Summer 2015

Labeled Trees and Spanning Trees: Computational Discrete Mathematics and Applications

Demet Yalman

LABELED TREES AND SPANNING TREES: COMPUTATIONAL DISCRETE MATHEMATICS AND APPLICATIONS

by

DEMET YALMAN

(Under the Direction of Hua Wang)

ABSTRACT

In this thesis, we examine two topics. In the first part, we consider Leech tree which is a tree of order n with positive integer edge weights such that the weighted distances between pairs of vertices are exactly $1, 2, \ldots, \binom{n}{2}$. Only five Leech trees are known and some non-existence results have been presented through the years. Variations of Leech trees such as the minimal distinct distance trees and modular Leech trees have been considered in recent years. In this thesis, such Leech-type questions on distances between leaves are studied as well as some other labeling questions related to the original motivation for Leech trees. In the second part, we consider the question of finding spanning trees under various restrictions. A "dense" tree, from graph theoretical point of view, has small total distances between vertices and large number of substructures. In this thesis, the "density" of a spanning tree is conveniently measured by the total distance of the tree. An edge-swap heuristic for generating "dense" spanning trees is developed by utilizing established conditions and relations between trees with the minimum total distance.

Key Words: edge-swap heuristic, dense tree, minimum spanning tree, Leech tree, modular Leech tree, distances between leaves

2009 *Mathematics Subject Classification*: 90C27, 05C78

LABELED TREES AND SPANNING TREES: COMPUTATIONAL DISCRETE MATHEMATICS AND APPLICATIONS

by

DEMET YALMAN

B.S. in Mathematics and Computer Science

B.S. in Computer Engineering

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment
of the Requirement for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

2015

LABELED TREES AND SPANNING TREES: COMPUTATIONAL DISCRETE MATHEMATICS AND APPLICATIONS

by

DEMET YALMAN

Major Professor: Hua Wang

Committee: Goran Lesaja

Colton Magnant

Electronic Version Approved:

July 23, 2015

DEDICATION

This thesis is dedicated to my all family and to all who believed and supported me during my whole education life.

ACKNOWLEDGEMENTS

First and foremost, I would like to show my deepest gratitude to my advisor, Dr. Hua Wang for his excellent guidance, motivation, patience, enthusiasm and providing me with an excellent atmosphere for doing research. I am immensely grateful to him for his insights and sharing his pearls of wisdom with me during this research. I could not have imagined having a better advisor and mentor for my study.

I want to thank my colleagues for their insight. Especially, I want to thank Mustafa Ozen for assistance with implementation, contributions and comments that greatly improved the manuscript. Also, I would like to thank to my department providing me to have a great opportunity for my thesis and expertise that greatly assisted the research.

I would never have been able to finish my thesis without guidance of my committee members, help from my friends, and support from my family who always encourage me with their best wishes.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Definitions and Notations

A *Leech tree* of order n is a tree whose edges are weighted by positive integers and the weighted distance between the $\binom{n}{2}$ pairs of vertices are exactly $1, 2, \ldots, \binom{n}{2}$. This concept was proposed by John Leech in 1975 [4], motivated from the question of finding an efficient design for electrical circuits.

Given an undirected graph G with vertex set V and edge set E, a subtree of G is a connected acyclic subgraph of G. A subtree with vertex set V is a spanning tree of G. The question of finding spanning trees (under various restrictions) of a given graph is of importance in many applications such as Information Technology and Network Design.

1.2 Previous Work

1.2.1 Leech Labeling

Throughout the years various properties of Leech trees have been presented [5, 6, 7]. The following are the five known Leech trees (Figure 1.1) and it is conjectured that they are the only Leech trees.

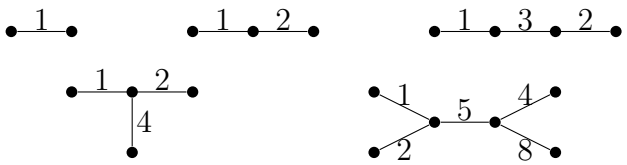


Figure 1.1: The known Leech trees

More recently, variations of Leech trees have been introduced and studied. Such concepts include the minimal distinct distance trees [1] and modular Leech trees [2, 3].

1.2.2 Spanning Tree

Many questions about spanning trees have been studied, including, but not limited to, the well-known minimum-weight spanning tree problem (MSTP), spanning trees with bounded degree, with bounded number of leaves, or with bounded number of branch vertices.

For MSTP which is one of the most popular combinatorial optimization problems, efficient methods have been studied over the years. In the last few decades, some faster algorithms for finding MSTP have been developed in order to make time bound one step closer to linearity[16]. Expected linear-time method was proposed by Karger, Klein and Tarjan[17].

Throughout years, many studies have been done for spanning trees with bounded degree [18]. In connected graphs, spanning trees with various degree restrictions have been studied [19]. Furthermore, biconnected spanning subgraphs with bounded degree have been considered [20, 21, 22].

The goal in such studies is usually to find efficient algorithms. In a recent work, an edge-swap heuristic for generating spanning trees with minimum number of branch vertices was presented [10], where an efficient algorithm resulted from iteratively reducing the number of branch vertices from a random spanning tree by swapping tree edges with edges not currently in the tree.

A tree of given number of vertices is considered "dense" if the number of substructures (including isomorphic subgraphs) is large or the total distance between vertices is small. In applications, such spanning trees have obvious advantages such as having more choices of sub-networks and efficient transfer of resources with minimum cost. In this thesis, we consider an edge-swap heuristic, inspired by similar work presented in [10], for finding dense spanning trees. Computational results are presented for randomly generated graphs and specific examples originated from applications.

CHAPTER 2

LEECH TYPE LABELING OF TREES

2.1 Leaf-Leech Trees

In many areas of study, the distances between leaves are of interest in addition to the distances between all vertices. Motivated by the concept of Leech tree, we define the *leaf-Leech tree* as follows.

Definition 1. *A tree T with n leaves is a leaf-Leech tree if the distances between pairs of leaves are exactly $3, 4, \ldots, \binom{n}{2} + 2$.*

Remark 1. *Since only distances between leaves are considered, we do not require the presence of 1 or 2 among the distances. Also, note that the distances considered for leaf-Leech trees are not weighted.*

For Leech trees, the beautiful Taylor's condition [6] asserts that the order n of a Leech tree must be a perfect square or a perfect square plus two. Below is an analogous statement following very similar argument as that in [6].

Proposition 2.1.1. *If there is a leaf-Leech tree on n leaves, we must have $n = k^2$ or $n = k^2 + 2$ for some k.*

Proof. Let T be a leaf-Leech tree on n leaves and v be one of the leaves. Define

- O to be the set of leaves at odd distance from v;
- E to be the set of leaves at even distance from v (note that $v \in E$).

First note that the distance between a pair of vertices in O (E) is even and the distance between a vertex in O and a vertex in E is odd. Now consider two cases:

- If $\binom{n}{2}$ is even, then the number of odd distances between leaves is the same as the number of even distances. Consequently

$$\binom{|O|}{2} + \binom{|E|}{2} = |O| \cdot |E|,$$

which can be rewritten as

$$n = |O| + |E| = (|O| - |E|)^2.$$

- If $\binom{n}{2}$ is odd, then the number of odd distances between leaves is one more than the number of even distances. Consequently

$$\binom{|O|}{2} + \binom{|E|}{2} + 1 = |O| \cdot |E|,$$

which can be rewritten as

$$n = |O| + |E| = (|O| - |E|)^2 + 2.$$

■

It is shown in [5], among more general results, that other than the ones shown in Figure 1.1, no Leech tree can be a star which is a tree with one internal node and n leaves. Similar arguments yield the analogous conclusion that there are only a few *starlike* (with exactly one vertex of degree at least 3) leaf-Leech trees.

Proposition 2.1.2. *There is no starlike leaf-Leech trees on more than 4 leaves.*

Proof. Let T be a starlike leaf-Leech tree on at least 5 leaves. In order to have distance 3 between leaves, we must have a pendant edge and a pendant path of length 2.

Figure 2.1: The tree with 2 leaves

Then, in order to have distance 4 between leaves, the next shortest pendant path must be of length 3. The distances between these three leaves are 3, 4, and 5, respectively.

Figure 2.2: The tree with 3 leaves

In order to obtain distance 6 between leaves, the next shortest pendant path must be of length 5 (for any shorter path will result in multiple appearance of the same distance from the existing leaves). Then, the distance between these four leaves are 3, 4, 5, 6, 7, 8.

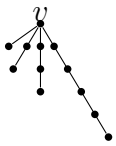


Figure 2.3: The tree with 4 leaves

For T to be a leaf-Leech tree, the fifth shortest pendant path must generate a distance 9 with some of the first four leaves. The only way to do so is for this pendant path to have length 8, generating new distances 9, 10, 11, 13 from the existing leaves.

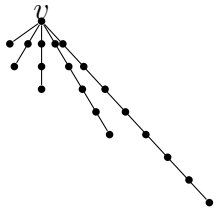


Figure 2.4: The tree with 5 leaves

Similarly, to generate 12 requires the next shortest pendant path to be of length

11, which also generate a second distance 13 between leaves. Thus, a starlike leaf-Leech tree can have at most four leaves. ∎

It is known that no path can be a Leech tree except for the ones in Figure 1.1. In fact, it is shown in [5] that a Leech tree cannot contain a long path. In regard to leaf-Leech trees, we show that there are only a few *caterpillars* (trees whose removal of leaves result in paths) that can be leaf-Leech trees.

Proposition 2.1.3. *There is no leaf-Leech caterpillars on more than 4 leaves.*

Figure 2.5: Caterpillar on n leaves

Proof. Let T be a leaf-Leech caterpillar in Figure 2.5 with leaves $v_1, v_2, \ldots, v_n$. It is easy to see that no v_i and v_j $(i \neq j)$ can share a common neighbor (which would create a distance 2 between leaves). Now let u_i be the unique neighbor of v_i and define an edge-weighted path P as follows:

- $V(P) = \{u_1, \ldots, u_n\}$;
- two vertices u_i and u_j are adjacent in P if no other u_k lies on the path connecting them in T;
- the weight of an edge u_iu_j in P is the distance between u_i and u_j in T.

