MELİKŞAH UNIVERSITY
KAYSERİ

The Graduate Institute of Sciences and Engineering

M.Sc. Thesis in Electrical and Computer Engineering

ON A GREEDY HEURISTIC FOR THE STEINER FOREST PROBLEM

by

Bilge Kağan DEDETÜRK

July 2014
Kayseri, Turkey

ON A GREEDY HEURISTIC FOR THE STEINER FOREST PROBLEM

by

Bilge Kağan DEDETÜRK

A thesis submitted to

the Graduate Institute of Sciences and Engineering

of

Melikşah University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical and Computer Engineering

July 2014
Kayseri, TURKEY

APPROVAL PAGE

This is to certify that I have read the thesis entitled "On a Greedy Heuristic for the Steiner Forest Problem" by Bilge Kağan DEDETÜRK and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Electrical and Computer Engineering, the Graduate Institute of Science and Engineering, Melikşah University.

July 3, 2014 Asst. Prof. Ali ÇİVRİL
Supervisor

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

July 3, 2014 Prof. Dr. Murat UZAM
Head of Department

<u>Examining Committee Members</u>

Title and Name		Approved
Asst. Prof. Ali ÇİVRİL	July 3, 2014	
Assoc. Prof. Ahmet UYAR	July 3, 2014	
Asst. Prof. Aytekin VARGÜN	July 3, 2014	
Assoc. Prof Emel KIZILKAYA AYDOĞAN	July 3, 2014	

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Science and Engineering.

Prof. Dr. M. Halidun KELEŞTİMUR
Director

July 2014

ON A GREEDY HEURISTIC FOR THE STEINER FOREST PROBLEM

Bilge Kağan DEDETÜRK

M.S. Thesis – Electrical and Computer Engineering
July 2014

Supervisor: Assist. Prof. Dr. Ali ÇİVRİL

ABSTRACT

The Steiner forest problem is one of the important NP-Complete problems in the field of approximation algorithms and combinatorial optimization through decades. In this work, we devolop three heuristics for Steiner forest problem inspired by the greedy algorithms for the problem of finding a minimum spanning tree. According to the experimental results on random geometric graphs and real-world geometric graphs, our algorithms yield solutions of comparable quality to that of the famous 2-approximate algorithm of Agrawal, Klein and Ravi, and a widely used greedy heuristic. Especially, for real-world geometric graphs they yield much better running time with similar costs.

Keywords: Steiner Forest, NP-Complete, Approximation Algorithms, Combinatorial Optimization, Greedy Algorithms

ON A GREEDY HEURISTIC FOR THE STEINER FOREST PROBLEM

Bilge Kağan DEDETÜRK

Yüksek Lisans Tezi – Elektrik ve Bilgisayar Mühendisliği
Temmuz 2014

Tez Yöneticisi: Yrd. Doç. Dr. Ali ÇİVRİL

ÖZ

Steiner ormanı problemi onlarca yıldır yaklaştırma algoritmaları ve kombinatoryel optimizasyon alanındaki önemli NP-Tam problemlerden biridir. Bu çalışmada, Steiner ormanı problemi için minimum bir tarayan ağaç bulma problemini çözen açgözlü algoritmalardan esinlenen üç adet sezgisel algoritma geliştiriyoruz. Rastgele geometrik çizgeler ve gerçek dünya geometrik çizgeleri üzerindeki deneysel sonuçlara göre algoritmalarımız Agrawal, Klein ve Ravi'nin ünlü 2 yaklaşık algoritması ve geniş bir şekilde kullanılan açgözlü sezgisel algoritma ile karşılaştırılabilir kalitede çözümler veriyor. Özellikle, gerçek dünya geometrik çizgeleri için algoritmalarımız benzer maliyetlerle çok daha iyi çalışma zamanı veriyor.

Anahtar Kelimeler: Steiner Ormanı, NP-Tam, Yaklaştıma Algoritmaları, Kombinatoryel Optimizasyon, Açgözlü Algoritmalar

DEDICATION

Dedicated to my parents for their endless support and patience during the forming phase of this thesis.

ACKNOWLEDGEMENT

I would like to thank TÜBİTAK (Türkiye Bilimsel ve Teknolojik Araştırma Kurumu) that has supported my thesis studies under the project number 112E192.

My special thanks to my advisor, Dr. Ali Çivril for his encouragement, motivation, guidance, and help on technical issues. Through his help, I get opportunities to learn new things in the development of the thesis.

