[VN][VN];
int cn;
int vn;
int bn;
public: void InitGraph(int vk, int ck, int bk); // bk is the frequency bound
void PrintGraph();
void RecordToFile(FILE *fp);
private: bool IsConnected();
};

void Graph::InitGraph(int vk, int ck, int bk) {
int i,j,k;
int counter[CN];
int path[VN];
bool visited[VN];
int current_label;

cn=ck; vn=vk; bn=bk;
for (i=0; i<cn; i++) counter[i]=0;
for (i=0; i<vn; i++) visited[i]=false;

for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=c[j][i]=cn;
}
}
// build an preset solution with bk labels
current_label=rand()%cn;
counter[current_label]=0;
path[0]=rand()%vn; visited[path[0]]=true;
i=1;
while (i<vn) {
path[i]=rand()%vn;
while (visited[path[i]]) path[i]=(path[i]+1)%vn;

```

```

visited[path[i]]=true;
j=rand()%i;
c[path[i]][path[j]]=c[path[j]][path[i]]=current_label;
counter[current_label]++;
if (counter[current_label]>=bn) {
current_label=rand()%cn;
while (counter[current_label]>=bn) current_label=(current_label+1)%cn;
counter[current_label]=0;
}
i++;
}
// Set random other labels with frequency bn for other edges
for (k=0; k<cn; k++) {
if (counter[k]>0) continue;
while (counter[k]<bn) {
i=rand()%vn; j=rand()%vn;
while ((i==j) || c[i][j]<cn) {
j=(j+1)%vn;
if (j==0) i=(i+1)%vn;
}
c[i][j]=c[j][i]=k; counter[k]++;
}
}
}

void Graph::PrintGraph() {
int i,j;

for (i=0; i<vn; i++) {
printf("%d: ",i);
for (j=0; j<vn; j++) {
printf("%d->%d(%d) ",i,j,c[i][j]);
}
printf("\n");
}
}

void Graph::RecordToFile(FILE *fp) {
int i,j;
int counter[CN];

for (i=0; i<cn; i++) counter[i]=0;
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {
fprintf(fp, "%d ",c[i][j]);
counter[c[i][j]]++;
}
}
}

```

```

fprintf(fp, "\n");
}
/*printf("Freq: \n");
for (i=0; i<cn; i++) printf("%d ", counter[i]);
printf("\n");*/
}

bool Graph::IsConnected() {
int i, j, k;
int temp[VN];
bool f[VN];

for (i=0; i<vn; i++) f[i]=false;
temp[0]=0; k=1;
while (k>0) {
i=temp[k-1]; f[i]=true; k--;
for (j=0; j<vn; j++) {
if (c[i][j]==cn) continue;
if (f[j]) continue;
temp[k]=j; k++;
}
}
for (i=0; i<vn; i++) {
if (!f[i]) return false;
}
return true;
}

void main() {
int cn, bn, vn, sn;
char buf[20];
int i;
Graph G;
FILE *fp;

printf("Number of vertices: ");
scanf("%d", &vn);
printf("Number of colors: ");
scanf("%d", &cn);
printf("Bound of the frequency: ");
scanf("%d", &bn);
printf("Number of samples: ");
scanf("%d", &sn);
sprintf(buf, "BGraph%d_%d_%d.txt", vn, cn, bn);
fp=fopen(buf, "wt");
if (fp==NULL) {
printf("cannot open file %s\n", buf);
}
}

```



```
return;
}
printf("vn=%d, cn=%d, bn=%d, sn=%d\n",vn,cn,bn,sn);
fprintf(fp,"%d %d\n",vn,cn);
for (i=0; i<20; i++) {
G.InitGraph(vn,cn,bn);
//G.PrintGraph();
G.RecordToFile(fp);
}
fclose(fp);
}
```

```

// MLST_MVCA.cpp
// Solve MLST by MVCA heuristic

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
void PrintSolution(); // Print the MVCA solution
int Eval(); // Find the value of the solution
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
void MVCA(Graph G); // MVCA heuristic
public: int NumComponents();
// Find the number of connected components of the graph

```

```

private: void Sort1(); // Sort by the general frequency
        void Sort2(); // Sort by the net frequency
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::Sort2() {
int i,j;
int temp;

```

```

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;
edge *e,*e1,*e2;

for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
// Add edges in the empty graph
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) continue;
k=ReadNumber(fp);
if (k==cn) continue;
// Add color k for edge (i,j)
e=new edge;
e->u=i; e->v=j; e->c=k; e->next=NULL;
e1=L[k].root;
if (e1==NULL) L[k].root=e;
else {
e2=e1;
while (e1!=NULL) { e2=e1; e1=e1->next; }
e2->next=e;
}
L[k].en++;
}
}
Sort1();
}

```

```

}

void Graph::InitEmptyGraph(Graph G) {
int i;

vn=G.vn; cn=G.cn;
for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

```

```

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

}

void Graph::MVCA(Graph G) {
int k,num;
int i,c;
bool flag;
bool temp[CN];

RemoveAllColors();
num=vn; c=0; flag=true;
for (i=0; i<G.cn; i++) temp[i]=false;
while (flag) {
for (i=0; i<G.cn; i++) {
if (temp[i]) continue;
AddColor(G,i);
k=NumComponents();
if (k<num) {
c=i; num=k;
}
RemoveColor(i);
}
AddColor(G,c); temp[c]=true;
if (num==1) flag=false;
}
/*printf("MVCA Solution: ");
for (i=0; i<G.cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");*/
}

void Graph::PrintSolution() {
int i;

printf("MVCA Solution: ");
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");
}

int Graph::Eval() {
int k,i;

k=0;
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) k++;
}
}

```

```

}
return k;
}

void main() {
FILE *fp;
Graph G,H;
int k;
int gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Group size: ");
scanf("%d",&gn);
printf("====Result====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
H.InitEmptyGraph(G);
value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
//u1=time(NULL);
H.MVCA(G);
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,H.Eval());
H.PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+H.Eval();
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```



```

// MLST_GA01.cpp
// Solve MLST by Genetic Algorithm (GA) using the general frequency
// for each label

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
public: int NumComponents();
// Find the number of connected components of the graph
private: void Sort1(); // Sort by the general frequency
void Sort2(); // Sort by the net frequency

```

```

};

class Solution {
public: int C[CN]; // set of colors
int cn; // number of colors
Graph H; // Corresponding graph
public: void InitSolution(Graph G); // Initialize a solution
void PrintSolution(); // Print a solution
void Backup(); // Back up the solution
void Restore(Graph G); // Restore the solution
bool IsBetter(); // Check if a better solution is achieved
void Mutation1(Graph G);
// Local 1-search, which removes one color
void Mutation2(Graph G);
// Add one color, then do mutation1
void Crossover1(Graph G, Solution t);
// Combine two color set and find a new solution from this set
private: void Sort1(Graph G);
// Sort a solution by the general frequency of the color.
void Sort2(Graph G);
// Sort a solution by the net frequency of the color.
bool IsExist(int c);
// Check if the color c already exists
private: int backup_C[CN];
int backup_cn;
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

```

```

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;
edge *e,*e1,*e2;

for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {

```

```

L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
// Add edges in the empty graph
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) continue;
k=ReadNumber(fp);
if (k==cn) continue;
// Add color k for edge (i,j)
e=new edge;
e->u=i; e->v=j; e->c=k; e->next=NULL;
e1=L[k].root;
if (e1==NULL) L[k].root=e;
else {
e2=e1;
while (e1!=NULL) { e2=e1; e1=e1->next; }
e2->next=e;
}
L[k].en++;
}
}
Sort1();
}

void Graph::InitEmptyGraph(Graph G) {
int i;

vn=G.vn; cn=G.cn;
for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;

```

```

while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

```

```

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

bool Solution::IsExist(int c) {
int i;

for (i=0; i<cn; i++) {
if (c==C[i]) return true;
}
return false;
}

void Solution::Sort1(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].en; f2=G.L[C[j]].en;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}

```

```

}

void Solution::Sort2(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].freq; f2=G.L[C[j]].freq;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}

void Solution::InitSolution(Graph G) {
int c;

H.InitEmptyGraph(G);
cn=0;
while (H.NumComponents(>1) {
c=rand()%G.cn;
while (IsExist(c)) {
c=rand()%G.cn;
}
if (G.L[c].root==NULL) continue;
H.AddColor(G,c);
C[cn]=c; cn++;
}
Sort1(G);
}

void Solution::PrintSolution() {
printf("Solution: ");
for (int i=0; i<cn; i++) printf("%d ",C[i]);
printf("\n");
}

void Solution::Backup() {
int i;

for (i=0; i<cn; i++) backup_C[i]=C[i];
backup_cn=cn;
}

```

```

void Solution::Restore(Graph G) {
int i;

H.RemoveAllColors();
for (i=0; i<backup_cn; i++) {
C[i]=backup_C[i];
H.AddColor(G,C[i]);
}
cn=backup_cn;
}

bool Solution::IsBetter() {
if (cn<backup_cn) return true;
return false;
}

void Solution::Mutation1(Graph G) {
int i,j;
int c;
int temp[CN];
bool flag[CN];

for (i=0; i<cn; i++) {
temp[i]=C[i]; flag[i]=true;
}
i=cn-1;
while (i>=0) {
c=temp[i];
H.RemoveColor(c);
if (H.NumComponents()<=1) {
flag[i]=false;
}
else H.AddColor(G,c);
i--;
}
j=0;
for (i=0; i<cn; i++) {
if (flag[i]) {
C[j]=temp[i]; j++;
}
}
cn=j;
}

void Solution::Mutation2(Graph G) {

```



```

int c,k;

c=rand()%G.cn; k=0;
while (IsExist(c)) {
c=(c+1)%G.cn;
k++;
if (k>=G.cn) break;
}
H.AddColor(G,c);
C[cn]=c; cn++;
Sort1(G);
Mutation1(G);
}

void Solution::Crossover1(Graph G, Solution t) {
int i;
int c;
int temp[CN];

H.RemoveAllColors();
for (i=0; i<t.cn; i++) {
c=t.C[i];
if (!IsExist(c)) {
C[cn]=c; cn++;
}
}
Sort1(G);
for (i=0; i<cn; i++) {
temp[i]=C[i];
}
cn=0; i=0;
while (H.NumComponents(>1) {
c=temp[i]; i++;
H.AddColor(G,c);
C[cn]=c; cn++;
}
}

Solution S[PN];

void InitPopulation(Graph G, int pn) { // Initial the population
for (int i=0; i<pn; i++) {
S[i].InitSolution(G);
}
}

int GenValue(int pn) {

```

```

// Find the member with the best value in each generation
int i,v;
int j;

v=VN; j=0;
for (i=0; i<pn; i++) {
if (v>S[i].cn) { v=S[i].cn; j=i; }
}
return j;
}

void main() {
FILE *fp;
Graph G;
int i,j,k,v;
int pn,gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Population size: ");
scanf("%d",&pn);
printf("Group size: ");
scanf("%d",&gn);
printf("====Result====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
InitPopulation(G,pn);
for (i=1; i<pn; i++) {
for (j=0; j<pn; j++) {
int t1,t2;
t1=j; t2=(j+i)%pn;
S[t1].Backup();

```

```

S[t1].Crossover1(G,S[t2]);
S[t1].Mutation2(G);
if (!S[t1].IsBetter()) S[t1].Restore(G);
}
v=GenValue(pn);
/*printf("Generation %d(%d): ",i,S[v].cn);
S[v].PrintSolution();*/
}
v=GenValue(pn);
G.ClearAll();
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,S[v].cn);
S[v].PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+S[v].cn;
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```

```

// MLST_GA02.cpp
// Solve MLST by Genetic Algorithm (GA) using the net frequency
// for each label

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
public: int NumComponents();
// Find the number of connected components of the graph
private: void Sort1(); // Sort by the general frequency
void Sort2(); // Sort by the net frequency

```

```

    void GetNetFreq(int k); // Get the net frequency for color k
};

class Solution {
public: int C[CN]; // set of colors
int cn; // number of colors
Graph H; // Corresponding graph
public: void InitSolution(Graph G); // Initialize a solution
void PrintSolution(); // Print a solution
void Backup(); // Back up the solution
void Restore(Graph G); // Restore the solution
bool IsBetter(); // Check if a better solution is achieved
void Mutation1(Graph G);
// Local 1-search, which removes one color
void Mutation2(Graph G);
// Add one color, then do mutation1
void Crossover1(Graph G, Solution t);
// Combine two color set and find a new solution from this set
private: void Sort1(Graph G);
// Sort a solution by the general frequency of the color.
void Sort2(Graph G);
// Sort a solution by the net frequency of the color.
bool IsExist(int c);
// Check if the color c already exists
private: int backup_C[CN];
int backup_cn;
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

```

```

}

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::GetNetFreq(int k) {
int i,j,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
if (L[k].root==NULL) {
L[k].freq=0; return;
}
}

```

```

}
e=L[k].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int m=0; m<vn; m++)
if (vertex[m]==j) vertex[m]=i;
}
else {
for (m=0; m<vn; m++)
if (vertex[m]==i) vertex[m]=j;
}
e=e->next;
}
m=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) m++;
L[k].freq=vn-m;
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;
edge *e,*e1,*e2;

for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
// Add edges in the empty graph
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) continue;
k=ReadNumber(fp);
if (k==cn) continue;
// Add color k for edge (i,j)
e=new edge;
e->u=i; e->v=j; e->c=k; e->next=NULL;
e1=L[k].root;
if (e1==NULL) L[k].root=e;
else {
e2=e1;
while (e1!=NULL) { e2=e1; e1=e1->next; }
}
}
}
}

```

```

e2->next=e;
}
L[k].en++;
}
}
for (i=0; i<cn; i++) GetNetFreq(i);
Sort2();
}

void Graph::InitEmptyGraph(Graph G) {
int i;

vn=G.vn; cn=G.cn;
for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {

```



```

L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
}
}
}

```

```

e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

bool Solution::IsExist(int c) {
int i;

for (i=0; i<cn; i++) {
if (c==C[i]) return true;
}
return false;
}

void Solution::Sort1(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].en; f2=G.L[C[j]].en;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}

void Solution::Sort2(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].freq; f2=G.L[C[j]].freq;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}
}

```