Since the distances between pairs of vertices in $\{v_1, \ldots, v_n\}$ are $\left\{3, \ldots, \binom{n}{2} + 2\right\}$, the weighted distances between pairs of vertices in $\{u_1, \ldots, u_n\}$ are exactly $\left\{1, \ldots, \binom{n}{2}\right\}$, implying that P is a Leech path. The rest of the proof simply follows from that in [5] on the non-existence of long Leech path. ∎

2.2 Leach v.s. Leaf Leach

Given the very similar properties of leaf-Leech trees and what is known for Leech trees, it is natural to ask if there is any obvious connection between the two. Intuitively, a leaf-Leech tree seems to be easier to find than a Leech tree. Indeed, this is confirmed by our next observation.

Given a tree T with positive integer weights on edges, the *expansion* T^e of T is defined as follows (Figure 2.6):

- For any edge $u_iu_j \in E(T)$ with weight w, subdivide this edge into a path of length w from u_i to u_j;
- To every vertex $u_i \in V(T)$, append a pendant edge u_iv_i.

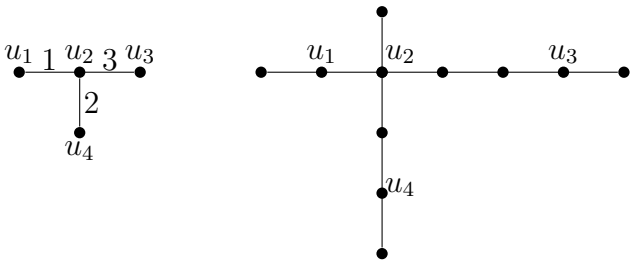


Figure 2.6: An edge-weighted tree T (on the left) and its expansion T^e (on the right)

Then, similar argument as that of Proposition 2.1.3 implies the following.

Theorem 2.2.1. *If there exists a Leech tree T on n vertices, there exists a leaf-Leech tree, namely T^e, on n leaves (See Appendix A for examples).*

On the other hand, it is not obvious whether the inverse is true. Figure 2.7 shows a tree on three leaves that is not the expansion of any tree. This is because of the vertex v with degree at least three and no leaf neighbors. We call such vertices *spanning* vertices.

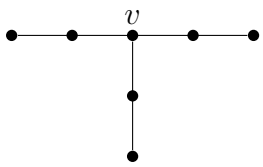


Figure 2.7: A tree with a spanning vertex v

To see why such a vertex prevent the tree to be the expansion of a tree, we argue as follows. At first, there must be a pendant edge and a pendant path of length 2 in order to have a leaf-Leech tree. If there exists a spanning vertex (existent vertices cannot be a spanning vertex), it should be pended with at least distance one to the root of the tree. Pendant paths of spanning vertex must be at least 2 by its definition. If we have a leaf-Leech tree consisting such a structure, it is easy to see that it is not the expansion of any tree.

It is also easy to see that the following is similar to that in the proof of Proposition 2.1.3. Note that Propositions 2.1.2 and 2.1.3 follow as immediate corollaries of Proposition 2.2.2 and known facts on the Leech trees.

Theorem 2.2.2. *Every tree with no spanning vertex is the expansion of some edge-weighted tree.*

A natural question follows:

Question 2.2.1. *Does there exist a leaf-Leech tree that contains at least one spanning vertex?*

A negative answer to this question would imply the equivalence of the Leech trees and leaf-Leech trees. In the other direction, we have not been able to find a leaf-Leech tree that is not the expansion of a Leech tree.

2.3 Concluding Remarks and Other Related Questions

We studied the characteristics of leaf-Leech trees and explored its connection with Leech trees.

Given Leech's original motivation for the concept of Leech trees, some other variations may also be interesting. Knowing that no Leech tree exists on n vertices, the minimal distinct distance tree [1] is one way to generalize the concept and to find the next best weighted tree in this aspect. It is also natural to pack, instead of distinct values, as many as possible values of $\{1, \ldots, \binom{n}{2}\}$ into the set of distances between vertices. Figure 2.8 is such an "almost Leech tree" on 5 vertices, where 6 is the only distance missing from the set $\{1, \ldots, 10\}$. Of course, it is easy to see that the expansion of an "almost Leech tree" yields an "almost leaf-Leech tree".

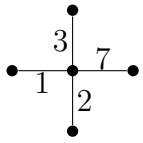


Figure 2.8: An "almost" Leech tree on 5 vertices

A tree with positive integer edge weights such that the weighted distances between vertices yields $1, 2, \ldots, \binom{n}{2}$ when taking modulo $\binom{n}{2} + 1$ is called as a *modular Leech tree* [2, 3]. It is known that there is no *modular Leech tree* of order 5. Figure 2.9 shows an "almost modulo Leech tree" of order 5.

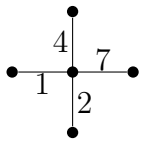


Figure 2.9: An "almost" modulo Leech tree on 5 vertices

Another question of obvious interest is to "pack" as many copies of each distance from a given set of values as possible, in a tree with as few vertices as possible. As an

example, Figure 2.10 is a weighted binary tree where every sub-star on four vertices is an exact copy of the Leech star on four vertices. When the structure is extended indefinitely, it is easy to see that each of the distances $1, 2, \ldots$ appear at least three times.

These are interesting topics but we will not work further on them in this thesis.

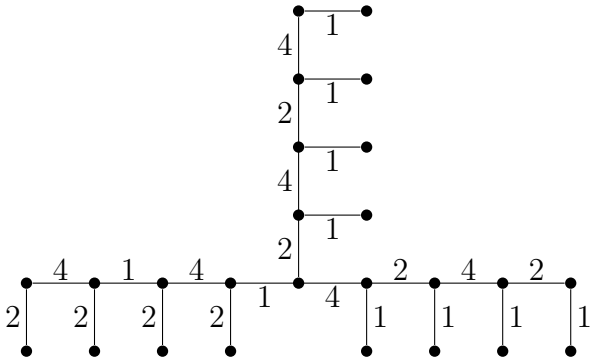


Figure 2.10: A binary tree that accomodate multiple copies of each distances

CHAPTER 3

HEURISTIC FOR GENERATING DENSE SPANNING TREES

3.1 Preliminaries

The number of subtrees and the total distance of a tree belong to a group of graph invariants, called topological indices, that are used in the literature as effective descriptors of graph structures. For instance:

- the sum of distances between all pairs of vertices is also known as the *Wiener index* as one of the most well known distance-based index in chemical graph theory; (see Appendix B.1 for an example)
- the number of subtrees is an example of counting-based indices first introduced from pure-mathematical point of view.

These two indices have been extensively studied in recent literature. In particular, it is well known that the star minimizes the Wiener index and maximizes the number of subtrees while the path maximizes the Wiener index and minimizes the number of subtrees. More interestingly, among tress of given degree sequence, the *greedy tree* (Definition 2 below) was shown to minimize the Wiener index [11, 14] and maximize the number of subtrees [15], where the degree sequence is simply the nonincreasing sequence of vertex degrees.

Definition 2 (Greedy trees)**.** *Given the sequence, the greedy tree is achieved through the following "greedy" algorithm:*

i) Start with a single vertex $v = v_1$ as the root and give v the appropriate number of children so that it has the largest degree;

ii) Label the neighbors of v as v_2, v_3, ..., assign to them the largest available degrees such that $deg(v_2) \geq deg(v_3) \geq \cdots$;

iii) Label the neighbors of v_2 (except v) as v_{21}, v_{22}, ... such that they take all the largest degrees available and that $deg(v_{21}) \geq deg(v_{22}) \geq \cdots$, then do the same for v_3, v_4, ...;

iv) Repeat (iii) for all the newly labeled vertices, always start with the neighbors of the labeled vertex with largest degree whose neighbors are not labeled yet.

For example, Fig. 3.1 shows a greedy tree with degree sequence

$$(4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 2, 2, 1, \ldots, 1).$$

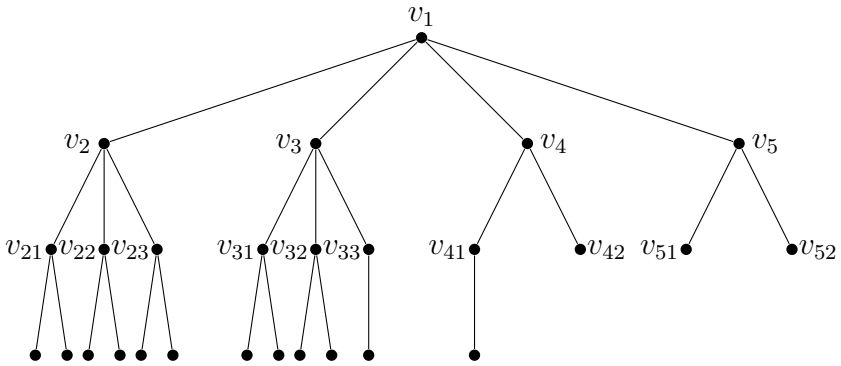


Figure 3.1: A greedy tree.

Interestingly, the greedy trees are also extremal with respect to many other graph indices, among which is the following special case of the *Randić index* [8], also called the *weight* of a tree [9]:

$$R(T) = \sum_{uv \in E(T)} deg(u)deg(v). \tag{3.1}$$

A comprehensive discussion of the extremal trees of given degree sequence with respect to functions defined on adjacent vertex degrees can be found in [12].

