ALL COLORS SHORTEST PATH PROBLEM ON TREES

AKÇAY, MEHMET BERKEHAN

JUNE 2015

ALL COLORS SHORTEST PATH PROBLEM ON TREES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF
NATURAL AND APPLIED SCIENCES OF
IZMIR UNIVERSITY OF ECONOMICS

BY
AKÇAY, MEHMET BERKEHAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

JUNE 2015

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Cüneyt GÜZELİŞ
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Turhan TUNALI
Head of Department

We have read the dissertation entitled **All Colors Shortest Path Problem on Trees** completed by **Mehmet Berkehan Akçay** under supervision of **Assoc. Prof. Dr. Hüseyin AKCAN** and **Assoc. Prof. Dr. Cem EVRENDİLEK** and we certify that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Master of Science.

Assoc. Prof. Dr. Cem EVRENDİLEK
Co-Supervisor

Assoc. Prof. Dr. Hüseyin AKCAN
Supervisor

Examining Committee Members

Date: 29.06.2015

Assoc. Prof. Dr. Hüseyin AKCAN
Dept. of Software Engineering, IUE

Asst. Prof. Dr. Gazihan ALANKUŞ
Dept. of Computer Engineering, IUE

Assoc. Prof. Dr. Deniz TÜRSEL ELİİYİ
Dept. of Industrial Engineering, Yaşar Uni.

Assoc. Prof. Dr. Cem EVRENDİLEK
Dept. of Computer Engineering, IUE

Asst. Prof. Dr. Zeynep SARGUT
Dept. of Industrial Engineering, IUE

Assoc. Prof. Dr. Mehmet Süleyman ÜNLÜTÜRK
Dept. of Software Engineering, IUE

Acknowledgments

I want to thank everybody who helped, and supported me during this thesis.

ABSTRACT

ALL COLORS SHORTEST PATH PROBLEM ON TREES

Akçay, Mehmet Berkehan

M.Sc. in Intelligent Systems Engineering
Graduate School of Natural and Applied Sciences

Supervisor: Assoc. Prof. Dr. Hüseyin Akcan
Co-Supervisor: Assoc. Prof. Dr. Cem Evrendilek
June 2015

Given an edge weighted tree $T(V, E)$, rooted at a designated base vertex $r \in V$, and a color from a set of colors $C = \{1, ..., k\}$ assigned to every vertex $v \in V$, All Colors Shortest Path problem on trees (*ACSP-t*) seeks the shortest, possibly non-simple, path starting from r in T such that at least one node from every distinct color in C is visited. We show that *ACSP-t* is NP-Hard, and also prove that it doesn't have a constant factor approximation algorithm. We give an Integer Linear Programming formulation of *ACSP-t*. Based on a Linear Programming relaxation of this formulation, several heuristics are proposed. The thesis also explores Genetic Algorithms, and Tabu Search to develop alternative heuristic solutions for *ACSP-t*. The performance of all the proposed heuristics are finally evaluated experimentally for a wide range of trees parametrically generated.

Keywords: Graph theory, integer linear programming, linear programming relaxation, genetic algorithm, tabu search, NP-Hardness, inapproximability, constant factor approximation

ÖZ

AĞAÇ YAPILARINDA TÜM RENKLERİ İÇEREN EN KISA YOL PROBLEMİ

Akçay, Mehmet Berkehan

Akıllı Mühendislik Sistemleri, Yüksek Lisans
Fen Bilimleri Enstitüsü

Tez Yöneticisi: Doç. Dr. Hüseyin Akcan
İkinci Tez Yöneticisi: Doç. Dr. Cem Evrendilek
Haziran 2015

Ağaç Yapılarında Tüm Renkleri İçeren En Kısa Yolu Bulma (TREKY-a) problemi, $r \in V$ düğümünde kökleşmiş bir $T = (V, E)$ ağacı verilip, ağaçtaki her bir düğüme $C = \{1, ..., k\}$ kümesinden bir renk atandığında, r düğümünden başlayarak, her bir farklı renkten en az bir düğüm içeren en kısa yolu bulma problemidir. Biz *TREKY-a* probleminin NP-Zor olduğunu gösteriyoruz. Ayrıca, *TREKY-a* için sabit faktörlü bir yakınsama algoritmasının olmadığını kanıtlıyoruz. *TREKY-a* problemi için tamsayı lineer programlama formülü veriyoruz. Bu formülün lineer programlama gevşetmesini temel alarak çeşitli sezgisel çözüm yöntemleri öneriliyor. Bu tez ayrıca *TREKY-a* için Genetik ve Tabu Arama algoritmalarını temel alan alternatif sezgisel çözüm yöntemleri de geliştirmektedir. Önerilen bütün sezgisel yöntemlerin performansı çeşitli parametrik tipte ağaçlarla deneysel olarak değerlendirilmektedir.

Anahtar Kelimeler: Çizge teorisi, tamsayılı lineer programlama, lineer programlama gevşetilmesi, genetik algoritma, tabu arama algoritması, NP-Zorluk, yakınsanamazlık, sabit faktör yakınsama

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In mathematics, and computer science, Graph Theory is one of the most popular research areas studying graphs. Graphs are mathematical structures used for modeling pairwise relationships between objects. There are thousands of papers, and research about graphs on a wide range of domains which affect our life, from telecommunication to transportation, from logistics to social networks, and from VLSI design to air traffic controlling.

In this thesis, a variant of All Colors Shortest Path (ACSP) problem first introduced in [1], All Colors Shortest Path Problem on trees (ACSP-t) is introduced, and explored with respect to its computational characteristics. Given an edge weighted, rooted tree with each node assigned to an apriori known color from a set of known colors, *ACSP-t* aims at finding the shortest, possibly non-simple, path starting from the root visiting every color at least once.

ACSP-t is a very generic problem with numerous applications. One typical scenario is related to item collection. In this scenario, a robot located at a specific base location is assumed to collect an instance of a list of items with one or more instances of each item distributed randomly among a known set of locations. The objective is to collect at least one instance of each item in the list by traveling the minimum distance from the base location. Once the robot has them all, it needs not move any further.

In another motivational scenario, we have a mobile agent which explores an outdoor area where various terrain types exist. These terrain types might be muddy terrains, roads, sand, meadows, forests with different types of trees, swamps, lakes, etc. The map of the area is known, and the objective of the mobile agent would be, starting from an initially known position, to explore the area, and collect sensor readings from each of the available terrain types by following the shortest path.

1.2 Contributions of the Thesis

Our main contributions in this thesis are:

1. We introduce a new, computationally unique variant of *ACSP* defined first in [1], namely All Colors Shortest Path Problem on trees.

2. We show that *ACSP-t* is NP-Hard.

3. We prove that there is no constant factor approximation algorithm for *ACSP-t*.

4. An ILP formulation of *ACSP-t* is provided.

5. Several heuristic solutions based on LP relaxation, Genetic Algorithm, and Tabu Search for *ACSP-t* are developed.

6. We conduct an intense experimental study to perform a comparative analysis of the proposed heuristics.

1.3 Organization of the Thesis

The organization of the thesis is as follows: In Chapter 1, we make an introduction to the thesis. In Chapter 2, we present the necessary terminology and the background. In Chapter 3, we discuss the related work. In Chapter 4, we introduce the problem formally, and prove its NP-Hardness, along with an inaproximability result. ILP formulation of the problem is presented in Chapter 5. In Chapter 6, some metaheuristic solutions for *ACSP-t* are developed. In Chapter 7, we present, and compare the results of the proposed heuristic solutions. Finally, the thesis is concluded in Chapter 8.

Chapter 2

Background and Terminology

A graph G is defined as an ordered pair of sets as $G = (V, E)$ where V is non-empty, finite set of nodes or vertices, and E, called edge set, is a set of connections between nodes i and j with $i, j \in V$. Nodes i and $j \in V$ are said to be adjacent if and only if $e = (i, j) \in E$. We also say, in this case, that e is incident on i and j. The vertices of an edge are called end points or end vertices. The degree of a vertex in an undirected graph is the number of edges incident on that vertex. If a weight function $w : E \rightarrow \mathbb{R}$ is defined on a graph G, then G is called an *edge weighted graph.* The weight of an edge $e = (i, j) \in E$ is represented by $w(e)$ or $w(i, j)$. If the edges in E are ordered, then the graph is called a *directed graph* or *digraph.* We also call an ordered pair (i, j) an arc. In a directed graph each arc has a direction. An arc can be traversed only in the direction of the arc. The arc (i, j) is an outgoing arc of i, and an incoming arc of j.

A *walk* is a sequence of vertices such that any two consecutive vertices i, j have an edge (i, j). If the start, and the end nodes are the same in a walk, it is called a *closed walk.* If all the edges are distinct, then it is called a *trail.* A closed trail is called a *cycle*, *circuit* or a *tour.* A *path* from i to j is a sequence of vertices $v_0, ..., v_n$ where $v_0 = i$, $v_n = j$, and each pair of successive vertices is connected by an edge. A path is simple if the vertices are distinct. A cycle is a circuit in which all vertices are visited at most once except the first (also happens to be the last). A graph is called a *cyclic graph*, if the graph contains cycles. In a *complete graph*, there is an edge between every pair of nodes. A graph is connected if all pairs of vertices $i, j \in V$ are connected by a path.

A *tree* T is a connected graph without any cycles. A node on a tree can be designated as its root, and all the edges can be thought of as directed away from it. Such a depiction in level order forms a rooted tree. A node on a higher level of a tree is called the *parent* of a *child* node one level below, and with an edge in between. Nodes with the same parent are *siblings.* A node reachable by repeatedly proceeding from the parent to a child is called a *descendant*, and a node repeatedly proceeding from a child to the parent called an *ancestor.* If a node has no children it is called a *leaf.* The *height* of a tree is the number of edges on the longest path from the root to any leaf. The *depth* of a node is the number of edges on the unique path to the root.

NP (Nondeterministic polynomial time) is one of the fundamental complexity classes in computational complexity theory [2]. A problem is considered in class NP, if it is solvable in polynomial time by a nondeterministic Turing machine. A problem is NP-Hard if every problem in NP can be reduced to it in time polynomial in the size of the problem instance. An NP-Hard problem which is also in NP is called NP-Complete.

An approximation algorithm is an algorithm which produces feasible solutions that are close to the optimal, and efficient. When the problem σ is a minimization problem, and ε is a function, ε: $Z^+ \rightarrow Q^+$ with $\varepsilon \geq 1$, an approximation algorithm A is said to be factor ε approximation algorithm for σ if, for each instance I, A produces a feasible solution s for I such that $A(I, s) \leq \varepsilon(|I|) * OPT(I)$, and the running time of A is bounded by a fixed polynomial in $|I|$. An algorithm is a *constant factor algorithm* if ε is a constant. Complexity class APX (approximable) is the set of optimization problems which allows for a polynomial time approximation algorithm with an approximation ratio bounded by a constant.

A linear programming (LP) is the problem of minimizing or maximizing a linear function subject to linear equality, and linear inequality constraints. Linear programming can be solved using either the simplex method [3], ellipsoid method [4], or a much more efficient algorithm by Karmarkar [5]. Integer Linear Programming is a special case of LP, in which all variables must take on integer values. ILP is shown to be NP-Hard in [6].

Chapter 3

Related Work

When the underlying graph in *ACSP* is restricted to be a tree in *ACSP-t*, the computational nature changes dramatically. As an example, a special case of *ACSP-t* arises when each node has a different color. In this case the objective becomes finding a possibly non-simple path traversing the entire tree with the shortest distance, which can be solved in polynomial time. However, *ACSP*, in this special case, turns into the Hamiltonian Path problem well-known to be NP-Complete [2].

Although there are several similar problems examined in the literature, *ACSP-t* is computationally unique. Generalized Minimum Spanning Tree (GMST) introduced by Myung, Lee, and Tcha in [7] is probably the closest problem. Given an undirected graph $G = (V, E)$ with its vertex set partitioned into m clusters *GMST* is defined to be the problem of finding the Minimum Spanning Tree that visits exactly one node from every cluster. This problem has been shown to be NP-Hard in [7]. Some inaproximability results for it are presented by Pop in [8]. In [9], [10], and [11], Integer Linear Programming formulations of *GMST* are also proposed. Feremans et al. study the polytope associated with the *GMST* problem in [9]. *GMST* problem restricted to trees has been studied by Pop, and has been shown to be NP-Hard in [12]. Another variant by Dror et al. in [13], called *l-GMST*, relaxes *GMST*, and allows more than one node from every cluster to be visited. They also present different heuristic solutions including a genetic algorithm. Although *l-GMST* appears to be computationally similar to *ACSP-t*, they differ in various ways.

In Figure 3.1, solutions of *l-GMST*, and *ACSP-t* for a given problem instance are presented. Each node, in this figure, is labeled with i/c where i corresponds to the node, and c to the color. The tree is rooted at r with color 0. All edge weights are assumed to be equal to one. When the given instance is viewed as an instance of *l-GMST*, the optimal solution is the sub-tree enclosed in a rectangle with a cost of 4 as shown in the figure. The best that could be obtained from it as also a solution to *ACSP-t* has a cost of 7. Yet the optimal solution by *ACSP-t* as shown to the bottom of Figure 3.1, has cost 6. So the shape of the solution

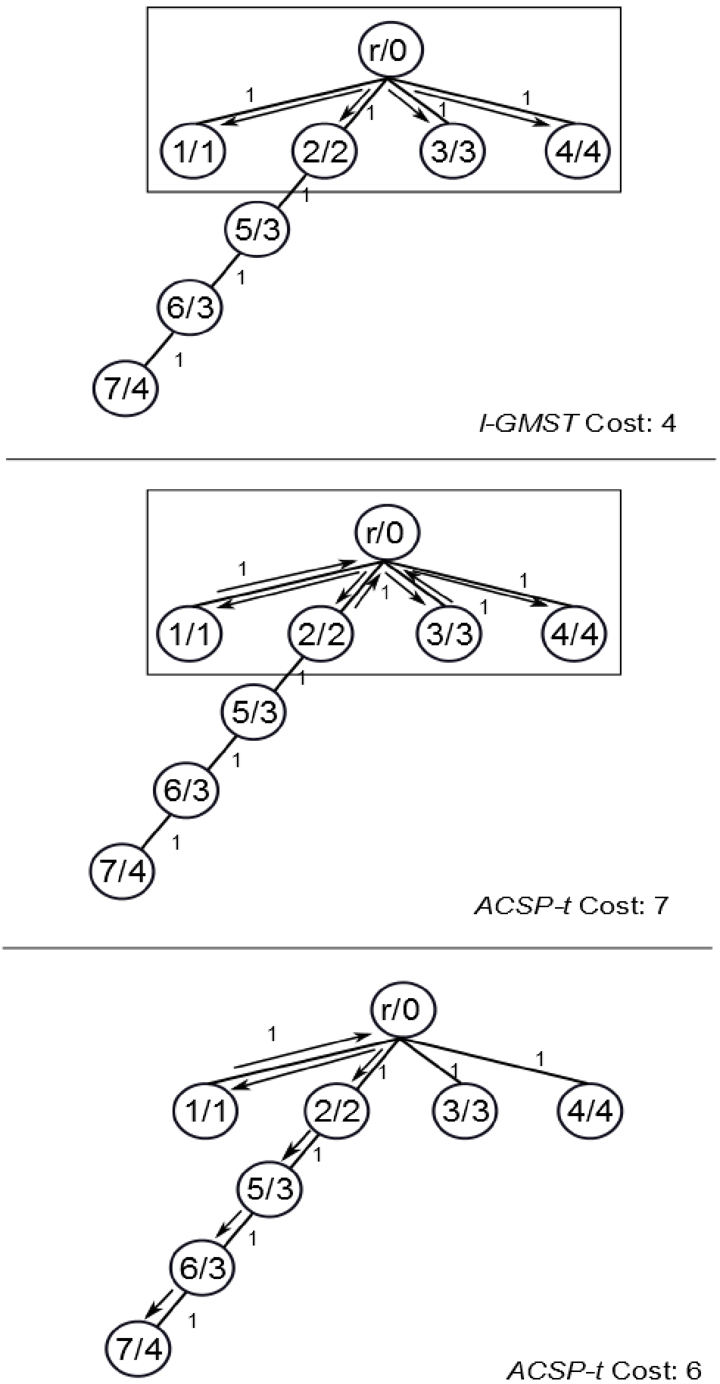

Figure 3.1: The optimal solution by *l-GMST* (above), the corresponding solution for ACSP-t (middle), and the optimal solution by *ACSP-t* (below) to the same problem instance.

has an impact which will be revisited later again.

Another problem similar to *ACSP-t* is the Generalized Steiner Tree Problem (GSTP) introduced by Reich, and Widmayer in [14]. *GSTP* is defined on a complete, edge weighted, undirected graph G with a subset of nodes S partitioned into m clusters to find the minimum cost tree in G that contains at least one node from each cluster. This NP-Hard problem is shown to be a direct generalization of the set cover problem in [15], [16], and [17]. Ihler et al. show that the problem is NP-Hard even on trees in [18]. Garg et al. introduce a polylogarithmic approximation algorithm for this problem in [19]. In [20], *GSTP* is proved to be not approximable to within $\Omega(log^{2-\epsilon}n)$ unless NP admits a quasipolynomialtime Las Vegas algorithm.