I also want to express my gratitude to my colleagues Ayub Rokhman Wakhid and Osman Melih Kürtüncü for their supports.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

CHAPTER 1

INTRODUCTION

Many optimization problems are NP-hard which means that there is no polynomial time algorithm that exactly solves the problem unless P = NP. If P ≠ NP, we can't find optimal solutions in polynomial time for NP hard problems. What can we do when we encounter with such NP hard problems? One approach is that we can try to find an approximate solution that runs in polynomial time instead of finding optimal solution. The goal is to find a solution in polynomial time which is as close as possible to the optimal solution under some objective function.

Definition 1.1: *An α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of α of the value of an optimal solution.* [1]

Here, α is the approximation ratio or approximation factor of the algorithm. $\alpha > 1$ for minimization problems while $\alpha < 1$ for maximization problems.

Based on this definition, note that an approximation algorithm has to guarantee the approximation factor for all instances of the problem. This requirement makes approximation algorithms hard and complicated to design and also they may need high running time to be implemented. However, very complicated instances (bad cases) are generally hard to be found in the real-world. Hence, to solve real-world problems, we can still benefit from other algorithms that even do not have proven approximation ratios, but can give reasonably good results in a reasonable running time.

In this thesis, we provide novel algorithms for the Steiner forest problem that are applicable to real-world data with better running time and comparable quality compared to the well known approximation algorithms which we briefly call AKR. We base our algorithms on the well-known minimum spanning tree algorithms of Kruskal, Prim, and Boruvka. We give an example in which our algorithms work worse than AKR in a specific case. However in real-world instances like the geometric graph representing the cities of specific countries, our algorithms work much faster than the primal dual algorithm AKR although they have similar costs. Also, our algorithms have very good running time for all instances and they are very simple compared to AKR.

1.1 The Steiner Forest Problem

In this thesis, we deal with the Steiner forest problem which is also known as the generalized Steiner tree problem. The Steiner forest problem is a natural generalization of the Steiner tree problem. So, we first mention about the Steiner tree problem.

In the Steiner tree problem, we are given a connected, undirected graph G = (V, E) with nonnegative costs c_e for all edges e ∈ E and a set of terminal vertices T ⊆ V. The vertices of V-T are nonterminal vertices or Steiner vertices. The goal is to find a minimum cost tree which contains all terminal vertices in T.

In the Steiner forest problem, we are given an undirected graph G = (V, E), nonnegative edge costs c_e for all edges e ∈ E. Also we are given a set of k terminal pairs R = $\{(s_1, t_1), \ldots, (s_k, t_k)\}$. The object is to find minimum cost subset of edges F ⊆ E such that there is at least one path between each terminal pair s_i and t_i for 1 ≤ i ≤ k.

The Steiner tree problem is an NP hard problem [2]. Since the Steiner forest problem is the natural generalization of the Steiner tree problem, the Steiner Forest problem is also an NP hard problem. That is to say, if t_i = s_{i+1} for i = 1,…, k-1 the Steiner Forest problem reduces to the Steiner tree problem. The Steiner forest problem is one of the major problems in the field of approximation algorithms. As we will see, there is a 2-approximate algorithm for this problem which is more than 2 decades old. A text written by Williamson and Shmoys [1] tells us that obtaining an algorithm having approximation factor strictly less than 2 for the Steiner forest problem is one of the ten major open problems in the field.

1.2 Related Work

The well-known approximation algorithm for the Steiner Forest problem was introduced by Agrawal, Klein and Ravi [3]. Approximation ratio of this algorithm is 2-$\frac{1}{k}$, where k is the number of terminal pairs. Then, Goemans and Williamson [4] introduced an algorithm mimicking this algorithm by using the primal-dual method. The approximation ratio of this algorithm is still 2-$\frac{1}{k}$. Analogously, recent paper by Könemann, Leonardi, Schafer and Van Zwam [5] give us an algorithm having the same approximation ratio with this algorithm by using a slightly different LP relaxation.

Besides of the Steiner forest problem itself, there has been numerous works associated with it. Here, we mention about some of them which are popular.