```

void Solution::InitSolution(Graph G) {
    int c;

    H.InitEmptyGraph(G);
    cn=0;
    while (H.NumComponents(>1) {
        c=rand()%G.cn;
        while (IsExist(c)) {
            c=rand()%G.cn;
        }
        if (G.L[c].root==NULL) continue;
        H.AddColor(G,c);
        C[cn]=c; cn++;
    }
    Sort2(G);
}

void Solution::PrintSolution() {
    printf("Solution: ");
    for (int i=0; i<cn; i++) printf("%d ",C[i]);
    printf("\n");
}

void Solution::Backup() {
    int i;

    for (i=0; i<cn; i++) backup_C[i]=C[i];
    backup_cn=cn;
}

void Solution::Restore(Graph G) {
    int i;

    H.RemoveAllColors();
    for (i=0; i<backup_cn; i++) {
        C[i]=backup_C[i];
        H.AddColor(G,C[i]);
    }
    cn=backup_cn;
}

bool Solution::IsBetter() {
    if (cn<backup_cn) return true;
    return false;
}

```

```

void Solution::Mutation1(Graph G) {
    int i,j;
    int c;
    int temp[CN];
    bool flag[CN];

    for (i=0; i<cn; i++) {
        temp[i]=C[i]; flag[i]=true;
    }
    i=cn-1;
    while (i>=0) {
        c=temp[i];
        H.RemoveColor(c);
        if (H.NumComponents()<=1) {
            flag[i]=false;
        }
        else H.AddColor(G,c);
        i--;
    }
    j=0;
    for (i=0; i<cn; i++) {
        if (flag[i]) {
            C[j]=temp[i]; j++;
        }
    }
    cn=j;
}

void Solution::Mutation2(Graph G) {
    int c,k;

    c=rand()%G.cn; k=0;
    while (IsExist(c)) {
        c=(c+1)%G.cn;
        k++;
        if (k>=G.cn) break;
    }
    H.AddColor(G,c);
    C[cn]=c; cn++;
    Sort2(G);
    Mutation1(G);
}

void Solution::Crossover1(Graph G, Solution t) {
    int i;

```

```

int c;
int temp[CN];

H.RemoveAllColors();
for (i=0; i<t.cn; i++) {
c=t.C[i];
if (!IsExist(c)) {
C[cn]=c; cn++;
}
}
Sort2(G);
for (i=0; i<cn; i++) {
temp[i]=C[i];
}
cn=0; i=0;
while (H.NumComponents(>1) {
c=temp[i]; i++;
H.AddColor(G,c);
C[cn]=c; cn++;
}
}

Solution S[PN];

void InitPopulation(Graph G, int pn) { // Initial the population
for (int i=0; i<pn; i++) {
S[i].InitSolution(G);
}
}

int GenValue(int pn) {
// Find the member with the best value in each generation
int i,v;
int j;

v=VN; j=0;
for (i=0; i<pn; i++) {
if (v>S[i].cn) { v=S[i].cn; j=i; }
}
return j;
}

void main() {
FILE *fp;
Graph G;
int i,j,k,v;
int pn,gn;

```

```

int value;
char filename[40];
double AvgValue;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Population size: ");
scanf("%d",&pn);
printf("Group size: ");
scanf("%d",&gn);
printf("====Result====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
value=0;
for (k=0; k<gn; k++) {
G.InitGraph(fp);
/*printf("Freq: ");
for (i=0; i<G.cn; i++) printf("%d(%d) ",i,G.L[i].freq);
printf("\n");*/
//G.PrintGraph();
InitPopulation(G,pn);
for (i=1; i<pn; i++) {
for (j=0; j<pn; j++) {
int t1,t2;
t1=j; t2=(j+i)%pn;
S[t1].Backup();
S[t1].Crossover1(G,S[t2]);
S[t1].Mutation2(G);
if (!S[t1].IsBetter()) S[t1].Restore(G);
}
}
v=GenValue(pn);
/*printf("Generation %d(%d): ",i,S[v].cn);
S[v].PrintSolution();*/
}
v=GenValue(pn);
G.ClearAll();
printf("Group %d: value=%d\n",k+1,S[v].cn);
S[v].PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+S[v].cn;
}
AvgValue=(value+0.0)/gn;

```

```
printf("Average Value: %f\n",AvgValue);  
fclose(fp);  
}
```

```

// MLST_MGA.cpp
// Solve MLST by Modified Genetic Algorithm (MGA)

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
public: int NumComponents();
// Find the number of connected components of the graph
private: void Sort1(); // Sort by the general frequency
void Sort2(); // Sort by the net frequency
};

```



```

class Solution {
public: int C[CN]; // set of colors
int cn; // number of colors
Graph H; // Corresponding graph
public: void InitSolution(Graph G); // Initialize a solution
void PrintSolution(); // Print a solution
void Backup(); // Back up the solution
void Restore(Graph G); // Restore the solution
bool IsBetter(); // Check if a better solution is achieved
void Mutation1(Graph G);
// Local 1-search, which removes one color
void Mutation2(Graph G);
// Add one color, then do mutation1
void Crossover1(Graph G, Solution t);
// Combine two color set and find a new solution from this set
private: void Sort1(Graph G);
// Sort a solution by the general frequency of the color.
void Sort2(Graph G);
// Sort a solution by the net frequency of the color.
bool IsExist(int c);
// Check if the color c already exists
private: int backup_C[CN];
int backup_cn;
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

```

```

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;
edge *e,*e1,*e2;

for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
}
}

```

```

L[i].c=i; L[i].root=NULL;
}
// Add edges in the empty graph
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) continue;
k=ReadNumber(fp);
if (k==cn) continue;
// Add color k for edge (i,j)
e=new edge;
e->u=i; e->v=j; e->c=k; e->next=NULL;
e1=L[k].root;
if (e1==NULL) L[k].root=e;
else {
e2=e1;
while (e1!=NULL) { e2=e1; e1=e1->next; }
e2->next=e;
}
L[k].en++;
}
}
Sort1();
}

void Graph::InitEmptyGraph(Graph G) {
int i;

vn=G.vn; cn=G.cn;
for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {

```

```

printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

```

```

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

bool Solution::IsExist(int c) {
int i;

for (i=0; i<cn; i++) {
if (c==C[i]) return true;
}
return false;
}

```

```

void Solution::Sort1(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].en; f2=G.L[C[j]].en;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}
}

```

```

void Solution::Sort2(Graph G) {
int i,j;
int temp;

for (i=0; i<cn-1; i++) {
for (j=i; j<cn; j++) {
int f1,f2;
f1=G.L[C[i]].freq; f2=G.L[C[j]].freq;
if (f1<=f2) {
temp=C[i]; C[i]=C[j]; C[j]=temp;
}
}
}
}

void Solution::InitSolution(Graph G) {
int c;

H.InitEmptyGraph(G);
cn=0;
while (H.NumComponents(>1) {
c=rand()%G.cn;
while (IsExist(c)) {
c=rand()%G.cn;
}
if (G.L[c].root==NULL) continue;
H.AddColor(G,c);
C[cn]=c; cn++;
}
//Sort1(G);
}

void Solution::PrintSolution() {
printf("Solution: ");
for (int i=0; i<cn; i++) printf("%d ",C[i]);
printf("\n");
}

void Solution::Backup() {
int i;

for (i=0; i<cn; i++) backup_C[i]=C[i];
backup_cn=cn;
}

```

```

void Solution::Restore(Graph G) {
int i;

H.RemoveAllColors();
for (i=0; i<backup_cn; i++) {
C[i]=backup_C[i];
H.AddColor(G,C[i]);
}
cn=backup_cn;
}

bool Solution::IsBetter() {
if (cn<backup_cn) return true;
return false;
}

void Solution::Mutation1(Graph G) {
int i,j;
int c;
int temp[CN];
bool flag[CN];

for (i=0; i<cn; i++) {
temp[i]=C[i]; flag[i]=true;
}
i=cn-1;
while (i>=0) {
c=temp[i];
H.RemoveColor(c);
if (H.NumComponents()<=1) {
flag[i]=false;
}
else H.AddColor(G,c);
i--;
}
j=0;
for (i=0; i<cn; i++) {
if (flag[i]) {
C[j]=temp[i]; j++;
}
}
cn=j;
}

void Solution::Mutation2(Graph G) {
int c,k;

```

```

int i;

c=rand()%G.cn; k=0;
while (IsExist(c)) {
c=(c+1)%G.cn;
k++;
if (k>=G.cn) break;
}
H.AddColor(G,c);
for (i=cn; i>0; i--) C[i]=C[i-1];
C[0]=c; cn++;
Mutation1(G);
}

void Solution::Crossover1(Graph G, Solution t) {
int i,j;
int c;
int num,k;
int temp[CN];
bool flag[CN];
bool flag1;

H.RemoveAllColors();
for (i=0; i<t.cn; i++) {
c=t.C[i];
if (!IsExist(c)) {
C[cn]=c; cn++;
}
}
//Sort1(G);
for (i=0; i<cn; i++) {
temp[i]=C[i];
flag[i]=false;
}
j=0; num=G.vn; flag1=true;
while (flag1) {
for (i=0; i<cn; i++) {
if (flag[i]) continue;
H.AddColor(G,temp[i]);
k=H.NumComponents();
if (k<num) { c=i; num=k; }
H.RemoveColor(temp[i]);
}
C[j]=temp[c]; j++;
H.AddColor(G,temp[c]); flag[c]=true;
if (num==1) flag1=false;
}
}

```



```

cn=j;
}

Solution S[PN];

void InitPopulation(Graph G, int pn) { // Initial the population
for (int i=0; i<pn; i++) {
S[i].InitSolution(G);
}
}

int GenValue(int pn) { // Find the member with the best value in each generation
int i,v;
int j;

v=VN; j=0;
for (i=0; i<pn; i++) {
if (v>S[i].cn) { v=S[i].cn; j=i; }
}
return j;
}

void main() {
FILE *fp;
Graph G;
int i,j,k,v;
int pn,gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Population size: ");
scanf("%d",&pn);
printf("Group size: ");
scanf("%d",&gn);
printf("====Result====\n");
G.vn=G.ReadNumber(fp);

```

```

G.cn=G.ReadNumber(fp);
value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
InitPopulation(G,pn);
for (i=1; i<=pn/2; i++) {
for (j=0; j<pn; j++) {
int t1,t2;
t1=j; t2=(j+i)%pn;
S[t1].Backup();
S[t1].Crossover1(G,S[t2]);
S[t1].Mutation2(G);
if (!S[t1].IsBetter()) S[t1].Restore(G);
}
v=GenValue(pn);
/*printf("Generation %d(%d): ",i,S[v].cn);
S[v].PrintSolution();*/
}
v=GenValue(pn);
G.ClearAll();
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,S[v].cn);
S[v].PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+S[v].cn;
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```

```

// MLST_MVCA01.cpp
// Solve MLST by the first version of Modified MVCA (MVCA1)
// using 100% of all labels as the pilot label.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
int best[10];
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
void PrintSolution(); // Print the MVCA solution
int Eval(); // Find the value of the solution
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
void CopyGraph(Graph G);

```

```

void MVCA(Graph G, int color); // MVCA heuristic
public: int NumComponents();
// Find the number of connected components of the graph
private: void Sort1(); // Sort by the general frequency
        void Sort2(); // Sort by the net frequency
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

```

```

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;
edge *e,*e1,*e2;

for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
// Add edges in the empty graph
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) continue;
k=ReadNumber(fp);
if (k==cn) continue;
// Add color k for edge (i,j)
e=new edge;
e->u=i; e->v=j; e->c=k; e->next=NULL;
e1=L[k].root;
if (e1==NULL) L[k].root=e;
else {
e2=e1;
while (e2!=NULL) { e2=e2->next; }
e2->next=e;
}
L[k].en++;
}
}
}

```

```

}
}
Sort1();
}

void Graph::InitEmptyGraph(Graph G) {
int i;

vn=G.vn; cn=G.cn;
for (i=0; i<vn; i++) {
vertex[i]=i;
}
for (i=0; i<cn; i++) {
L[i].en=0; L[i].freq=0;
L[i].c=i; L[i].root=NULL;
}
}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

```

```

void Graph::CopyGraph(Graph H) {
int i;

RemoveAllColors();
for (i=0; i<H.cn; i++) {
if (H.L[i].root!=NULL) {
AddColor(H,i);
}
}
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
}
}
}

```

```

if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

void Graph::MVCA(Graph G, int color) {
int k,num;
int i,c;
bool flag;
bool temp[CN];

RemoveAllColors();
AddColor(G,color);
num=NumComponents(); c=0; flag=true;
if (num==1) return;
for (i=0; i<G.cn; i++) temp[i]=false;
temp[color]=true;
while (flag) {
for (i=0; i<G.cn; i++) {
if (temp[i]) continue;
AddColor(G,i);
k=NumComponents();
if (k<num) {
c=i; num=k;
}
RemoveColor(i);
}
AddColor(G,c); temp[c]=true;
if (num==1) flag=false;
}
/*printf("MVCA Solution: ");
for (i=0; i<G.cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");*/

```



```

}

void Graph::PrintSolution() {
int i;

printf("MVCA Solution: ");
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");
}

int Graph::Eval() {
int k,i;

k=0;
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) k++;
}
return k;
}

void main() {
FILE *fp;
Graph G,H,H1;
int k,i;
int gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Group size: ");
scanf("%d",&gn);
printf("=====Result=====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
H.InitEmptyGraph(G);
H1.InitEmptyGraph(G);

```

```

value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
H.MVCA(G,0);
for (i=0; i<G.cn; i++) {
H1.MVCA(G,i);
if (H1.Eval()<H.Eval()) {
H.CopyGraph(H1);
}
}
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,H.Eval());
H.PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+H.Eval();
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```

```

// MLST_MVCA02.cpp
// Solve MLST by the second version of Modified MVCA (MVCA2)
// using the top 10% of high frequency labels as the pilot label