Generalized Traveling Salesman Problem (GTSP) formulated by Labordere in [21] is another problem that has similar features to *ACSP-t* problem. Given a graph with the vertex set partitioned into m disjoint clusters, *GTSP* is, then, finding the shortest Hamiltonian tour containing exactly one or at least one node from each cluster. Laporte and Nobert [22] prove that "exactly one" version corresponds to "at least one" node version when the distance matrix is Euclidean. They also develop the first ILP formulation for *GTSP*. A dynamic programming formulation is proposed as a solution procedure in [23]. In [24], an ILP formulation to this problem is presented when the distance matrix is asymmetrical. Lien, Ma, and Wah show that a given instance of *GTSP* can be transformed into standard Traveling Salesman Problem [25] efficiently with the same number of nodes in [26].

Chapter 4

Formal Problem Definition and Its Computational Complexity

In this chapter, we give a formal definition of *ACSP-t* problem, prove its NP-Hardness, and make some observations leading to an inapproximability result for *ACSP-t*.

4.1 Formal Definition

Given a tree $T(V, E)$ rooted at $r \in V$, a function $w : E \rightarrow \mathbb{R}^+$ associating positive weights to the edges, and another function *color* : $V \rightarrow C$ mapping vertices in $V = \{1, ..., n\}$ to colors in $C = \{1, ..., k\}$, *ACSP-t* is then to find the shortest, possibly non-simple, path starting from the root $r \in V$ such that every distinct color is visited at least once.

4.1.1 Example

A tree for an example instance of *ACSP-t* is given in Figure 4.1. The tree is rooted at the root r which has color 0. Each node, in this figure, is labeled with i/c where i corresponds to the node, and c to the color. The instance has 15 nodes, and 5 colors including the color of the root. A feasible solution is $r, 3, r, 2, r, 4, r, 1$ with a total cost of 26, in which all nodes visited are on the first level of the tree. Another solution is $r, 1, 5, 11, 14$ with a total cost 20, in which all colors are visited on a single branch of the tree. Optimal solution, on the other hand, is $r, 2, 7, 12, 7, 13$ with a total cost of 10.

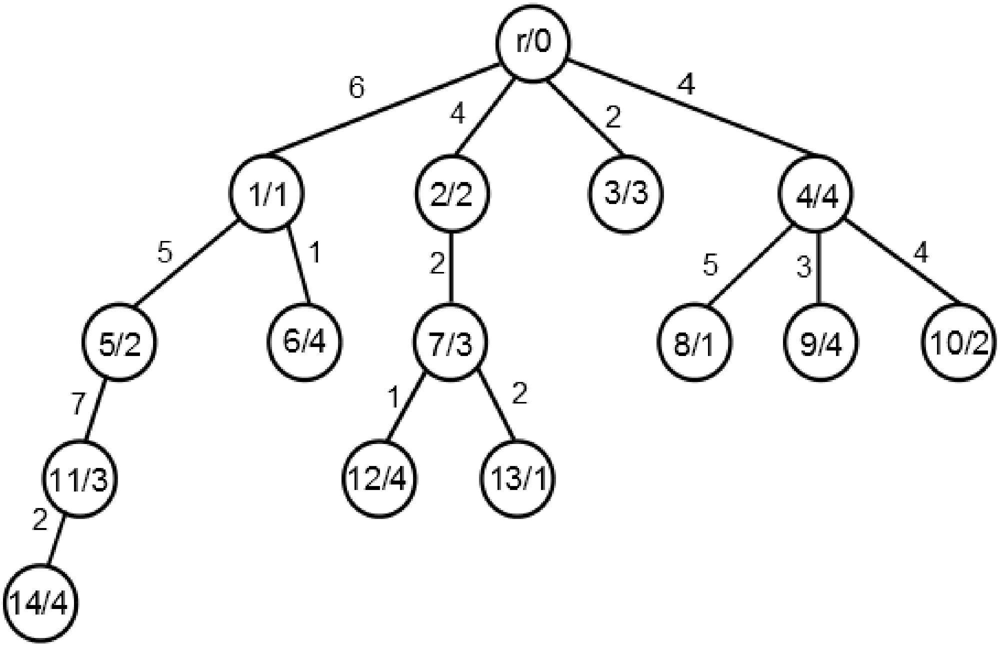

Figure 4.1: A tree for an example instance of *ACSP-t*.

4.2 NP-Hardness

We prove that *ACSP-t* is NP-Hard by a polynomial time reduction from the Hitting Set Problem (HSP) [2]. HSP is known to be NP-Hard [2], and also a variant of the well-known Set Cover (SC) problem [27].

HSP: Given $X = \{x_1, x_2, ..., x_n\}$ as a base set, $k \in N^+$, and a collection of m sets $S_1, S_2, ..., S_m$ with $S_i \subseteq X$, the objective of *HSP* is to find $Y \subset X$ such that $|Y| \leq k$, and $\forall i \; S_i \bigcap Y \neq \emptyset$ hold.

A given instance of *HSP* can be transformed into a corresponding instance of *ACSP-t* as follows:

The color set C is initialized to have $n+m+1$ colors as $C = \{c_0, c_1, ..., c_{n+m}\}$.

• We first create a node r as the root such that $color(r) = c_0$.

• For each $x_i \in X$, two new nodes x_i, and x'_i are created, both assigned to the same color as $color(x_i) = color(x'_i) = c_i$.

• Lastly, for each $S_i \in \{S_1, ..., S_m\}$ in the given instance of *HSP*, we create $|S_i|$ nodes. For each element $x_j \in S_i$, a node $S_{i,j}$ is created. All the nodes $S_{i,j}$ for a given i, are assigned to the same color c_{n+i}. The replication of nodes ensures that a tree structure will be maintained in the subsequent construction.

Once the nodes with the corresponding colors are created, the tree in the corresponding instance of *ACSP-t* is constructed as shown in Figure 4.2.

• Each node x_i is connected to the root r with an edge of cost (weight, distance) one while node x_i' is connected to r with an edge cost zero.

• For each node $S_{i,j}$, an edge from $S_{i,j}$ to x_i as its parent is created with weight 0.

All colors must be visited at least once. The color 0 is visited by a visit to the root r. For colors c_1 through c_n either x_i or x_i' can be visited. As there are no edges from nodes x_i' to any $S_{j,i}$ however, some x_i nodes must also be visited to cover the colors c_{n+1} through c_{n+m}.

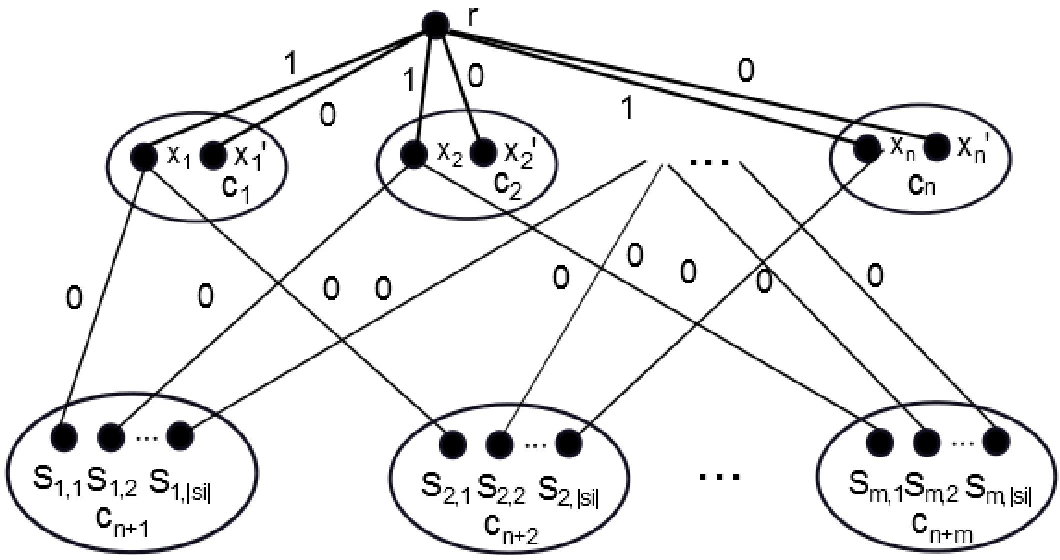

Figure 4.2: Reduction from Hitting Set.

This transformation is obviously polynomial in the size of the given instance of *HSP*. A total of $1 + 2n + mn$ nodes including the root are created in the worst case, assuming each S_i covers all the elements in the base set X. The total number of edges created, on the other hand, is $2n + mn$ in the worst case. So the entire transformation takes $O(mn)$ time which is directly proportional to the size of $S = \{S_1, S_2, ..., S_m\}$ in the given instance of *HSP*.

Lemma 4.2.1 *A given instance of HSP has a solution with size less than or equal to k if and only if ACSP-t has a solution of path length less than or equal to $2k - 1$.*

Proof (If part): If there is a solution P in the corresponding instance of *ACSP-t*, obtained through the transformation described, with path length less than or equal to $2k - 1$, then by choosing $Y = \{x_i | x_i \in P\}$ we obtain a *hitting set* with size less than or equal to k.

(Only if part): If a *hitting set* Y with $|Y| \leq k$ exists in the given instance of HSP, a DFS (Depth First Search) traversal of the subtree constrained to the nodes $x_i \in Y$, and their descendants only, has a cost less than or equal to $2k-1$. □

Theorem 4.2.2 *ACSP-t is NP-hard.*

Proof The transformation is polynomial. This coupled with Lemma 4.2.1 readily proves the theorem. □

4.3 Inapproximability

It is shown in [18] that *l-GMST*, referred to as *CLASS TREE* problem there, does not admit a constant factor polynomial time approximation algorithm, even when the underlying graph is restricted to be a tree. Equipped with this knowledge, we can make the following similar observation for *ACSP-t*, in the same way it has been previously formulated for *ACSP* in [1].

Observation 1 *For a given valid instance I of l-GMST on trees (l-GMST-t),*

$$OPT_{l\text{-}GMST\text{-}t}(I) \leq \min_{J \in V}\{OPT_{ACSP\text{-}t}(I_j)\} < 2 * OPT_{l\text{-}GMST\text{-}t}(I)$$

where I_j is the corresponding instance of ACSP-t obtained by designating $j \in V$ as the root.

Proof We prove the two inequalities separately for the given expression. Let us assume, by contradiction that $OPT_{l\text{-}GMST\text{-}t}(I) > \min_{j \in V}\{OPT_{ACSP\text{-}t}(I_j)\}$. In this case, *l-GMST-t* can simply adopt the solution that gives the minimum over all such instances for *ACSP-t*. All it takes is to cast the non-simple path to a tree by disregarding any duplicate edges, and hence a contradiction.

Let us assume the latter inequality does not, once more, hold, and

$$\min_{j \in V}\{OPT_{ACSP\text{-}t}(I_j)\} \geq 2 * OPT_{l\text{-}GMST\text{-}t}(I).$$

But we know that the optimal solution of *l-GMST-t* is a tree spanning all colors, and a DFS traversal of all nodes in it gives a non-simple path with length strictly less than twice the cost of this tree. This, obviously is a solution for one of the instances I_j to *ACSP-t*, contradicting the assumption.

This observation lets us prove the following theorem easily for *ACSP-t* in the same way it was established for *ACSP* in [1]:

Theorem 4.3.1 *There is no constant factor polynomial time approximation (apx) for ACSP-t unless $P = NP$.*

Proof Let us assume, contrary to the theorem, that there is such an algorithm $apx_{ACSP\text{-}t}$ satisfying $apx_{ACSP\text{-}t}(I) \leq c * OPT_{ACSP\text{-}t}(I)$ for all valid instances I, and a constant $c > 1$. Now, given an instance I of *l-GMST-t*, let us feed I_j obtained by designating j as the root in the corresponding *ACSP-t* instance into apx_{ACSP-t} for each $j \in V$. We know, by Observation 1, that $OPT_{l\text{-}GMST\text{-}t}(I) \leq \min_{J \in V}\{OPT_{ACSP\text{-}t}(I_j)\} < 2 * OPT_{l\text{-}GMST\text{-}t}(I)$.

As $\forall j \in V$ $apx_{ACSP\text{-}t}(I_j) \leq c * OPT_{ACSP\text{-}t}(I_j)$ holds by the assumption made,

$$\min_{j \in V}\{apx_{ACSP\text{-}t}(I_j)\} \leq c * \min_{j \in V}\{OPT_{ACSP\text{-}t}(I_j)\} < 2 * c * OPT_{l\text{-}GMST\text{-}t}(I)$$

is readily obtained. This, by definition, indicates the existence of a $2c$ *apx* for *l-GMST-t*, and hence a contradiction as it certainly takes polynomial time to run apx_{ACSP-t} $O(n)$ times.

Chapter 5

Integer Linear Programming Formulation of ACSP-t

In this chapter, an ILP formulation is developed for *ACSP-t*. We then, relax it to LP, and propose several heuristic solutions based on this LP relaxation.

5.1 ILP Model

We use 0-1 Integer Programming for *ACSP-t*, where variables are restricted to be either 0 or 1. A given instance of *ACSP-t* is represented by an edge weighted tree $T(V, E)$ rooted at $r \in V$ such that *color* is a function mapping each vertex $\in V = \{1, 2, ..., n\}$ to a color from $C = \{1, 2, ..., k\}$, and the weight of an edge $(i, j) \in E$ is denoted by $w_{i,j}$. In order to give a compact ILP formulation, we treat each edge $(i, j) \in E$ as two directed edges through a transformation. The binary variable $x_{i,j} \in \{0, 1\}$ is then, easily defined to be set to 1 if and only if the directed edge (i, j) is visited in the solution. For each undirected edge $(i, j) \in E$, both of the directed edges (i, j) and (j, i) after the transformation are assumed to have the same weight $w_{i,j}$. We also introduce two more nodes as the source, and the sink. While the source is denoted by 0, the sink is numbered as $n + 1$. These two nodes are assigned to a brand new color 0. We add a directed edge $(0, r)$ from the source to the original root of weight zero, as well as edges $(i, n+1)$ for all $i \in V$ each with a weight of zero. Sink node $n + 1$ must be the last node visited in any feasible solution. When all colors are visited, edge to $n + 1$ is taken, and the path terminates. Now the transformed instance has $C' = C \cup \{0\}$, $V' = V \cup \{0, n + 1\}$, $E' = \{(0, r)\} \cup \{(i, n + 1)|i \in V\} \cup \{(i, j), (j, i)|(i, j) \in E\}$, and the weight and color functions, using the same notation as before, have been augmented by $color(0) = color(n + 1) = 0$, $w_{0,r} = 0$, $w_{i,n+1} = 0\ \forall i \in V$, and finally $w_{i,j} = w_{j,i}\ \forall (i, j) \in E$.

The ILP formulation follows:

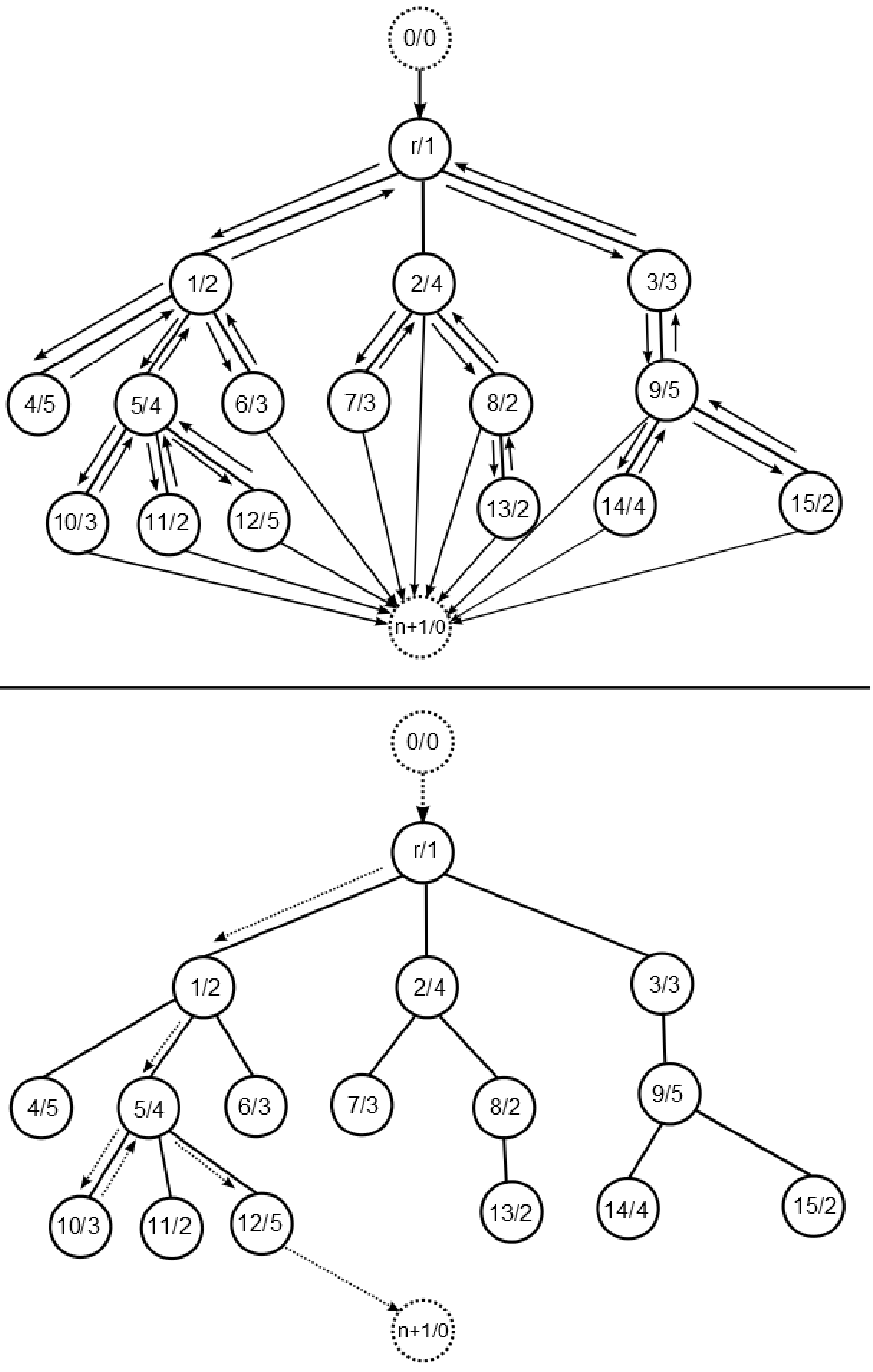

Figure 5.1: The transformed instance with directed edges shown by arrows, and additional nodes shown by dashed circles is given above. A feasible solution path shown by dashed arrows is depicted below. n is equal to 15, hence node id of additional node is equal to 16.

$$\text{minimize} \sum_{(i,j):(i,j)\in E'} x_{i,j} w_{i,j} \tag{5.1.1}$$

subject to

$$\sum_{j:(j,i)\in E'} x_{j,i} - \sum_{j:(i,j)\in E'} x_{i,j} = 0, \quad \forall i \in V \tag{5.1.2}$$

$$\sum_{(i,j):(i,j)\in E' \wedge color(j)=c} x_{i,j} \geq 1, \quad \forall c \in C \tag{5.1.3}$$

$$x_{parent(parent(i)),parent(i)} \geq x_{parent(i),i}, \quad \forall i \in V - \{r\} \tag{5.1.4}$$

$$x_{i,j} \in \{0,1\}, \quad \forall (i,j) \in E' \tag{5.1.5}$$

Our objective function in 5.1.1 computes the length of feasible path in an effort to minimize it. The result is the total cost of the selected edges. In order to restrict the shape of the solution to a possibly non-simple path, constraint 5.1.2 is used to ensure that the number of edges that enter a node is equal to the ones that exit. In order to overcome the difficulty of dealing with exceptional nodes such as the root r, and the last node on a feasible path, the source, and the sink have been introduced. Constraint 5.1.3 ensures that all colors are visited at least once. Constraint 5.1.4 is used to enforce the connectivity of the nodes. It is actually a kind of sub-tour elimination constraint. A node cannot be visited before its parent is visited. With constraint 5.1.4, if there is no edge that enters a node from its parent as part of the solution, there can't be an edge that exits from that node. It should be noted that, for all $i \in V$ the $parent(i)$ is already defined. Constraint 5.1.5 dictates that all the decision variables are either 0 or 1 in any feasible solution.