The multicommodity rent-or-buy problem (MRoB) is a generalization of the Steiner Forest problem. In this problem, we are given a weighted graph G = (V, E), nonnegative edge costs c_e for all edges e ∈ E, a set of k terminal pairs R = {(s_1, t_1),…, (s_k, t_k)}, and a parameter M ≥ 1. We are also given a positive demand d_i for each terminal pair (s_i, t_i) ∈ R for 1 ≤ i ≤ k. The object is to install capacities on the edges of G such that for all (s_i, t_i) ∈ R, we can simultaneously route an amount of flow d_i from s_i *to* t_i. We can either buy infinite capacity on edge e at cost $M.w(e)$ or we can rent capacity on an edge e at cost $f(e).w(e)$, where $f(e)$ is the flow on edge e. For M = 1 and unit demands, this problem reduces to the Steiner Forest problem. Kumar, Gupta and Roughgarden [6] gave first approximation algorithm for this problem. Later Gupta, Kumar, Pal and Roughgarden [7, 8] produced an algorithm whose approximation ratio is 12 (Actually, it can be tightened to 8-approximation). Then Becchetti, Könemann, Leonardi and Pal [9] gave the improved approximation ratio which is 6.828. Fleischer et al. [10] have given the best approximation ratio which is 5 for this problem so far.

Another well-known problem which is related with the Steiner Forest problem is the survivable network design problem. It is the natural generalization of the Steiner forest problem, that is, if we require that there exists r_i edge-disjoint paths between s_i and t_i, we have the survivable network design problem. For this problem was given a 2k approximation algorithm by Williamson et al. [11]. Goemans et al. [12] gave a $2H_k$ approximation algorithm, where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$. Later, Jain [13] produced a 2 approximation algorithm for this problem.

CHAPTER 2

PRELIMINARIES

2.1 Dijkstra's Algorithm

In section 4.1 where we present our algorithms, we will need to compute all the shortest paths between terminals. In addition, in section 3.2 where we will talk about a well known greedy heuristic, we will need to compute all the shortest paths between all terminal pairs. There are several algorithms for finding shortest paths between vertices. Here, we consider Dijkstra's algorithm. Dijkstra's algorithm is an algorithm which solves the single source shortest paths problem on a weighted connected graph G = (V, E) with nonnegative weights. This algorithm calculates all the shortest paths from a given vertex called the source to all other vertices. In the following, we give the pseudocode of Dijkstra's algorithm [14].

Algorithm 2.1.1: Dijkstra's Algorithm

Initialize minimum priority queue(Q)
for every vertex v in V
    $d_v \leftarrow \infty$
    $p_v \leftarrow$ null
    Enqueue(v, d_v)
$d_s \leftarrow 0$
DecreaseKey(Q, s, d_s) // s is the element in the priority queue whose priority will be decreased
S = ∅
while Q ≠ ∅
    $u \leftarrow$ DequeueMin(Q)
    S ← S ∪ { u }
    for every vertex v in V – S that is adjacent to u
        if $d_u + w(u,v) < d_v$
            $d_v \leftarrow d_u + w(u,v)$
            $p_v \leftarrow u$
            DecreaseKey(Q, u, d_u)

2.2 Disjoint Set Operations

We will use the disjoint set operations in section 3.1 and 4.1 while giving detailed descriptions of algorithms there in. MAKE-SET operation will be used to make a set for each terminal. We will use FIND-SET operation to check whether two nodes are in the same set and UNION operation to combine two different sets into one set. We usually use the definitions and notation from [15]. Below are the main functions that are used inside disjoint set operations.

- MAKE-SET(u): creates a new set whose only member and representative is u.
- UNION(u, v): combines the different two sets that contain u and v into a new set which is the union of these two sets.
- FIND-SET(u): finds and returns the representative of the set which contain u.

2.3 Minimum Spanning Tree Algorithms

Before we discuss our algorithms in section 4.1, it is of a great benefit to remember the well-known minimum spanning tree algorithms of Kruskal [16], Prim [17], and Boruvka [18]. We can choose any one of these algorithms as a basis to form our algorithms.

Kruskal’s algorithm works by sorting the edges in non-decreasing order and sequentially processes all of them by starting from the smallest weight edge. The algorithm takes the next smallest weight edge and checks whether a cycle is formed. To check whether a cycle is formed, a disjoint set data structure is used. If a cycle is not formed, it includes the edge into the tree. It repeats this process until finally a tree that includes all nodes in graph is formed. Prim’s algorithm achieves the same goal differently. It finds the smallest weight edge on the boundary of current set to connect this set to the rest of the nodes. This process continues until all of the nodes included to the tree. As for Boruvka’s algorithm, it works similar to Kuruskal’s algorithms. At each step, the cheapest edges that are going out of each set are considered. At the final iteration, algorithm merges the sets appropriately and continues until just one set remains which is the minimum spanning tree. The pseudocodes of the algorithms are provided as follows:

Algorithm 2.3.1: Kruskal's Algorithm

```
G=(V, E), w: E → Q+
F ← ∅
for each vertex v ∈ V do
     MAKE-SET(v)
Sort the edges of E in non-decreasing order by w
for each edge (u, v) ∈ E, taken in non-decreasing order by w do
    if FIND-SET(u) ≠ FIND-SET(v) then
        F ← F ∪ {(u, v)}
        UNION(u, v)
return F
```

Algorithm 2.3.2: Prim's Algorithm

```
G=(V, E), w: E → Q+, r
F ← ∅
S ← MAKE-SET(r)
while |S| < |V| do
    F' ← ∅
    for each vertex u in S do
        F' ← F' ∪ {the cheapest edge (u,v) such that FIND-SET(u) ≠ FIND-SET(v) }
    F ← F ∪ {the cheapest edge (u', v') in F' }
    UNION(u', v')
return F
```

Algorithm 2.3.3: Boruvka's Algorithm

```
G=(V, E), w: E → Q+
F ← ∅
for each vertex v ∈ V do
     MAKE-SET(v)
while there are more than 1 set do
    for each set S do
        F_s ← ∅
      for each vertex u in S do
          F_s ← F_s ∪ {the cheapest edge (u,v) such that FIND-SET(u) ≠ FIND-SET(v)}
        F ← F ∪ {the cheapest edge (u_s, v_s) in F_s}
    for each set S do
        UNION(u_s, v_s)
return F
```

CHAPTER 3

ALGORITHMS FOR THE STEINER FOREST PROBLEM

3.1 A Primal Dual Algorithm for the Steiner Forest Problem

We now return to the Steiner Forest problem. Recall that in this problem we are given an undirected graph $G = (V, E)$ with nonnegative edge costs c_e for all edges $e \in E$ and a set of k terminal pairs R = $\{(s_1, t_1), \dots, (s_k, t_k)\} \subseteq$ V × V. The goal is to find a minimum cost subset of edges $F \subseteq E$ such that there is at least one path between each terminal pair s_i and t_i for 1 ≤ i ≤ k.

Here, we consider the algorithm of Agrawal, Klein, and Ravi [1], which we briefly call AKR. AKR provides a 2-$\frac{1}{k}$ approximate solution for the Steiner Forest problem. This algorithm is a primal dual algorithm, that is to say, it performs both a feasible integral primal and a feasible dual solution. In the standart integer programming formulation of AKR, there is a binary variable x_e for all edges $e \in E$ such that this variable has value 1 if e is in the resulting forest $F \subseteq E$ and 0 otherwise. A subset $C \subseteq V$ is a Steiner cut if it separates at least one terminal pair in R. In other words, C is a Steiner cut iff there is a pair $(s, t) \in R$ that satisfies $|\{s, t\} \cap C| = 1$. Let S be the set of all Steiner cuts. Let also $\delta(C)$ include all the edges which is one endpoint in C and other endpoint not in C. In other words, we define $\delta(C)$ to denote the set of edges crossing the cut (C,$\bar{C}$). Now, we can give the following integer linear programming formulation for the Steiner Forest problem.

$$minimize \quad \sum_{e \in E} c_e x_e \qquad\qquad (IP)$$

$$subject\ to \quad \sum_{e \in \delta(C)} x_e \ \geq 1, \qquad \forall C \subseteq S,$$

$$x_e \in \{0, 1\}, \qquad \forall e \in E$$

In the dual program, there is a variable y_C for each Steiner cut $C \in S$. The program tries to maximize the sum of the dual variables y_C subject to the condition that the cost of each edge e in $\delta(C)$ can not be smaller than the dual variable y_C. The dual of the linear programming relaxation (LP) of integer programming (IP) is as follows:

$$\begin{array}{llr} \textit{maximize} & \sum_{C \in S} y_C & (D) \\ \textit{subject to} & \sum_{C \in S: e \in \delta(C)} y_C \le c_e, \quad \forall e \in E, & \\ & y_C \ge 0, \qquad \forall C \in S. & \end{array}$$

At each iteration, the algorithm raises the dual variables y_C uniformly for each Steiner cut $C \in S$ until the some edge e $\in \delta(C)$ goes tight. It maintains the iterations until all terminal pairs are connected. After all iterations are completed and the resulting forest F is generated, the algorithm performs the reverse-delete operation. In this operation, the algorithm removes the edge or edges from the resulting forest F provided that the resulting forest remains feasible; that is, it excludes the edge or edges from the the resulting forest as long as all s_i and t_i pairs are connected.