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
int *top;
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
void PrintSolution(); // Print the MVCA solution
int Eval(); // Find the value of the solution
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
void CopyGraph(Graph G);

```

```

void MVCA(Graph G, int color); // MVCA heuristic
public: int NumComponents();
// Find the number of connected components of the graph
int NumComponents(int color);
// Find the number of connected components of the color
private: void Sort1(); // Sort by the general frequency
        void Sort2(); // Sort by the net frequency
        void FindTop10();
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);

```

```

printf("\n");*/
}

void Graph::FindTop10() {
int i,j,j1,m;
int num;
int *value;

m=cn/10;
top=new int[m];
value=new int[m];
for (i=0; i<m; i++) { top[i]=vn; value[i]=vn; }
for (i=0; i<cn; i++) {
num=NumComponents(i);
for (j=m-1; j>=0; j--) {
if (num>=value[j]) break;
}
if (j==m-1) continue;
if (j<0) {
for (j1=m-1; j1>0; j1--) top[j1]=top[j1-1];
top[0]=i; value[0]=num;
}
else {
for (j1=m-1; j1>j+1; j1--) top[j1]=top[j1-1];
top[j+1]=i; value[j+1]=num;
}
}
}

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

```

```

void Graph::InitGraph(FILE *fp) {
    int i,j,k;
    edge *e,*e1,*e2;

    for (i=0; i<vn; i++) {
        vertex[i]=i;
    }
    for (i=0; i<cn; i++) {
        L[i].en=0; L[i].freq=0;
        L[i].c=i; L[i].root=NULL;
    }
    // Add edges in the empty graph
    for (i=0; i<vn; i++) {
        for (j=i; j<vn; j++) {
            if (i==j) continue;
            k=ReadNumber(fp);
            if (k==cn) continue;
            // Add color k for edge (i,j)
            e=new edge;
            e->u=i; e->v=j; e->c=k; e->next=NULL;
            e1=L[k].root;
            if (e1==NULL) L[k].root=e;
            else {
                e2=e1;
                while (e1!=NULL) { e2=e1; e1=e1->next; }
                e2->next=e;
            }
            L[k].en++;
        }
    }
    Sort1();
    FindTop10();
}

void Graph::InitEmptyGraph(Graph G) {
    int i;

    vn=G.vn; cn=G.cn;
    for (i=0; i<vn; i++) {
        vertex[i]=i;
    }
    for (i=0; i<cn; i++) {
        L[i].en=0; L[i].freq=0;
        L[i].c=i; L[i].root=NULL;
    }
}

```

```

}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::CopyGraph(Graph H) {
int i;

RemoveAllColors();
for (i=0; i<H.cn; i++) {
if (H.L[i].root!=NULL) {
AddColor(H,i);
}
}
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {

```

```

L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```



```

int Graph::NumComponents(int color) {
int i,j,k;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
if (L[color].root==NULL) return vn;
e=L[color].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

void Graph::MVCA(Graph G, int color) {
int k,num;
int i,c;
bool flag;
bool temp[CN];

RemoveAllColors();
AddColor(G,color);
num=NumComponents(); c=0; flag=true;
if (num==1) return;
for (i=0; i<G.cn; i++) temp[i]=false;
temp[color]=true;
while (flag) {
for (i=0; i<G.cn; i++) {
if (temp[i]) continue;
AddColor(G,i);
k=NumComponents();
if (k<num) {
c=i; num=k;
}
}
}
}

```

```

RemoveColor(i);
}
AddColor(G,c); temp[c]=true;
if (num==1) flag=false;
}
/*printf("MVCA Solution: ");
for (i=0; i<G.cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");*/
}

void Graph::PrintSolution() {
int i;

printf("MVCA Solution: ");
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");
}

int Graph::Eval() {
int k,i;

k=0;
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) k++;
}
return k;
}

void main() {
FILE *fp;
Graph G,H,H1;
int k,i;
int gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {

```

```

printf("cannot open file.\n");
return;
}
printf("Group size: ");
scanf("%d",&gn);
printf("=====Result=====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
H.InitEmptyGraph(G);
H1.InitEmptyGraph(G);
value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
//u1=time(NULL);
H.MVCA(G,G.top[0]);
for (i=1; i<G.cn/10; i++) {
H1.MVCA(G,G.top[i]);
if (H1.Eval()<H.Eval()) {
H.CopyGraph(H1);
}
}
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,H.Eval());
H.PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+H.Eval();
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```

```

// MLST_MVCA03.cpp
// Solve MLST by the third version of Modified MVCA (MVCA3)
// using 30% of high frequency labels as the pilot label.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 1000 // number of vertices
#define CN 1000 // number of colors
#define PN 100 // population size

typedef struct edge_type {
int u; // one vertex
int v; // the other vertex
int c; // color
struct edge_type *next; // next edge
} edge;

typedef struct color_type {
int c; // color
int en; // the number of edges
int freq; // the net frequency of the color
edge *root; // point to a list of edges
} color;

class Graph {
public: color L[CN]; // list of colors
int cn; // number of colors
int freq1[CN]; // general frequency
int freq2[CN]; // net frequency
int vn; // number of vertices
int *top;
private: int vertex[VN];
// The representative element of the set with this vertex
public: int ReadNumber(FILE *fp);
void InitGraph(FILE *fp);
void InitEmptyGraph(Graph G);
void PrintGraph(); // Print this graph
void PrintSolution(); // Print the MVCA solution
int Eval(); // Find the value of the solution
public: void AddColor(Graph G, int c);
void RemoveColor(int c);
void ClearAll();
void RemoveAllColors();
void CopyGraph(Graph G);

```

```

void MVCA(Graph G, int color); // MVCA heuristic
public: int NumComponents();
// Find the number of connected components of the graph
int NumComponents(int color);
// Find the number of connected components of the color
private: void Sort1(); // Sort by the general frequency
        void Sort2(); // Sort by the net frequency
        void FindTop10();
};

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::Sort1() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq1[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq1[i]].en; f2=L[freq1[j]].en;
if (f1<f2) {
temp=freq1[i]; freq1[i]=freq1[j]; freq1[j]=temp;
}
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);

```

```

printf("\n");*/
}

void Graph::FindTop10() {
int i,j,j1,m;
int num;
int *value;

m=cn*3/10;
top=new int[m];
value=new int[m];
for (i=0; i<m; i++) { top[i]=vn; value[i]=vn; }
for (i=0; i<cn; i++) {
num=NumComponents(i);
for (j=m-1; j>=0; j--) {
if (num>=value[j]) break;
}
if (j==m-1) continue;
if (j<0) {
for (j1=m-1; j1>0; j1--) top[j1]=top[j1-1];
top[0]=i; value[0]=num;
}
else {
for (j1=m-1; j1>j+1; j1--) top[j1]=top[j1-1];
top[j+1]=i; value[j+1]=num;
}
}
}

void Graph::Sort2() {
int i,j;
int temp;

for (i=0; i<cn; i++) freq2[i]=L[i].c;
for (i=0; i<cn-1; i++) {
for (j=i+1; j<cn; j++) {
int f1,f2;
f1=L[freq2[i]].freq; f2=L[freq2[j]].freq;
if (f1<f2) {
temp=freq2[i]; freq2[i]=freq2[j]; freq2[j]=temp;
}
}
}
/*printf("Sort List: ");
for (i=0; i<cn; i++) printf("%d(%d) ",sortL[i],L[sortL[i]].en);
printf("\n");*/
}

```

```

void Graph::InitGraph(FILE *fp) {
    int i,j,k;
    edge *e,*e1,*e2;

    for (i=0; i<vn; i++) {
        vertex[i]=i;
    }
    for (i=0; i<cn; i++) {
        L[i].en=0; L[i].freq=0;
        L[i].c=i; L[i].root=NULL;
    }
    // Add edges in the empty graph
    for (i=0; i<vn; i++) {
        for (j=i; j<vn; j++) {
            if (i==j) continue;
            k=ReadNumber(fp);
            if (k==cn) continue;
            // Add color k for edge (i,j)
            e=new edge;
            e->u=i; e->v=j; e->c=k; e->next=NULL;
            e1=L[k].root;
            if (e1==NULL) L[k].root=e;
            else {
                e2=e1;
                while (e1!=NULL) { e2=e1; e1=e1->next; }
                e2->next=e;
            }
            L[k].en++;
        }
    }
    Sort1();
    FindTop10();
}

void Graph::InitEmptyGraph(Graph G) {
    int i;

    vn=G.vn; cn=G.cn;
    for (i=0; i<vn; i++) {
        vertex[i]=i;
    }
    for (i=0; i<cn; i++) {
        L[i].en=0; L[i].freq=0;
        L[i].c=i; L[i].root=NULL;
    }
}

```

```

}

void Graph::PrintGraph() {
int i;
edge *e;

printf("-----Graph-----\n");
for (i=0; i<cn; i++) {
if (L[i].root==NULL) continue;
printf("color %d(%d): ",i,L[i].en);
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Graph::AddColor(Graph G, int c) {
L[c].root=G.L[c].root;
L[c].en=G.L[c].en;
L[c].freq=G.L[c].freq;
L[c].c=G.L[c].c;
}

void Graph::RemoveColor(int c) {
L[c].root=NULL;
L[c].en=0; L[c].freq=0;
}

void Graph::CopyGraph(Graph H) {
int i;

RemoveAllColors();
for (i=0; i<H.cn; i++) {
if (H.L[i].root!=NULL) {
AddColor(H,i);
}
}
}

void Graph::ClearAll() {
int i;
edge *e,*e1;

for (i=0; i<cn; i++) {

```



```

L[i].c=i; L[i].en=0; L[i].freq=0;
e=L[i].root;
while (e!=NULL) {
e1=e; e=e->next;
delete e1;
}
L[i].root=NULL;
}
}

void Graph::RemoveAllColors() {
int i;

for (i=0; i<cn; i++) {
L[i].c=i; L[i].en=0;
L[i].freq=0; L[i].root=NULL;
}
}

int Graph::NumComponents() {
int i,j,k,m;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (L[m].root==NULL) continue;
e=L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

int Graph::NumComponents(int color) {
int i,j,k;
edge *e;

for (i=0; i<vn; i++) vertex[i]=i;
if (L[color].root==NULL) return vn;
e=L[color].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

void Graph::MVCA(Graph G, int color) {
int k,num;
int i,c;
bool flag;
bool temp[CN];

RemoveAllColors();
AddColor(G,color);
num=NumComponents(); c=0; flag=true;
if (num==1) return;
for (i=0; i<G.cn; i++) temp[i]=false;
temp[color]=true;
while (flag) {
for (i=0; i<G.cn; i++) {
if (temp[i]) continue;
AddColor(G,i);
k=NumComponents();
if (k<num) {
c=i; num=k;
}
}
}
}

```

```

RemoveColor(i);
}
AddColor(G,c); temp[c]=true;
if (num==1) flag=false;
}
/*printf("MVCA Solution: ");
for (i=0; i<G.cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");*/
}

void Graph::PrintSolution() {
int i;

printf("MVCA Solution: ");
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) printf("%d ",i);
}
printf("\n");
}

int Graph::Eval() {
int k,i;

k=0;
for (i=0; i<cn; i++) {
if (L[i].root!=NULL) k++;
}
return k;
}

void main() {
FILE *fp;
Graph G,H,H1;
int k,i;
int gn;
int value;
char filename[40];
double AvgValue;
time_t u1,u2;
double u;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {

```

```

printf("cannot open file.\n");
return;
}
printf("Group size: ");
scanf("%d",&gn);
printf("=====Result=====\n");
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
H.InitEmptyGraph(G);
H1.InitEmptyGraph(G);
value=0; u=0;
for (k=0; k<gn; k++) {
u1=time(NULL);
G.InitGraph(fp);
//G.PrintGraph();
//u1=time(NULL);
H.MVCA(G,G.top[0]);
for (i=1; i<G.cn*3/10; i++) {
H1.MVCA(G,G.top[i]);
if (H1.Eval()<H.Eval()) {
H.CopyGraph(H1);
}
}
u2=time(NULL);
printf("Group %d: value=%d\n",k+1,H.Eval());
H.PrintSolution();
//printf("GA value: %d\n",S[v].cn);
value=value+H.Eval();
u=u+difftime(u2,u1);
}
AvgValue=(value+0.0)/gn;
printf("Average Value: %f\n",AvgValue);
printf("Average Time: %f\n",u/gn);
fclose(fp);
}

```

```

// BuildCompleteGraph.cpp
// Build complete graphs for LC-MST problems
// with random position for each vertex and
// random label for each edge.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
public: void InitGraph(int vk, int ck); // bk is the frequency bound
void PrintGraph();
void RecordToFile(FILE *fp);
private: bool IsConnected();
};

void Graph::InitGraph(int vk, int ck) {
int i,j;

vn=vk; cn=ck;
for (i=0; i<vn; i++) {
v[i].x=rand()%RANGE;
v[i].y=rand()%RANGE;
}
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) c[i][j]=c[j][i]=cn+1;
else {
c[i][j]=c[j][i]=rand()%cn;
}
}
}
}

void Graph::PrintGraph() {

```

```

int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {
printf("%d: (%d, %d)\n",i+1,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
}

void Graph::RecordToFile(FILE *fp) {
int i,j;

fprintf(fp, "Vertices: \n");
for (i=0; i<vn; i++) {
fprintf(fp, "(%d, %d)\n",v[i].x,v[i].y);
}
fprintf(fp, "Labels: \n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
fprintf(fp, "%d ",c[i][j]);
}
fprintf(fp, "\n");
}
}

void main() {
int cn,vn;
char buf[20];
Graph G;
FILE *fp;

printf("Number of vertices: ");
scanf("%d",&vn);
printf("Number of colors: ");
scanf("%d",&cn);
sprintf(buf,"CGraph%d_%d.txt",vn,cn);
fp=fopen(buf,"wt");
if (fp==NULL) {
printf("cannot open file %s\n",buf);
return;
}
}

```