5.2 Linear Programming Relaxation

Although we can get optimal solutions via the given ILP formulation for small sized instances of *ACSP-t* in a reasonable amount of time, it cannot solve large instances of *ACSP-t* in polynomial time as ILP is NP-Hard. For that reason, the ILP formulation in Section 5.1 is relaxed to a Linear Programming formulation by replacing the last constraint with a weaker constraint that ensures that each variable is in the [0,1] interval. This LP model is presented next.

$$\text{minimize} \sum_{(i,j):(i,j)\in E'} x_{i,j} w_{i,j} \tag{5.2.1}$$

subject to

$$\sum_{j:(j,i)\in E'} x_{j,i} - \sum_{j:(i,j)\in E'} x_{i,j} = 0, \quad \forall i \in V \tag{5.2.2}$$

$$\sum_{(i,j):(i,j)\in E' \wedge color(j)=c} x_{i,j} \geq 1, \quad \forall c \in C \tag{5.2.3}$$

$$x_{parent(parent(i)),parent(i)} \geq x_{parent(i),i}, \quad \forall i \in V - \{r\} \tag{5.2.4}$$

$$0 \leq x_{i,j} \leq 1, \quad \forall (i,j) \in E' \tag{5.2.5}$$

5.2.1 Heuristics based on the LP-relaxation

Linear Programming is able to give us a solution in polynomial time in the size of the instances. But in a feasible solution, fractional values for $x_{i,j}$ as opposed to integer values are returned by the LP-relaxation. In order to round the fractional values returned by LP to either 0 or 1, we propose 2 different strategies used at the core of the corresponding heuristics. It should be noted that the inapproximability result previously reported in this thesis lowers our expectations for promising results via these types of approaches.

Our first heuristic, called LP-oneshot, modifies the solution obtained by LP in such a way that highest 20% of $x_{i,j}$ variables are rounded to 1 while the others are set to 0 in a single iteration. After this process, a possible disconnectivity among the nodes is fixed via a post processing algorithm. First, we connect each disconnected edge to the root. Then, we find all unvisited colors. For each unvisited color we search for the closest node within the set of already visited nodes. Finally, we connect the node with the minimum distance to an unvisited color. This process is repeated until all colors are visited.

In the second heuristic which we name LP-iterative, the rounding of variables are done in a decreasing order of their values iteratively: We find the highest $x_{i,j}$ variable, and round it to 1 by adding $x_{i,j} = 1$ to the current LP formulation for a subsequent call to LP. This process is repeated until all $x_{i,j}$ values are either 0 or 1. LP is called n times in worst case, and runtime is $O(n).RT(LP)$. If there are edges with equal $x_{i,j}$ values, we break the ties in favor of those minimizing $distance/\#ofcolors$ where $distance$ is the total distance to get to this edge, and $\#colors$ is the number of visited colors on this path.

Chapter 6

Metaheuristic Solutions

In this chapter we present several metaheuristic approaches for solving *ACSP-t* problem. In Section 6.1 we suggest a Genetic Algorithm based solution for *ACSP-t* while, in Section 6.2, we present a Tabu Search Algorithm.

6.1 Genetic Algorithm

Genetic Algorithm (GA) is a metaheuristic, first introduced by Holland [28], based on evolutionary aspects of natural selection, and genetics in order to solve combinatorial optimization problems. GA uses natural selection, recombination, and mutation to solve problems. GA mimics natural evolutionary process by focusing on survival of the fittest among individuals in a population over generations. Therefore, in a search space, only the finest solution may survive, and evolve towards better solutions. GA has five main phases: initialization, fitness, selection, crossover, and mutation. In the initialization step we create a population using randomly generated initial solutions. In GA, a solution is referred to as a chromosome, and represents an encoding to the original problem. Each chromosome is composed of genes, which are individual pieces of the encoding. In the fitness phase, a fitness function is used to evaluate the quality of the proposed solution. In the selection phase, chromosomes are chosen among the population to perform crossover, by which the fittest individuals transfer their genes to the next generations. The selection phase ensures that the fittest individuals in a population will produce more offspring than those that perform poorly. Roulette Wheel Selection algorithm [28] is a popular selection algorithm which uses fitness values f_i of each chromosome to associate a probability of selection. Probability of selection for a solution is, then, $p_i = \frac{f_i}{\sum_{i=1}^{N} f_i}$, where N is number of the chromosomes. Even though, having a higher fitness value ensures a higher chance to be selected for crossover, to avoid local optimals, GA also gives chance to individuals with lower fitness values. Crossover phase is initiated after candidates are selected. In the crossover phase, selected chromosomes exchange genes with each other similar to genetic crossover in nature. There are several ways to perform crossover such as single point crossover, two points crossover, cut and splice crossover, uniform crossover. In this phase, two new chromosomes are created from the parent chromosomes. After the crossover, GA enters the mutation phase, in which individual genes are changed randomly based on a mutation rate. Mutation is a necessary operator to maintain the diversity among the generations in GA.

We developed two variants of GA to solve *ACSP-t*. In the first one we use color encoding which is presented in Section 6.1.1. The algorithm presented in Section 6.1.2 uses path encoding. Details of the algorithms are given below.

6.1.1 Color Encoding Approach

In this version, called GA-color, we use strings of size m, where m is equal to the number of colors, to represent a feasible solution for a given *ACSP-t* instance. The root r is not added to the chromosome, since it, and its color are always included in the solution. An example representation is shown in Figure 6.1 for the tree given in Figure 6.2. The tree is rooted at node 0. Each node, in this figure, is labeled with i/c where i corresponds to the node, and c to the color. The algorithmic outline of a typical GA is given in Figure 6.3.

Nodes	7	6	11	3
Colors	1	2	3	4

Figure 6.1: An example representation for GA-color.

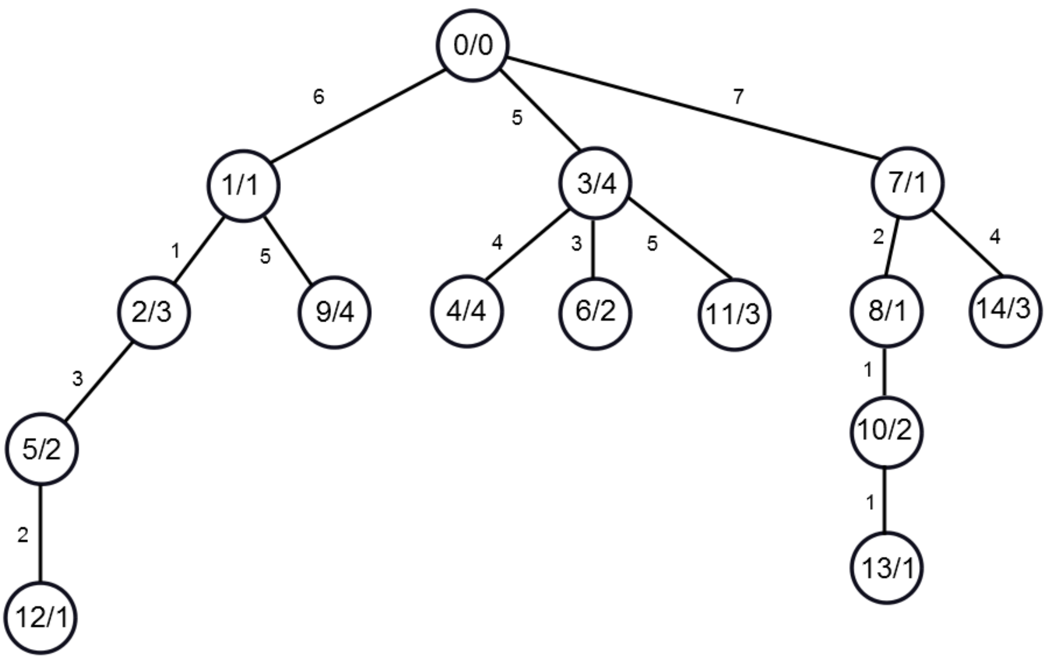

Figure 6.2: An example tree with 15 nodes and 5 colors. Each node is identified by node id/color.

```
input: ACSP-t instance identified with T(V, E) rooted at r.
output: Best solution in population
1: function GeneticAlgorithm(T)
2:     Population ← {};
3:     for i = 0 to populationSize do
4:         chromosome ← createPopulation(T);
5:         calculateCost(T,chromosome);
6:         add chromosome to Population;
7:     end for
8:     for i = 0 to iterationsize do
9:         selectedParents ← rouletteWheelSelection(Population);
10:        children ← Crossover(selectedParents);
11:        for each child c in children do
12:            r ← random(0,1) ;
13:            if r < mutationRate then
14:                mutation(c) ;
15:            end if
16:            calculateCost(T,c);
17:        end for
18:        remove the worst 2 chromosome from Population;
19:        add children to Population;
20:    end for
21:    bestSolution ← best solution in Population;
22:    formPath(T,bestSolution);
23: end function
```

Figure 6.3: Genetic Algorithm

In the initialization step, first, we create an empty population in line 2. In line 4, we create a chromosome by choosing a random node from each color. In line 5, we calculate the cost of the chromosome, using Algorithm calculateCost presented in Figure 6.9 which we explain in detail at the end of this section, and add the chromosome to the population in line 6. This process is repeated until the population size saturates in lines 3 - 7.

Line 8 through 20, GA iterates. Iteration size has a huge impact on quality of the solution, and fine tuned in the experiments. First, we select two parent chromosomes using Roulette Wheel Selection algorithm in line 9. Then, we perform crossover in line 10. We use uniform crossover in GA-color. In this phase, we change genes of the parent chromosomes with each other. For each node belonging to a color, with a crossover probability we swap nodes between parent chromosomes. This crossover process is done in linear time. With crossover process we get two new chromosomes which have features from both of the parent chromosomes. An example crossover is shown in Figure 6.4.

Mutation phase ensures diversification among the population. In the mutation phase, first, we select a random number between 0 and 1 for each node in line 12. If this number is lower than mutation rate, we select 10% of the colors, and for

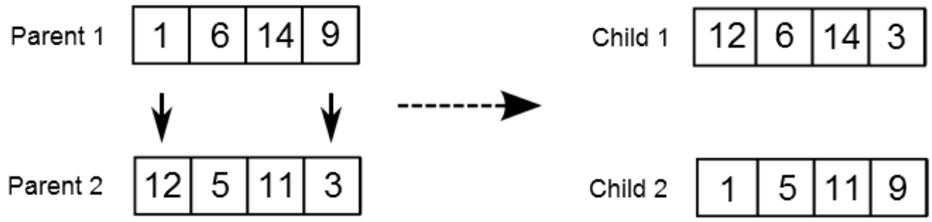

Figure 6.4: Crossover: Parent 1, and Parent 2 on the left side, and Child 1, and 2 on the right side after crossover.

each selected color, we change it with another random node belonging to the same color in line 14.

After mutation, we calculate the cost of the child chromosome in line 16. When crossover and mutation are completed, the two worst chromosomes in the population are replaced by chromosomes generated in crossover phase in line 18, and 19.

GA-color is iterated for a fixed number of times dictated by the parameter iteration size. When the iterations are completed, the best solution found is represented a chromosome. The construction of the shortest, possibly non-simple, path corresponding to this chromosome is obtained by a simple traversal of the sub-tree spanning the entire chromosome such that the nodes on the path leading to the farthest away leaf to be visited.

We use Algorithm FORMPATH given in Figure 6.5 which, after initialization, calls the function that actually does the construction. The algorithm uses a global variable $pathCost$ which is a precomputed $n \times 2$ matrix that contains the pairwise shortest distances of the nodes in T.

```
global: pathCost is a precomputed global n×2 matrix which contains the pairwise
    shortest path costs of the nodes in T.
input: ACSP-t instance identified with T(V, E) rooted at r, node set S denotes the
    set of nodes that must be visited in T.
output: shortest, possibly non-simple, path path passing through all the nodes in S.
function formPath(T, S)
    stack ← empty stack;
    path ← ∅;
    mustPass[i] ← false, ∀i ∈ V;
    mustPass[r] ← true;
    for each node v in S do
        while !mustPass[v] do
            mustPass[v] ← true ;
            v ← parent[v];
        end while
    end for
    furthestfromRoot ← max_{v∈S} pathCost[r][v];
    while furthestfromRoot ≠ r do
        push(furthestfromRoot, stack);
        furthestfromRoot = parent[furthestfromRoot];
    end while
    path ← path || r;          /* || is append operator.*/
    path ← path || PathFurthestLast(T, path, mustPass, stack)
end function
```

Figure 6.5: formPath function which does the necessary initializations and call the function that construct the shortest path.

```
input: ACSP-t instance identified with T(V, E) rooted at r, partially constructed
    path path, the set of nodes that must be visited mustPass, a stack keeping nodes
    on the path leading to the farthest away leaf in mustPass.
output: shortest, possibly non-simple, path path passing through all the nodes in S.
function PathFurthestLast(T, path, mustPass, stack)
    if stack is empty then
        return path;
    end if
    n ← pop(stack);
    path ← path || ConstructPath_Stub(T, n, mustPass);
    path ← path || n;
    return PathFurthestLast(T, path, mustPass, stack)
end function
```

Figure 6.6: PathFurthestLast function to construct the shortest path for given nodes by traversing the farthest away leaf last.

input: *ACSP-t* instance identified with $T(V, E)$ rooted at r, partially constructed path *path*, a node v on the path leading to farthest away node whose siblings are traversed, the set of nodes that must be visited *mustPass*.
output: Path that traverses the nodes not saved for subsequent visit.

```
function ConstructPath_Stub(T, v, mustPass)
    constPath ← ∅
    for each child c in parent[v].children - v do
        constPath ← constPath || constructPath(T, c, mustPass, constPath)
        constPath ← constPath || parent[c];
    end for
end function
```

Figure 6.7: ConstructPath_Stub function to traverse the nodes not saved for subsequent visit.

input: *ACSP-t* instance identified with $T(V, E)$ rooted at r, a root node v for traversal, the set of nodes that must be visited *mustPass*, a partially constructed path *constPath*
output: a path traversing the subtree rooted at v which is given as input.

```
function constructPath(T, v, mustPass, constPath)
    if mustPass[v] then
        constPath ← constPath || v;
    end if
    for each child c of v do
        if mustPass[c] then
            constPath ← constructPath(T, v, mustPass, constPath);
            constPath ← constPath || v;
        end if
    end for
    return constPath;
end function
```

Figure 6.8: constructPath function to traverse subtree rooted at given node in input.