Subsequently, we use the term of “active set” instead of the term of “Steiner Cut”. A set is an active set iff it includes a cut that separates at least one terminal pair in R. We give the following example to better explain when a set is active or not.

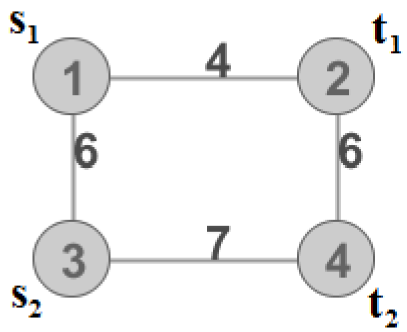

Figure 3.1.1: An example to explain when a set is active or not

Initially, the following four sets are active: {1}, {2}, {3}, and {4} and each of these active sets has the dual variable $y_C = 0$. In the first iteration, the algorithm raises the dual variables by 2 for each active set and the edge between the nodes 1 and 2 goes tight. It then adds this edge to resulting forest F. The sets {1} and {2} replace {1,2} as a set. However, the set {1,2} isn't an active set any more because it doesn't separate any terminal pair. So, this set doesn't have the dual variable any longer. The active sets are {3} and {4}. When the dual variables are raised by 1.5, edge between 3 and 4 goes tight and the sets {3} and {4} replace {3,4} as a set. The algorithm adds this edge to resulting forest. We now have a feasible solution since all s_i and t_i pairs (1-2) and (3-4) are connected.

In the following, we give pseudocode of the primal dual algorithm for the Steiner forest problem [1].

Algorithm 3.1.1: Primal-dual Algorithm for the Steiner forest problem

1: $y \leftarrow 0$
2: F $\leftarrow \emptyset$
3: $l \leftarrow 0$
4: **while** not all $s_i - t_i$ pairs are connected in (V, F) **do**
5: $\quad l \leftarrow l + 1$
6: $\quad$ Let S be the set of all connected components C of (V, F) such that $|C \cap \{s_i, t_i\}| = 1$
7: $\quad$ for some i
8: $\quad$ Increase y_C for all C in S uniformly until for some $e_l \in \delta(C'), C' \in S$,
9: $\quad c_{e_l} = \sum_{S: e_l \in \delta(S)} y_S$
10: $\quad F \leftarrow F \cup \{e_l\}$
11: $F' \leftarrow F$
12: **for** $k \leftarrow l$ down to 1 **do**
13: $\quad$ **if** $F' - e_k$ is a feasible solution **then**
14: $\quad\quad$ Remove e_k from F'
15: **Return** F'

Line 1 initializes the dual variable to 0. Line 2 initializes the resulting forest F to empty set and line 3 initializes the variable l to 0. Until all terminal pairs (s_i - t_i) are connected, the algorithm maintains the while loop of line 4-10. Each time through the "while" loop of lines 4-10, line 5 raises the variable l by 1, lines 8 and 9 increase the dual variable y_C uniformly for all connected components (active sets) C until the dual equality provide for some edge e $\in \delta(C)$ and line 10 adds this edge to resulting forest F. Each time through the "for" loop of lines 12-14, the algorithm removes the edge if the solution is still feasible when this edge is removed from the resulting forest F. Finally, line 15 returns the solution.

In the following, we give an example on which we explain how the primal-dual algorithm works.

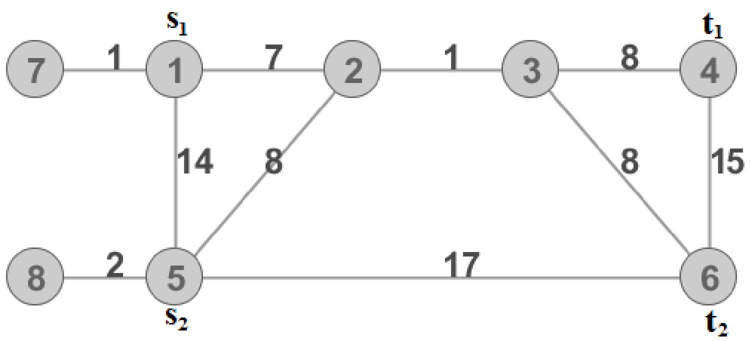

Figure 3.1.2: An example to explain how AKR works

Initially, the following four sets are active: {1}, {4}, {5}, and {6} and each of these active sets has the dual variable $y_C = 0$. In the first iteration, the algorithm raises the dual variables by 1 for each active set and the edge between the nodes 1 and 7 goes tight. In the second iteration, it then adds this edge to resulting forest F and the active set {1} and the set {7} replace {1,7} as an active set.