```
fprintf(fp, "%d %d\n", vn, cn);
G.InitGraph(vn, cn);
//G.PrintGraph();
G.RecordToFile(fp);
fclose(fp);
}
```

```

// LCMST_LS1.cpp
// Solve LC-MST by Local Search #1 for small cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();
void SetupLabelList();
};

class Tree {
public: Graph *G;

```



```

bool *labelset;
node *nd;
int cn;
int vn;
public: void InitTree(Graph *graph);
void MSTByPrim();
void MLSTByMVCA();
float LocalSearch1();
void KMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
private: int bd;
        int solution[CN];
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {

```

```

k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {

```

```

printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph) {
int i;

G=graph;
cn=G->cn; vn=G->vn;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
}

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];

```

```

if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
labelset[j]=true;
if (num==1) flag=false;
}
}

```

```

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;

```

```

for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;
for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
}
}

```

```

}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];
set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

float Tree::LocalSearch1() {

```

```

int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=solution[i]; labelset[k]=true;
}
if (NumComponents(>1) cost=(float) 1.0e10;
else {
MSTByPrim();
cost=TotalCost();
}
for (i=0; i<bd; i++) {
k=solution[i];
labelset[k]=false;
for (j=0; j<cn; j++) {
if (labelset[j]) continue;
labelset[j]=true;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[j]=false;
}
solution[i]=k;
labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return -1;
MSTByPrim();
cost=TotalCost();
return cost;
}

void Tree::KMST(int bound) {
int i,k;
float cost1,cost2;

bd=bound;
while (true) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
solution[i]=rand()%cn;
k=solution[i];

```

```

while (labelset[k]) {
solution[i]=rand()%cn;
k=solution[i];
}
labelset[k]=true;
}
cost1=LocalSearch1();
if (cost1>=0) break;
}
while (true) {
cost2=LocalSearch1();
if (cost2==cost1) break;
cost1=cost2;
}
MSTByPrim();
}

void main() {
Graph G;
Tree T;
char filename[80];
int num1,num2;
time_t u1,u2;
int i,j;
float cost,cost1;

printf("filename: ");
scanf("%s",filename);
G.InitGraph(filename);
//G.PrintGraph();
T.InitTree(&G);
T.MSTByPrim();
num1=T.NumLabels();
T.PrintTree();
T.MLSTByMVCA();
T.MSTByPrim();
num2=T.NumLabels();
T.PrintTree();
printf("*****\n");
/*T.KMST(20);
T.PrintTree();*/
for (i=num2; i<=num1; i++) {
u1=time(NULL);
cost=1.0e10;
for (j=0; j<5; j++) {
T.KMST(i);
cost1=T.TotalCost();

```



```
if (cost1<cost) cost=cost1;
}
u2=time(NULL);
printf("Value(%d): %f\n",i,cost);
printf("Running time: %f\n",difftime(u2,u1));
}
}
```

```

// LCMST_LS1a.cpp
// Solve LC-MST by Local Search #1 for large cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();
void SetupLabelList();
};

class Tree {
public: Graph *G;

```

```

bool *labelset;
node *nd;
int cn;
int vn;
public: void InitTree(Graph *graph);
void MSTByPrim();
void MLSTByMVCA();
float LocalSearch1();
void KMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
private: int bd;
        int solution[CN];
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {

```

```

k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {

```

```

printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph) {
int i;

G=graph;
cn=G->cn; vn=G->vn;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
}

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];

```

```

if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
labelset[j]=true;
if (num==1) flag=false;
}
}

```

```

int Tree::NumLabels() {
edge *e1;
int set[CN];
int i,j;
int num;

```

```

for (i=0; i<cn; i++) set[i]=0;
for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;
for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;

```

```

if (nd[k].key<nd[j].key) j=k;
}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];
set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

```



```

float Tree::LocalSearch1() {
int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=solution[i]; labelset[k]=true;
}
if (NumComponents(>1) cost=(float) 1.0e10;
else {
MSTByPrim();
cost=TotalCost();
}
for (i=0; i<bd; i++) {
k=solution[i];
labelset[k]=false;
for (j=0; j<cn; j++) {
if (labelset[j]) continue;
labelset[j]=true;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[j]=false;
}
solution[i]=k;
labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return -1;
MSTByPrim();
cost=TotalCost();
return cost;
}

void Tree::KMST(int bound) {
int i,k;
float cost1,cost2;

bd=bound;
while (true) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
solution[i]=rand()%cn;

```

```

k=solution[i];
while (labelset[k]) {
solution[i]=rand()%cn;
k=solution[i];
}
labelset[k]=true;
}
cost1=LocalSearch1();
if (cost1>=0) break;
}
while (true) {
cost2=LocalSearch1();
if (cost2==cost1) break;
cost1=cost2;
}
MSTByPrim();
}

void main() {
Graph G;
Tree T;
char filename[80];
time_t u1,u2;
int kn;

printf("filename: ");
scanf("%s",filename);
G.InitGraph(filename);
T.InitTree(&G);
printf("K=");
scanf("%d",&kn);
//G.PrintGraph();
T.MSTByPrim();
T.PrintTree();
for (int i=0; i<5; i++) {
u1=time(NULL);
T.KMST(kn);
u2=time(NULL);
T.PrintTree();
printf("Running time: %f\n",difftime(u2,u1));
}
}

```

```

// LCMST_LS2.cpp
// Solve LC-MST by Local Search #2 for small cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();
void SetupLabelList();
};

class Tree {
public: Graph *G;

```

```

bool *labelset;
node *nd;
int cn;
int vn;
public: void InitTree(Graph *graph);
void MSTByPrim();
void MLSTByMVCA();
float LocalSearch2();
void KMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
private: int bd;
        int solution[CN];
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {

```

```

k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {

```

```

printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph) {
int i;

G=graph;
cn=G->cn; vn=G->vn;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
}

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];

```

```

if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
labelset[j]=true;
if (num==1) flag=false;
}
}

```

```

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;

```

```

for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;
for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
}
}

```



```

}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];
set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

float Tree::LocalSearch2() {

```

```

int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=solution[i]; labelset[k]=true;
}
if (NumComponents(>1) cost=(float) 1.0e10;
else {
MSTByPrim();
cost=TotalCost();
}
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=-1;
for (j=0; j<bd; j++) {
labelset[solution[j]]=false;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[solution[j]]=true;
}
if (k<0) labelset[i]=false;
else {
labelset[solution[k]]=false;
for (j=k; j<bd-1; j++) {
solution[j]=solution[j+1];
}
solution[j]=i;
}
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return -1;
MSTByPrim();
cost=TotalCost();
return cost;
}

void Tree::KMST(int bound) {
int i,k;
float cost1,cost2;

```

```

bd=bound;
while (true) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
solution[i]=rand()%cn;
k=solution[i];
while (labelset[k]) {
solution[i]=rand()%cn;
k=solution[i];
}
labelset[k]=true;
}
cost1=LocalSearch2();
if (cost1>=0) break;
}
while (true) {
cost2=LocalSearch2();
if (cost2==cost1) break;
cost1=cost2;
}
MSTByPrim();
}

void main() {
Graph G;
Tree T;
char filename[80];
int num1,num2;
int i,j;
float cost,cost1;
time_t u1,u2;

printf("filename: ");
scanf("%s",filename);
G.InitGraph(filename);
//G.PrintGraph();
T.InitTree(&G);
T.MSTByPrim();
num1=T.NumLabels();
T.PrintTree();
T.MLSTByMVCA();
T.MSTByPrim();
num2=T.NumLabels();
T.PrintTree();
printf("*****\n");
/*T.KMST(20);
T.PrintTree();*/

```

```
for (i=num2; i<=num1; i++) {
u1=time(NULL);
cost=1.0e10;
for (j=0; j<5; j++) {
T.KMST(i);
cost1=T.TotalCost();
if (cost1<cost) cost=cost1;
}
u2=time(NULL);
printf("Value(%d): %f\n",i,cost);
printf("Running time: %f\n",difftime(u2,u1));
}
}
```

```

// LCMST_LS2a.cpp
// Solve LC-MST by Local Search #2 for large cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();
void SetupLabelList();
};

class Tree {
public: Graph *G;

```

```

bool *labelset;
node *nd;
int cn;
int vn;
public: void InitTree(Graph *graph);
void MSTByPrim();
void MLSTByMVCA();
float LocalSearch2();
void KMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
private: int bd;
        int solution[CN];
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {

```

```

k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {

```

```

printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph) {
int i;

G=graph;
cn=G->cn; vn=G->vn;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
}

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];

```



```

if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
labelset[j]=true;
if (num==1) flag=false;
}
}

```

```

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;

```

```

for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;
for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
}
}

```

```

}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];
set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

float Tree::LocalSearch2() {

```

```

int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=solution[i]; labelset[k]=true;
}
if (NumComponents(>1) cost=(float) 1.0e10;
else {
MSTByPrim();
cost=TotalCost();
}
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=-1;
for (j=0; j<bd; j++) {
labelset[solution[j]]=false;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[solution[j]]=true;
}
if (k<0) labelset[i]=false;
else {
labelset[solution[k]]=false;
for (j=k; j<bd-1; j++) {
solution[j]=solution[j+1];
}
solution[j]=i;
}
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return -1;
MSTByPrim();
cost=TotalCost();
return cost;
}

void Tree::KMST(int bound) {
int i,k;
float cost1,cost2;

```

```

bd=bound;
while (true) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
solution[i]=rand()%cn;
k=solution[i];
while (labelset[k]) {
solution[i]=rand()%cn;
k=solution[i];
}
labelset[k]=true;
}
cost1=LocalSearch2();
if (cost1>=0) break;
}
while (true) {
cost2=LocalSearch2();
if (cost2==cost1) break;
cost1=cost2;
}
MSTByPrim();
}

void main() {
Graph G;
Tree T;
char filename[80];
time_t u1,u2;
int kn;

printf("filename: ");
scanf("%s",filename);
printf("K=");
scanf("%d",&kn);
G.InitGraph(filename);
//G.PrintGraph();
for (int i=0; i<5; i++) {
u1=time(NULL);
T.InitTree(&G);
T.KMST(kn);
u2=time(NULL);
T.PrintTree();
printf("Running time: %f\n",difftime(u2,u1));
}
}

```

```

// LCMST_GA.cpp
// Solve LC-MST by Genetic algorithm for small cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

typedef struct chromosome_type {
int s[CN];
float value;
} chromosome;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();

```

```

    void SetupLabelList();
};

class Tree {
public: Graph *G;
bool *labelset;
node *nd;
int cn;
int vn;
int pn;
public: void InitTree(Graph *graph,int popsize);
void InitChromosome();
void MSTByPrim();
void MLSTByMVCA();
void Mutation(int n1);
void Mutation1(int n1);
void Crossover(int n1,int n2);
void LCMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
        float GetBestValue();
        int GetQueenBee();
private: int bd;
        chromosome *cs;
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
}

```

```

return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {
k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
}

```



```

}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {
printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph,int popsize) {
int i;

G=graph;
cn=G->cn; vn=G->vn; pn=popsize;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
cs=new chromosome[popsize];
}

```

```

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
}
labelset[i]=false;
}
}

```

```

}
labelset[j]=true;
if (num==1) flag=false;
}
}

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;
for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;

```

```

for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];

```

```

set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

```

```

void Tree::Crossover(int n1,int n2) {
bool Q[VN];
int i,j,k;
float dis;
int count[CN];
int usedlabels;
float cost;

for (i=0; i<cn; i++) {
labelset[i]=false;
count[i]=0;
}
for (i=0; i<bd; i++) {
labelset[cs[n1].s[i]]=true;
labelset[cs[n2].s[i]]=true;
}
for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[rand()%vn].key=0;
for (i=0; i<vn; i++) Q[i]=false;
usedlabels=0;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
Q[j]=true; i=nd[j].pi;
}

```

```

if (i>=0) {
k=G->c[i][j];
if (count[k]==0) {
count[k]++; usedlabels++;
}
if (usedlabels==bd) break;
}
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
for (i=0; i<cn; i++) {
if (count[i]>0) labelset[i]=true;
else labelset[i]=false;
}
if (NumComponents(>1) return;
MSTByPrim();
cost=TotalCost();
if (cost<cs[n1].value) {
k=0;
for (i=0; i<cn; i++) {
if (count[i]>0) {
cs[n1].s[k]=i; k++;
}
}
cs[n1].value=cost;
}
}

void Tree::Mutation(int n1) {
int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=cs[n1].s[i]; labelset[k]=true;
}
cost=cs[n1].value;

```

```

for (i=0; i<bd; i++) {
k=cs[n1].s[i];
labelset[k]=false;
for (j=0; j<cn; j++) {
if (labelset[j]) continue;
labelset[j]=true;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[j]=false;
}
cs[n1].s[i]=k;
labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return;
cs[n1].value=cost;
}

void Tree::Mutation1(int n1) {
int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=cs[n1].s[i]; labelset[k]=true;
}
cost=cs[n1].value;
j=rand()%cn;
while (labelset[j]==true) j=rand()%cn;
labelset[j]=true;
for (i=0; i<bd; i++) {
k=cs[n1].s[i];
labelset[k]=false;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cs[n1].s[i]=j;
cs[n1].value=cost1;
break;
}
}
}
}

```