The cost of a chromosome is calculated by using Algorithm calculateCost. It takes *ACSP-t* tree $T(V, E)$, and a chromosome S as input, and returns the cost of the chromosome. First, in line 2 of this algorithm, all nodes are set as unvisited, and in line 3, the root is set as visited. Then, for each node in chromosome, starting from that node we sum the edge weights until we encounter a visited ancestor in line 7, and set nodes visited in line 8. In line 12, we find the farthest node to the root, and its path cost in line 13. Finally, we multiply sum of all calculated edge weights by 2, and subtract the total weight of the path from the maximum distance node to the root in line 14.

```
input: ACSP-t instance identified with T(V,E) rooted at r, a chromosome S.
output: Total Cost of S
function calculateCost(T, S)
    visited[i] = 0; ∀i ∈ V          /* boolean array for visited nodes*/
    visited[r] = 1;
    edgeCost = 0;          /*total weight of selected edges*/
    for each node n ∈ S do
        while visited[n] ≠ 1 do
            edgeCost+ = weight[parent[n],n];
            visited[n] = 1;
            n ← parent[n];
        end while
    end for
    furthestfromRoot ← max_{v∈S} pathCost[r][v];
    pathCostofFurthest = pathCost[r][furthestfromRoot];
    return (2 * edgeCost − pathCostofFurthest);
end function
```

Figure 6.9: calculateCost function to calculate the cost of the feasible solution.

6.1.2 Path Encoding Approach

In our second algorithm, named GA path, we represent our chromosome as a sequence of edges, which represents the visiting order of the edges. An example chromosome which forms a path visiting each color at least once is given in Figure 6.10 for the tree in Figure 6.2.

<0,7>	<7,0>	<0,3>	<3,6>	<6,3>	<3,11>

Figure 6.10: An example representation for GA-path.

The outline of the main algorithm does not change. Moreover, computing the cost is now much easier. In the initialization step, we select a node for each color. Then, using Algorithm formPath given in Figure 6.5, we create a path which visits all colors. The edges of the path are corresponding to the genes of the chromosome. We compute the cost by summing all of the selected edges. After that, we calculate the fitness of the chromosome.

In the crossover phase, we use Roulette Wheel Selection algorithm to select the parent chromosomes. For GA-path, we choose single point crossover as shown in Figure 6.11. To the top of the figure, the parent chromosomes, Parent1, and Parent2, are seen, where the arrows represent the crossover points. With the crossover, we exchange parts of the chromosomes. To the bottom of it, newly generated children, Child1, and Child2, are given.

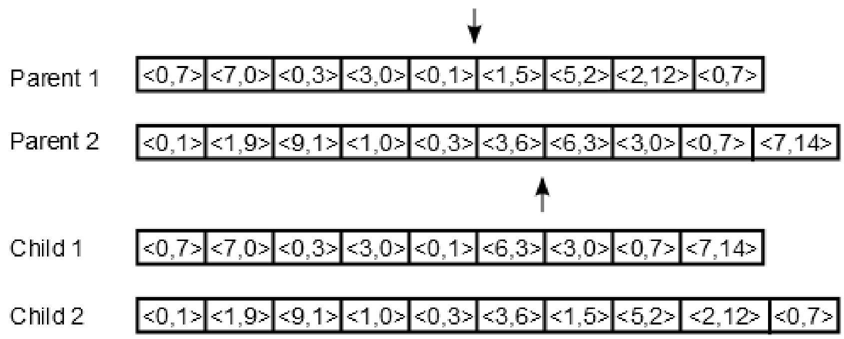

Figure 6.11: GA-path: Parents (above) and children (below).

Mutation phase follows the crossover. For each child, we select a random number. If this number is lower than the mutation rate, we perform mutation. For mutation, we delete 10 % of the genes at the end of the chromosome. Then, we select an unvisited edge, and connect it to the node with the minimum distance.

The chromosome returned by the crossover might be a disconnected path. In such a case, we use the pairwise shortest paths to reconnect the paths. After reconnection, we check the solution to see whether there is any missing color. If missing colors exist, for each missing color, we find a node with the minimum distance to the end of the path, and connect it to the last node in path. This process is repeated until all colors are visited. Duplicate edges, if any, are deleted, and a valid solution that visits each color at least once is generated. Then, we calculate the fitness of the chromosome. After crossover and mutation the two worst chromosomes in the population are replaced by the chromosomes generated in the crossover phase.

When the iterations are completed, best chromosome found so far is returned, and the path is rearranged using Algorithm FORMPATH in Figure 6.5. Then, starting from the beginning of the chromosome, we check the colors of the nodes on the path. When all of the colors are visited, we crop the rest of the chromosome, calculate the cost of the path, and finally return it.

6.2 Tabu Search

Tabu Search is a metaheuristic used to solve optimization problems that employs iterative local search methods. Tabu search algorithm was first developed by Glover [29] in 1986. It is used to solve a wide range of classical and practical problems ranging from graph theory to scheduling problems, and telecommunications. Tabu search starts from an initial solution, iteratively searches a subset of the solution neighborhood, and returns the best solution within the neighborhood. A neighbor solution is obtained by replacing elements in a solution. A mechanism called a short term memory is used to prevent from visiting the same solutions over and over again. This mechanism provides a way to escape from local minima. The attributes of the recently visited solutions are declared as forbidden or tabu; and not considered as a neighbor solution unless they fulfill an aspiration criterion. If a solution declared tabu is good enough to reconsider, its tabu status is overridden. By using aspiration criteria, we have a chance to catch good solutions that are considered as tabu. One of the mostly used aspiration criteria is that when an attribute declared tabu yields a better solution than the best known solution, tabu status of the attribute is overridden, and the solution is allowed to be reconsidered as a neighbor solution.

Tabu attributes are kept in a tabu list for a certain time period which is called tabu tenure. After the tabu tenure elapses, tabu attributes are freed to be considered within a new solution. The search continues for a fixed number of iterations attributed by the iteration count. When a specified number of iterations have elapsed in total, or since the last best solution was found, the search stops. Then, the tabu search may either be terminated, or a new search iteration start for diversification.

In the rest of this section we present tabu search methods developed specifically for *ACSP-t* problem with different neighborhood search methods.

For *ACSP-t*, we employ a tabu search algorithm which similar to the one used to solve *GMST* by Öncan et al. in [30]. A solution is encoded as an integer array of size k, where each element corresponds to a node with color $c_i \in C$. The algorithmic outline of a typical TS is given in Figure 6.13.

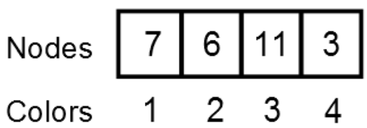

Figure 6.12: An example representation for Tabu Search.

In order to generate a feasible solution, we first select a random node for each color $c_i \in C$. The set of selected nodes is denoted by $A(s)$ where s is the current solution. We calculate the cost of the solution using Algorithm CALCULATECOST

```
input: ACSP-t instance identified with T(V, E) rooted at r
output: Best solution
function TABU SEARCH(T)
    colorselectionFrequency[i] = 0 ∀i ∈ V;
    tabuList ← ∅;
    tabuTenure ← ∅;
    initialSolution ← createInitialSolution(T) ;
    bestSolution ← initialSolution;
    for i = 0 to iterationsize do
        sc ← selectColors();
        update colorselectionFrequency for all colors in sc;
        bestCandidate ← neighborhoodSearch(T,sc,initialSolution);
        if cost(bestCandidate) < cost(bestSolution) then;
            bestSolution ← bestCandidate;
        end if
        initialSolution ← bestCandidate;
        update tabulist;
        update tabuTenure;
    end for
end function
```

Figure 6.13: Tabu Search Algorithm

in Figure 6.9.

After creating an initial solution, we continue with the iterative neighborhood search. In each iteration, we look for a lower cost solution using tabu mechanism. For neighborhood search, we restrict the search to a neighborhood with c colors, denoted by $N^c(s)$. First, we select as many as c colors for searching neighborhood. To overcome repeatedly selecting the same colors, we select the colors based on their rate of selection. If a color has a lower selection rate, it has a higher chance for selection. For each color $c_i \in C$, we assign an attribute δ_{c_i}, which denotes the number of times color c_i has been selected. All such attributes are initially set to zero. When a color $c_i \in C$ is selected δ_{c_i} is increased by one. For each color, we then calculate its selection rate, which is equal to $\frac{1}{\delta_{c_i}}$.

Neighborhood search follows the color selection. For each selected color, we substitute in a new node with the same color in the solution, so that we get a new neighbor solution $\overline{s}$ in $N^c(s)$. We consider all such neighbors. Next, we discuss different methods for the neighborhood search.

Neighborhood Search I for Color Encoding

In the first version for neighborhood search called Tabu, we employ $N^3(s)$. We consider all $(|c_p| - 1) \times (|c_q| - 1) \times (|c_r| - 1)$ solutions as possible neighbors where c_p, c_q, and c_r are different colors, and $|c_i|$ where $i \in \{p, q, r\}$ denotes the number of nodes with color c_i. Cost of the candidate solution is calculated with Algorithm CALCULATECOST in Figure 6.9. Neighbors are evaluated, and put into a candidate list if a replaced node is either not in the tabu list, or the node is in the tabu list, but it meets the aspiration criteria. The candidate list is initially empty. If a solution is considered as a tabu, and it has a lower cost than the best solution found so far, we override the tabu status, and put the solution into the candidate list. This simple aspiration criteria results a faster runtime performance. After evaluating all neighbors, and forming the candidate list, we select the solution with the lowest cost. If the newly obtained solution is better than the best solution, we designate it as the new best solution. Then, we update the tabu status of modified nodes. If the modified node is not in the tabu list, we add it into the tabu list.

Neighborhood Search II for Color Encoding

The second search method, called Tabu Visit Frequency (Tabu-VF) is the same as the one used by Öncan et al. [30] for *GMST* problem. In this method, a new attribute α_i is associated with each node $i \in V$, corresponding to the aspiration level of that node. Initially, α_i is set to $cost(s)$, the cost of solution, if $i \in A(s)$, and otherwise to infinity. Another new attribute τ_i is also associated with each node $i \in V$, corresponding to the visit frequency, which is the number of times a node has been selected as part of a solution. Initially, τ_i is set to 1 if $i \in A(s)$, and otherwise to 0.

We obtain the neighbors by selecting colors, and replacing nodes in the solution. A node is considered in forming a new neighbor, if it is either not in the tabu list, or meets the aspiration criteria. In this version, the aspiration criterion is met if the cost of a neighbor solution $\overline{s}$ is lower than one of the aspiration levels α_i such that $i \in A(s)$. We use a penalty function to penalize frequently visited vertices of the tree. The penalized cost function is defined in [30] as follows:

$$p(\overline{s}) = \begin{cases} cost(\overline{s}) + \beta cost(s)\sqrt{kn} \sum\limits_{i \in A(\overline{s}) \setminus A(s)} \alpha_i/\lambda, & \text{if } cost(\overline{s}) \geq cost(s) \\ cost(\overline{s}), & \text{otherwise.} \end{cases}$$

The penalty is added to a neighbor solution, if its cost is greater than the cost of the current solution. $\sqrt{kn}$ is used to compensate for the instance size, where k is the number of colors, and n is the number of nodes. β is used as a factor that adjust intensity of diversification, and λ is the iteration count. Finally, considering all solutions in the Candidate list, we return the solution with the minimum penalized cost in it.

In Tabu-VF, for neighborhood search, we use $N^1(s)$, $N^2(s)$ and $N^3(s)$. In order to control the increased computational demand, we propose a hybrid neighborhood search that uses all three of them. We iteratively increase the neighborhood size until a better solution is found. First, Tabu search starts to iterate in $N^1(s)$ neighborhoods. When it doesn't give us a better solution for a predetermined number of iterations, we switch to $N^2(s)$ neighborhood, and then, to $N^3(s)$, and then back to $N^1(s)$ in a cyclic order. After evaluating all neighbor solutions, we get the solution with the lowest cost as the current solution. If the newly obtained solution is better than the best solution, we designate it as the new best. Then, we update tabu status, increase α_i by 1 for the modified nodes, and for each $i \in A(s)$, update τ_i to $min\{\tau_i, cost(s)\}$.

Tabu-VF is iterated for a fixed number of iterations dictated by the iteration count. When the iterations are over, the best solution found is returned, and the path is formed using Algorithm formPath in Figure 6.5.

Chapter 7

Experimental Study

In this chapter, we present experimental results for the heuristic solutions proposed, namely LP-relaxation, genetic algorithm, and tabu search based heuristics. All algorithms are implemented using C++ on a computer with an Intel i7 2.79 GHz CPU, 8GB 1333 MHz DDR 3 RAM, and running on Windows* 7 operating system. For ILP, and LP solutions, IBM ILOG CPLEX Optimizer [31] is used. We conduct our experiments with several types of datasets generated. The details of the datasets are presented in Section 7.1.

The rest of this chapter is organized as follows. In Section 7.2, we present the parameters used in metaheuristics and how they are chosen. Finally in Section 7.3, we report the results of the experiments conducted.

7.1 Datasets

We conduct our experiments on several types of trees with various weight distributions, and bushiness to see how the proposed algorithms behave on these trees. The trees used in our experiments are classified based on two criteria excluding the number of nodes, and colors: bushiness, and edge weight distribution. Bushiness of trees depends on the average branching factor, and has an impact on the height of the tree generated. This parameter can be set in 3 different ways:

1) Random: Trees in which the nodes are distributed randomly.

2) Shallow: Trees which has a relatively small height, and high average branching factor.

3) Deep: Trees which are tall, and have a low average branching factor.

*Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

For each bushiness level, we also use four different types of edge weight distributions that are:

1) randomly distributed edge weights,

2) all edge weights are set to one,

3) weights decreasing from the root to the leaves,

4) weights increasing from the root to the leaves.

The complete list of parameters used in generating different types of trees are given in Table 7.1.

Parameter	**Description**	**Values**
n	number of nodes	211-1111
k	number of colors	26-556
b	bushiness type	S (Shallow), D (Deep), R (Random)
w	edge weight distribution	Random (R), 1 (All 1), I (Increasing), D (Decreasing)
bf	branching factor	2-20
h	height of the tree	2-10

Table 7.1: The complete list of parameters used in generating different types of trees.

Each specific type of a tree used throughout the experiments is labeled with a string of the form $nV_1kV_2bV_3bfV_4wV_5$, where V_i with $i \in \{1..6\}$ are the corresponding values for the parameters right next to them. The possible values for each parameter are specified in the third column of Table 7.1.

b	**bf**	**Description**
R	5-20	Trees are created randomly. Branching factor changes from node to node. Tree height is dynamic.
D	2	Branching factor is low. Trees are balanced.
S	9-20	Branching factor is high. Trees are balanced.

Table 7.2: The parameters indirectly set for each bushiness type.

For each bushiness type, the parameters indirectly determined, and their descriptions are given in the first, and third columns of Table 7.2 respectively. In the second column of the table, branching factors which compatible with the respective bushiness type dictated by this setting are shown, which also have an impact on height of the tree.

W	Edge Weight	Description
R	1-10	Weights are randomly distributed among the edges.
1	1	All edge weights are set to one.
D	1-10	Starting from level 0 with edge weight 10, weights decrease gradually.
I	4-10	Starting from level 0 with edge weight 1, weights increase gradually.

Table 7.3: Weight types, and the values of the corresponding parameters along with their description.

In Table 7.3, weight types are shown in the first column. Parameter values along with their description are given in the second, and the third columns respectively.

Furthermore, for each tree type generated, we have three different n/k ratios used to see the impact on the quality of the solution obtained by the heuristics. The ratio can take on the values 2, 4, and 10 throughout the experiments.

7.2 Parameter Tuning

Parameter tuning is an important part of metaheuristic algorithms to get good solutions. In this section, we present algorithm specific parameters used in metaheuristic algorithms, and decide on their values. In Section 7.2.1, we explain meta-parameters, and perform tuning for Genetic Algorithm, while in Section 7.2.2, we do it for Tabu Search.

7.2.1 Parameter Tuning for Genetic Algorithm

In this section, we present results of the experiments conducted for selecting the values of the parameters used in Genetic Algorithm. GA-path takes at least 2352 seconds to generate result even in the smallest tree instance which has 255 nodes and 26 colors, therefore is not included in experiments. In GA-color, we use iteration size, population size, crossover rate, and mutation rate, which have a huge impact on the quality of the solutions obtained. For *ACSP-t*, we conduct experiments to decide on the values of these parameters on *random trees* with randomly distributed weights. First, we conduct experiments for iteration size. We keep the values of other parameters fixed at population size = 500, crossover rate = 0.5, and mutation rate = 0.1. Then, we perform tests on all tree instances with iteration sizes 1000, 2000, 3000, 4000, 5000, 10000, 20000, 30000, 40000, and 50000. As seen in Table 7.4, we get 6 of the best results with 30000, and 50000 iterations, 5 with 40000 iterations, 2 with 20000 iterations, and 1 with 10000 iterations. Best results are obtained with 30000, and 50000 iterations. When we compare the runtimes of those two iteration sizes, as seen in Table 7.5, 50000 iterations case gives results higher by a factor of 1.75 than the 30000 iterations case on the average. As iteration size increases, runtime increases. Hence we select 30000 as the iteration size to be used throughout the experiments.