For trees of different given degree sequences, much work has been done in comparing the greedy trees (of the same order) of different degree sequence. In particular, for two nonincreasing sequences $\pi' = (d'_1, \cdots, d'_n)$ and $\pi'' = (d''_1, \cdots, d''_n)$, π'' is said

to *majorize* π' if for $k = 1, \cdots, n-1$

$$\sum_{i=0}^{k} d'_i \leq \sum_{i=0}^{k} d''_i \qquad \text{and} \qquad \sum_{i=0}^{n} d'_i = \sum_{i=0}^{n} d''_i. \tag{3.2}$$

The concept of majorization has been applied to the comparison of greedy trees of different degree sequences in order to find the dense structure (with minimal distance function or maximal number of subtrees) under various constraints. See [15] for an example of such discussions. For convenience we also say that π'' is higher in the majorization ladder than π' if π'' majorizes π'. (See appendix B.1 for an example)

In terms of finding dense spanning trees, the edge-swap heuristic starts with a random spanning tree and continuously remove a "bad" edge and add a "good" edge in order to improve the density of the spanning tree. From the aspect of distance-based and structure-based graph indices, evaluating the corresponding index of the resulted tree at each step would be extremely time consuming.

We propose an edge-swap heuristic that is based on the above results and use $R(T)$ defined in (3.1) instead of the distance or number of subtrees as an effective measure. In every step, we consider the degrees of the end vertices of the edge to be removed or added, as well as the resulted change in $R(T)$. Such a strategy simultaneously optimizes the value of the $R(T)$ and improves the degree sequence in the ladder of majorization. The consideration of $R(T)$ results in an efficient algorithm that quickly finds a dense spanning tree, which we present in the next section. Computational results will be provided for both randomly generated graphs and specific examples from applications. We also comment on potential improvements with the degree sequences taken into account.

3.2 The Edge-Swap Heuristic

In this section we present an edge-swap heuristic in details. The following algorithm takes a graph $G = (V, E)$ as input and return a dense spanning tree T as output.

ALGORITHM: Finding a dense spanning tree T for a given graph $G = (V, E)$.

Step 1.

Input $G(V, E)$ and generate a random spanning tree T for G. Let *SPARSE* be "true".

Step 2.

Step 2-1: Find the candidate edge e to be removed from T.

For each edge $e = uv \in E(T)$, let

$$f(e) = d_u d_v + \sum_{i=1}^{d_u - 1} d_{u_i} + \sum_{i=1}^{d_v - 1} d_{v_i}$$

be contribution of e, where d_u and d_v are the degrees of the vertices u and v respectively (in T), d_{u_i} for $1 \leq i \leq d_u - 1$ (d_{v_i} for $1 \leq j \leq d_v - 1$) are the degrees of the other neighbors of u (v) in T.

Let e be an edge with the minimum contribution.

Step 2-2: Generate the spanning forest $T' = T - e$ with two components T_u and T_v.

Step 2-3: Find the candidate edge e'' to be added to T.

For each edge $e' = u'v' \in E(G)$ with $u' \in T_u$ and $v' \in T_v$, let

$$g(e') = (d_{u'} + 1)(d_{v'} + 1) + \sum_{i=1}^{d_{u'}} d_{u'_i} + \sum_{i=1}^{d_{v'}} d_{v'_i}$$

be contribution of e', where $d_{u'}$ and $d_{v'}$ are the degrees of the vertices u' and v' respectively (in T'), $d_{u'_i}$ for $1 \leq i \leq d_{u'}$ ($d_{v'_i}$ for $1 \leq j \leq d_{v'}$) are the degrees of the neighbors of u' (v') in T'.

Let e'' be such an edge with the maximum contribution calculated by $g(.)$.

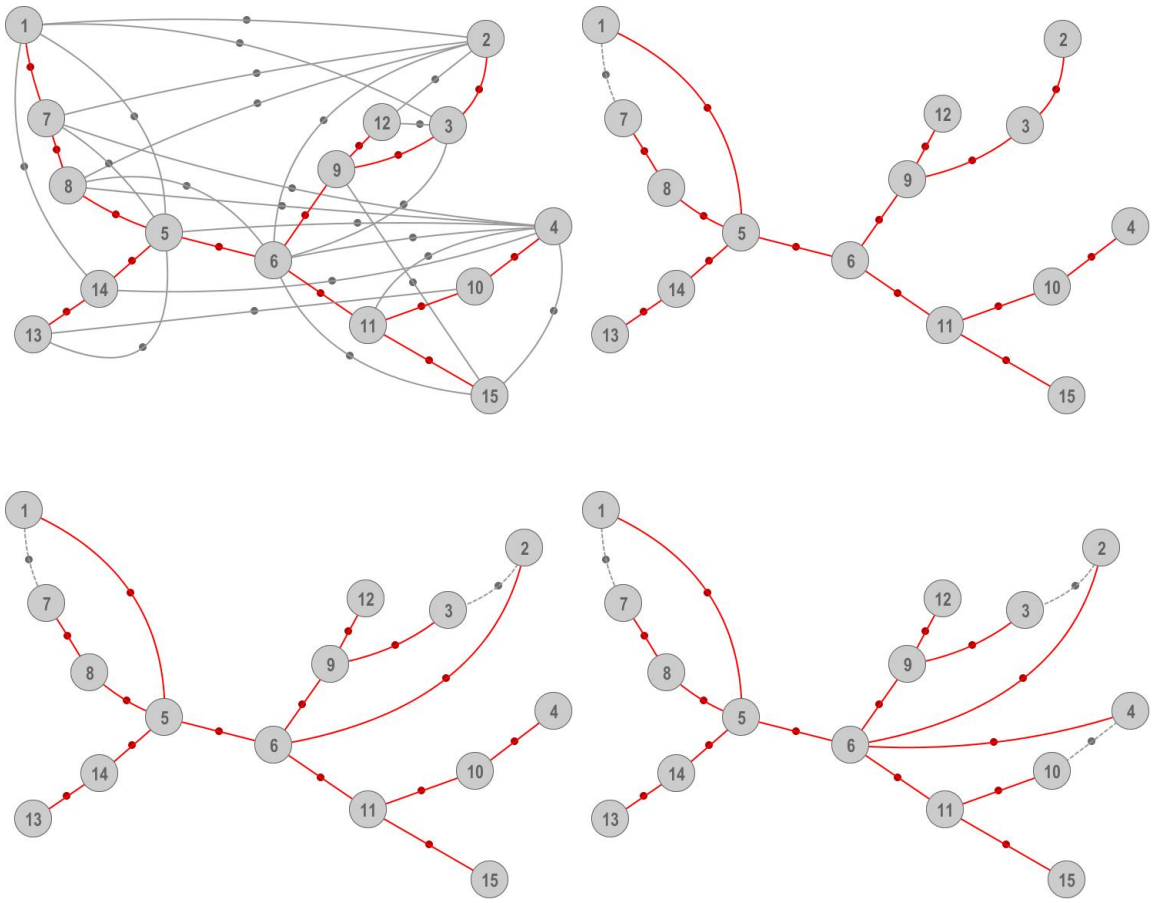
Step 2-4: Generate the spanning tree $T'' = T' + e''$.

Step 2-5: If $f(e) < g(e'')$, let *SPARSE* be "true". Otherwise let *SPARSE* be "false".

Step 3.

While *SPARSE* is "true", let $T = T''$ and repeat Step 2. Return T when *SPARSE* is "false".

Figure 3.2 and Figure 3.3 below presents a step by step illustration of the algorithm, where the spanning trees in each step is shown in red and the removed edge in each step is shown as dotted.