```

labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (i>=bd) {
labelset[j]=false;
cs[n1].value=cost;
}
}

void Tree::InitChromosome() {
int i,j,k;

for (j=0; j<pn; j++) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
cs[j].s[i]=rand()%cn;
k=cs[j].s[i];
while (labelset[k]) {
cs[j].s[i]=rand()%cn;
k=cs[j].s[i];
}
labelset[k]=true;
}
if (NumComponents(>1) cs[j].value=1.0e10;
else {
MSTByPrim();
cs[j].value=TotalCost();
}
}
}

float Tree::GetBestValue() {
int i;
float cost;

cost=cs[0].value;
for (i=1; i<pn; i++) {
if (cs[i].value<cost) cost=cs[i].value;
}
return cost;
}

int Tree::GetQueenBee() {
int i;
float cost;
int qb;

```



```

cost=cs[0].value; qb=0;
for (i=1; i<pn; i++) {
if (cs[i].value<cost) {
cost=cs[i].value;
qb=i;
}
}
return qb;
}

void Tree::LCMST(int bound) {
int i,j,k;
float cost1,cost2;
int qb;
int count;

bd=bound;
InitChromosome();
cost1=GetBestValue(); count=0;
while (true) {
qb=GetQueenBee();
k=rand()%100;
if (k<20) Mutation1(qb);
for (i=0; i<pn; i++) {
if (i==qb) continue;
k=rand()%100;
if (k<70) {
Crossover(i,qb);
}
}
cost2=GetBestValue();
if (cost2==cost1) count++;
else count=0;
if (count>2) break;
cost1=cost2;
//printf("cost=%f\n",cost1);
}
qb=GetQueenBee();
Mutation(qb); Mutation(qb);
printf("Bound: %d, cost: %f\n",bd, cs[qb].value);
}

void main() {
Graph G;
Tree T;
char filename[80];
int num1,num2;

```

```

int pn;

printf("filename: ");
scanf("%s",filename);
printf("population size: ");
scanf("%d",&pn);
G.InitGraph(filename);
//G.PrintGraph();
T.InitTree(&G,pn);
T.MSTByPrim();
num1=T.NumLabels();
T.PrintTree();
T.MLSTByMVCA();
T.MSTByPrim();
num2=T.NumLabels();
T.PrintTree();
printf("*****\n");
//T.LCMST(20);
for (int i=num2; i<=num1; i++) {
T.LCMST(i);
}
}

```

```

// LCMST_GAa.cpp
// Solve LC-MST by Genetic algorithm for large cases

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

typedef struct chromosome_type {
int s[CN];
float value;
} chromosome;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();

```

```

    void SetupLabelList();
};

class Tree {
public: Graph *G;
    bool *labelset;
    node *nd;
    int cn;
    int vn;
    int pn;
public: void InitTree(Graph *graph,int popsize);
    void InitChromosome();
    void MSTByPrim();
    void MLSTByMVCA();
    void Mutation(int n1);
    void Mutation1(int n1);
    void Crossover(int n1,int n2);
    void LCMST(int bound);
    int NumLabels();
    float TotalCost();
    void PrintTree();
private: float Distance(int k);
        int NumComponents();
        float GetBestValue();
        int GetQueenBee();
private: int bd;
        chromosome *cs;
};

int Graph::ReadNumber() {
    char buf[10];
    char c;
    int i,n;
    int num;

    c=fgetc(fp);
    i=0;
    while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
    while (!feof(fp) && (c>='0' && c<='9')) {
        buf[i]=c; c=fgetc(fp); i++;
    }
    //buf[i]='\0';
    num=0;
    n=i;
    for (i=0; i<n; i++) {
        num=num*10+(int)(buf[i]-'0');
    }
}

```

```

return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {
k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
}

```

```

}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {
printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph,int popsize) {
int i;

G=graph;
cn=G->cn; vn=G->vn; pn=popsize;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
cs=new chromosome[popsize];
}

```

```

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];
if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
}
}

```

```

}
labelset[j]=true;
if (num==1) flag=false;
}
}

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;
for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;

```



```

for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];

```

```

set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

```

```

void Tree::Crossover(int n1,int n2) {
bool Q[VN];
int i,j,k;
float dis;
int count[CN];
int usedlabels;
float cost;

for (i=0; i<cn; i++) {
labelset[i]=false;
count[i]=0;
}
for (i=0; i<bd; i++) {
labelset[cs[n1].s[i]]=true;
labelset[cs[n2].s[i]]=true;
}
for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[rand()%vn].key=0;
for (i=0; i<vn; i++) Q[i]=false;
usedlabels=0;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
Q[j]=true; i=nd[j].pi;
}

```

```

if (i>=0) {
k=G->c[i][j];
if (count[k]==0) {
count[k]++; usedlabels++;
}
if (usedlabels==bd) break;
}
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
for (i=0; i<cn; i++) {
if (count[i]>0) labelset[i]=true;
else labelset[i]=false;
}
if (NumComponents(>1) return;
MSTByPrim();
cost=TotalCost();
if (cost<cs[n1].value) {
k=0;
for (i=0; i<cn; i++) {
if (count[i]>0) {
cs[n1].s[k]=i; k++;
}
}
cs[n1].value=cost;
}
}

void Tree::Mutation(int n1) {
int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=cs[n1].s[i]; labelset[k]=true;
}
cost=cs[n1].value;

```

```

for (i=0; i<bd; i++) {
k=cs[n1].s[i];
labelset[k]=false;
for (j=0; j<cn; j++) {
if (labelset[j]) continue;
labelset[j]=true;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cost=cost1; k=j;
}
}
labelset[j]=false;
}
cs[n1].s[i]=k;
labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (NumComponents(>1) return;
cs[n1].value=cost;
}

void Tree::Mutation1(int n1) {
int i,j,k;
float cost,cost1;

for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
k=cs[n1].s[i]; labelset[k]=true;
}
cost=cs[n1].value;
j=rand()%cn;
while (labelset[j]==true) j=rand()%cn;
labelset[j]=true;
for (i=0; i<bd; i++) {
k=cs[n1].s[i];
labelset[k]=false;
if (NumComponents()==1) {
MSTByPrim();
cost1=TotalCost();
if (cost1<cost) {
cs[n1].s[i]=j;
cs[n1].value=cost1;
break;
}
}
}
}

```

```

labelset[k]=true;
//printf("cost=%f\n",cost);
}
if (i>=bd) {
labelset[j]=false;
cs[n1].value=cost;
}
}

void Tree::InitChromosome() {
int i,j,k;

for (j=0; j<pn; j++) {
for (i=0; i<cn; i++) labelset[i]=false;
for (i=0; i<bd; i++) {
cs[j].s[i]=rand()%cn;
k=cs[j].s[i];
while (labelset[k]) {
cs[j].s[i]=rand()%cn;
k=cs[j].s[i];
}
labelset[k]=true;
}
if (NumComponents(>1) cs[j].value=1.0e10;
else {
MSTByPrim();
cs[j].value=TotalCost();
}
}
}

float Tree::GetBestValue() {
int i;
float cost;

cost=cs[0].value;
for (i=1; i<pn; i++) {
if (cs[i].value<cost) cost=cs[i].value;
}
return cost;
}

int Tree::GetQueenBee() {
int i;
float cost;
int qb;

```

```

cost=cs[0].value; qb=0;
for (i=1; i<pn; i++) {
if (cs[i].value<cost) {
cost=cs[i].value;
qb=i;
}
}
return qb;
}

void Tree::LCMST(int bound) {
int i,j,k;
float cost1,cost2;
int qb;
int count;

bd=bound;
InitChromosome();
cost1=GetBestValue(); count=0;
while (true) {
qb=GetQueenBee();
k=rand()%100;
if (k<20) Mutation1(qb);
for (i=0; i<pn; i++) {
if (i==qb) continue;
k=rand()%100;
if (k<70) {
Crossover(i,qb);
}
}
cost2=GetBestValue();
if (cost2==cost1) count++;
else count=0;
if (count>2) break;
cost1=cost2;
//printf("cost=%f\n",cost1);
}
qb=GetQueenBee();
Mutation(qb);
Mutation(qb);
printf("Bound: %d, cost: %f\n",bd, cs[qb].value);
}

void main() {
Graph G;
Tree T;
char filename[80];

```

```
time_t u1,u2;
int pn,kn;

printf("filename: ");
scanf("%s",filename);
printf("population size: ");
scanf("%d",&pn);
printf("K=");
scanf("%d",&kn);
G.InitGraph(filename);
u1=time(NULL);
T.InitTree(&G,pn);
T.LCMST(kn);
u2=time(NULL);
printf("Running time: %f\n",difftime(u2,u1));
}
```

```

// LCMST_OPT.cpp
// Solve LC-MST by an optimal solution
// with back-track search

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define VN 500
#define CN 500
#define RANGE 1000

typedef struct Point_type {
int x; int y;
} Point;

typedef struct edge_type {
int u;
int v;
struct edge_type *next;
} edge;

typedef struct label_type{
int c;
edge *root;
} label;

typedef struct node_type {
float key;
int pi;
} node;

class Graph {
public: int c[VN][VN];
Point v[VN];
int cn;
int vn;
label *L;
public: void InitGraph(char *filename); // bk is the frequency bound
void PrintGraph();
private: FILE *fp;
private: int ReadNumber();
void SetupLabelList();
};

class Tree {

```



```

public: Graph *G;
bool *labelset;
node *nd;
int cn;
int vn;
public: void InitTree(Graph *graph);
void MSTByPrim();
void MLSTByMVCA();
void KMST(int bound);
int NumLabels();
float TotalCost();
void PrintTree();
private: float Distance(int k);
        int NumComponents();
private: int bd;
        int solution[CN];
};

int Graph::ReadNumber() {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Graph::SetupLabelList() {
int i,j,k;
edge *e1,*e2;

L=new label[cn];
for (i=0; i<cn; i++) { L[i].root=NULL; L[i].c=i; }
for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {

```

```

k=c[i][j];
if (L[k].root==NULL) {
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
L[k].root=e1;
}
else {
e1=L[k].root; e2=e1;
while (e1!=NULL) {
e2=e1; e1=e1->next;
}
e1=new edge;
e1->u=i; e1->v=j; e1->next=NULL;
e2->next=e1;
}
}
}
}

void Graph::InitGraph(char *filename) {
int i,j;

fp=fopen(filename,"rt");
if (fp==NULL) {
printf("Can not open file %s\n",filename);
exit(1);
}
vn=ReadNumber();
cn=ReadNumber();
for (i=0; i<vn; i++) {
v[i].x=ReadNumber();
v[i].y=ReadNumber();
}
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
c[i][j]=ReadNumber();
}
}
fclose(fp);
SetupLabelList();
}

void Graph::PrintGraph() {
int i,j;

printf("***** Vertices *****\n");
for (i=0; i<vn; i++) {

```

```

printf("%d: (%d, %d)\n",i,v[i].x,v[i].y);
}
printf("***** Labels *****\n");
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
printf("%d ",c[i][j]);
}
printf("\n");
}
printf("***** Label List *****\n");
for (i=0; i<cn; i++) {
printf("%d: ",L[i].c);
edge *e;
e=L[i].root;
while (e!=NULL) {
printf("(%d,%d) ",e->u,e->v);
e=e->next;
}
printf("\n");
}
}

void Tree::InitTree(Graph *graph) {
int i;

G=graph;
cn=G->cn; vn=G->vn;
labelset=new bool[cn];
for (i=0; i<cn; i++) labelset[i]=true;
nd=new node[vn];
for (i=0; i<cn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
}

int Tree::NumComponents() {
int i,j,k,m;
edge *e;
int vertex[VN];

for (i=0; i<vn; i++) vertex[i]=i;
for (m=0; m<cn; m++) {
if (!labelset[m]) continue;
e=G->L[m].root;
while (e!=NULL) {
i=vertex[e->u]; j=vertex[e->v];

```

```

if (i==j) { e=e->next; continue; }
if (i<j) {
for (int k=0; k<vn; k++)
if (vertex[k]==j) vertex[k]=i;
}
else {
for (k=0; k<vn; k++)
if (vertex[k]==i) vertex[k]=j;
}
e=e->next;
}
}
k=0;
for (i=0; i<vn; i++)
if (vertex[i]==i) k++;
return k;
}

```

```

void Tree::MLSTByMVCA() {
int i,j,k,num;
bool flag;

for (i=0; i<cn; i++) {
labelset[i]=false;
}
flag=true; num=vn+1;
while (flag) {
for (i=0; i<cn; i++) {
if (labelset[i]) continue;
labelset[i]=true;
k=NumComponents();
if (k<num) {
j=i; num=k;
}
labelset[i]=false;
}
labelset[j]=true;
if (num==1) flag=false;
}
}

```

```

int Tree::NumLabels() {
int set[CN];
int i,j;
int num;

for (i=0; i<cn; i++) set[i]=0;

```

```

for (j=0; j<vn; j++) {
if (nd[j].pi<0) continue;
i=G->c[j][nd[j].pi];
set[i]++;
}
num=0;
for (i=0; i<cn; i++) {
if (set[i]>0) num++;
}
return num;
}

float Tree::Distance(int k) {
int i,j;
float x,y;
float d;

if (nd[k].pi<0) return 0;
i=k; j=nd[k].pi;
x=(float) G->v[i].x-G->v[j].x+0.0;
y=(float) G->v[i].y-G->v[j].y+0.0;
d=(float) sqrt(x*x+y*y);
return d;
}

void Tree::MSTByPrim() {
bool Q[VN];
int i,j,k;
float dis;

for (i=0; i<vn; i++) {
nd[i].key=(float) 1.0e10;
nd[i].pi=-1;
}
nd[0].key=0;
for (i=0; i<vn; i++) Q[i]=false;
while (true) {
// Check if Q is empty
for (i=0; i<vn; i++) {
if (!Q[i]) break;
}
if (i>=vn) break;
// Extract minimum element from Q
j=i; // The first unvisited node
for (k=i; k<vn; k++) {
if (Q[k]) continue;
if (nd[k].key<nd[j].key) j=k;
}
}
}

```