	Iteratin Size									
Instances	1000	2000	3000	4000	5000	10000	20000	30000	40000	50000
n255k26bRbf5wR	347	277	266	189	183	178	179	**177**	181	179
n255k64bRbf5wR	901	782	677	634	588	552	548	**543**	554	555
n255k128bRbf5wR	1445	1329	1257	1191	1166	1140	1138	**1134**	**1134**	1138
n421k43bRbf7wR	649	527	447	384	349	**298**	304	**298**	303	302
n421k106bRbf7wR	1482	1330	1193	1085	995	852	**832**	836	833	836
n421k211bRbf7wR	2545	2376	2248	2142	2064	1948	1944	**1941**	1945	1946
n511k52bRbf7wR	945	769	653	569	515	428	423	**419**	423	424
n511k128bRbf7wR	1909	1714	1531	1371	1281	1029	**1010**	1018	**1010**	1014
n511k256bRbf7wR	3025	2849	2704	2552	2459	2265	2241	**2240**	2247	2243
n820k82bRbf8wR	1590	1384	1217	1038	935	646	625	632	613	**612**
n820k205bRbf8wR	3342	3093	2837	2615	2460	1892	1764	1758	1758	**1754**
n820k410bRbf8wR	5471	5242	4961	4748	4748	4131	4037	4019	4037	**4009**
n1023k103bRbf15wR	2097	1868	1602	1438	1280	878	798	798	791	**772**
n1023k256bRbf15wR	4328	3999	3688	3533	3266	2558	2280	2279	2277	**2273**
n1023k512bRbf15wR	6937	6715	6434	6205	6005	5321	5059	5049	**5041**	5071
n1111k112bRbf20wR	2302	2058	1822	1564	1442	933	**840**	850	**840**	851
n1111k278bRbf20wR	4573	4319	4020	3768	3566	2800	2472	2467	2487	**2462**
n1111k556bRbf20wR	7778	7413	7158	6913	6636	5863	5580	5580	**5578**	5582

Table 7.4: Average path cost for iteration size tuning in GA-color on random trees with randomly distributed edge weights. The best results are given in bold.

	Iteration Size									
Instances	1000	2000	3000	4000	5000	10000	20000	30000	40000	50000
n255k26bRbf5wR	0.483	0.982	1.453	1.921	2.365	5.381	11.245	19.352	26.417	33.452
n255k64bRbf5wR	0.671	1.331	1.953	2.581	3.221	6.772	15.16	23.32	31.56	40.53
n255k128bRbf5wR	0.954	1.784	2.631	3.462	4.336	8.961	19.384	29.635	40.811	51.358
n421k43bRbf7wR	0.672	1.261	1.851	2.472	3.057	6.165	14.471	22.152	30.424	38.145
n421k106bRbf7wR	0.951	1.797	2.613	3.455	4.218	8.253	18.422	28.954	39.796	49.894
n421k211bRbf7wR	1.279	2.418	3.526	4.649	5.772	11.373	25.132	38.033	52.125	65.707
n511k52bRbf7wR	0.671	1.264	1.857	2.433	3.057	6.147	13.931	22.106	29.531	38.189
n511k128bRbf7wR	0.967	1.794	2.636	3.479	4.306	8.486	18.331	29.281	39.546	50.435
n511k256bRbf7wR	1.435	2.683	3.931	5.211	6.459	12.777	26.384	46.124	57.096	72.182
n820k82bRbf8wR	0.796	1.498	2.184	2.871	3.588	6.869	14.569	24.264	33.341	42.541
n820k205bRbf8wR	1.279	2.371	3.464	4.555	5.633	11.154	23.042	36.566	49.452	63.398
n820k410bRbf8wR	1.435	2.621	3.791	4.961	4.945	11.981	24.538	39.415	53.149	67.408
n1023k103bRbf15wR	0.795	1.445	2.093	2.766	3.417	6.682	14.043	23.218	31.903	54.529
n1023k256bRbf15wR	1.256	2.299	3.328	4.381	5.404	10.611	31.433	34.374	46.926	60.116
n1023k512bRbf15wR	1.972	3.633	5.282	6.942	8.583	16.823	33.891	53.245	72.288	91.969
n1111k112bRbf20wR	0.827	1.492	2.177	2.862	3.544	6.872	14.595	23.799	33.028	41.452
n1111k278bRbf20wR	1.022	1.883	2.731	3.597	4.445	8.698	17.378	28.581	39.831	50.533
n1111k556bRbf20wR	1.688	3.013	4.361	5.673	7.001	13.636	27.353	43.363	58.919	75.491

Table 7.5: Average runtime for iteration size tuning in GA-color on random trees with randomly distributed edge weights.

After tuning iteration size, we conduct experiments for population size. We fix iteration size to 30000, crossover rate to 0.5, and mutation rate to 0.1, and we carry out the experiments on random trees with random weights for population size values 100, 250, 500, 1000, 2000, 3000, 4000, and 5000. As seen in Table 7.6, we get 11 of the best results with a population size of 1000, 7 with population size 500, 1 with population size 250, and 1 with population size 2000. Thus, we select the population size as 1000.

	Population Size							
Instances	100	250	500	1000	2000	3000	4000	5000
n255k26bRbf5wR	189	176	**177**	178	179	179	187	204
n255k64bRbf5wR	564	546	**543**	544	547	567	611	658
n255k128bRbf5wR	1148	1137	1134	**1132**	1137	1153	1187	1228
n421k43bRbf7wR	324	297	298	297	**294**	326	375	416
n421k106bRbf7wR	892	**832**	836	**832**	862	958	1066	1144
n421k211bRbf7wR	1969	1946	**1941**	1944	1958	2051	2136	2204
n511k52bRbf7wR	444	426	**419**	**419**	428	477	560	626
n511k128bRbf7wR	1081	1039	1018	**1011**	1077	1238	1388	1481
n511k256bRbf7wR	2280	2254	2240	**2234**	2302	2424	2548	2637
n820k82bRbf8wR	720	634	632	**601**	710	876	1037	1145
n820k205bRbf8wR	1988	1826	1758	**1753**	2047	2401	2617	2788
n820k410bRbf8wR	4166	4056	4019	**4001**	4271	4563	4767	4937
n1023k103bRbf15wR	1027	886	798	**774**	936	1229	1419	1564
n1023k256bRbf15wR	2709	2395	2279	**2276**	2791	3121	3398	3512
n1023k512bRbf15wR	5344	5134	**5049**	5056	5547	5946	6197	6316
n1111k112bRbf20wR	1162	900	850	**813**	1050	1362	1574	1758
n1111k278bRbf20wR	2851	2592	**2467**	2475	3079	3474	3769	3967
n1111k556bRbf20wR	5940	5643	**5580**	5587	6153	6600	6866	7055

Table 7.6: Average path cost for population size tuning in GA-color on random trees with randomly distributed edge weights. The best results are given in bold.

Then, we conduct experiments for tuning the crossover rate. We fix iteration size at 30000, population size at 1000, and mutation rate to 0.1. We run experiments on random trees with randomly distributed weights for crossover rates ranging from 0.1 through 0.9 in increments of 0.1. As seen in Table 7.7, we obtain 10 of the best solutions with crossover rate 0.6, 5 with 0.5, 2 with 0.8, and 1 for each of 0.3, 0.4, and 0.7. We selected the crossover rate as 0.6.

	Crossover Rate								
Instances	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
n255k26bRbf5wR	175	177	175	178	179	**173**	177	174	178
n255k64bRbf5wR	559	549	**543**	545	544	544	5446	**543**	565
n255k128bRbf5wR	1146	1136	1137	1134	**1133**	1136	1134	**1133**	1137
n421k43bRbf7wR	312	309	298	297	299	**296**	298	302	316
n421k106bRbf7wR	856	839	847	836	834	829	**826**	836	875
n421k211bRbf7wR	1954	1948	1948	1944	1943	**1941**	1944	1949	1965
n511k52bRbf7wR	436	434	428	431	**422**	419	423	426	442
n511k128bRbf7wR	1058	1023	1016	**1012**	1016	1014	**1012**	1030	1072
n511k256bRbf7wR	2259	2251	2243	2242	2242	**2241**	2245	2242	2275
n820k82bRbf8wR	695	642	621	625	629	**601**	628	650	703
n820k205bRbf8wR	1919	1801	1774	1763	**1754**	1763	1775	1806	1983
n820k410bRbf8wR	4095	4046	4012	4030	4018	**4005**	4026	4050	4141
n1023k103bRbf15wR	938	831	805	**778**	796	787	788	868	1001
n1023k256bRbf15wR	2546	2352	2284	2289	**2279**	2283	2304	2347	2562
n1023k512bRbf15wR	5209	5083	5068	5067	5067	**5062**	5085	5090	5315
n1111k112bRbf20wR	1007	907	872	840	**825**	851	858	898	1046
n1111k278bRbf20wR	2710	2505	2507	2483	**2458**	2465	2505	2559	2851
n1111k556bRbf20wR	5702	5631	5583	5595	5592	**5586**	5598	5636	5783

Table 7.7: Average path cost for crossover rate tuning in GA-color on random trees with randomly distributed edge weights. The best results are given in bold.

Finally, we conduct experiments for mutation rate on random trees with randomly distributed edge weights. We fix iteration size to 30000, population size to 1000, and crossover rate to 0.6. Experiments are conducted for mutation rates in the interval [0.1, 0.9] in increments of 0.1. As seen in Table 7.8, we get the best results between 0.1 and 0.8. We get 8 of the best results with mutation rate 0.5, 3 with each of 0.4, 0.6, and 0.7, 2 with each of 0.3, and 0.2, and 1 with each of 0.1, and 0.2. Therefore we set the mutation rate to 0.5.

	Mutation Rate								
Instances	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
n255k26bRbf5wR	179	179	177	178	**175**	177	176	180	177
n255k64bRbf5wR	546	548	541	542	546	542	544	**540**	551
n255k128bRbf5wR	1132	1139	1132	**1131**	1133	1136	1135	**1131**	1136
n421k43bRbf7wR	298	297	295	296	298	298	**292**	296	295
n421k106bRbf7wR	**829**	838	840	830	833	831	832	833	842
n421k211bRbf7wR	1945	1945	1947	1942	1943	**1939**	1941	1945	1946
n511k52bRbf7wR	422	**416**	421	**416**	418	419	418	421	422
n511k128bRbf7wR	1017	1008	1013	1015	**1006**	**1006**	**1006**	1017	1035
n511k256bRbf7wR	2248	2247	**2238**	2239	2240	2239	2239	2242	2265
n820k82bRbf8wR	627	619	611	610	**605**	609	**605**	608	641
n820k205bRbf8wR	1753	1755	**1746**	1767	1775	1753	1772	1848	1932
n820k410bRbf8wR	4021	4035	4021	4025	**4011**	4017	4032	4085	4170
n1023k103bRbf15wR	785	798	806	786	**769**	783	778	804	877
n1023k256bRbf15wR	2273	2271	2263	2270	**2258**	2307	2340	2451	2586
n1023k512bRbf15wR	5052	5059	5055	5045	**5037**	5054	5099	5195	5366
n1111k112bRbf20wR	842	855	834	837	852	**826**	828	856	902
n1111k278bRbf20wR	2482	2458	2460	2462	**2441**	2484	2526	2623	2817
n1111k556bRbf20wR	5592	5595	5568	**5559**	5588	5595	5657	5780	5961

Table 7.8: Average path cost for mutation rate tuning in GA-color on random trees with randomly distributed edge weights. The best results are given in bold.

7.2.2 Parameter Tuning for Tabu Search Algorithm

For tabu search algorithms, the iteration size, and the tabu tenure are the parameters to be fine-tuned for the quality of the solutions. Tabu and Tabu-VF give similar results for parameter tuning. Hence we only present the results of the experiments for Tabu. First, we fix the tabu tenure as 10, and conduct experiments on random trees with randomly distributed edge weights. We perform tests with iteration sizes 1000, 2000, 3000, 4000, 5000, 10000, 20000, 30000, 40000, and 50000. As seen in Table 7.9, 12 of the best results are obtained with iteration size of 50000, 4 with iteration size of 10000, 2 with iteration size of 30000, and 1 with iteration size of 40000. Hence, we select the iteration size as 50000.

In order to determine the best tabu tenure value for Tabu, we, then fix iteration size to 50000, and conduct experiments on random trees with randomly distributed edge weights. Tests are performed with tabu tenures 1, 5, 10, 15, 20, 25, 30, 40, and 50. As seen in Table 7.10 we obtain 8 of the best results with tabu tenure value 10, 4 of them with 40, 3 with tabu tenure value 20, 2 with 50, and 1 with each of 1, 5, and 15. Thus we set tabu tenure as 10.

	Iteration Size									
Instances	1000	2000	3000	4000	5000	10000	20000	30000	40000	50000
n255k26bRbf5wR	177	191	190	177	176	**171**	179	187	180	179
n255k64bRbf5wR	575	573	554	567	561	563	561	539	542	**539**
n255k128bRbf5wR	1181	1159	1153	1151	1169	**1143**	1151	1152	1146	1144
n421k43bRbf7wR	313	310	312	308	315	307	304	309	305	**301**
n421k106bRbf7wR	883	867	848	841	837	839	835	845	828	**820**
n421k211bRbf7wR	2002	1994	2001	1988	1979	1983	1987	1964	1960	**1950**
n511k52bRbf7wR	448	421	440	426	435	**412**	431	417	414	415
n511k128bRbf7wR	1113	1079	1072	1067	1069	1050	1027	1050	1026	**1015**
n511k256bRbf7wR	2331	2287	2302	2272	2286	2276	2272	2268	2260	**2245**
n820k82bRbf8wR	639	612	615	636	641	623	626	619	620	**603**
n820k205bRbf8wR	1846	1851	1837	1847	1821	1830	1803	1756	**1750**	1763
n820k410bRbf8wR	4516	4471	4517	4508	4516	4479	4536	**4441**	4455	4455
n1023k103bRbf15wR	850	841	838	802	801	796	809	797	785	**778**
n1023k256bRbf15wR	2437	2421	2450	2441	2381	2331	2334	2318	2311	**2293**
n1023k512bRbf15wR	6107	6104	6092	6067	6078	6089	6089	**5978**	6113	6039
n1111k112bRbf20wR	938	938	914	856	873	**831**	840	863	861	**831**
n1111k278bRbf20wR	2610	2614	2591	2576	2573	2557	2524	2495	2493	**2479**
n1111k556bRbf20wR	6773	6807	6728	6767	6736	6653	6711	6756	6725	**6651**

Table 7.9: Average path cost for iteration size tuning in Tabu on random trees with randomly distributed edge weights. The best results are given in bold.

	Tabu Tenure								
Instances	1	5	10	15	20	25	30	40	50
n255k26bRbf5wR	**171**	175	179	182	178	182	174	180	178
n255k64bRbf5wR	562	572	539	557	548	567	566	543	**531**
n255k128bRbf5wR	1143	1145	**1134**	1138	1136	1145	1141	1144	1147
n421k43bRbf7wR	299	305	301	311	308	297	300	**296**	306
n421k106bRbf7wR	842	834	**820**	836	829	827	841	**820**	830
n421k211bRbf7wR	1952	1957	**1950**	1969	1955	1959	1955	1967	1960
n511k52bRbf7wR	421	426	**396**	406	428	400	417	397	407
n511k128bRbf7wR	1035	1047	**1015**	1041	1028	1051	1039	**1015**	1040
n511k256bRbf7wR	2258	2247	2245	2247	2247	2263	2267	2258	**2240**
n820k82bRbf8wR	630	625	**603**	612	611	614	618	607	609
n820k205bRbf8wR	1768	**1737**	1750	1766	1776	1764	1753	1772	1776
n820k410bRbf8wR	4449	4440	**4414**	4468	4478	4473	4434	4497	4458
n1023k103bRbf15wR	790	778	778	**772**	785	796	779	779	775
n1023k256bRbf15wR	2326	2314	**2293**	2322	2310	2319	2298	2329	2346
n1023k512bRbf15wR	6061	6026	6039	6088	**6005**	6027	6019	6076	6052
n1111k112bRbf20wR	849	848	**831**	848	**831**	864	841	848	852
n1111k278bRbf20wR	2477	2464	2458	2467	2485	**2456**	2467	2500	2490
n1111k556bRbf20wR	6873	6738	6620	6730	**6592**	6706	6733	6674	6706

Table 7.10: Average path cost for tabu tenure tuning in Tabu on random trees with randomly distributed edge weights. The best results are given in bold.

As a summary, for GA-color, we select iteration size 30000, population size 10000, crossover rate 0.6, and mutation rate 0.5. For Tabu and Tabu-VF, we select iteration size 50000, and tabu tenure 10.

7.3 Experimental Results

In Section 7.3.1, we present the results for random trees. In Section 7.3.2, we present the results for shallow trees, and in Section 7.3.3, we present the results for deep trees.

7.3.1 Results for Random Trees

In Section 7.3.1.1, we present the results of *random trees* with *random weights*, and the results when the weights are all equal to one are presented in Section 7.3.1.2

7.3.1.1 Random Trees with Randomly Distributed Weights

In this section, we show, and compare, the results of the proposed heuristics on *random trees* with randomly distributed weights. These trees are generated as described in Section 7.1. Tests are repeated 10 times for each metaheuristic algorithm. As seen in Table 7.11, we obtain the optimal values using ILP. LP-oneshot gives us values that are, on the average, within a factor of 2.3, 1.53, and 1.18 of the optimal when n/k ratios are 10, 4, and 2 respectively. As n/k ratio increases, the solution by LP-oneshot moves away from the optimal. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.71, 1.39, and 1.12 when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.11, GA-color returns solution within a factor of 1.07 of the optimal, while GA-path, Tabu, and Tabu-VF can provide solutions that are away by a factor of 1.18, 1.09, and 1.12 respectively from the optimal. For the best path cost shown in Table 7.12, GA-color, Tabu, Tabu-VF have values worse off by a factor of 1.053, 1.054, and 1.082 of the optimal respectively. However, as number of colors (k) increases, we get worse results with Tabu, and Tabu-VF specifically on the datasets n820k82bRbf8wR, n1023k103bRbf15wR, and n1111k112bRbf20wR.