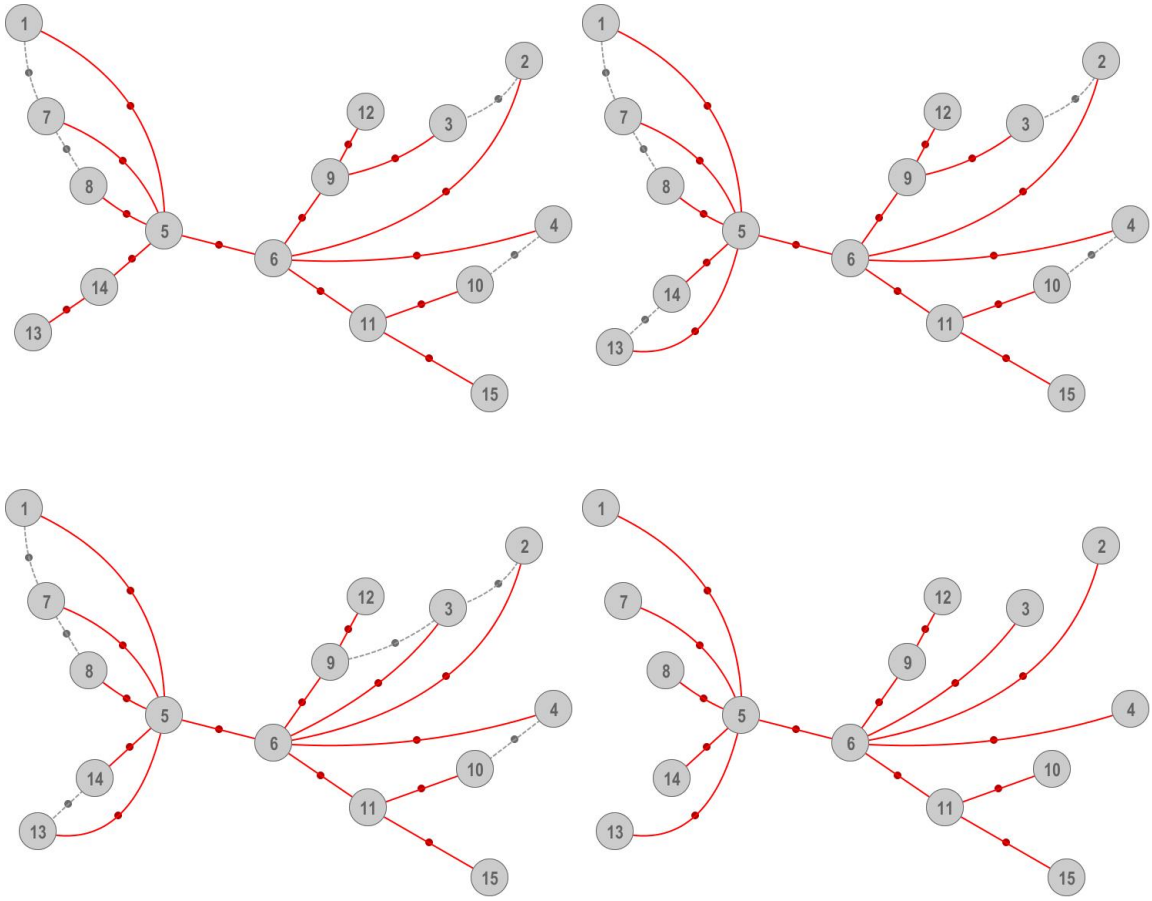


Figure 3.2: Step by step illustration of the algorithm

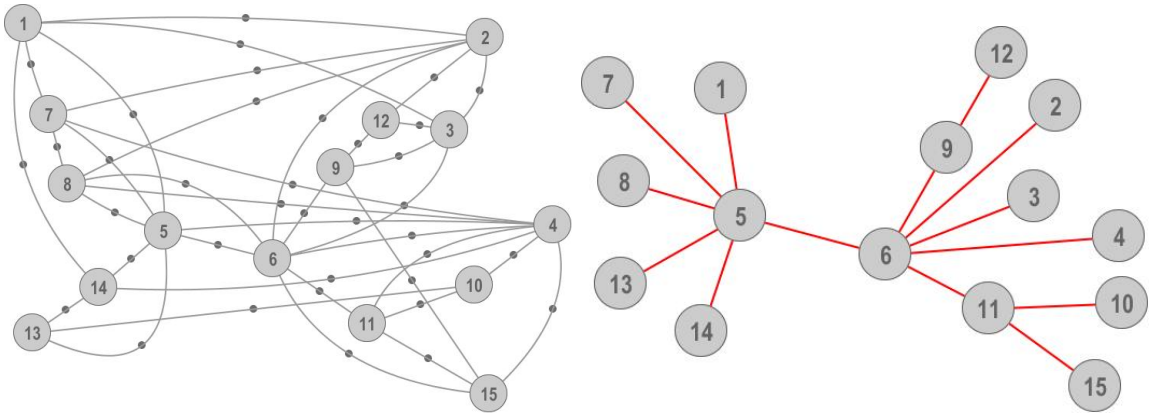


Figure 3.3: The original graph (on the left) and the resulted spanning tree (on the right)

In the above algorithm, the value

$$g(e'') - f(e) = R(T'') - R(T)$$

is the maximum possible improvement in $R(.)$ over one swap. In the case of a tie (i.e., multiple edges can serve as e or e''), we simply pick one of them. Since after each swap, the value of $R(T)$ is strictly increasing, this process terminates after finitely many steps.

3.3 Computational Results

Of course, the heuristic proposed in the previous section does not guarantee the densest spanning tree as an output. But as experimental results show, this heuristic effectively finds a dense spanning tree within very few swaps and hence is of great practical interests. When tested the algorithm on 100 randomly generated graphs, each of order 15 and containing a spanning star, the algorithm returns a star in over 60 runs. Part of this data is shown in Table 3.1.

Note that a star on 15 vertices has total distance 196. As shown in Table 3.1, all resulted spanning trees are dense (even if it is not a star) with only one exception.

In the following example (Figure 3.4), 7 edge-swaps resulted in the final spanning tree from the original graph on 15 vertices and 37 edges.

Graphs	Number of swaps	Initial distance	Final distance	Returns a star
A	9	386	238	N
B	15	384	196	Y
C	10	404	196	Y
D	0	348	348	N
E	10	432	196	Y
F	10	382	196	Y
G	8	374	232	N
H	13	348	196	Y
I	16	382	196	Y
J	10	382	196	Y

Table 3.1: Results of ten randomly generated graphs on 15 vertices

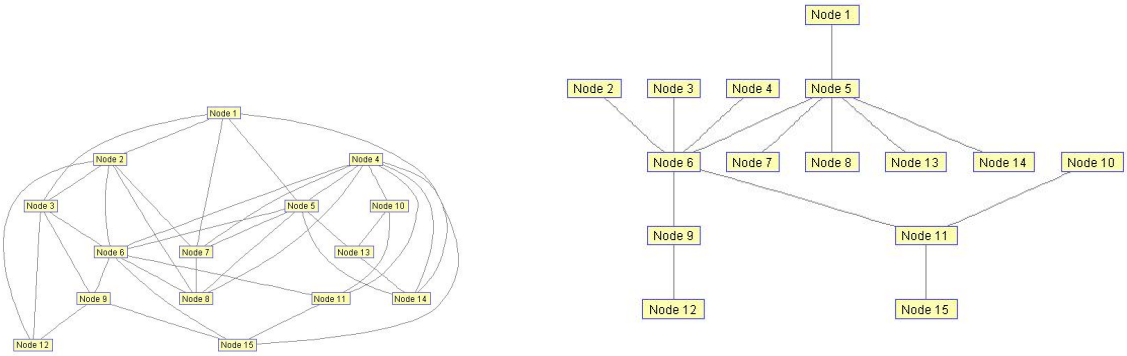


Figure 3.4: The original graph (left) and the resulted spanning tree (right)

When applied to the US Airports data set of 332 vertices and 2126 edges [13], only 15 edge-swaps were needed to obtain the final spanning tree in Figure 3.6). During edge-swaps, the total distance decreases from 1444880 to 1421327.

Figure 3.5: Initial spanning tree of the US Airport data set

Figure 3.6: Final spanning tree for the US Airport data set

A simple way of improving the likelihood of achieving the denser spanning tree can be obtained by replacing Step 2-5 of the algorithm with the following:

(Step 2-5)′: If $f(e) < g(e'')$ or $f(e) = g(e'')$ and the degree sequence of T'' majorizes that of T, let $SPARSE$ be "true". Otherwise let $SPARSE$ be "false".

In this case, after each swap, the value of $R(T)$ is strictly increasing or non-decreasing with the degree sequence moving up in the majorization ladder. Take, for instance, two of the randomly generated graphs on 15 vertices as discussed in the previous section, Figure 3.7 and 3.8 show improvements in the resulted spanning tree.

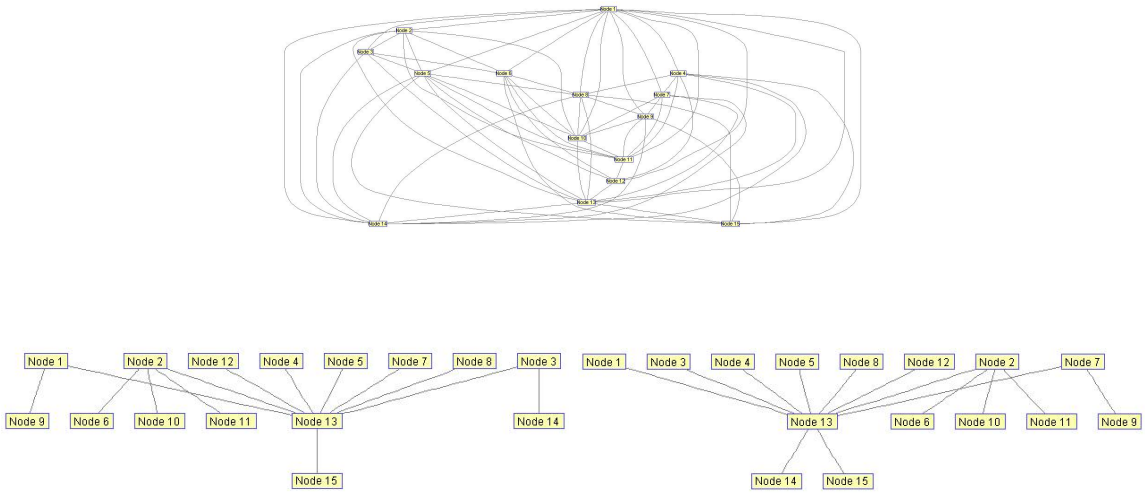


Figure 3.7: A random graph (up), the first resulted spanning tree (left) and the improved spanning tree (right)

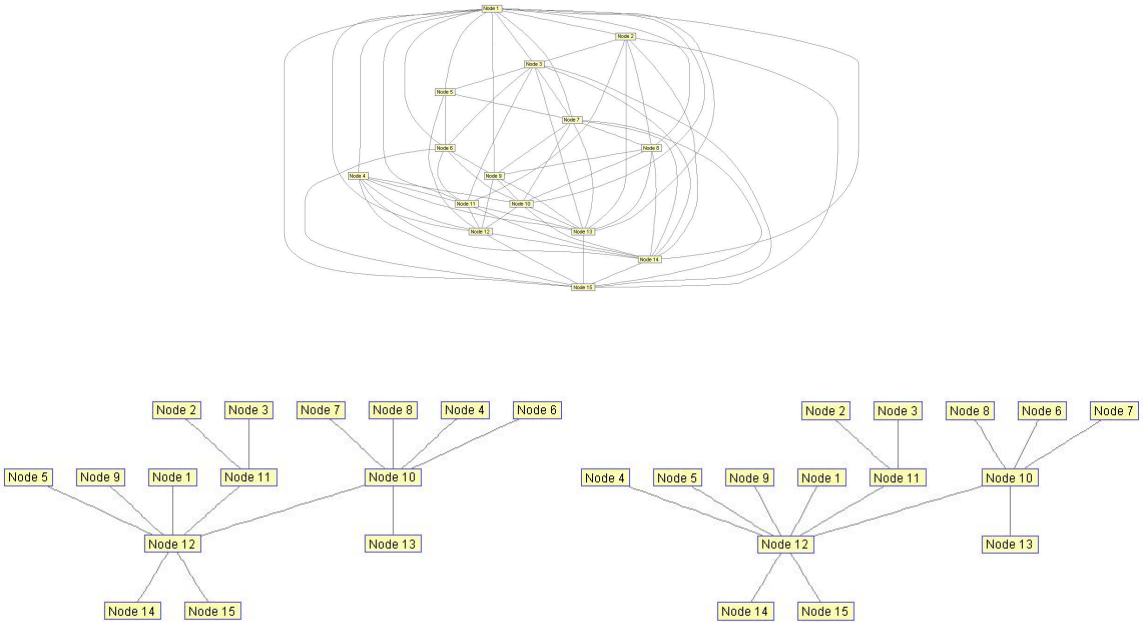


Figure 3.8: A random graph (up), the first resulted spanning tree (left) and the improved spanning tree (right)

When applying the modified algorithm to the US Airports data set, the new resulted spanning tree is shown in Figure 3.9, with significant improvements over the previous result in Figure 3.6. The total distance of the improved result is 1412038.