```

}
Q[j]=true;
// Set up keys for adjacent element of j
for (i=0; i<vn; i++) {
if (j==i) continue;
if (Q[i]) continue;
k=G->c[i][j];
if (!labelset[k]) continue;
dis=(float) sqrt((G->v[i].x-G->v[j].x)*(G->v[i].x-G->v[j].x)
+(G->v[i].y-G->v[j].y)*(G->v[i].y-G->v[j].y));
if (dis<nd[i].key) {
nd[i].key=dis; nd[i].pi=j;
}
}
}
}

float Tree::TotalCost() {
int i;
float cost;

cost=0;
for (i=0; i<vn; i++) cost=cost+Distance(i);
return cost;
}

void Tree::PrintTree() {
bool set[VN];
int i,j,k;

printf("***** Tree *****\n");
for (i=0; i<cn; i++) set[i]=false;
for (i=0; i<vn; i++) {
if (nd[i].pi<0) continue;
k=G->c[i][nd[i].pi];
set[k]=true;
}
printf("Total Cost: %f\n",TotalCost());
printf("Labels: ");
j=0;
for (i=0; i<cn; i++) {
if (set[i]) { printf("%d ",i); j++; }
}
printf("(%d)\n",j);
}

void Tree::KMST(int bound) {

```

```

int i,j,k;
bool set[CN];
float cost1,cost2;
int end;
int *index;

for (i=0; i<cn; i++) {
if (i<bound) labelset[i]=true;
else labelset[i]=false;
}
cost1=(float) 1.0e10;
index=new int[bound];
for (i=0; i<bound; i++) index[i]=i;
while (true) {
end=index[bound-1];
if (NumComponents()==1) {
MSTByPrim();
cost2=TotalCost();
if (cost2<cost1) {
cost1=cost2;
for (i=0; i<cn; i++) set[i]=labelset[i];
}
}
labelset[end]=false;
end=end+1;
if (end<cn) {
labelset[end]=true;
index[bound-1]=end; continue;
}
j=bound-2;
if (j<0) break;
while (index[j]+bound-j==cn) {
k=index[j]; j--;
if (j<0) break;
}
if (j<0) break;
labelset[index[j]]=false;
index[j]=index[j]+1;
labelset[index[j]]=true;
for (k=j+1; k<bound; k++) {
labelset[index[k]]=false;
index[k]=index[j]+(k-j);
labelset[index[k]]=true;
}
}
for (i=0; i<cn; i++) labelset[i]=set[i];
MSTByPrim();

```

```

}

void main() {
Graph G;
Tree T;
char filename[80];
int num1,num2;
time_t u1,u2;

printf("filename: ");
scanf("%s",filename);
G.InitGraph(filename);
//G.PrintGraph();
T.InitTree(&G);
T.MSTByPrim();
num1=T.NumLabels();
T.PrintTree();
T.MLSTByMVCA();
T.MSTByPrim();
num2=T.NumLabels();
T.PrintTree();
printf("*****\n");
for (int i=num2; i<=num1; i++) {
u1=time(NULL);
T.KMST(i);
u2=time(NULL);
T.PrintTree();
printf("Running time: %f\n",difftime(u2,u1));
}
}

```



```

// BuildGraph.cpp
// Build complete labeled graphs for CTSP

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VN 500
#define CN 500

class Graph {
public: int c[VN][VN];
int cn;
int vn;
public: void InitGraph(int vk, int ck);
void RecordToFile(FILE *fp);
};

void Graph::InitGraph(int vk, int ck) {
int i,j;

cn=ck; vn=vk;
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) {
if (i==j) c[i][j]=-1;
else c[i][j]=rand()%cn;
}
}
}

void Graph::RecordToFile(FILE *fp) {
int i,j;

for (i=0; i<vn; i++) {
for (j=i+1; j<vn; j++) {
fprintf(fp, "%d ",c[i][j]);
}
fprintf(fp,"\n");
}
}

void main() {
int cn,vn,sn;
char buf[20];
int i;
Graph G;
FILE *fp;

```

```

printf("Number of vertices: ");
scanf("%d",&vn);
printf("Number of colors: ");
scanf("%d",&cn);
printf("Number of samples: ");
scanf("%d",&sn);
sprintf(buf,"Graph%d_%d.txt",vn,cn);
fp=fopen(buf,"wt");
if (fp==NULL) {
printf("cannot open file %s\n",buf);
return;
}
fprintf(fp,"%d %d\n",vn,cn);
for (i=0; i<sn; i++) {
G.InitGraph(vn,cn);
G.RecordToFile(fp);
}
fclose(fp);
}

```

```

// CTSP_MPEA.cpp
// Solve CTSP by the MPEA heuristic

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define VN 500
#define CN 500

class Graph {
public: int *c[VN];
int cn;
int vn;
int freq[CN];
public: void InitGraph(FILE *fp);
void ClearGraph();
int ReadNumber(FILE *fp);
void PrintGraph();
};

class Tour {
public: int *t;
int tn;
int vn;
int cn;
public: void InitTour(Graph G);
void ClearTour();
void EMSh(Graph G);
void CopyTour(Tour tour);
void PrintTour();
int Value();
private: bool *v;
int *c;
};

void Graph::InitGraph(FILE *fp) {
int i,j,k;

for (i=0; i<vn; i++) {
c[i]=new int[VN];
}
for (i=0; i<cn; i++) freq[i]=0;
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) { c[i][j]=-1; continue; }
k=ReadNumber(fp);
}
}
}

```

```

c[i][j]=k; c[j][i]=k; freq[k]++;
}
}
}

void Graph::ClearGraph() {
int i;

for (i=0; i<vn; i++) {
delete c[i];
}
}

void Graph::PrintGraph() {
int i,j;

printf("----- Graph -----\n");
for (i=0; i<vn; i++) {
printf("%d: ",i);
for (j=0; j<vn; j++) {
if (i==j) continue;
printf("%d->%d(%d) ",i,j,c[i][j]);
}
printf("\n");
}
}

int Graph::ReadNumber(FILE *fp) {
char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

```

```

void Tour::InitTour(Graph G) {
    int i;

    vn=G.vn;
    cn=G.cn;
    tn=0;
    t=new int[VN];
    v=new bool[VN];
    c=new int[CN];
    for (i=0; i<vn; i++) v[i]=false;
    for (i=0; i<cn; i++) c[i]=0;
}

void Tour::ClearTour() {
    int i;

    tn=0;
    for (i=0; i<vn; i++) v[i]=false;
    for (i=0; i<cn; i++) c[i]=0;
}

int Tour::Value() {
    int i;
    int value=0;

    for (i=0; i<cn; i++) {
        if (c[i]>0) value++;
    }
    return value;
}

void Tour::CopyTour(Tour tour) {
    int i;

    vn=tour.vn; cn=tour.cn; tn=tour.tn;
    for (i=0; i<tn; i++) t[i]=tour.t[i];
    for (i=0; i<vn; i++) v[i]=tour.v[i];
    for (i=0; i<cn; i++) c[i]=tour.c[i];
}

void Tour::PrintTour() {
    int i;

    printf("Tour: \n");
    for (i=0; i<tn; i++) {
        printf("%d->",t[i]);
    }
}

```

```

}
printf("%d\n",t[0]);
}

void Tour::EMSH(Graph G) {
int i,j;
int k,color,color1,color2;
int k1,colorA,colorB;
int temp[VN];
int maxcolor,fq;

// Begin with a random city
t[0]=rand()%vn; v[t[0]]=true;
// Find the second city with the highest frequency label.
maxcolor=0; k=0; fq=-1;
if (k==t[0]) k=1;
for (i=0; i<vn; i++) {
if (i==t[0]) continue;
maxcolor=G.c[t[0]][i];
if (G.freq[maxcolor]>fq) {
fq=G.freq[maxcolor];
k=i;
}
else if (G.freq[maxcolor]==fq) {
if (rand()%2==1) k=i;
}
}
t[1]=k; v[t[1]]=true;
color=G.c[t[0]][t[1]];
c[color]++;

// Find the third city
for (i=0; i<vn; i++) { // Find a suitable city
if (v[i]) continue;
color=G.c[t[1]][i];
if (c[color]>0) {
t[2]=i; c[color]++;
v[t[2]]=true; break;
}
color=G.c[t[0]][i];
if (c[color]>0) {
t[2]=t[1]; t[1]=t[0]; t[0]=i;
c[color]++;
v[t[0]]=true; break;
}
}
if (i>=vn) {

```

```

// Fail to find a suitable city, new label is added
k=0; fq=-1;
for (i=0; i<vn; i++) {
if (v[i]) continue;
maxcolor=G.c[t[1]][i];
if (G.freq[maxcolor]>fq) {
fq=G.freq[maxcolor];
k=i;
}
else if (G.freq[maxcolor]==fq) {
if (rand()%2==1) k=i;
}
}
t[2]=k; v[t[2]]=true;
color=G.c[t[1]][t[2]];
c[color]++;
}
tn=3;

// Find the rest of cities
while (tn<vn) {
/*printf("Tour: ");
for (i=0; i<tn; i++) printf("%d ",t[i]);
printf("\n");*/
// Find the next suitable city
for (i=0; i<vn; i++) {
if (v[i]) continue;
// Consider the two end points of the partial tour
color=G.c[t[tn-1]][i];
if (c[color]>0) {
t[tn]=i; tn++; v[i]=true;
c[color]++; break;
}
color=G.c[t[0]][i];
if (c[color]>0) {
for (j=tn; j>=1; j--) t[j]=t[j-1];
t[0]=i; tn++; c[color]++;
v[t[0]]=true; break;
}
// Consider three cases to insert into the partial tour
for (j=0; j<tn-1; j++) {
color1=G.c[t[j]][i];
color2=G.c[t[j+1]][i];
// Case 1
if (c[color1]>0 && c[color2]>0) {
color=G.c[t[j]][t[j+1]];
c[color]--; // cut edge (t[j],t[j+1])
}
}
}

```

```

for (k=tn; k>j+1; k--) t[k]=t[k-1];
t[j+1]=i; tn++; // add edges (t[j],i) and (i,t[j+1]);
v[i]=true; c[color1]++; c[color2]++;
break;
}
// Case 2
if (c[color1]>0 && c[color2]==0) {
colorA=G.c[t[0]][t[j+1]];
colorB=G.c[t[0]][t[tn-1]];
color=G.c[t[j]][t[j+1]];
if (c[colorA]>0) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0; t[k1]=i; k1++;
for (k=j; k>=0; k--) {
t[k1]=temp[k]; k1++;
}
for (k=j+1; k<=tn-1; k++) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorA]++; c[color1]++;
v[i]=true; break;
}
if (c[colorB]>0) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0; t[k1]=i; k1++;
for (k=j; k>=0; k--) {
t[k1]=temp[k]; k1++;
}
for (k=tn-1; k>=j+1; k--) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorB]++; c[color1]++;
v[i]=true; break;
}
}
// Case 3
if (c[color1]==0 && c[color2]>0) {
colorA=G.c[t[tn-1]][j];
colorB=G.c[t[0]][t[tn-1]];
color=G.c[t[j]][t[j+1]];
if (c[colorA]>0) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0;
for (k=0; k<=j; k++) {
t[k1]=temp[k]; k1++;
}
}
}

```



```

}
for (k=tn-1; k>=j+1; k--) {
t[k1]=temp[k]; k1++;
}
t[k1]=i; tn++;
c[color]--; c[colorA]++; c[color2]++;
v[i]=true; break;
}
if (c[colorB]>0) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0;
t[k1]=i; k1++;
for (k=j+1; k<=tn-1; k++) {
t[k1]=temp[k]; k1++;
}
for (k=0; k<=j; k++) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorB]++; c[color2]++;
v[i]=true; break;
}
}
}
if (j<tn-1) break; // Already find a suitable city
}
if (i>=vn) {
// Can not find a suitable city,
// then find the one with the highest frequency label
k=0; fq=-1;
for (j=0; j<vn; j++) {
if (v[j]) continue;
maxcolor=G.c[t[tn-1]][j];
if (G.freq[maxcolor]>fq) {
fq=G.freq[maxcolor];
k=j;
}
}
else if (G.freq[maxcolor]==fq) {
if (rand()%2==1) k=j;
}
}
t[tn]=k; v[t[tn]]=true;
color=G.c[t[tn-1]][t[tn]];
c[color]++; tn++;
}
}
color=G.c[t[vn-1]][t[0]];

```

```

c[color]++;
}

void main() {
FILE *fp;
Graph G;
Tour T1,T2;
int i,j;
int value1,value2;
char filename[40];
time_t u1,u2;
double utime;
double AvgValue;
int gn;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Group size: ");
scanf("%d",&gn);
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
T1.InitTour(G);
T2.InitTour(G);
utime=0.0;
AvgValue=0;
for (j=0; j<gn; j++) {
G.InitGraph(fp);
u1=time(NULL);
T1.ClearTour();
T1.EMSH(G);
value1=T1.Value();
for (i=0; i<200; i++) {
T2.ClearTour();
T2.EMSH(G);
value2=T2.Value();
if (value2<value1) {
value1=value2;
T1.CopyTour(T2);
}
}
u2=time(NULL);
utime=utime+difftime(u2,u1);

```

```
T1.PrintTour();
printf("value(%d): %d\n",j+1,T1.Value());
AvgValue=AvgValue+T1.Value();
G.ClearGraph();
}
printf("Running time: %f\n",utime/gn);
printf("Average Value: %f\n",AvgValue/gn);
fclose(fp);
}
```

```

// CTSP_GA.cpp
// Solve CTSP by Genetic Algorithm (GA)