When n/k ratio increases, the runtime of ILP increases dramatically on the datasets n820k82bRbf8wR, n1023k103bRbf15wR, n1111k112bRbf20wR, and n11-11k278bRbf20wR as shown in Table 7.13. Average runtime for these four trees is 2617 seconds while the average runtime for the rest of *random trees* with randomly distributed edge weights is only 4.9 seconds. Average runtime for LP-oneshot is 5.94 seconds. As the LP-iterative calls the LP solver multiple times, its runtime is higher as would be expected. GA-color, Tabu, and Tabu-VF have average runtimes of 39 seconds, 28 seconds, and 8 seconds respectively. The average runtime of GA-path is 12273 seconds, hence is only executed once for each instance. Since GA-path has high runtime, it isn't included in the experiment of other tree types. The increase in the number of colors (k), increases the runtime for GA-color due to the crossover operation presented in Section 6.1.1. As a result of the neighborhood search, runtime increases in Tabu, and Tabu-VF as n/k ratio increases.

	ILP	LP-oneshot	LP-iterative	GA-color	GA-path	Tabu	Tabu-VF
n255k26bRbf5wR	155	2.6194	1.6839	1.1613	1.3354	1.1548	1.2194
n255k64bRbf5wR	511	1.5538	1.4716	1.0626	1.1976	1.0548	1.1174
n255k128bRbf5wR	1131	1.1424	1.1034	1.0044	1.1520	1.0115	1.0203
n421k43bRbf7wR	262	2.5725	1.7137	1.1527	1.2938	1.1489	1.2176
n421k106bRbf7wR	780	1.6385	1.4064	1.0718	1.3358	1.0718	1.0731
n421k211bRbf7wR	1933	1.2111	1.1107	1.0052	1.1908	1.0088	1.0207
n511k52bRbf7wR	368	2.3777	1.7554	1.1467	1.5652	1.1467	1.2065
n511k128bRbf7wR	984	1.4533	1.3923	1.0366	1.4115	1.0549	1.0691
n511k256bRbf7wR	2227	1.2003	1.1486	1.0063	1.1508	1.0117	1.0216
n820k82bRbf8wR	557	2.0718	1.6535	1.0987	1.4111	1.0826	1.1149
n820k205bRbf8wR	1659	1.5624	1.4286	1.0645	1.3267	1.0549	1.1121
n820k410bRbf8wR	3954	1.1725	1.1477	1.0167	1.1765	1.1267	1.1343
n1023k103bRbf15wR	685	2.3620	1.6350	1.1620	NA	1.1460	1.1577
n1023k256bRbf15wR	2157	1.5369	1.3848	1.0552	NA	1.0709	1.1205
n1023k512bRbf15wR	4989	1.1790	1.1189	1.0138	NA	1.2105	1.2129
n1111k112bRbf20wR	750	2.0280	1.8213	1.1320	NA	1.1080	1.1653
n1111k278bRbf20wR	2340	1.4513	1.2872	1.0590	NA	1.0628	1.1120
n1111k556bRbf20wR	5499	1.1879	1.1200	1.0160	NA	1.2199	1.2219

Table 7.11: Average path costs as factor of the optimal solution for random trees with randomly distributed weights.

	ILP	LP-oneshot	LP-iterative	GA-color	GA-path	Tabu	Tabu-VF
n255k26bRbf5wR	155	2.6194	1.6839	1.1548	1.3354	1.1097	1.1032
n255k64bRbf5wR	511	1.5538	1.4716	1.0509	1.1976	1.0313	1.0470
n255k128bRbf5wR	1131	1.1424	1.1034	1.0000	1.1520	1.0000	1.0053
n421k43bRbf7wR	262	2.5725	1.7137	1.1069	1.2938	1.0534	1.1374
n421k106bRbf7wR	780	1.6385	1.4064	1.0526	1.3358	1.0269	1.0218
n421k211bRbf7wR	1933	1.2111	1.1107	1.0021	1.1908	1.0041	1.0072
n511k52bRbf7wR	368	2.3777	1.7554	1.1332	1.5652	1.0000	1.1196
n511k128bRbf7wR	984	1.4533	1.3923	1.0244	1.4115	1.0142	1.0366
n511k256bRbf7wR	2227	1.2003	1.1486	1.0031	1.1508	1.0009	1.0085
n820k82bRbf8wR	557	2.0718	1.6535	1.0610	1.4111	1.0323	1.0467
n820k205bRbf8wR	1659	1.5624	1.4286	1.0536	1.3267	1.0289	1.0573
n820k410bRbf8wR	3954	1.1725	1.1477	1.0086	1.1765	1.1088	1.1179
n1023k103bRbf15wR	685	2.3620	1.6350	1.1255	NA	1.0847	1.0876
n1023k256bRbf15wR	2157	1.5369	1.3848	1.0408	NA	1.0315	1.0890
n1023k512bRbf15wR	4989	1.1790	1.1189	1.0072	NA	1.1812	1.1920
n1111k112bRbf20wR	750	2.0280	1.8213	1.0813	NA	1.0347	1.1107
n1111k278bRbf20wR	2340	1.4513	1.2872	1.0376	NA	1.0436	1.0897
n1111k556bRbf20wR	5499	1.1879	1.1200	1.0105	NA	1.2035	1.2042

Table 7.12: Best path costs as factor of the optimal solution for random trees with randomly distributed weights. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	GA-path	Tabu	Tabu-VF
n255k26bRbf5wR	1.641	1.634	17.321	18.322	2352	19.341	8.432
n255k64bRbf5wR	2.145	1.688	36.598	23.455	5242	3.508	2.541
n255k128bRbf5wR	1.293	1.712	79.658	31.558	16731	1.411	1.495
n421k43bRbf7wR	3.841	3.431	41.278	22.613	2809	33.906	8.112
n421k106bRbf7wR	2.988	3.369	101.774	23.802	8975	8.234	5.854
n421k211bRbf7wR	2.227	3.721	167.013	50.488	15031	3.716	2.341
n511k52bRbf7wR	14.637	8.231	62.587	35.744	5573	57.603	17.718
n511k128bRbf7wR	4.726	5.101	143.692	51.953	8555	10.455	6.715
n511k256bRbf7wR	2.048	5.106	255.504	48.584	20517	6.502	6.513
n820k82bRbf8wR	750.046	7.158	154.392	41.545	6623	101.617	18.757
n820k205bRbf8wR	16.718	7.806	344.059	70.345	16495	18.809	7.577
n820k410bRbf8wR	2.741	7.106	618.323	64.361	38380	7.234	7.318
n1023k103bRbf15wR	3960.33	7.683	248.196	32.643	NA	88.725	20.289
n1023k256bRbf15wR	491.993	8.446	574.221	33.851	NA	13.825	6.007
n1023k512bRbf15wR	4.497	6.602	972.131	52.596	NA	5.218	4.451
n1111k112bRbf20wR	3941	9.023	302.541	29.674	NA	111.567	15.189
n1111k278bRbf20wR	1818.82	9.621	589.667	28.598	NA	14.554	6.069
n1111k556bRbf20wR	4.575	9.588	1034.56	48.421	NA	7.605	6.757

Table 7.13: Average runtimes for random trees with randomly distributed weights.

7.3.1.2 Random Trees with All Weights Set to One

In this section, we show, and compare the results for *random trees* when all the edge weights are equal to one. As seen in Table 7.14 we could obtain the optimal values using ILP except for the datasets n802k82bRbf8w1, n1023k103bRbf15w1, and n1111k112bRbf20w1. For those datasets, we present the LP results instead of optimal. LP-oneshot gives us values that are, on the average within a factor of 1.9, 1.42, and 1.16 of the optimal when n/k ratios are 10, 4, and 2 respectively. As n/k ratio increases, the solution by LP-oneshot moves away from the optimal. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.40, 1.19, and 1.11 of when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.14, GA-color returns solutions within a factor of 1.03 of optimal values, while Tabu, and Tabu-VF give 1.11, and 1.12. For the best path cost shown in Table 7.15, GA-color, Tabu, and Tabu-VF have values within a factor of 1.02, 1.09, and 1.11 of the optimal values respectively. An inspection of Table 7.16 reveals that the average running time for ILP is 8.4 seconds except for the datasets n1023k256bRbf15w1, and n1111k278bRbf20w1. Average runtimes for these two trees are 48268, and 7393 seconds respectively. While LP-oneshot runs in 6.79 seconds, LP-iterative has a run time of 317 seconds on the average. GA-color, Tabu, and Tabu-VF have average runtimes of 39, 26.5, and 8.5 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bRbf5w1	42	1.9524	1.4762	1.0476	1.0476	1.0714
n255k64bRbf5w1	120	1.4333	1.2167	1.0417	1.0417	1.0500
n255k128bRbf5w1	256	1.1406	1.0938	1.0117	1.0234	1.0313
n421k43bRbf8w1	73	2.0274	1.2877	1.0548	1.0685	1.0822
n421k106bRbf8w1	203	1.4877	1.2808	1.0345	1.0591	1.0640
n421k211bRbf8w1	410	1.1780	1.1024	1.0098	1.0268	1.0317
n511k52bRbf8w1	92	1.9783	1.4565	1.0435	1.0652	1.1196
n511k128bRbf8w1	240	1.3583	1.1542	1.0333	1.0500	1.0708
n511k256bRbf8w1	512	1.1563	1.1445	1.0234	1.0352	1.0469
n820k82bRbf8w1	81	3.7037	2.3457	1.9506	2.0000	2.0247
n820k205bRbf8w1	397	1.4282	1.1864	1.0453	1.0630	1.1033
n820k410bRbf8w1	824	1.1335	1.1092	1.0182	1.0801	1.0947
n1023k103bRbf15w1	104	3.9608	2.5000	2.0294	2.0588	2.0588
n1023k256bRbf15w1	498	1.3815	1.1807	1.0562	1.0703	1.0984
n1023k512bRbf15w1	1032	1.2064	1.1231	1.0213	1.1308	1.1434
n1111k112bRbf20w1	114	3.5946	2.4234	2.0090	2.0180	2.0541
n1111k278bRbf20w1	541	1.4603	1.1811	1.0518	1.0702	1.0887
n1111k556bRbf20w1	1116	1.1756	1.1344	1.0251	1.1577	1.1676

Table 7.14: Average path costs as factor of the optimal solution for random trees with all weights equal to one.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bRbf5w1	42	1.9524	1.4762	1.0238	**1.0000**	1.0238
n255k64bRbf5w1	120	1.4333	1.2167	1.0417	1.0083	1.0250
n255k128bRbf5w1	256	1.1406	1.0938	1.0078	1.0117	1.0234
n421k43bRbf8w1	73	2.0274	1.2877	1.0411	1.0137	1.0411
n421k106bRbf8w1	203	1.4877	1.2808	1.0197	1.0443	1.0394
n421k211bRbf8w1	410	1.1780	1.1024	1.0049	1.0146	1.0146
n511k52bRbf8w1	92	1.9783	1.4565	1.0326	1.0217	1.0217
n511k128bRbf8w1	240	1.3583	1.1542	1.0250	1.0167	1.0333
n511k256bRbf8w1	512	1.1563	1.1445	1.0195	1.0156	1.0391
n820k82bRbf8w1	81	3.7037	2.3457	1.9012	1.9630	1.9630
n820k205bRbf8w1	397	1.4282	1.1864	1.0327	1.0353	1.0680
n820k410bRbf8w1	824	1.1335	1.1092	1.0158	1.0704	1.0777
n1023k103bRbf15w1	104	3.9608	2.5000	2.0000	1.9412	2.0098
n1023k256bRbf15w1	498	1.3815	1.1807	1.0442	1.0542	1.0783
n1023k512bRbf15w1	1032	1.2064	1.1231	1.0165	1.1231	1.1231
n1111k112bRbf20w1	114	3.5946	2.4234	1.9820	1.9550	1.9910
n1111k278bRbf20w1	541	1.4603	1.1811	1.0370	1.0536	1.0702
n1111k556bRbf20w1	1116	1.1756	1.1344	1.0197	1.1452	1.1541

Table 7.15: Best path costs for random trees with all weights equal to one. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bRbf5w1	6.591	3.635	18.607	34.957	29.412	13.592
n255k64bRbf5w1	4.006	1.632	45.678	48.245	5.694	5.781
n255k128bRbf5w1	1.171	1.632	87.783	39.295	2.402	2.139
n421k43bRbf8w1	15.503	6.237	52.293	25.750	38.851	12.756
n421k106bRbf8w1	10.819	4.233	127.487	28.451	6.710	3.651
n421k211bRbf8w1	2.579	4.021	256.484	43.312	3.174	2.846
n511k52bRbf8w1	23.692	4.541	99.524	28.489	47.182	12.759
n511k128bRbf8w1	10.751	7.385	230.505	38.414	8.333	4.127
n511k256bRbf8w1	2.968	4.578	38.234	41.985	3.650	3.991
n820k82bRbf8w1	2.456	6.839	254.953	21.997	73.859	13.743
n820k205bRbf8w1	281.860	7.625	573.437	50.517	14.117	7.983
n820k410bRbf8w1	18.353	8.129	952.115	54.513	5.124	4.037
n1023k103bRbf15w1	6.671	7.204	376.979	36.739	110.652	22.815
n1023k256bRbf15w1	48268	13.512	718.946	48.352	18.315	8.124
n1023k512bRbf15w1	20.251	7.288	1095.26	63.751	7.002	5.331
n1111k112bRbf20w1	7.001	14.493	428.337	24.862	84.623	20.541
n1111k278bRbf20w1	7393.44	9.253	822.434	28.615	14.721	6.011
n1111k556bRbf20w1	8.252	10.026	1224.04	44.642	4.571	4.322

Table 7.16: Average runtimes for random trees with all weights equal to one.

7.3.2 Results for Shallow Trees

In this section, we present the results for shallow trees. In Section 7.3.2.1, we present the results for *shallow trees* with randomly distributed edge weights. In Section 7.3.2.2, we present the results for *shallow trees* with all edge weights equal to 1. In Section 7.3.2.3, we present the results for *shallow trees* when weights are decreasing. And finally in Section 7.3.2.4, we present the results when weights are increasing.

7.3.2.1 Shallow Trees with Randomly Distributed Weights

In this section, we present, and compare the results of the proposed heuristics on *shallow trees* when the weights of the edges are distributed randomly. As seen in Table 7.17, we obtain the optimal values using ILP for all trees. LP-oneshot obtains solutions that are on the average within a factor of 1.4, 1.09, and 1.006 of the optimal when n/k ratios are 10, 4, and 2 respectively. As n/k ratio increases, the solution by LP-oneshot moves away from the optimal. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.4, 1.09, and 1.003 when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.17, GA-color returns solutions within a factor of 1.02 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away by a factor of 1.06, and 1.07 respectively from the optimal on the average. For the best path cost shown in Table 7.18, GA-color, Tabu, and Tabu-VF have values worse off by a factor of 1.014, 1.038, and 1.051 of the optimal respectively. As presented in Table 7.19, the average runtime is 5.1 seconds for ILP. LP-oneshot has a runtime of 5.93 seconds while LP-iterative has an average runtime of 349 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 39, 22.7, and 6.6 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wR	152	1.3158	1.3026	1.0395	1.0724	1.1184
n421k106bSbf20wR	573	1.0297	1.0244	1.0087	1.0052	1.0070
n421k211bSbf20wR	1592	1.0031	1.0000	1.0000	1.0000	1.0000
n820k82bSbf9wR	420	1.4643	1.4405	1.0714	1.0619	1.0976
n820k205bSbf9wR	1322	1.1362	1.1362	1.0182	1.0234	1.0386
n820k410bSbf9wR	3382	1.0103	1.0065	1.0009	1.0849	1.0837
n1111k112bSbf10wR	552	1.5018	1.5489	1.0670	1.0888	1.1105
n1111k278bSbf10wR	1849	1.1174	1.1201	1.0319	1.0460	1.0595
n1111k556bSbf10wR	4532	1.0049	1.0033	1.0024	1.1825	1.1845

Table 7.17: Average path costs as factor of the optimal solution for shallow trees with randomly distributed weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wR	152	1.3158	1.3026	1.0132	1.0263	1.0132
n421k106bSbf20wR	573	1.0297	1.0244	1.0035	**1.0000**	1.0070
n421k211bSbf20wR	1592	1.0031	**1.0000**	**1.0000**	**1.0000**	**1.0000**
n820k82bSbf9wR	420	1.4643	1.4405	1.0381	1.0143	1.0595
n820k205bSbf9wR	1322	1.1362	1.1362	1.0045	1.0151	1.0272
n820k410bSbf9wR	3382	1.0103	1.0065	1.0006	1.0716	1.0585
n1111k112bSbf10wR	552	1.5018	1.5489	1.0453	1.0308	1.0815
n1111k278bSbf10wR	1849	1.1174	1.1201	1.0249	1.0249	1.0552
n1111k556bSbf10wR	4532	1.0049	1.0033	1.0013	1.1615	1.1593

Table 7.18: Best path costs as factor of the optimal solution for shallow trees with randomly distributed weights. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wR	2.036	3.244	38.681	23.542	23.455	5.853
n421k106bSbf20wR	1.037	2.765	51.822	43.653	3.856	2.452
n421k211bSbf20wR	1.162	2.766	55.836	40.145	1.765	1.784
n820k82bSbf9wR	10.072	7.375	330.252	35.931	76.133	16.131
n820k205bSbf9wR	3.472	5.840	342.550	45.123	12.933	6.318
n820k410bSbf9wR	2.701	6.581	575.765	62.400	5.710	5.663
n1111k112bSbf10wR	14.803	9.911	351.998	30.567	66.442	12.023
n1111k278bSbf10wR	7.024	7.224	701.217	28.642	11.234	6.051
n1111k556bSbf10wR	3.791	7.709	698.401	44.675	3.476	3.648

Table 7.19: Average runtimes for shallow trees with randomly distributed weights.