Figure 3.9: Final spanning tree for the US Airport data set through modified algorithm

CHAPTER 4

RESULTS AND ANALYSIS

In this chapter, we present computational results and examples for Leech type tree labeling and pseudo-code for dense minimum spanning trees.

4.1 Examples for Leaf-Leech Trees

As shown in the Proposition 2.1.2, trees in figure 4.1 are starlike leaf-Leech trees of at most 4 leaves giving distances from 3 to $\binom{n}{2} + 2$.

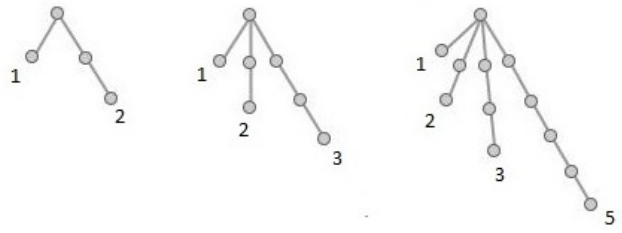


Figure 4.1: Leaf-Leech trees on 1, 2, 3 and 4 leaves

Since there are no more leaf-Leech trees on n leaves where $n > 4$, we present some examples of almost leaf-Leech trees below.

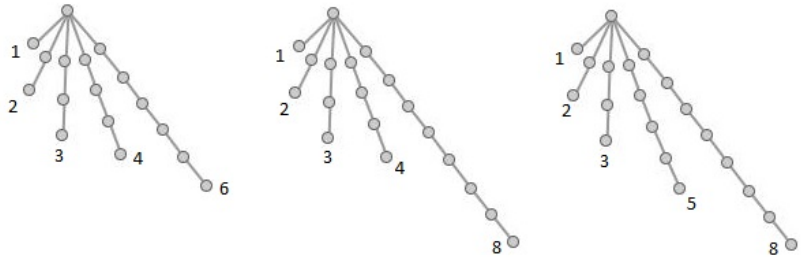
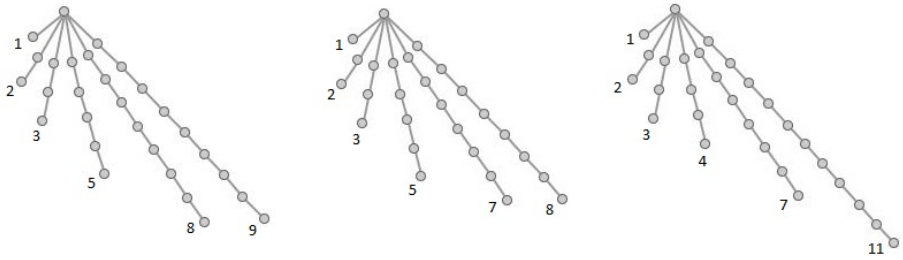


Figure 4.2: Almost leaf-Leech tree examples on 5 leaves

Last two distances are missing in the first tree in figure 4.2. Second tree has just one missing distance, 8. For the last example on 5 leaves, there is one missing distance, $\binom{5}{2} + 2$ and one extra distance which is $\binom{5}{2} + 3$.

For trees on 6 leaves in figure 4.3, there are at least two missing distances. For the last tree, there is also an unsolicited distance. In order to get rid of that distance, path of length 11 can be placed as in figure 4.4. In that case, unwanted edge is removed and number of missing distances decreases to one.



Missing distances: {15,16}, {14,16,17}, {16,17}

Figure 4.3: Almost leaf-Leech tree examples on 6 leaves

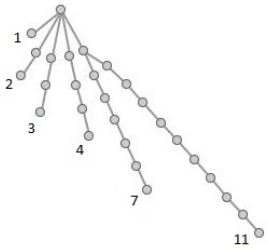


Figure 4.4: Modifed almost leaf-Leech tree example on 6 leaves

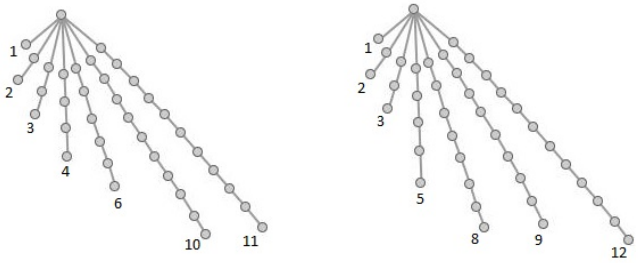


Figure 4.5: Almost leaf-Leech tree examples on 7 leaves

4.2 Pseudo-code for Heuristic

In this section, we describe pseudo-code for the edge-swap heuristic for generating dense spanning trees in detail in Algorithm 1. For MATLAB code, see *Appendix B*.

The algorithm starts with loading data set in the format of a $n \times 3$ matrix. First two columns represent an edge with two vertices and last column shows weights between these vertices. In addition to some specific data sets, we also have a function "*rand_starlike_graph*" for generating a data set of random, undirected graphs, each of which contains a spanning star. After loading data, a minimum weighted spanning tree T is computed through Kruskal algorithm. As described earlier, we set *true* as initial value of "*sparse*" since we assume that the tree is not dense enough at the beginning. Edge-swap heuristic continues to be made until stopping criterion is satisfied (i.e."*sparse*" is $false$).

Each iteration includes removing a "*bad*" edge and adding a "*good*" edge which is not in the current tree. The first task in the loop is to find the edge to be removed. Function "$findRemovalEdges$" gets adjacency matrix of the tree T as an input and returns list of the candidate edges with the minimum value of $f(.)$. From candidate list, one of the edges, $e = (u, v)$ is chosen randomly to be removed. After removing the edge from the adjacency matrix of the tree T, we obtain "T_r" as updated tree. Function "*split*" is used to split the adjacency matrices of new two subtrees up and return the lists of neighbors of vertices u and v.

Function "$findInsertionEdges$" takes neighbor lists, adjacency matrices of updated tree and the original graph as inputs. After calculating $g(.)$ for each candidate edge, it returns minimum $g(.)$ and list of corresponding edges. One of the candidate edges $e'' = (u'', v'')$ is chosen to be inserted and the updated tree "T_a" is obtained. After this process, $f(e)$ is compared with $g(e'')$. If it is less than $g(e'')$, it means that new tree is denser than previous. Thus, iterations continue. If $f(e) = g(e'')$,

the degree sequences of the initial tree and current tree are calculated. If the degree sequence of the current tree "T_a" majorizes that of T (and the two degree sequences are not the same), then an edge-swap is made; otherwise process is terminated.

Data: $G = (V, E)$
Result: Updated tree T

Load data set: $G \leftarrow$ data set;
$T \leftarrow \text{MST}(G)$;
sparse $\leftarrow$ true;
while *sparse is true* **do**
    $(L\_remove, \min f(e)) \leftarrow$ findRemovalEdges(T);
    Select removal edge: $e \leftarrow (u, v)$;
    $T\_r \leftarrow T \setminus (u, v)$;
    $(neighbors\_u, neighbors\_v) \leftarrow \text{split}(T\_r, (u, v))$;
    $(L\_add, \max g(e)) \leftarrow$ findInsertionEdges(G,T_r,$neighbors\_u$,$neighbors\_v$)
    Select insertion edge: $e'' \leftarrow (u'', v'')$;
    $T\_a \leftarrow T\_r \cup (u'', v'')$;
    if *min* $f(e) <$ *max* $g(e'')$ **then**
        $T \leftarrow T\_a$
    else if *min* $f(e) ==$ *max* $g(e'')$ **then**
        if *degree sequence of* T_a *majorizes degree sequences of* T **then**
            T $\leftarrow T\_a$;
        else
            sparse $\leftarrow$ false;
        end
    else
        sparse $\leftarrow$ false;
    end
end

Algorithm 1: Pseudo-code for modified edge-swap heuristic

CHAPTER 5

CONCLUSION

In this thesis, we considered questions of labeled trees and generating dense spanning trees.

In the first part, we studied questions motivated from a concept called the Leech trees. In addition to presenting known Leech trees, some non-existence results and recently studied variations such as modular Leech trees; Leech-type questions on distances between pairs of leaves which we define as leaf-Leech trees are proposed. Besides, relation between Leech and leaf-Leech trees and some related questions are examined such as"almost" Leech and "almost" modulo Leech trees.

In the second part, an edge-swap heuristic for generating dense spanning trees from a given graph structures is studied. In this case, "dense" trees are determined by measuring total distance of trees calculated by using *Randić index*. A MATLAB code is implemented according to developed efficient algorithm. Computational results are provided for randomly generated graphs and specific examples from applications. For further improvement of the results, the concept of majorization between degree sequences is included into the algorithm. The outcomes of experiments show significant improvements over the previous results.

REFERENCES

[1] B. Calhoun, K. Ferland, L. Lister, and J. Polhill, Minimal distinct distance trees, Journal of Combinatorial Mathematics and Combinatorial Computing, 61 (2007), 33-57.

[2] D. Leach, Modular Leech trees of order at most 8, International Journal of Combinatorics, (2014), Article ID 218086.

[3] D. Leach and M. Walsh, Generalized Leech trees, Journal of Combinatorial Mathematics and Combinatorial Computing, 78 (2011), 15-22.

[4] J. Leech, Research problems: another tree labelling problem, The American Mathematical Monthly, 82(9) (1975), 923-925.