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define VN 500
#define CN 500

class Graph {
public: int *c[VN];
int cn;
int vn;
int freq[CN];
public: void InitGraph(FILE *fp);
void ClearGraph();
int ReadNumber(FILE *fp);
void PrintGraph();
};

class Tour {
public: int *t;
int tn;
int vn;
int cn;
public: void InitTour(Graph G);
void ClearTour();
void GetLabels(Graph G);
void EMSh(Graph G);
void Crossover(Graph G, Tour second);
void Mutation(Graph G);
void CopyTour(Tour tour);
void PrintTour();
int Value();
private: bool *v;
int *c;
private: int NumComponents(Graph G);
};

typedef struct node_type {
int k;
struct node_type *next;
} node;

class List {
public: node *root;
};

```

```

public: List() { root=NULL; }
public: bool Iseempty();
void Insert(int i);
int Get();
void PrintList();
};

bool List::Iseempty() {
if (root==NULL) return true;
return false;
}

void List::Insert(int i) {
node *r1,*r2;

r1=root;
if (r1==NULL) {
r1=new node;
r1->k=i; r1->next=NULL;
root=r1; return;
}
r2=r1;
while (r1!=NULL) {
if (r1->k==i) return;
r2=r1; r1=r1->next;
}
r1=new node;
r1->k=i; r1->next=NULL;
r2->next=r1;
}

int List::Get() {
node *r;
int m;

r=root;
if (r==NULL) return -1;
m=r->k;
root=r->next;
delete r;
return m;
}

void List::PrintList() {
node *r;

r=root;

```

```

printf("list: ");
while (r!=NULL) {
printf("%d ",r->k);
r=r->next;
}
printf("\n");
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;

for (i=0; i<vn; i++) {
c[i]=new int[VN];
}
for (i=0; i<cn; i++) freq[i]=0;
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) { c[i][j]=-1; continue; }
k=ReadNumber(fp);
c[i][j]=k; c[j][i]=k; freq[k]++;
}
}
}

void Graph::ClearGraph() {
int i;

for (i=0; i<vn; i++) {
delete c[i];
}
}

void Graph::PrintGraph() {
int i,j;

printf("----- Graph ----- \n");
for (i=0; i<vn; i++) {
printf("%d: ",i);
for (j=0; j<vn; j++) {
if (i==j) continue;
printf("%d->%d(%d) ",i,j,c[i][j]);
}
printf("\n");
}
}

int Graph::ReadNumber(FILE *fp) {

```

```

char buf[10];
char c;
int i,n;
int num;

c=fgetc(fp);
i=0;
while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
while (!feof(fp) && (c>='0' && c<='9')) {
buf[i]=c; c=fgetc(fp); i++;
}
//buf[i]='\0';
num=0;
n=i;
for (i=0; i<n; i++) {
num=num*10+(int)(buf[i]-'0');
}
return num;
}

void Tour::InitTour(Graph G) {
int i;

vn=G.vn;
cn=G.cn;
tn=0;
t=new int[VN];
v=new bool[VN];
c=new int[CN];
for (i=0; i<vn; i++) v[i]=false;
for (i=0; i<cn; i++) c[i]=0;
}

void Tour::ClearTour() {
int i;

tn=0;
for (i=0; i<vn; i++) v[i]=false;
for (i=0; i<cn; i++) c[i]=0;
}

int Tour::Value() {
int i;
int value=0;

for (i=0; i<cn; i++) {
if (c[i]>0) value++;
}
}

```

```

}
return value;
}

void Tour::CopyTour(Tour tour) {
int i;

vn=tour.vn; cn=tour.cn; tn=tour.tn;
for (i=0; i<tn; i++) t[i]=tour.t[i];
for (i=0; i<vn; i++) v[i]=tour.v[i];
for (i=0; i<cn; i++) c[i]=tour.c[i];
}

void Tour::PrintTour() {
int i;

printf("Tour: \n");
for (i=0; i<tn; i++) {
printf("%d->",t[i]);
}
printf("%d\n",t[0]);
}

int Tour::NumComponents(Graph G) {
List list;
int i,k,j;
int num,color;
bool visited[VN];

for (i=0; i<vn; i++) visited[i]=false;
num=0;
for (i=0; i<vn; i++) {
if (visited[i]) continue;
list.Insert(i); num++;
while (!list.Isempty()){
//list.PrintList();
k=list.Get(); visited[k]=true;
for (j=0; j<vn; j++) {
if (k==j) continue;
if (visited[j]) continue;
color=G.c[k][j];
if (c[color]>0) list.Insert(j);
}
}
}
return num;
}

```



```

void Tour::GetLabels(Graph G) {
int num;
int i;

num=NumComponents(G);
while (num>1) {
//printf("num=%d\n",num);
i=rand()%cn;
while (c[i]>0) i=rand()%cn;
c[i]=1;
num=NumComponents(G);
}
/*for (i=0; i<cn; i++) {
if (c[i]>0) printf("%d ",i);
}
printf("\n");*/
}

void Tour::EMSH(Graph G) {
int i,j;
int k,color,color1,color2;
int k1,colorA,colorB;
int temp[VN];
int maxcolor,fq;
bool flag[CN];

for (i=0; i<cn; i++) {
if (c[i]>0) flag[i]=true;
else flag[i]=false;
c[i]=0;
}
// Begin with a random city
t[0]=rand()%vn; v[t[0]]=true;
// Find the second city
for (i=0; i<vn; i++) {
if (i==t[0]) continue;
color=G.c[t[0]][i];
if (flag[color]) break;
}
if (i>=vn) {
printf("Error\n"); return;
}
t[1]=i; v[i]=true;
color=G.c[t[0]][t[1]]; c[color]++;

// Find the third city

```

```

for (i=0; i<vn; i++) { // Find a suitable city
if (v[i]) continue;
color=G.c[t[1]][i];
if (flag[color]) {
t[2]=i; c[color]++;
v[t[2]]=true; break;
}
color=G.c[t[0]][i];
if (flag[color]) {
t[2]=t[1]; t[1]=t[0]; t[0]=i;
c[color]++;
v[t[0]]=true; break;
}
}
if (i>=vn) {
printf("Error\n"); return;
}
tn=3;

// Find the rest of cities
while (tn<vn) {
/*printf("Tour: ");
for (i=0; i<tn; i++) printf("%d ",t[i]);
printf("\n");*/
// Find the next suitable city
for (i=0; i<vn; i++) {
if (v[i]) continue;
// Consider the two end points of the partial tour
color=G.c[t[tn-1]][i];
if (flag[color]) {
t[tn]=i; tn++; v[i]=true;
c[color]++; break;
}
color=G.c[t[0]][i];
if (flag[color]) {
for (j=tn; j>=1; j--) t[j]=t[j-1];
t[0]=i; tn++; c[color]++;
v[t[0]]=true; break;
}
// Consider three cases to insert into the partial tour
for (j=0; j<tn-1; j++) {
color1=G.c[t[j]][i];
color2=G.c[t[j+1]][i];
// Case 1
if (flag[color1] && flag[color2]) {
color=G.c[t[j]][t[j+1]];
c[color]--; // cut edge (t[j],t[j+1])
}
}
}

```

```

for (k=tn; k>j+1; k--) t[k]=t[k-1];
t[j+1]=i; tn++; // add edges (t[j],i) and (i,t[j+1]);
v[i]=true; c[color1]++; c[color2]++;
break;
}
// Case 2
if (flag[color1] && !flag[color2]) {
colorA=G.c[t[0]][t[j+1]];
colorB=G.c[t[0]][t[tn-1]];
color=G.c[t[j]][t[j+1]];
if (flag[colorA]) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0; t[k1]=i; k1++;
for (k=j; k>=0; k--) {
t[k1]=temp[k]; k1++;
}
for (k=j+1; k<=tn-1; k++) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorA]++; c[color1]++;
v[i]=true; break;
}
if (flag[colorB]) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0; t[k1]=i; k1++;
for (k=j; k>=0; k--) {
t[k1]=temp[k]; k1++;
}
for (k=tn-1; k>=j+1; k--) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorB]++; c[color1]++;
v[i]=true; break;
}
}
// Case 3
if (!flag[color1] && flag[color2]) {
colorA=G.c[t[tn-1]][j];
colorB=G.c[t[0]][t[tn-1]];
color=G.c[t[j]][t[j+1]];
if (flag[colorA]) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0;
for (k=0; k<=j; k++) {
t[k1]=temp[k]; k1++;
}
}
}

```

```

}
for (k=tn-1; k>=j+1; k--) {
t[k1]=temp[k]; k1++;
}
t[k1]=i; tn++;
c[color]--; c[colorA]++; c[color2]++;
v[i]=true; break;
}
if (flag[colorB]) {
for (k=0; k<tn; k++) temp[k]=t[k];
k1=0;
t[k1]=i; k1++;
for (k=j+1; k<=tn-1; k++) {
t[k1]=temp[k]; k1++;
}
for (k=0; k<=j; k++) {
t[k1]=temp[k]; k1++;
}
tn++;
c[color]--; c[colorB]++; c[color2]++;
v[i]=true; break;
}
}
}
if (j<tn-1) break; // Already find a suitable city
}
if (i>=vn) {
// Can not find a suitable city,
// then find the one with the highest frequency label
k=0; fq=-1;
for (j=0; j<vn; j++) {
if (v[j]) continue;
maxcolor=G.c[t[tn-1]][j];
if (G.freq[maxcolor]>fq) {
fq=G.freq[maxcolor];
k=j;
}
}
t[tn]=k; v[k]=true;
color=G.c[t[tn-1]][t[tn]];
c[color]++; tn++;
flag[color]=true;
}
}
color=G.c[t[vn-1]][t[0]];
c[color]++;
}

```

```

void Tour::Crossover(Graph G, Tour second) {
Tour child;
int labelset[CN];
int set1[CN];
int i,j,k;
int i1,j1;
int temp;

// Union the two labelsets
k=0;
for (i=0; i<cn; i++) {
if (c[i]>0 || second.c[i]>0) {
labelset[k]=i; k++;
}
}
// Sort the labelset
for (i=0; i<k; i++) {
i1=labelset[i];
for (j=i+1; j<k; j++) {
j1=labelset[j];
if (G.freq[i1]<G.freq[j1]) {
temp=labelset[i]; labelset[i]=labelset[j];
labelset[j]=temp;
}
}
}
// Determine the label set.
child.InitTour(G);
k=0;
while (child.NumComponents(G)>1) {
i=labelset[k]; child.c[i]=1;
set1[k]=i; k++;
}
for (j=k-1; j>=0; j--) {
i=set1[j]; child.c[i]=0;
if (child.NumComponents(G)>1) child.c[i]=1;
}
child.EMSH(G);
if (child.Value()<=Value()) CopyTour(child);
}

void Tour::Mutation(Graph G) {
int i,j,k;
int i1,j1,k1;
int c1,c2,nc1,nc2;
int value1,value2;

```

```

bool flag;
int temp[VN];

for (i=0; i<vn; i++) {
j=(i+2)%vn; flag=false;
while ((j+vn-i)%vn<=1) {
i1=(i+1)%vn; j1=(j+1)%vn;
c1=G.c[t[i]][t[i1]]; c2=G.c[t[j]][t[j1]];
nc1=G.c[t[i]][t[j]]; nc2=G.c[t[i1]][t[j1]];
value1=Value();
c[c1]--; c[c2]--;
c[nc1]++; c[nc2]++;
value2=Value();
if (value2<value1) {
flag=true;
for (k=0; k<vn; k++) temp[k]=t[k];
k1=0;
t[k1]=temp[i]; k1++;
for (k=j; k>=i1; k--) {
t[k1]=temp[k]; k1++;
}
for (k=j1; k<vn; k++) {
t[k1]=temp[k]; k1++;
}
for (k=0; k<i; k++) {
t[k1]=temp[k]; k1++;
}
break;
}
c[c1]++; c[c2]++;
c[nc1]--; c[nc2]--;
j=(j+1)%vn;
}
if (flag) break;
}
}

void main() {
FILE *fp;
Graph G;
Tour *T;
int i,j,k,k1;
int value,value1;
char filename[40];
time_t u1,u2;
double utime;
double AvgValue;

```

```

int gn,pn;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
printf("Population size: ");
scanf("%d",&pn);
printf("Group size: ");
scanf("%d",&gn);
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
T=new Tour[pn];
for (i=0; i<pn; i++) {
T[i].InitTour(G);
}
utime=0.0;
AvgValue=0;
for (j=0; j<gn; j++) {
G.InitGraph(fp);
u1=time(NULL);
for (i=0; i<pn; i++) {
T[i].ClearTour();
T[i].GetLabels(G);
T[i].EMSH(G);
}
for (i=1; i<pn; i++) {
for (k=0; k<pn; k++) {
k1=(k+i)%pn;
T[k].Crossover(G,T[k1]);
T[k].Mutation(G);
}
}
value=T[0].Value(); k=0;
for (i=1; i<pn; i++) {
value1=T[i].Value();
if (value1<value) {
value=value1; k=i;
}
}
u2=time(NULL);
utime=utime+difftime(u2,u1);
T[k].PrintTour();
printf("value(%d): %d\n",j+1,T[k].Value());

```

```
AvgValue=AvgValue+T[k].Value();
G.ClearGraph();
}
printf("Running time: %f\n",utime/gn);
printf("Average Value: %f\n",AvgValue/gn);
fclose(fp);
}
```