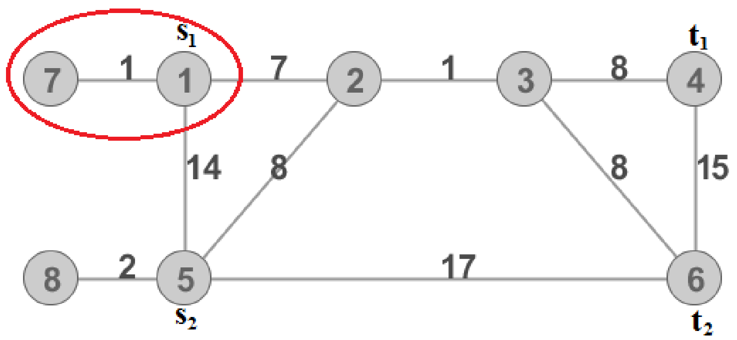

Figure 3.1.3: Illustration of union 1 and 7

The active sets are {1,7}, {4}, {5} and {6}. When the dual variables are raised by 1, edge between 5 and 8 goes tight. The algorithm adds this edge into resulting forest and the active set {5} and the set {8} replace {5,8} as an active set as follows:

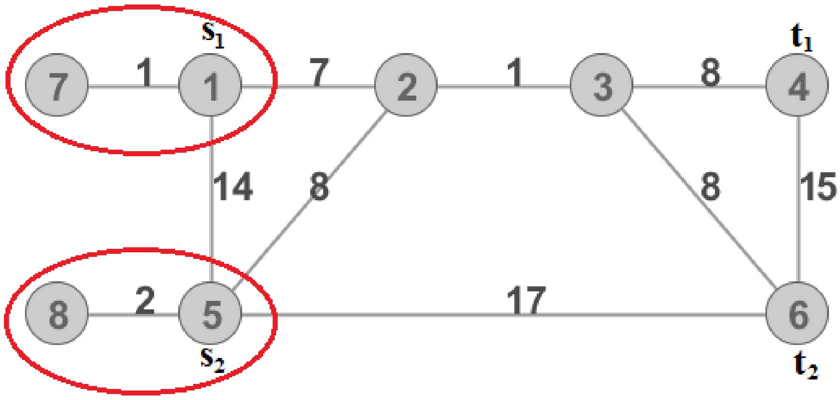

Figure 3.1.4: Illustration of union 5 and 8

The active sets are {1,7}, {4}, {5,8} and {6}. When the dual variables are raised by 5, edges (1-2) and (1-5) go tight at same time. Let's assume that the algorithm select the edge between 1 and 2 first. In the next iteration, it adds this edge into resulting forest and the active set {1,7} and the set {2} replace {1,7,2} as an active set. In the next iteration, it adds edge (1-5) into resulting forest and the active set {1,7,2} and the active set {5,8} replace {1,2,7,5,8} as an active set as follows:

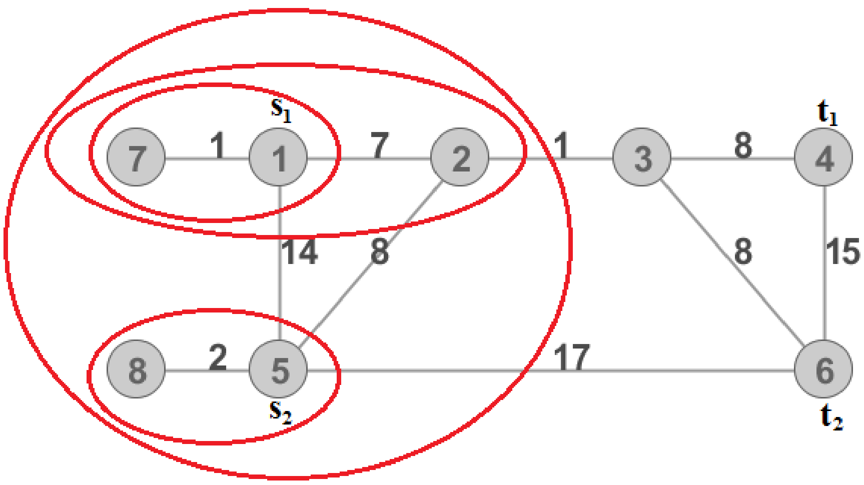

Figure 3.1.5: Union of the sets corresponding to 1, 2 and 5

The active sets are {1,2,7,5,8}, {4}, and {6}. When the dual variables are raised by 0.5, edge between 4 and 6 goes tight. The algorithm adds this edge into resulting forest and the active set {4} and the active set {6} replace {4,6} as an active set as follows:

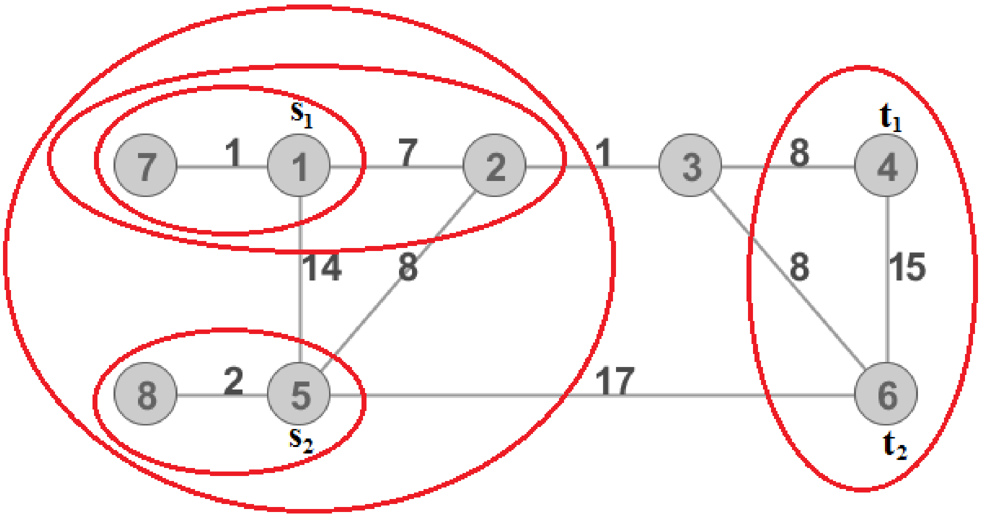

Figure 3.1.6: Illustration of union 4 and 6

The active sets are {1,2,7,5,8} and {4,6}. When the dual variables are raised by 0.5, edges (2-3), (3-4) and (3,6) go tight at same time. Let's assume that the algorithm select the edge between 3 and 6 first. In the next iteration, it adds this edge into resulting forest and the active set {4,6} and the set {3} replace {4,6,3} as an active set. Then, let the algorithm select the edge between 3 and 4. Since set 3 and set 4 are in same set, the algorithm can not add this edge into resulting forest. In the next iteration, it adds edge (2-3) into resulting forest. We now have a feasible solution since all s_i and t_i pairs (1-4) and (5-6) are connected. In the reverse-delete step, edges (1-7) and (5-8) are excluded from resulting forest. Finally, solution obtained by the algorithm includes edges (1-2), (2-3), (3-6), (1-5), (4-6). The cost of this solution is 45. Edges which the solution includes are marked thick:

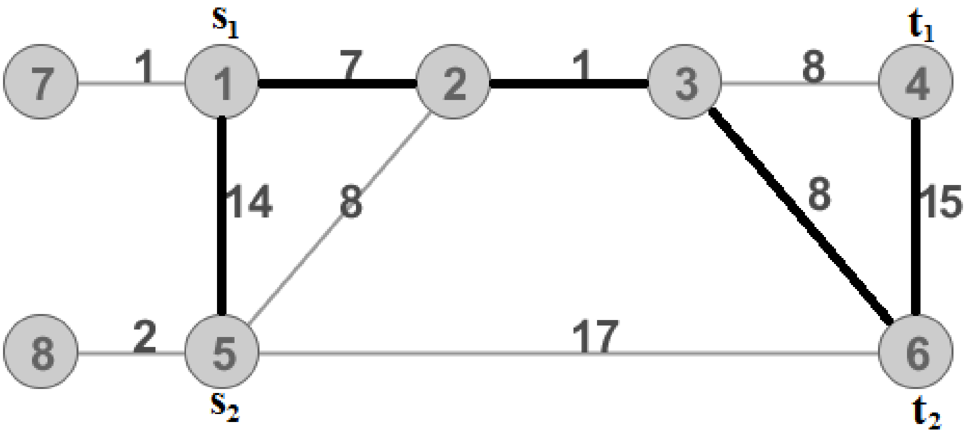

Figure 3.1.7: Illustration of the solution returned by AKR

The dual variables in the primal dual algorithm have a very nice geometric interpretion as growing moats. We can interpret the dual variables y_C as moats in order to better understand the primal dual algorithm for the Steiner forest problem. Initially each s_i and t_i considers as a component and each of them has a moat. At each iteration, moats are grown uniformly around the components and each time during the growing, if they cover an edge e, it is added to the solution F. Moats continue to grow until collision occurs. After the collision, these moats are considered as a single component. Figure 3.1.8 illustrates us how the idea of growing moats works on a specific example.