[5] L.A. Szkely, H. Wang, and Y. Zhang, Some non-existence results on Leech trees, Bulletin of the Institute of Combinatorics and its Applications, 44 (2005), 37-45.

[6] H. Taylor, Odd path sums in an edge-labeled tree, Mathematics Magazine, 50(5) (1977), 258-259.

[7] H. Taylor, A distinct distance set of 9 nodes in a tree of diameter 36, Discrete Mathematics, 93 (1991), 167-168.

[8] M. Randić, On characterization of molecular branching, *J. Amer. Chem. Soc.* 97 (1975) 6609–6615.

[9] D. Rautenbach, A note on trees of maximum weight and restricted degrees, *Discrete Math.* 271 (2003) 335–342.

[10] R. Silva, D. Silva, M. Resende, G. Mateus, J. Goncalves, P. Festa, An edge-swap heuristic for generating spanning trees with minimum number of branch vertices, *Optim. Lett.* 8 (2014) 1225–1243.

[11] H. Wang, The extremal values of the Wiener index of a tree with given degree sequence, *Discrete Applied Mathematics,* 156 (2008), 2647–2654.

[12] H. Wang, Functions on adjacent vertex degrees of trees with given degree sequence, *Central European J. Math.* 12 (2014) 1656–1663.

[13] Pajek datasets, *US Air lines*: http://vlado.fmf.uni-lj.si/pub/networks/data/

[14] X.-D. Zhang, Q.-Y. Xiang, L.-Q. Xu, R.-Y. Pan, The Wiener index of trees with given degree sequences, *MATCH Commun.Math.Comput.Chem.*, 60 (2008), 623–644.

[15] X.-M. Zhang, X.-D. Zhang, D. Gray, H. Wang, The number of subtrees of trees with given degree sequence, *J. Graph Theory,* 73(3) (2013), 280–295.

[16] C.-F. Bazlamacci, K.-S. Hindi, Minimum-weight spanning tree algorithms: a survey and emprical study, *Computers and Operations Research,* 28 (2001), 767–785.

[17] D.R. Karger, P.-N. Klein, R.-E. Tarjan, A randomized linear-time algorithm to find minimum spanning trees, *J. Association for Computing Machinery,* 42(2) (1995), 321–328.

[18] W.B. Strothmann, Bounded degree spanning trees. PhD thesis, Department of Computer Science, University of Paderborn, 1997.

[19] R.-J. Douglas, NP-completeness and degree restricted spanning trees, *Discrete Mathematics,* 105 (1992), 41–47.

[20] D.W. Barnette, 2-connected spanning subgraphs of planar 3-connected graphs, *J. Combinatorial Theory, Series B,* 61(1994), 210–216.

[21] Z. Gao, 2-connected coverings of bounded degree in 3-connected graphs, *J. Graph Theory,* 20(3)(1995), 327–338.

[22] D.P. Sanders, Y. Zhao,On 2-connected spanning subgraphs with low maximum degree. Revised manuscript, htp://www.math.ohio-state.edu/ dsanders/papers/lmd.ps, 1996.

Appendix A

FIRST APPENDIX

A.1 Leech vs leaf-Leech

In this section, expansion of known Leech trees are shown. In leaf-Leech trees, nodes which are shown as star are roots of the tree and dotted edges show pendant edges.

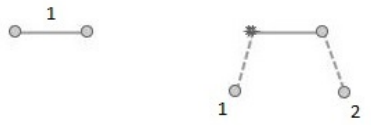


Figure A.1: Leech tree on 2 vertices (left) and leaf-Leech tree on 2 leaves (right)

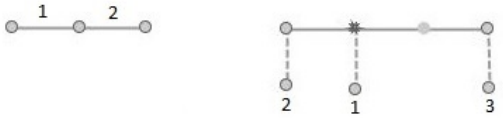


Figure A.2: Leech tree on 3 vertices (left) and leaf-Leech tree on 3 leaves (right)

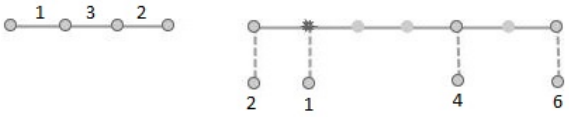


Figure A.3: Leech tree on 4 vertices (left) and leaf-Leech tree on 4 leaves (right)

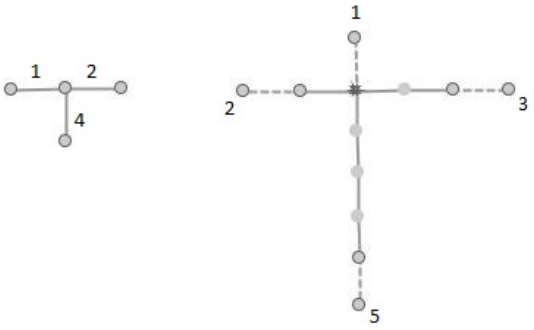


Figure A.4: Leech tree on 4 vertices (left) and leaf-Leech tree on 4 leaves (right)

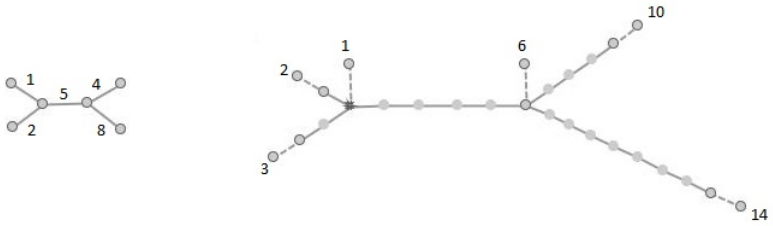


Figure A.5: Leech tree on 6 vertices (left) and leaf-Leech tree on 6 leaves (right)

A.2 Almost Leech vs Almost leaf-Leech

In this section, one example which demonstrates relation between almost Leech and leaf-Leech tree is presented.

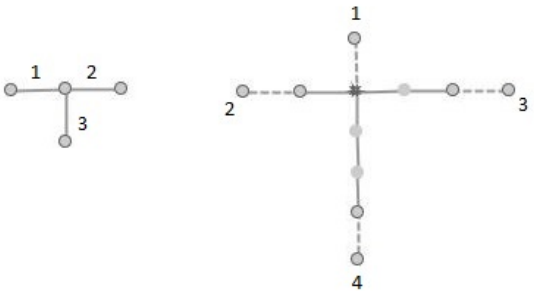


Figure A.6: Almost Leech on 4 vertices (left) and almost leaf-Leech on 4 leaves (right)

Appendix B

SECOND APPENDIX

B.1 Examples

In this section, examples for *Wiener index*, *Randić index* and *concept of majorization* are presented.

Example B.1.1. *Wiener index of a tree is calculated by the sum of distances between all pairs of vertices.*

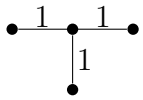


Figure B.1: Tree on 4 vertices

In the tree shown in figure B.1, there are three distance 1 and three distance 2 which is the largest distance in the tree. Thus,

$$W(T) = 3 \times 1 + 3 \times 2 = 9.$$

Example B.1.2. *Suppose two nonincreasing degree sequences* $\pi' = (4, 2, 2, 1, 1, 1, 1)$ *and* $\pi'' = (4, 3, 1, 1, 1, 1, 1)$ *are given. Comparison for the first and summation of the first two components in the degree sequences, inequality in* (3.2) *is satisfied. After summation of first two components, equality in* (3.2) *is satisfied to the end. As it is seen on the figure B.2,* π'' *majorizes* π'*.*

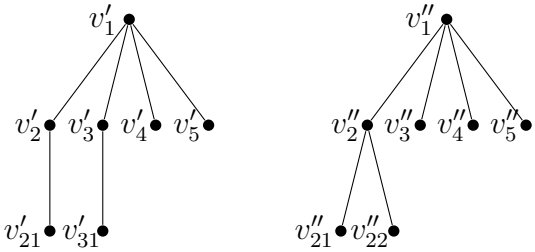


Figure B.2: π' on the left and π'' on the right

Example B.1.3. *Randić index of a tree is calculated by the summation of products of adjacent vertex degrees which is shown in* (3.1).

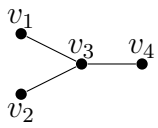


Figure B.3: Tree on 4 vertices

Randić index of the tree shown in figure B.3 is:

$$\begin{aligned} R(x) &= (deg(v_1) \times deg(v_3)) + (deg(v_2) \times deg(v_3)) + (deg(v_3) \times deg(v_4)) \\ &= (1 \times 3) + (1 \times 3) + (3 \times 1) \\ &= 9. \end{aligned}$$

Next, MATLAB code for modified algorithm shown in Section 4.2 i presented.