7.3.2.2 Shallow Trees with All Weights Set to One

For *shallow trees* when all the edge weights are equal to one, the average cost of the solutions are shown in Table 7.20. We could obtain the optimal values using ILP except for the datasets n820k82bSbf9w1, and n1111k112bSbf10w1. For those datasets, we present the LP results instead of optimal. LP-oneshot operates on the average, within a factor of 1.03, and 1.011 of the optimal when n/k ratios are 4, and 2 respectively. A solution within a feasible amount of time is returned when $n/k = 10$ only for the dataset n421k43bSbf20w1. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.02, and 1.008 when n/k ratios are 4, and 2 respectively. As seen in Table 7.20, GA-color returns solutions within a factor of 1.0005 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away by a factor of 1.013, and 1.019 respectively. For the best path cost shown in Table 7.21, GA-color returns the optimal for all trees while Tabu, and Tabu-VF have values worse off by a factor of 1.007, and 1.011 of the optimal respectively. As presented in Table 7.22, the average runtime is 9.5 seconds for ILP. LP-oneshot has an average runtime of 5.85 seconds while LP-iterative has an average runtime of 336 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 37, 22.1, and 8.12 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20w1	82	1.0244	1.0000	1.0000	1.0000	1.0122
n421k106bSbf20w1	208	1.0192	1.0192	1.0000	1.0096	1.0144
n421k211bSbf20w1	418	1.0096	1.0048	1.0048	1.0048	1.0048
n820k82bSbf9w1	82	2.1358	2.0370	1.9877	1.9877	2.0123
n820k205bSbf9w1	405	1.0444	1.0296	1.0000	1.0074	1.0148
n820k410bSbf9w1	815	1.0172	1.0147	1.0025	1.0221	1.0245
n1111k112bSbf10w1	111	2.1982	2.0270	1.9730	2.0000	1.9910
n1111k278bSbf10w1	551	1.0290	1.0145	1.0000	1.0163	1.0254
n1111k556bSbf10w1	1109	1.0090	1.0054	1.0000	1.0343	1.0388

Table 7.20: Average path costs as factor of the optimal solution for shallow trees with all weights equal to one.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20w1	82	1.0244	**1.0000**	**1.0000**	**1.0000**	**1.0000**
n421k106bSbf20w1	208	1.0192	1.0192	**1.0000**	**1.0000**	**1.0000**
n421k211bSbf20w1	418	1.0096	1.0048	**1.0000**	1.0048	1.0048
n820k82bSbf9w1	82	2.1358	2.0370	1.9877	1.9877	1.9877
n820k205bSbf9w1	405	1.0444	1.0296	**1.0000**	**1.0000**	1.0099
n820k410bSbf9w1	815	1.0172	1.0147	**1.0000**	1.0147	1.0196
n1111k112bSbf10w1	111	2.1982	2.0270	1.9730	1.9730	1.9730
n1111k278bSbf10w1	551	1.0290	1.0145	**1.0000**	1.0073	1.0145
n1111k556bSbf10w1	1109	1.0090	1.0054	**1.0000**	1.0252	1.0325

Table 7.21: Best path costs as factor of the optimal solution for shallow trees with all weights equal to one. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20w1	5.694	3.292	59.403	24.866	25.591	15.554
n421k106bSbf20w1	4.758	2.980	135.023	39.031	5.179	3.713
n421k211bSbf20w1	1.108	2.824	56.737	41.668	2.371	2.590
n820k82bSbf9w1	6.596	7.176	257.926	31.887	58.810	14.540
n820k205bSbf9w1	19.748	6.234	712.413	35.741	9.641	5.039
n820k410bSbf9w1	3.812	6.001	772.628	48.703	4.852	4.711
n1111k112bSbf10w1	7.126	7.996	360.772	32.786	76.705	15.584
n1111k278bSbf10w1	26.341	8.316	1138.25	33.431	11.281	6.942
n1111k556bSbf10w1	5.037	7.894	1261.77	50.435	4.649	4.967

Table 7.22: Average runtimes for shallow trees with all weights equal to one.

7.3.2.3 Shallow Trees with Decreasing Weights

In this section, we show, and compare the results on *shallow trees* when all the edge weights are decreasing from root to the leaves. As presented in Table 7.23, we could obtain the optimal values using ILP except for the datasets n820k82bSbf9wD, n1111k112bSbf10wD, and n1111k278bSbf10wD. For those datasets we give the LP results. LP-oneshot gives us values within a factor of 1.09 of the optimal on the average. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.07. As seen in Table 7.23, GA-color has values worse off by a factor on the average of 1.03 of the optimal, while Tabu, and Tabu-VF have values within a factor of 1.04, and 1.05 respectively. For the best path cost shown in Table 7.24, GA-color, Tabu, and Tabu-VF return solutions within a factor of 1.02, 1.03, and 1.04 of the optimal respectively. As seen in Table 7.25, average runtime is 5.5 seconds for ILP except for n820k205bSbf9wD which takes 142651 seconds to complete. LP-oneshot has an average runtime of 6.34 seconds, while LP-iterative has average runtime of 293 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 30, 20, and 9.3 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wD	520	1.2692	1.2000	1.0596	1.0577	1.1058
n421k106bSbf20wD	1316	1.0973	1.0851	1.0464	1.0729	1.0783
n421k211bSbf20wD	2640	1.0197	1.0133	1.0136	1.0136	1.0136
n820k82bSbf9wD	486	2.6173	2.4527	2.3148	2.3272	2.3292
n820k205bSbf9wD	2604	1.1298	1.0998	1.0476	1.0438	1.0568
n820k205bSbf9wD	5200	1.0465	1.0385	1.0129	1.0352	1.0354
n1111k112bSbf10wD	666	2.5976	2.5766	2.2808	2.2913	2.2958
n1111k278bSbf10wD	2230	1.7516	1.6915	1.6561	1.6623	1.6874
n1111k556bSbf10wD	7024	1.0310	1.0216	1.0142	1.0500	1.0473

Table 7.23: Average path costs as factor of the optimal solution for shallow trees with decreasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wD	520	1.2692	1.2000	1.0462	1.0462	1.0615
n421k106bSbf20wD	1316	1.0973	1.0851	1.0426	1.0547	1.0699
n421k211bSbf20wD	2640	1.0197	1.0133	1.0136	1.0136	1.0136
n820k82bSbf9wD	486	2.6173	2.4527	2.2881	2.2798	2.2305
n820k205bSbf9wD	2604	1.1298	1.0998	1.0445	1.0292	1.0461
n820k205bSbf9wD	5200	1.0465	1.0385	1.0100	1.0285	1.0269
n1111k112bSbf10wD	666	2.5976	2.5766	2.2583	2.2342	2.2402
n1111k278bSbf10wD	2230	1.7516	1.6915	1.6556	1.6502	1.6771
n1111k556bSbf10wD	7024	1.0310	1.0216	1.0125	1.0433	1.0450

Table 7.24: Best path costs as factor of the optimal solution for shallow trees with decreasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wD	6.274	2.711	65.172	32.356	26.482	17.978
n421k106bSbf20wD	5.528	3.653	133.621	43.212	4.957	3.131
n421k211bSbf20wD	1.144	3.173	55.552	45.683	2.886	
n820k82bSbf9wD	5.001	5.483	153.442	23.924	53.044	13.365
n820k205bSbf9wD	142651	7.634	553.068	30.726	7.888	6.438
n820k205bSbf9wD	5.761	6.682	519.719	40.844	2.934	4.812
n1111k112bSbf10wD	10.965	11.287	272.692	24.211	67.462	14.544
n1111k278bSbf10wD	8.245	8.893	1045.22	28.902	10.447	8.734
n1111k556bSbf10wD	9.172	7.489	881.823	46.15	3.444	5.347

Table 7.25: Average runtimes for shallow trees with decreasing weights.

7.3.2.4 Shallow Trees with Increasing Weights

Experimental results are presented in this section for *shallow trees*, when the edge weights are increasing. As shown in Table 7.26, we could obtain the optimal values using ILP except for the dataset n820k82bSbf9wI. For this dataset we give the LP results. LP-oneshot gives us values that are, on the average, within a factor of 1.01, 1.01, and 1.004 of the optimal when n/k ratios are 10, 4, and 2 respectively. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.01, 1.008, and 1.004 when n/k ratios are 10, 4, and 2 respectively. As presented in Table 7.26, GA-color returns solutions within a factor of 1.0003 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away by a factor of 1.009, and 1.011 respectively from the optimal. For the best cost shown in Table 7.27 GA-color returns the optimal for all tree instances while, Tabu, and Tabu-VF have values worse off by a factor of 1.007, and 1.001 of the optimal respectively. Furthermore, Tabu returns the optimal values for 4 different types of trees, while Tabu-VF is superior in only 1 of them. As seen in Table 7.28, average runtime is 20.61 seconds for ILP. LP-oneshot has average runtime of 6.35 seconds, while LP-iterative has an average runtime of 310 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 36.47, 20.9, and 8.06 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wI	278	1.0000	1.0000	1.0000	1.0000	1.0000
n421k106bSbf20wI	900	1.0044	1.0044	1.0000	1.0022	1.0033
n421k211bSbf20wI	1942	1.0010	1.0010	1.0010	1.0010	1.0010
n820k82bSbf9wI	371	1.5768	1.5768	1.5067	1.5067	1.5148
n820k205bSbf9wI	1675	1.0215	1.0143	1.0000	1.0024	1.0107
n820k410bSbf9wI	3709	1.0081	1.0097	1.0016	1.0197	1.0200
n1111k112bSbf10wI	777	1.0309	1.0386	1.0000	1.0000	1.0013
n1111k278bSbf10wI	2337	1.0231	1.0077	1.0000	1.0090	1.0145
n1111k556bSbf10wI	5083	1.0035	1.0035	1.0000	1.0395	1.0427

Table 7.26: Average path costs as factor of the optimal solution for shallow trees with increasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wI	278	**1.0000**	**1.0000**	**1.0000**	**1.0000**	**1.0000**
n421k106bSbf20wI	900	1.0044	1.0044	**1.0000**	**1.0000**	1.0022
n421k211bSbf20wI	1942	1.0010	1.0010	**1.0000**	**1.0010**	1.0010
n820k82bSbf9wI	371	1.5768	1.5768	1.5067	1.5067	1.5067
n820k205bSbf9wI	1675	1.0215	1.0143	**1.0000**	**1.0000**	1.0072
n820k410bSbf9wI	3709	1.0081	1.0097	**1.0000**	1.0135	1.0151
n1111k112bSbf10wI	777	1.0309	1.0386	**1.0000**	**1.0000**	**1.0000**
n1111k278bSbf10wI	2337	1.0231	1.0077	**1.0000**	1.0051	1.0077
n1111k556bSbf10wI	5083	1.0035	1.0035	**1.0000**	1.0315	1.0311

Table 7.27: Best path costs as factor of the optimal solution for shallow trees with increasing weights. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n421k43bSbf20wI	2.587	2.684	82.174	31.124	34.515	18.267
n421k106bSbf20wI	3.907	2.859	158.575	34.415	5.123	3.806
n421k211bSbf20wI	1.123	2.754	63.885	53.214	3.034	3.021
n820k82bSbf9wI	7.362	7.985	150.788	20.124	49.872	12.741
n820k205bSbf9wI	6.967	7.038	460.999	40.875	8.277	4.119
n820k410bSbf9wI	3.851	6.815	469.717	45.131	3.599	2.995
n1111k112bSbf10wI	138.324	10.745	255.284	24.746	69.284	18.523
n1111k278bSbf10wI	8.124	8.367	840.613	33.311	10.727	5.382
n1111k556bSbf10wI	4	7.925	1099.42	45.341	4.044	3.713

Table 7.28: Average runtimes for shallow trees with increasing weights.

7.3.3 Results for Deep Trees

In this section we present the results for *deep trees*. In Section 7.3.3.1, we present the results for deep trees with randomly distributed edge weights. In Section 7.3.3.2, we present the results for *deep trees* with all the edge weights equal to 1. In Section 7.3.3.3, we present the results for *deep trees* when the weights are decreasing from root to the leaves. Finally, in Section 7.3.3.4, we present the results when the weights are increasing.

7.3.3.1 Deep Trees with Randomly Distributed Weights

In this section, we show, and compare the results of the proposed heuristics on *deep trees* when the edge weights are randomly distributed. As seen in Table 7.29, we could obtain the optimal values using ILP except for the dataset n1023k103bDbf2wR. For this dataset we give the LP results. LP-oneshot gives us values that are, on the average, within a factor of 2.5, 1.7, and 1.17 of the optimal when n/k ratios are 10, 4, and 2 respectively. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.82, 1.31, and 1.21 when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.29, GA-color is worse off by a factor of 1.04 of the optimal, while Tabu, and Tabu-VF returns solutions, on the average, within a factor of 1.06, and 1.09 of the optimal respectively. For the best path cost shown in Table 7.30, GA-color, Tabu, and Tabu-VF can obtain values within a factor of 1.018, 1.029, and 1.048 of the optimal respectively. As presented in Table 7.31, average runtime is 7.38 seconds for ILP except for the dataset n1023k256bDbf2wR which takes 9180 seconds. LP-oneshot has an average runtime of 5.23 seconds while LP-iterative has an average runtime of 391 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 37.26, 27.04, and 5.57 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wR	201	3.0149	2.0647	1.0597	1.0995	1.1244
n255k64bDbf2wR	561	1.7398	1.3939	1.0250	1.0285	1.0998
n255k128bDbf2wR	1244	1.2074	1.1270	1.0080	1.0153	1.0217
n511k52bDbf2wR	396	1.9899	1.5404	1.0884	1.0909	1.0934
n511k128bDbf2wR	1150	1.5591	1.2183	1.0426	1.0470	1.0617
n511k256bDbf2wR	2486	1.1581	1.1279	1.0169	1.0257	1.0334
n1023k103bDbf2wR	140	13.9143	11.5071	7.5786	7.4214	7.5429
n1023k256bDbf2wR	2441	1.4928	1.3290	1.0795	1.0782	1.1200
n1023k512bDbf2wR	5357	1.1589	1.1088	1.0162	1.1697	1.1833

Table 7.29: Average path costs as factor of the optimal solution for deep trees with randomly distributed weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wR	201	3.0149	2.0647	**1.0000**	**1.0000**	**1.0000**
n255k64bDbf2wR	561	1.7398	1.3939	1.0160	1.0053	1.0374
n255k128bDbf2wR	1244	1.2074	1.1270	1.0048	1.0032	1.0064
n511k52bDbf2wR	396	1.9899	1.5404	1.0202	**1.0000**	1.0480
n511k128bDbf2wR	1150	1.5591	1.2183	1.0270	1.0270	1.0287
n511k256bDbf2wR	2486	1.1581	1.1279	1.0068	1.0032	1.0193
n1023k103bDbf2wR	140	13.9143	11.5071	7.4500	7.1429	7.1643
n1023k256bDbf2wR	2441	1.4928	1.3290	1.0610	1.0414	1.0901
n1023k512bDbf2wR	5357	1.1589	1.1088	1.0134	1.1572	1.1581

Table 7.30: Best path costs as factor of the optimal solution for deep trees with randomly distributed weights. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wR	2.882	2.162	22.651	21.358	20.816	8.127
n255k64bDbf2wR	2.467	2.066	57.836	23.466	3.590	2.323
n255k128bDbf2wR	0.820	1.786	101.757	31.706	1.543	1.541
n511k52bDbf2wR	17.035	5.314	65.015	20.639	56.129	8.498
n511k128bDbf2wR	13.391	4.572	145.161	29.156	9.517	3.342
n511k256bDbf2wR	1.503	4.555	292.502	28.439	4.134	3.213
n1023k103bDbf2wR	7.793	8.312	479.848	41.761	121.993	13.251
n1023k256bDbf2wR	9180.50	8.959	917.34	47.564	19.235	5.819
n1023k512bDbf2wR	7.036	9.362	1441	91.322	6.474	4.045

Table 7.31: Average runtimes for deep trees with randomly distributed weights.