```

// CTSP_OPT.cpp
// Solve CTSP by an optimal algorithm
// using the backtrack search and
// a recursive algorithm to find a tour.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define VN 80
#define CN 100

class Graph {
public: int *c[VN];
int cn;
int vn;
int freq[CN];
int sortL[CN];
public: void InitGraph(FILE *fp);
void ClearGraph();
int ReadNumber(FILE *fp);
void PrintGraph();
private: void Sort();
};

class Tour {
public: int *t;
int tn;
int vn;
int cn;
public: void InitTour(Graph G);
void ClearTour();
bool TryOpT(Graph G, int pk);
bool ExistTour(Graph G, int currentpt, int position);
bool ExistTour1(Graph G);
void GetTour(Graph G);
bool IsConnect(Graph G);
void CopyTour(Tour tour);
void PrintTour();
int Value();
private: bool *v;
bool *c;
private: bool edge[VN][VN];
int degree[VN];
};

typedef struct node_type {

```

```

int k;
struct node_type *next;
} node;

class List {
public: node *root;
public: List() { root=NULL; }
public: bool Isempy();
void Insert(int i);
int Get();
};

bool List::Isempy() {
if (root==NULL) return true;
return false;
}

void List::Insert(int i) {
node *r1,*r2;

r1=root;
if (r1==NULL) {
r1=new node;
r1->k=i; r1->next=NULL;
root=r1; return;
}
r2=r1;
while (r1!=NULL) {
if (r1->k==i) return;
r2=r1; r1=r1->next;
}
r1=new node;
r1->k=i; r1->next=NULL;
r2->next=r1;
}

int List::Get() {
node *r;
int m;

r=root;
if (r==NULL) return -1;
m=r->k;
root=r->next;
delete r;
return m;
}

```

```

void Graph::Sort() {
int i,j;
int temp;

for (i=0; i<cn; i++) sortL[i]=i;
for (i=0; i<cn; i++) {
for (j=i+1; j<cn; j++) {
if (freq[sortL[i]]<freq[sortL[j]]) {
temp=sortL[i]; sortL[i]=sortL[j];
sortL[j]=temp;
}
}
}
}

void Graph::InitGraph(FILE *fp) {
int i,j,k;

for (i=0; i<vn; i++) {
c[i]=new int[VN];
}
for (i=0; i<cn; i++) freq[i]=0;
for (i=0; i<vn; i++) {
for (j=i; j<vn; j++) {
if (i==j) { c[i][j]=-1; continue; }
k=ReadNumber(fp);
c[i][j]=k; c[j][i]=k; freq[k]++;
}
}
Sort();
}

void Graph::ClearGraph() {
int i;

for (i=0; i<vn; i++) {
delete c[i];
}
}

void Graph::PrintGraph() {
int i,j;

printf("----- Graph -----\\n");
for (i=0; i<vn; i++) {
printf("%d: ",i);

```

```

for (j=0; j<vn; j++) {
    if (i==j) continue;
    printf("%d->%d(%d) ",i,j,c[i][j]);
}
printf("\n");
}
for (i=0; i<cn; i++) printf("%d ",sortL[i]);
printf("\n");
}

int Graph::ReadNumber(FILE *fp) {
    char buf[10];
    char c;
    int i,n;
    int num;

    c=fgetc(fp);
    i=0;
    while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
    while (!feof(fp) && (c>='0' && c<='9')) {
        buf[i]=c; c=fgetc(fp); i++;
    }
    //buf[i]='\0';
    num=0;
    n=i;
    for (i=0; i<n; i++) {
        num=num*10+(int)(buf[i]-'0');
    }
    return num;
}

void Tour::InitTour(Graph G) {
    int i;

    vn=G.vn;
    cn=G.cn;
    tn=0;
    t=new int[VN];
    v=new bool[VN];
    c=new bool[CN];
    for (i=0; i<vn; i++) v[i]=false;
    for (i=0; i<cn; i++) c[i]=false;
}

void Tour::ClearTour() {
    int i;

```

```

tn=0;
for (i=0; i<vn; i++) t[i]=0;
for (i=0; i<vn; i++) v[i]=false;
for (i=0; i<cn; i++) c[i]=false;
}

int Tour::Value() {
int i;
int value=0;

for (i=0; i<cn; i++) {
if (c[i]) value++;
}
return value;
}

void Tour::CopyTour(Tour tour) {
int i;

vn=tour.vn; cn=tour.cn; tn=tour.tn;
for (i=0; i<tn; i++) t[i]=tour.t[i];
for (i=0; i<vn; i++) v[i]=tour.v[i];
for (i=0; i<cn; i++) c[i]=tour.c[i];
}

void Tour::PrintTour() {
int i;

printf("Tour: \n");
for (i=0; i<tn; i++) {
printf("%d->",t[i]);
}
printf("%d\n",t[0]);
}

bool Tour::IsConnect(Graph G) {
int i,j,k;
List list;
bool visit[VN];

for (i=0; i<vn; i++) visit[i]=false;
i=0;
list.Insert(i);
while (!list.Isempty()) {
i=list.Get();
visit[i]=true;
for (j=0; j<vn; j++) {

```

```

if (i==j) continue;
if (visit[j]) continue;
k=G.c[i][j];
if (c[k] && edge[i][j]) list.Insert(j);
}
}
for (i=0; i<vn; i++) {
if (!visit[i]) return false;
}
return true;
}

void Tour::GetTour(Graph G) {
int i,j,k,i1,k1,j1;
int nt[VN];
int len;
bool visit[VN];
bool backedge[VN][VN];
int color;
int temp[VN];
int flag;
bool flag1;

printf("Label set: ");
for (i=0; i<cn; i++) {
if (c[i]) printf("%d ",i);
}
printf("\n");
for (i=0; i<vn; i++) visit[i]=false;
for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) backedge[i][j]=false;
}
nt[0]=0; visit[0]=true;
for (i=1; i<vn; i++) {
color=G.c[0][i];
if (c[color]) break;
}
nt[1]=i; visit[i]=true; len=2; flag=0; flag1=true;
while (len<vn) {
/*for (i=0; i<len; i++) printf("%d->",nt[i]);
printf("\n");
getchar();*/
if (flag>2 && !flag1) break;
if (flag>2 && flag1) {
for (k1=0; k1<len; k1++) temp[k1]=nt[k1];
for (k1=0; k1<len; k1++) nt[k1]=temp[len-k1-1];
flag1=false;
}
}
}

```

```

}
i=nt[len-1];
for (j=0; j<vn; j++) {
if (i==j) continue;
if (visit[j]) continue;
color=G.c[i][j];
if (c[color]) break;
}
if (j<vn) { // Find a new node
nt[len]=j; visit[j]=true;
len++; flag=0;
continue;
}
// Refine the partial tour
i1=0;
for (j=0; j<vn; j++) {
if (i==j) continue;
color=G.c[i][j];
if (!c[color]) continue;
if (visit[j] && !backedge[i][j]) {
for (k=0; k<len; k++) {
if (j==nt[k]) break;
}
//printf("k=%d, j=%d\n",k,j);
if (k==len-2) continue;
//if (degree[nt[k+1]]==2) continue;
for (k1=0; k1<len; k1++) temp[k1]=nt[k1];
k1=k+1;
for (j1=len-1; j1>=k+1; j1--) {
nt[k1]=temp[j1]; k1++;
}
backedge[i][j]=true;
break;
}
}
if (j<vn) flag++;
else if (flag1) {
for (k1=0; k1<len; k1++) temp[k1]=nt[k1];
for (k1=0; k1<len; k1++) nt[k1]=temp[len-k1-1];
flag1=false;
}
else break;
}
printf("len=%d\n",len);
printf("Get tour: ");
for (i=0; i<len; i++) {
printf("%d->",nt[i]);

```

```

}
printf("%d\n",nt[0]);
}

bool Tour::ExistTour1(Graph G) {
int i,j,k;
int temp;
bool flag;

for (i=0; i<vn; i++) {
for (j=0; j<vn; j++) edge[i][j]=true;
}
// Return false if a node has degree less than 1
for (i=0; i<vn; i++) {
degree[i]=0;
for (j=0; j<vn; j++) {
if (i==j) continue;
k=G.c[i][j];
if (c[k]) degree[i]++;
}
if (degree[i]<2) return false;
}
for (i=0; i<vn; i++) {
if (degree[i]<3) continue;
temp=0;
for (j=0; j<vn; j++) {
if (i==j) continue;
k=G.c[i][j];
if (c[k] && degree[j]==2) temp++;
}
if (temp>=3) return false;
if (temp==2) {
for (j=0; j<vn; j++) {
if (i==j) continue;
k=G.c[i][j];
if (c[k] && degree[j]>2) {
edge[i][j]=false; edge[j][i]=false;
}
}
}
}
if (!IsConnect(G)) return false;
flag=ExistTour(G,0,0);
if (flag) GetTour(G);
return flag;
}

```



```

bool Tour::ExistTour(Graph G, int currentpt, int position) {
int i,j;
bool flag;

if (position==vn-1) {
t[position]=currentpt;
v[currentpt]=true;
j=G.c[t[0]][currentpt];
if (c[j]) return true;
else {
v[currentpt]=false;
return false;
}
}
t[position]=currentpt; v[currentpt]=true;
/*printf("Tour: ");
for (j=0; j<=position; j++) printf("%d ",t[j]);
printf("\n");*/
for (i=0; i<vn; i++) {
if (i==currentpt) continue;
if (v[i]) continue;
j=G.c[i][currentpt];
if (!c[j]) continue;
if (!edge[i][currentpt]) continue;
flag=false;
flag=ExistTour(G,i,position+1);
if (flag) return true;
v[i]=false;
}
return false;
}

bool Tour::TryOpT(Graph G, int pk) {
int i,j,k,k1;
int end;
bool flag;
int index[CN];
int total;

total=0;
for (i=0; i<pk; i++) {
index[i]=i;
k=G.sortL[i]; c[k]=true;
total=total+G.freq[k];
}
if (total<G.vn) return false;
flag=true;

```

```

while (flag) {
  if (ExistTour1(G)) {
    printf("OPT solution: ");
    for (i=0; i<vn; i++) printf("%d->",t[i]);
    printf("%d\n",t[0]);
    printf("Colors: ");
    for (i=0; i<pk; i++) printf("%d ",G.sortL[index[i]]);
    printf("\n");
    return true;
  }
  end=index[pk-1]; k=G.sortL[end]; c[k]=false;
  total=total-G.freq[k];
  for (i=end+1; i<cn; i++) {
    index[pk-1]=i; k=G.sortL[i]; c[k]=true;
    total=total+G.freq[k];
    if (total<G.vn) {
      c[k]=false;
      total=total-G.freq[k];
      break;
    }
    if (ExistTour1(G)) {
      printf("OPT solution: ");
      for (j=0; j<vn; j++) printf("%d->",t[j]);
      printf("%d\n",t[0]);
      for (j=0; j<pk; j++) printf("%d ",G.sortL[index[j]]);
      printf("\n");
      return true;
    }
    c[k]=false;
    total=total-G.freq[k];
  }
  k=pk-2; k1=2;
  if (k<0) return false;
  while (index[k]+k1>=cn) {
    j=index[k];
    c[G.sortL[j]]=false;
    total=total-G.freq[G.sortL[j]];
    k--; k1++;
    if (k<0) return false;
  }
  k1=index[k]; c[G.sortL[k1]]=false;
  total=total-G.freq[G.sortL[k1]];
  for (i=k; i<pk; i++) {
    index[i]=k1+1+(i-k);
    j=index[i];
    if (j>=cn) return false;
    c[G.sortL[j]]=true;
  }
}

```

```

total=total+G.freq[G.sortL[j]];
}
}
return false;
}

void main() {
FILE *fp;
Graph G;
Tour T;
int i,j;
char filename[40];
time_t u1,u2;
double utime;
double AvgValue;

printf("filename: ");
scanf("%s",filename);
fp=fopen(filename,"rt");
if (fp==NULL) {
printf("cannot open file.\n");
return;
}
G.vn=G.ReadNumber(fp);
G.cn=G.ReadNumber(fp);
T.InitTour(G);
utime=0.0;
AvgValue=0;
for (j=0; j<10; j++) {
G.InitGraph(fp);
/*for (i=0; i<G.cn; i++) printf("%d(%d) ",G.sortL[i],G.freq[G.sortL[i]]);
printf("\n");*/
u1=time(NULL);
T.ClearTour();
for (i=1; i<=G.cn; i++) {
//printf("-----Try %d colors-----\n",i);
if (T.TryOpT(G,i)) break;
}
u2=time(NULL);
utime=utime+difftime(u2,u1);
printf("value(%d): %d \t time=%f\n",j+1,T.Value(),difftime(u2,u1));
AvgValue=AvgValue+T.Value();
G.ClearGraph();
}
printf("Running time: %f\n",utime/10);
printf("Average Value: %f\n",AvgValue/10);
fclose(fp);
}

```

}

## BIBLIOGRAPHY

- [1] B. Brassard and P. Bratley, "Algorithmics Theory and Practice," Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [2] T. Brüggemann, J. Monnot, and G.J. Woeginger, "Local search for the minimum label spanning tree problem with bounded color classes," *Oper. Res. Lett.*, vol. 31, pp. 195-201, 2003.
- [3] R.-S. Chang and S.-J. Leu, "The minimum labeling spanning trees," *Inf. Process. Lett.*, vol. 63, no. 5, pp. 277-282, 1997.
- [4] H. Chou, G. Premkumar and C.-H. Chu, "Genetic algorithms for communications network design - An empirical study of the factors that influence performance," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 3, pp. 236-249, 2001.
- [5] C. Duin and S. Voss, "The pilot method," *Networks*, vol. 34, pp. 181-191, 1999.
- [6] D. Eppstein, "Finding the  $k$  smallest spanning trees," *BIT*, vol. 32, pp. 237-248, 1992.
- [7] P.M. Camerini, F. Maffinoli, S. Martello, and P.Toth, "Most and least uniform spanning tree," *Discrete Appl. Math*, vol. 15, pp. 181-197, 1986.
- [8] Z. Galil and B.Schieber, "On finding most uniform spanning trees," *Discrete Appl. Math.*, vol. 20, pp. 173-175, 1988.
- [9] B. Golden, S. Raghavan and D. Stanojević, "Heuristic search for the generalized minimum spanning tree problem," *INFORMS Journal on Computing*, to appear, 2004.
- [10] J.-H. Ho, D.T. Lee, C.-H. Chang, and C.K. Wong. "Minimum diameter spanning trees and related problems," *SIAM J. Comput.*, vol. 20, pp. 987-997, 1991.

- [11] S.O. Krumke and H.-C. Wirth, "On the minimum label spanning tree problem," *Inf. Process. Lett.*, vol. 66, no. 2, pp. 81-85, 1998.
- [12] C.C. Palmer and A. Kershenbaum, "An approach to a problem in network design using genetic algorithms," *Networks*, vol. 26, no. 3, pp. 151-163, 1995.
- [13] S. Voss, R. Cerulli, C. Duin, M. Fernandes, A. Fink, M. Gentili, and L. Gouveia, "Applications of the pilot method to hard modifications of the minimum spanning tree problem," Presented at the *INFORMS Annual Meeting*, Denver, Colorado, October 2004.
- [14] Y. Wan, G. Chen, and Y. Xu, "A note on the minimum label spanning tree," *Inf. Process. Lett.*, vol. 84, pp. 99-101, 2002.
- [15] Y. Xiong, B. Golden, and E. Wasil, "Worst-case behavior of the MVCA heuristic for the minimum labeling spanning tree problem," *Oper. Res. Lett.*, vol. 33, no. 1, pp. 77-80, 2005.
- [16] Y. Xiong, B. Golden, and E. Wasil, "A one-parameter genetic algorithm for the minimum labeling spanning tree problem," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 1, pp. 55-60, 2005.
- [17] X. Shen and W. Liang. "A parallel algorithm for multiple edge updates of minimum spanning trees." *Proc. 7th Internet. Parallel Processing Symp.*, pp. 310-317, 1993.