B.2 MainProgram.m

```
% ------------------------ THE EDGE-SWAP HEURISTIC ------------------------
% Georgia Southern University, 2014-2015
% Department of Mathematical Sciences
% This program finds Dense Spanning Tree from a given graph.
% -------------------------------------------------------------------------
%% Loading special data sets:
% load g_small_15.txt
% [T adj_G] = MST(g_small_15);
%-----------------------------------
% load USAirports_332.txt
% [T adj_G] = MST(USAirports_332);

% view(biograph(triu(adj_G), [], 'ShowArrows', 'off', 'ShowWeights', 'off'))
% view(biograph(triu(T), [], 'ShowArrows', 'off', 'ShowWeights', 'off'))
% init_wiener = wiener_index(T)
% -------------------------------------------------------------------------
%% Generating and loading data set of random, undirected, connected graph:
% H = rand_starlike_graph(15);
% load MyFile.txt
% [T adj_G] = MST(MyFile);
% view(biograph(triu(adj_G), [], 'ShowArrows', 'off', 'ShowWeights', 'off'))
% view(biograph(triu(T), [], 'ShowArrows', 'off', 'ShowWeights', 'off'))
% [init_wiener D1] = wiener_index(T);
```

```
%% ------------------------------------------------------------------------
iter_num = 0;
sparse = 1;
visited_edgeu = 0;
visited_edgev = 0;
swap_num = 0;
counter = 0;
init_deg_seq = degree_seq(T);
prev_last_wiener = init_wiener;

while sparse > 0
    iter_num = iter_num + 1;

    % calculates min(f(e)) and returns list of the selected edges to be removed
    [L_remove f_e_min] = findRemovalEdges(T, visited_edgeu, visited_edgev);

    T_r = T;
    % choose e* = (u*,v*) to be removed
    u = L_remove(1,1);
    v = L_remove(1,2);

    % delete selectec edge : T_r = T\(u*,v*)
    T_r(u,v) = 0;
    T_r(v,u) = 0;

    % split tree into two parts
    % split function returns two separate lists (two subtrees)
    [T1 T2 list_u list_v] = split(T_r, u, v);

    % calculates max(g(e)) and returns candidate edges to be added
    [L_add max_g] = findInsertionEdges(adj_G, T_r, list_u, list_v);

    % if selected edge to be added is the same with removed edge and
    % if the list includes another candidate edge to be added:
    if (L_add(1,1) == u) & (L_add(1,2) == v) & (L_add(2,1) ~= 0)
        u_new = L_add(2,1); % choose second edge
        v_new = L_add(2,2);
    else   % otherwise, choose same edge
        u_new = L_add(1,1);
        v_new = L_add(1,2);
    end

    if (u_new == u)&(v_new == v)
        counter = counter + 1;
        visited_edgeu = u;
        visited_edgev = v;
    else
        swap_num = swap_num + 1;
        visited_edgeu = 0;
        visited_edgev = 0;
    end

    T_a = T_r;
    T_a(u_new,v_new) = 1;
```

```
78      T_a(v_new,u_new) = 1;
79
80      last_deg_seq = degree_seq(T_a);
81
82      deg_sum1 = 0;
83      deg_sum2 = 0;
84      mark = 0;
85
86      % if new result is better, update the tree and continue.
87      if (f_e_min < max_g)
88          T = T_a;
89          % show tree in each step
90          view(biograph(triu(T), [], 'ShowArrows', 'off', 'ShowWeights', 'off'));
91
92      % if nothing changed in terms of function values, check degree sequences
93      elseif f_e_min == max_g
94          for e = 1:size(T_a,1)
95              deg_sum1 = deg_sum1 + init_deg_seq(e);
96              deg_sum2 = deg_sum2 + last_deg_seq(e);
97
98              % compare degree sequences of previous and current trees in each step
99              % if new swap makes the degree sequence better:
100             % (if degree sequence of new tree majorizes degree sequences of
101             % previous tree, then make swap)
102             if deg_sum2 > deg_sum1
103                 mark = 1;
104             end
105
106             % if there is an equivalence between degree sums, then
107             % keep going to at the end of the degree sums
108             if deg_sum2 >= deg_sum1
109                 continue;
110             % if degree sum isn't affected in a good way, stop swapping.
111             else
112                 sparse = 0;
113                 mark = 0;
114                 break;
115             end
116         end
117
118         % if new swap makes the degree sequence better (from previous if block):
119         if mark == 1
120             % make that swap and update the tree
121             T = T_a;
122
123             %view(biograph(triu(T), [], 'ShowArrows', 'off', 'ShowWeights', 'off'));
124             [last_wiener D] = wiener_index(T);
125
126             % if wiener index of updated tree is better than previous,
127             % then update wiener index
128             if last_wiener < prev_last_wiener
129                 prev_last_wiener = last_wiener;
130             %if wiener does not change, stop.
131             elseif last_wiener == prev_last_wiener
```

```
132                 sparse = 0;
133             end
134         end
135
136     % if new result is not better, stop.
137     else
138         sparse = 0;
139     end
140 end
141
142 [last_wiener D] = wiener_index(T);
143
144 disp('iter_num = ')
145 disp(iter_num)
146 disp('swap_num =')
147 disp(swap_num)
148 view(biograph(triu(T), [], 'ShowArrows', 'off', 'ShowWeights', 'off'));
149 disp('init_wiener')
150 disp(init_wiener)
151 disp('last_wiener =')
152 disp(last_wiener)
```

B.3 rand_starlike_graph.m

```
1  function [adj_G] = rand_starlike_graph(n)
2
3  % function [adj_G] = rand_starlike_graph(n)
4  % This function generates data set of random undirected graph
5  % since we want to have a connected graph,
6  % 1st column and 1st row of the adjacency matrix of graph include just 1's
7  % input:
8        % n = size of the adjacency matrix of the data set to be generated
9  % output:
10       % adj_G = adjacency matrix of generated data set
11 % variables:
12       % random = upper triangular part of random (n-1)x(n-1) matrix
13       % M = (n-1)x(n-1) matrix whose elements either 1 or 0
14       % A = nx3 matrix. [vertex vertex weight]
15           % edge is defined by first two columns and third column shows
16           % weights between vertices
17 %-------------------------------------------------------------------------
18
19 adj_G = zeros(n,n);
20 first_row = ones(1,n);
21 first_column = ones(n,1);
22
23 first_row(1) = 0;
24 first_column(1) = 0;
25
26 % 1st row and 1st column includes 1s in order to guarantee a connected graph
27 adj_G(1,:) = first_row;
```

```
28  adj_G(:,1) = first_column;
29
30  random = triu(rand(n-1));
31  M = random + tril(random',-1)>.5;
32  M(logical(eye(size(M)))) = 0;
33
34  % rest of the adjacency matrix includes randomly generated matrix M
35  adj_G(2:n, 2:n) = M;
36
37  % counts number of edges (according to 1s in the adjacency martix)
38  counter = 0;
39  for i=1:n
40      for j=i:n
41          if adj_G(i,j) == 1
42              counter = counter + 1;
43          end
44      end
45  end
46
47  weights = zeros(counter,1);
48  i_index = zeros(counter,1);
49  j_index = zeros(counter,1);
50
51  ind = 0;
52  for k=1:n
53      for l=k:n
54          if adj_G(k,l) == 1
55              ind = ind+1;
56              weights(ind) = rand(1);
57              i_index(ind) = k;
58              j_index(ind) = l;
59          end
60      end
61  end
62
63  A = [i_index j_index weights];
64
65  % writing into a text file
66  fid=fopen('MyFile.txt','w');
67  formatSpec = '%i %i %0.2f \n';
68  fprintf(fid, formatSpec, A');
69  fclose(fid);
70
71  end
```

B.4 MST.m

```
function [T, adj_G] = MST(G)

% function [T, adj_G] = MST(G)
% Kruskal algorithm is used to find minimum spanning tree from a given graph
% Input:
        % G = nx3 matrix. 1st and 2nd columns define the edge (2 vertices) and
        % 3rd column shows the weight of the edge
% Output:
        % T = adjacency matrix of the minimum spanning tree
        % adj_G = adjacency matrix of the graph
% variables:
        % row = number of rows of the given graph matrix G
        % n = size of graph matrix

%-------------------------------------------------------------------------

row_number = size(G,1);

% create adjacency matrix of the graph which is a symmetric matrix
% read line by line.
for i = 1:row_number
    adj_G(G(i,1),G(i,2)) = 1;
    adj_G(G(i,2),G(i,1)) = 1;
end

n = size(adj_G,1);

% sort G by ascending order according to 3rd column(including weights)
G = sort(G, 3, 'ascend')
vertices = zeros(1,n);
T = zeros(n);

for i = 1 : row_number
% check when we insert edge[i,j] in the graph whether it has cycle
    inserted_edge = G(i,[1 2]);
    [vertices, cycle] = cycle_control(vertices, inserted_edge);
    if cycle == 1
        G(i,:) = [0 0 0];
    end
end

% Create minimum spanning tree's adjacency matrix
for i = 1 : row_number
    if G(i,[1 2]) ~= [0 0]
        T(G(i,1),G(i,2)) = 1;
        T(G(i,2),G(i,1)) = 1;
    end
end
end
```

B.5 cycle_control.m

```
function [vertices, cycle] = cycle_control(vertices, inserted_edge)

% Reference : http://www.mathworks.com/matlabcentral/fileexchange/13457-kruskal-algorithm/content
    //MST_Kruskal/iscycle.m
% function [vertices, cycle] = cycle_control(vertices, inserted_edge)
% input:
        % vertices = set of vertices in the graph
        % inserted_edge = edge we insert in graph
% output:
        % vertices = The "new: set of vertices
        % cycle = 1 if there is a cycle, else cycle = 0
%-------------------------------------------------------------------------

g = max(vertices)+1;
cycle = 0;
n = length(vertices);

if vertices(inserted_edge(1)) == 0 & vertices(inserted_edge(2)) == 0
    vertices(inserted_edge(1))=g;
    vertices(inserted_edge(2))=g;
elseif vertices(inserted_edge(1))==0
    vertices(inserted_edge(1))=vertices(inserted_edge(2));
elseif vertices(inserted_edge(2))==0
    vertices(inserted_edge(2)) = vertices(inserted_edge(1));
elseif vertices(inserted_edge(1)) == vertices(inserted_edge(2))   % check self-cycle
    cycle = 1;
    return
else
    m = max(vertices(inserted_edge(1)),vertices(inserted_edge(2)));
    for i=1:n
        if vertices(i)== m
            vertices(i) = min(vertices(inserted_edge(1)),vertices(inserted_edge(2)));
        end
    end
end
```