7.3.3.2 Deep Trees with All Weights Set to One

In this section, the results for *deep trees* when all the edge weights are equal to one are shown, and compared. As seen in Table 7.32, we could obtain the optimal values using ILP except for the datasets n1023k103bDbf2w1, and n1023k256bDbf2w1. For those datasets we give the LP results. LP-oneshot gives us values that are, on the average, within a factor of 2.32, 1.59, and 1.14 of the optimal when n/k ratios are 10, 4, and 2 respectively. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.3, 1.14, and 1.08 when n/k ratios are 10, 4, and 2 respectively. As presented in Table 7.32, GA-color returns solutions within a factor of 1.034 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away from the optimal by a factor of 1.065, and 1.082 respectively. For the best path cost shown in Table 7.33, GA-color, Tabu, and Tabu-VF have values worse off by a factor of 1.017, 1.031, and 1.051 of the optimal respectively. We obtain one of the optimal solutions with GA-color, and 3 of optimal solutions with Tabu. As shown in Table 7.34, the average runtime is 18.15 seconds for ILP. LP-oneshot has an average runtime of 5.14 seconds, while LP-iterative has an average runtime of 255 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 32.28, 21.20, and 6.42 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2w1	45	2.3556	1.4000	1.0444	1.0444	1.0889
n255k64bDbf2w1	121	1.6860	1.1653	1.0331	1.0496	1.0661
n255k128bDbf2w1	257	1.1673	1.0856	1.0078	1.0195	1.0195
n511k52bDbf2w1	98	2.2857	1.2653	1.0714	1.0816	1.1224
n511k128bDbf2w1	254	1.5118	1.1260	1.0433	1.0787	1.0748
n511k256bDbf2w1	520	1.1308	1.0769	1.0192	1.0327	1.0462
n1023k103bDbf2w1	102	4.4118	2.3824	2.1078	2.1471	2.1765
n1023k256bDbf2w1	255	2.9490	2.3569	2.1647	2.2039	2.2431
1023N512C_D1_2d_h9	1061	1.1329	1.0980	1.0207	1.1517	1.1593

Table 7.32: Average path costs as factor of the optimal solution for deep trees with all weights equal to one.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2w1	45	2.3556	1.4000	**1.0000**	**1.0000**	1.0444
n255k64bDbf2w1	121	1.6860	1.1653	1.0165	**1.0000**	**1.0000**
n255k128bDbf2w1	257	1.1673	1.0856	1.0078	**1.0000**	1.0156
n511k52bDbf2w1	98	2.2857	1.2653	1.0408	1.0204	1.0612
n511k128bDbf2w1	254	1.5118	1.1260	1.0315	1.0472	1.0630
n511k256bDbf2w1	520	1.1308	1.0769	1.0115	1.0231	1.0385
n1023k103bDbf2w1	102	4.4118	2.3824	2.0490	2.1078	2.0882
n1023k256bDbf2w1	255	2.9490	2.3569	2.1294	2.1451	2.2000
1023N512C_D1_2d_h9	1061	1.1329	1.0980	1.0170	1.1320	1.1357

Table 7.33: Best path costs as factor of the optimal solution for deep trees with all weights equal to one. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2w1	10.577	1.570	14.045	20.867	19.756	8.533
n255k64bDbf2w1	5.217	1.966	40.891	23.571	3.697	2.449
n255k128bDbf2w1	1.146	1.981	73.971	30.512	1.732	1.638
n511k52bDbf2w1	39.302	4.636	57.207	23.856	40.851	9.314
n511k128bDbf2w1	20.688	4.698	154.707	29.171	6.864	3.775
n511k256bDbf2w1	9.454	5.070	275.352	45.181	3.881	3.511
n1023k103bDbf2w1	7.347	7.966	296.099	27.623	92.895	16.952
n1023k256bDbf2w1	7.145	7.816	679.947	39.912	16.092	6.898
n1023k512bDbf2w1	40.712	10.561	707.155	49.841	5.054	4.727

Table 7.34: Average runtimes for deep trees with all weights equal to one.

7.3.3.3 Deep Trees with Decreasing Weights

This section presents, and compares, the results for *deep trees* when the edge weights are decreasing from root to the leaves. As shown in Table 7.35, we could obtain the optimal values using ILP except for the datasets n1023k103bDbf2wD, and n1023k256bDbf2wD. For those datasets we give the LP results. LP-oneshot gives us values that are, on the average, within a factor of 2.57, 1.74, and 1.31 of the optimal when n/k ratios are 10, 4, and 2 respectively. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.6, 1.36, and 1.15 when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.35, GA-color returns solutions within a factor of 1.098 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away from the optimal by a factor of 1.096, and 1.125 respectively. For the best path cost shown in Table 7.36, GA-color, Tabu, and Tabu-VF have values worse off by a factor of 1.071, 1.060, and 1.073 of the optimal respectively. As shown in Table 7.37, the average runtime is 14.04 seconds for ILP. LP-oneshot has average runtime of 5.81 seconds, while LP-iterative has an average runtime of 161.63. GA-color, Tabu, and Tabu-VF have average runtimes of 39.49, 21.82, and 6.58 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wD	277	2.5162	1.6931	1.1227	1.1047	1.1480
n255k64bDbf2wD	707	1.7383	1.4187	1.0651	1.0636	1.1061
n255k128bDbf2wD	1371	1.3173	1.1648	1.0306	1.0379	1.0365
n511k52bDbf2wD	488	2.6332	1.5082	1.2582	1.1721	1.2541
n511k128bDbf2wD	1230	1.7472	1.3106	1.1244	1.1065	1.1236
n511k256bDbf2wD	2298	1.2977	1.1340	1.0379	1.0461	1.0579
n1023k103bDbf2wD	204	5.8627	5.8627	5.4804	5.0784	5.1324
n1023k256bDbf2wD	512	6.7734	5.2773	4.5059	4.3867	4.5117
n1023k512bDbf2wD	3652	1.3280	1.1632	1.0507	1.1443	1.1528

Table 7.35: Average path costs as factor of the optimal solution for deep trees with decreasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wD	277	2.5162	1.6931	1.0505	1.0289	1.0866
n255k64bDbf2wD	707	1.7383	1.4187	1.0396	1.0226	1.0820
n255k128bDbf2wD	1371	1.3173	1.1648	1.0160	1.0117	1.0058
n511k52bDbf2wD	488	2.6332	1.5082	1.2172	1.1475	1.0984
n511k128bDbf2wD	1230	1.7472	1.3106	1.1057	1.0683	1.0683
n511k256bDbf2wD	2298	1.2977	1.1340	1.0296	1.0226	1.0348
n1023k103bDbf2wD	204	5.8627	5.8627	5.2843	4.7745	4.8725
n1023k256bDbf2wD	512	6.7734	5.2773	4.4336	4.2070	4.4102
n1023k512bDbf2wD	3652	1.3280	1.1632	1.0433	1.1238	1.1369

Table 7.36: Best path costs as factor of the optimal solution for deep trees with decreasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wD	6.145	2.531	12.995	35.464	28.712	7.891
n255k64bDbf2wD	7.984	2.160	30.981	28.119	4.977	2.876
n255k128bDbf2wD	2.102	2.141	109.687	42.432	1.592	1.638
n511k52bDbf2wD	12.891	6.101	41.543	21.639	43.514	8.573
n511k128bDbf2wD	36.372	5.288	186.061	39.766	6.958	3.349
n511k256bDbf2wD	18.581	4.211	356.265	57.765	3.713	3.214
n1023k103bDbf2wD	8.988	9.706	445.844	23.725	89.402	19.202
n1023k256bDbf2wD	10.741	11.502	109.699	25.654	13.326	5.803
n1023k512bDbf2wD	13.953	8.720	1655.38	80.886	4.227	6.724

Table 7.37: Average runtimes for deep trees with decreasing weights.

7.3.3.4 Deep Trees with Increasing Weights

In this section, results for *deep trees* when the edge weights are increasing are presented. As seen in Table 7.38, we could obtain the optimal values using ILP except for the dataset n1023k103bDbf2wI. For this dataset we give the LP results. LP-oneshot gives us values that are, on the average, within a factor of 1.06 of the optimal for all n/k ratios. LP-iterative returns solutions that are, on the average, farther from the optimal by a factor of 1.6, 1.08, and 1.04 when n/k ratios are 10, 4, and 2 respectively. As seen in Table 7.38, GA-color returns solutions within a factor of 1.026 of the optimal, while Tabu, and Tabu-VF can provide solutions that are away by a factor of 1.041, and 1.056 respectively. For the best path cost shown in Table 7.39, GA-color, Tabu, and Tabu-VF have values worse off by a factor of 1.013, 1.025, and 1.033 of the optimal respectively. An inspection of Table 7.40 reveals that the average running time for ILP is 9.85 seconds. LP-oneshot has an average runtime of 5.36 seconds, while LP-iterative has an average runtime of 189.87 seconds. GA-color, Tabu, and Tabu-VF have average runtimes of 30.24, 19.96, and 6.15 seconds respectively.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wI	172	1.0872	1.0814	1.0233	1.0233	1.0581
n255k64bDbf2wI	574	1.0418	1.0418	1.0261	1.0314	1.0366
n255k128bDbf2wI	1400	1.0400	1.0271	1.0000	1.0057	1.0129
n511k52bDbf2wI	462	1.0411	1.1385	1.0368	1.0216	1.0325
n511k128bDbf2wI	1464	1.0820	1.0669	1.0355	1.0321	1.0403
n511k256bDbf2wI	3342	1.0718	1.0437	1.0180	1.0197	1.0311
n1023k103bDbf2wI	600	2.0100	1.9450	1.9383	1.9000	1.9050
n1023k256bDbf2wI	3475	1.0639	1.0829	1.0521	1.0397	1.0645
n1023k512bDbf2wI	7877	1.0764	1.0551	1.0171	1.1603	1.1714

Table 7.38: Average path costs as factor of the optimal solution for deep trees with increasing weights.

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wI	172	1.0872	1.0814	**1.0000**	**1.0000**	1.0116
n255k64bDbf2wI	574	1.0418	1.0418	1.0174	1.0139	**1.0000**
n255k128bDbf2wI	1400	1.0400	1.0271	**1.0000**	**1.0000**	**1.0000**
n511k52bDbf2wI	462	1.0411	1.1385	1.0216	**1.0000**	1.0173
n511k128bDbf2wI	1464	1.0820	1.0669	1.0164	1.0178	1.0109
n511k256bDbf2wI	3342	1.0718	1.0437	1.0132	1.0108	1.0197
n1023k103bDbf2wI	600	2.0100	1.9450	1.8800	1.8550	1.8617
n1023k256bDbf2wI	3475	1.0639	1.0829	1.0265	1.0224	1.0541
n1023k512bDbf2wI	7877	1.0764	1.0551	1.0135	1.1353	1.1582

Table 7.39: Best path costs as factor of the optimal solution for deep trees with increasing weights. The optimal results are given in bold

	ILP	LP-oneshot	LP-iterative	GA-color	Tabu	Tabu-VF
n255k26bDbf2wI	3.289	2.502	25.319	24.261	19.932	8.155
n255k64bDbf2wI	2.915	2.218	38.865	25.727	3.888	2.591
n255k128bDbf2wI	1.710	1.934	96.296	26.645	1.467	2.512
n511k52bDbf2wI	16.251	6.921	71.713	21.178	45.839	8.751
n511k128bDbf2wI	12.567	10.495	138.435	30.972	7.745	3.495
n511k256bDbf2wI	4.154	4.352	371.571	35.492	2.949	5.335
n1023k103bDbf2wI	6.751	7.923	252.549	27.788	80.831	12.402
n1023k256bDbf2wI	13.564	5.093	524.277	26.641	12.725	5.319
n1023k512bDbf2wI	27.128	6.848	1027.38	53.491	4.305	6.873

Table 7.40: Average runtimes for deep trees with increasing weights.

7.3.4 Discussion

In this section, we show which algorithm performs best with which type of trees. In Table 7.41 number of best results of heuristics are given for each type of tree. In random trees, both GA-color, and Tabu can be used. In shallow trees, GA-color outperforms all the other heuristics except for decreasing weights from root to the leaves. In deep trees, Tabu perform best except for the trees with all weights are equal to 1.

b/w	**R**	**1**	**D**	**I**
R	Tabu: 11 GA-color: 8	GA-color:10 Tabu: 8	-	-
S	GA-color: 6 Tabu: 5	GA-color: 9 Tabu: 5 Tabu-VF: 4	Tabu: 6 GA-color: 5	GA-color: 9 Tabu: 5
D	Tabu: 8 GA-color: 3	GA-color: 6 Tabu: 4	Tabu: 6 Tabu-VF: 3	Tabu: 6 Tabu-VF: 3 GA-color: 3

Table 7.41: Number of best results for heuristic algorithms

Chapter 8

Conclusion

In this chapter, we present concluding remarks, and point at the future research directions.

8.1 Summary

We introduce the problem informally, and give some motivational scenarios. Then, we present the contribution of the thesis, and we give organization of the thesis. We examine similar problems, and compare them with *ACSP-t*. We define the problem formally. Then, we prove that *ACSP-t* problem is NP-Hard, using a reduction from *HSP*. We show that there isn't any constant factor approximation algorithm for the problem. Based on this formulation, an LP-relaxation is obtained for *ACSP-t*. The LP-relaxation is then exploited to propose heuristic algorithms for *ACSP-t*. We present metaheuristic algorithms for *ACSP-t* problem based on Genetic Algorithm, and Tabu Search. We present experimental results for the proposed heuristics in an effort to compare them.

8.2 Future Work

For our future research, we would like to develop more sophisticated heuristic methods based on LP-relaxation might be developed to obtain better results. Other metaheuristic methods might be explored for *ACSP-t*. Genetic algorithm using path encoding might be improved using more sophisticated methods to get solutions within a reasonable runtime.

Chapter 9

Bibliography

[1] Y. C. Bilge, D. Çağatay, B. Genç, M. Sarı, H. Akcan, and C. Evrendilek, "All Colors Shortest Path Problem," *ArXiv e-prints*, July 2015, 1507.06865.

[2] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness.* San Francisco, LA: Freeman, 1979.

[3] G. B. Dantzig, *Linear programming and extensions.* Princeton University Press, 1998.

[4] L. G. Khachiyan, "Polynomial algorithms in linear programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 20, no. 1, pp. 53–72, 1980.

[5] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of Computing*, pp. 302–311, ACM, 1984.

[6] M. R. Garey and D. S. Johnson, "Strong NP-completeness results: Motivation, examples, and implications," *Journal of the ACM (JACM)*, vol. 25, no. 3, pp. 499–508, 1978.

[7] Y.-S. Myung, C.-H. Lee, and D.-W. Tcha, "On the generalized minimum spanning tree problem," *Networks*, vol. 26, no. 4, pp. 231–241, 1995.

[8] P. C. Pop, "New models of the generalized minimum spanning tree problem," *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 2, pp. 153–166, 2004.

[9] C. Feremans, M. Labbé, and G. Laporte, "A comparative analysis of several formulations for the generalized minimum spanning tree problem," *Networks*, vol. 39, no. 1, pp. 29–34, 2002.

[10] P. C. Pop, W. Kern, and G. Still, "A new relaxation method for the generalized minimum spanning tree problem," *European Journal of Operational Research*, vol. 170, no. 3, pp. 900–908, 2006.

[11] P. C. Pop, W. Kern, and G. Still, "An approximation algorithm for the generalized minimum spanning tree problem with bounded cluster size," Tech. Rep. 1577, Department of Applied Mathematics, University of Twente, 2001.

[12] P. C. Pop, *The generalized minimum spanning tree problem*. PhD thesis, University of Twente, 2002.

[13] M. Dror, M. Haouari, and J. Chaouachi, "Generalized spanning trees," *European Journal of Operational Research*, vol. 120, no. 3, pp. 583–592, 2000.

[14] G. Reich and P. Widmayer, "Beyond steiner's problem: A VLSI oriented generalization," in *Graph-theoretic Concepts in Computer Science*, pp. 196–210, Springer, 1990.

[15] E. Ihler, "The complexity of approximating the class steiner tree problem," in *Graph-Theoretic Concepts in Computer Science*, pp. 85–96, Springer, 1992.

[16] P. Klein and R. Ravi, "A nearly best-possible approximation algorithm for node-weighted Steiner trees," *Journal of Algorithms*, vol. 19, no. 1, pp. 104–115, 1995.

[17] P. Slavik, "The errand scheduling problem," tech. rep., Department of Computer Science, SUNY, Buffalo NY, 1997.

[18] E. Ihler, G. Reich, and P. Widmayer, "Class steiner trees and VLSI-design," *Discrete Applied Mathematics*, vol. 90, no. 1, pp. 173–194, 1999.

[19] N. Garg, G. Konjevod, and R. Ravi, "A polylogarithmic approximation algorithm for the group Steiner tree problem," *Journal of Algorithms*, vol. 37, no. 1, pp. 66–84, 2000.

[20] E. Halperin and R. Krauthgamer, "Polylogarithmic inapproximability," in *Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pp. 585–594, ACM, 2003.

[21] H. Labordere, "Record balancing problem: A dynamic programming solution of a Generalized Travelling Salesman problem," *Revue Francaise D Informatique De Recherche Operationnelle*, vol. 3, no. NB 2, p. 43, 1969.

[22] G. Laporte and Y. Nobert, "Generalized traveling salesman problem through n-sets of nodes-an integer programming approach," *Information Systems and Operational Research*, vol. 21, no. 1, pp. 61–75, 1983.

[23] S. Srivastava, S. Kumar, R. Garg, and P. Sen, "Generalized traveling salesman problem through n sets of nodes," *Canadian Operational Research Society Journal*, vol. 7, pp. 97–101, 1969.

[24] G. Laporte, H. Mercure, and Y. Nobert, "Generalized travelling salesman problem through n sets of nodes: The Asymmetrical case," *Discrete Applied Mathematics*, vol. 18, no. 2, pp. 185–197, 1987.

[25] E. L. Lawler, "The traveling salesman problem: a guided tour of combi-

natorial optimization," *Wiley-Interscience Series In Discrete Mathematics*, 1985.

[26] Y.-N. Lien, E. Ma, and B. W.-S. Wah, "Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem," *Information Sciences*, vol. 74, no. 1, pp. 177–189, 1993.

[27] V. V. Vazirani, *Approximation algorithms.* Springer Science & Business Media, 2001.

[28] J. H. Holland, *Adaptation in natural and artificial systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* U Michigan Press, 1975.

[29] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & Operations Research*, vol. 13, no. 5, pp. 533–549, 1986.

[30] T. Öncan, J.-F. Cordeau, and G. Laporte, "A tabu search heuristic for the generalized minimum spanning tree problem," *European Journal of Operational Research*, vol. 191, no. 2, pp. 306–319, 2008.

[31] "IBM ILOG CPLEX Optimizer." http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/. Accessed: 2015-06-20.