B.6 findRemovalEdges.m

```
function [L f_e_min] = findRemovalEdges(T, visited_edgeu, visited_edgev)

% function [L f_e_min] = findRemovalEdges(T, visited_edgeu, visited_edgev)
% This function finds candiate edge(s) to be removed.
% inputs:
        % T = adjacency matrix of the tree
        % visited_edgeu = checks if the edge between u and v is visited before
        % visited_edgev = checks if the edge between u and v is visited before
% outputs:
        % L = list of candidate edges to be removed
        % f_e_min = minimum f(e)
```

```
% variables:
        % list = nx2 matrix storing candidates
        % count = counts degrees of vertices
        % deg = vector which stores degree of each vertex
        % f_e_min = f(e) = deg_u * deg_v + deg_of_neighbors_of_u + deg_of_neighbors_of_v
        % ind = number of edges which gives minimum f(e)
% ------------------------------------------------------------------------

n = size(T,1);
deg = zeros(n,1);
count = 0;
list = zeros(n,2);
% finds degree of each vertex.
for i = 1:n
    for j = 1:n
        if T(i,j) == 1
            count = count +1;
        end
    end
    deg(i) = count;
    count = 0;
end
%-----------------------------------------

% initialize f(e)=0 and min(f(e))= total degree
f_e = 0;
f_e_min = sum(deg);
ind = 0;

% if u is not visited before:
if visited_edgeu == 0
    % Looks for each connected e=(u,v) and calculates f(e).
    for i = 1:n
        for j = i:n
            if T(i,j) == 1   % If u and v are connected
                f_e = deg(i)*deg(j);
                for k = 1:n  % looks for degree of other neighbors of u
                    if k ~= j
                        if T(i,k) == 1
                            f_e = f_e + deg(k);
                        end
                    end
                end
                for l = 1:n % looks for degree of other neighbors of v
                    if l ~= i
                        if T(l,j) == 1
                            f_e = f_e + deg(l);
                        end
                    end
                end
                % Check if there are two edge having the same min f(e)!!!
                if f_e < f_e_min   % controls to find min f(e).
                    ind = 1;
                    list = zeros(n,2);
```

```
66                      f_e_min = f_e;
67                      list(ind,:) = [i j]; % coordinates of vertices giving min f(e).
68                  elseif f_e == f_e_min
69                      ind = ind + 1;
70                      list(ind,:) = [i j];
71                  end
72              end
73          end
74      end
75
76  % if u is visited before
77  else
78      for i = 1:n   % Looks for each connected e=(u,v) and calculates f(e).
79          for j = i:n
80              if (T(i,j) == 1) & (i ~= visited_edgeu) & (j ~= visited_edgev) % If u and v are
                    connected
81                  f_e = deg(i)*deg(j);
82                  for k = 1:n  % looks for degree of other neighbors of u
83                      if k ~= j
84                          if T(i,k) == 1
85                              f_e = f_e + deg(k);
86                          end
87                      end
88                  end
89                  for l = 1:n  % looks for degree of other neighbors of v
90                      if l ~= i
91                          if T(l,j) == 1
92                              f_e = f_e + deg(l);
93                          end
94                      end
95                  end
96                  % Check if there are two edges having the same min f(e)!!!
97                  if f_e < f_e_min   % controls to find min f(e).
98                      ind = 1;
99                      list = zeros(n,2);
100                     f_e_min = f_e;
101                     list(ind,:) = [i j]; % coordinates of vertices giving min f(e).
102                 elseif f_e == f_e_min
103                     ind = ind + 1;
104                     list(ind,:) = [i j];
105                 end
106             end
107         end
108     end
109 end
110
111 L = zeros(ind,2);
112 for m=1:ind
113     L(m,:) = list(m,:);
114 end
115 end
```

B.7 split.m

```
function [T1 T2 list_u list_v] = split(T_r, u, v)

% function [T1 T2 list_u list_v] = split(T,u,v)
% This function splits the tree into two parts
% i.e. deletes the edge that is chosen to be removed

% inputs:
        % T_r = adjacency matrix of tree whose edge was deleted
        % u,v = vertices showing the edge chosen to be removed
% outputs:
        % T1 = one part of the adjacency matrix of original tree after removing
        % T2 = other part of the adjacency matrix of original tree after removing
        % list_u = list of u and neighbors of u (according to matrix T1)
        % list_v = list of v and neighbors of v (according to matrix T2)
% ------------------------------------------------------------------------

n = size(T_r,1);
T1 = zeros(n,n);
T2 = zeros(n,n);
list_u = zeros(n,1);
list_v = zeros(n,1);
list_u(1) = u;
list_v(1) = v;

%%% Process for list_u %%%
flag = 0;
ind = 1;
for i = 1:n-1
    if list_u(i) == 0
        break;
    else
        for j = 1:n
            flag = 0;
            if T_r(list_u(i),j) == 1
                ind = ind + 1;
                for k = 1:n-1
                    if isempty(find(list_u(k) == j))
                        flag = 1;
                    else
                        flag = 0;
                        ind = ind - 1;
                        break;
                    end
                end
                if flag == 1
                    list_u(ind) = j;
                    flag = 0;
                end
            end
        end
    end
```

```
52  end
53
54  %%% Process for list_v %%%
55  ind = 1;
56  for i = 1:n-1
57      if list_v(i) == 0
58          break;
59      else
60          for j = 1:n
61              flag = 0;
62              if T_r(list_v(i),j) == 1
63                  ind = ind + 1;
64                  for k = 1:n-1
65                      if isempty(find(list_v(k) == j))
66                          flag = 1;
67                      else
68                          flag = 0;
69                          ind = ind - 1;
70                          break;
71                      end
72                  end
73                  if flag == 1
74                      list_v(ind) = j;
75                      flag = 0;
76                  end
77              end
78          end
79      end
80  end
81
82  % constructing adjacency matrices (T1, T2) of two subtrees
83  for m = 1:n
84      if list_u(m) ~= 0
85          T1(list_u(m),1:n) = T_r(list_u(m),1:n);
86      end
87      if list_v(m) ~= 0
88          T2(list_v(m),1:n) = T_r(list_v(m),1:n);
89      end
90  end
91  end
```

B.8 findInsertionEdges.m

```
1  function [L max_g]= findInsertionEdges(adj_G, T_r, list_u, list_v)
2
3  % function [L max_g]= findInsertionEdge(adj_G, T, list_u, list_v)
4  % This function finds candidate edge(s) to be added.
5  % inputs:
6          % adj_G = adjacency matrix of graph
7          % T_r = adjacency matrix of tree whose edge was deleted
8          % list_u = list of u and neighbors of u
9          % list_v = list of v and neighbors of v
```

```
% outputs:
        % L = list of candidate edges to be added
        % max_g = maximum g(e)
% variables:
        % G_dif_T = difference matrix of adj_G and T_r
        % m_g = output of the g_e_max() function
            % = (deg_u+1) * (deg_v+1) + deg_of_neighbors_of_u + deg_of_neighbors_of_v
%-------------------------------------------------------------------------

n = size(T_r,1);
u = list_u(1);
v = list_v(1);

G_dif_T = adj_G - T_r;

m_g = 0;
max_g = 0;
L = zeros(n,2);
ind = 0;
u_n = 0;
v_n = 0;
temp = 0;

for i = 1:n
    if list_u(i) == 0
        break;
    else
        temp = list_u(i);
        for j = 1:n
            if G_dif_T(temp,j) == 1
                for k = 1:n
                    % check whether u and v is connected:
                    if ~isempty(find(list_v(k) == j))
                        u_n = temp;
                        v_n = j;
                        m_g = g_e_max(T_r, u_n, v_n);
                        % control for finding max g(e)
                        if m_g > max_g
                            ind = 1;
                            max_g = m_g;
                            L(ind,:) = [u_n v_n];
                        elseif m_g == max_g
                            ind = ind + 1;
                            L(ind,:) = [u_n v_n];
                        end
                    end
                end
            end
        end
    end
end
end
```

B.9 g_e_max.m

```
function max_g = g_e_max(T_r, u_new, v_new)

% function max_g = g_e_max(T_r,u_new, v_new)
% This function helps findInsertionEdge() function to calculate max(g(e))
% inputs:
        % T_r = adjacency matrix of tree whose edge was deleted
        % u_new, v_new = vertices defining new edge which is added
% outputs:
        % max_g = maximum g(e)
              % = (deg_u+1)*(deg_v+1)+deg_of_neighbors_of_u+deg_of_neighbors_of_v
%-------------------------------------------------------------------------

n = size(T_r,1);
deg = zeros(n,1);
max_g = 0;
count = 0;

% finds degree of each vertex of T_r.
for i = 1:n
    for j = 1:n
        if T_r(i,j) == 1
            count = count +1;
        end
    end

    deg(i) = count;
    count = 0;
end

max_g = (deg(u_new)+1)*(deg(v_new)+1);

% looks for degree of other neighbors of u
for k = 1:n
    if T_r(u_new,k) == 1
        max_g = max_g + deg(k);
    end
end

% looks for degree of other neighbors of v
for l = 1:n
    if T_r(v_new,l) == 1
        max_g = max_g + deg(l);
    end
end
end
```

B.10 degree_seq.m

```
function deg = degree_seq(T)

% function deg = degree_seq(T)
% This function calculates degree sequences of a given tree T
%-------------------------------------------------------------------------
n = size(T,1);
deg1 = zeros(n,1);
count = 0;
L1 = zeros(n,2);

for i = 1:n    % finds degree of each vertex.
    for j = 1:n
        if T(i,j) == 1
            count = count +1;
        end
    end
    deg1(i) = count;
    count = 0;
end

deg = sort(deg1,'descend');
end
```

B.11 wiener_index.m

```
function [ind D] = wiener_index(T)

% Reference: K. THILAKAM & A. SUMATHI, HOW TO COMPUTE THE WIENER INDEX OF A
% GRAPH USING MATLAB, International Journal of Applied Mathematics &
% Statistical Sciences (IJAMSS), Vol. 2, Issue 5, Nov 2013, 143-148
%-------------------------------------------------------------------------
% function [ind D] = wiener_ind(T)
% This function calculates the Wiener index of given tree
% input:
      % T = adjacency matrix of the tree T
% outputs:
      % ind = wiener index of T
      % D = distance matrix of T
%-------------------------------------------------------------------------
A = T;
% converts a sparse or full matrix to sparse form by squeezing out any zero elements.
G = sparse(A);
% finds all the shortest paths in graph.
D = graphallshortestpaths(G, 'directed', false);
% calculates total distances
M = sum(sum(D));
% take half of the total distance since it is calculated for 'directed'
ind = M/2;
end
```