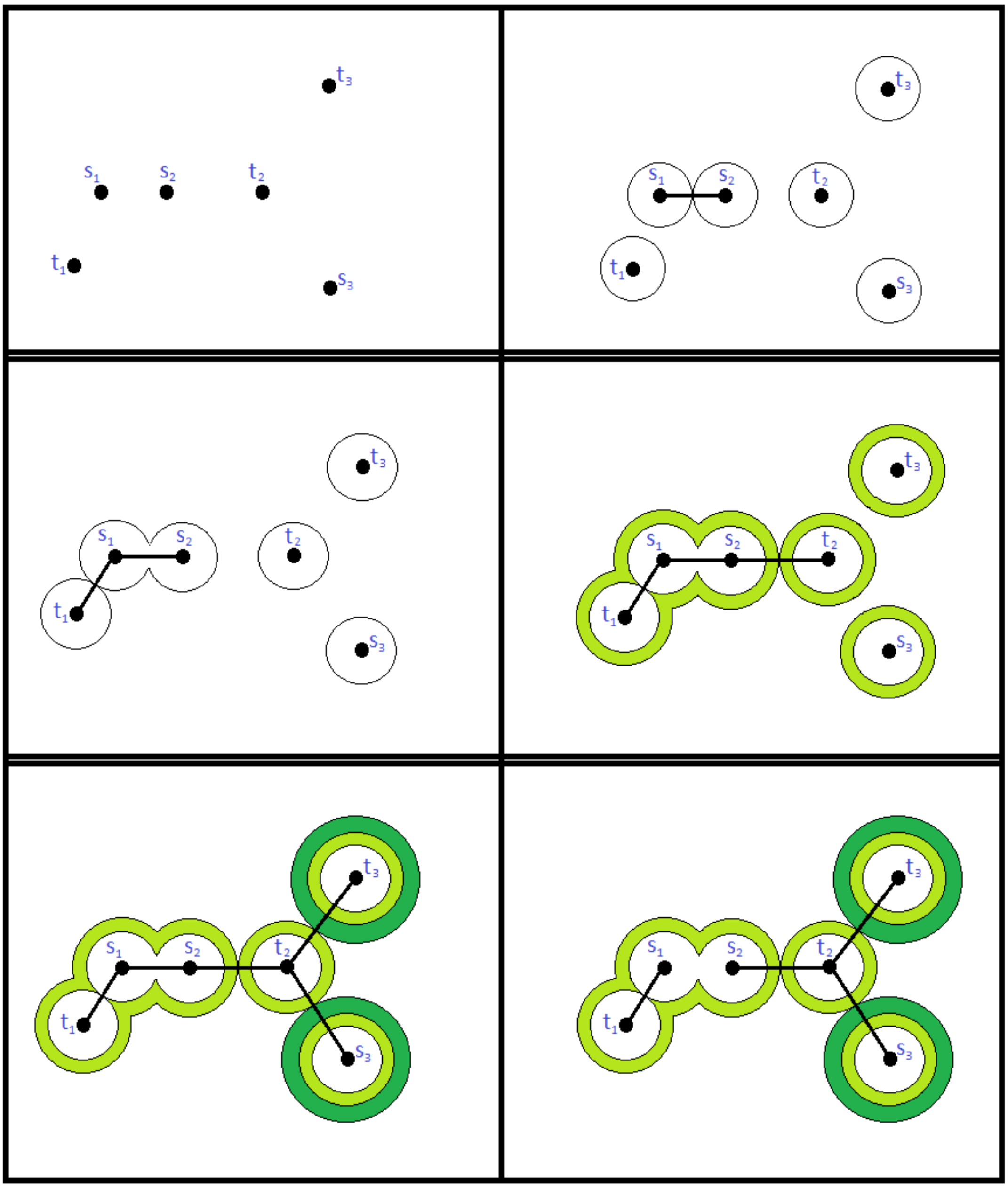

Figure 3.1.8: Illustration of growing moats on the specific example

In the following, we give a specific example for which the cost of the solution returned by AKR is as bad as $2 - \frac{1}{k}$ times the cost of the optimum solution. In this example, all terminal pairs s_i and t_i are adjacent to each other with cost of $1 + \epsilon$ for $i = 1, \ldots, k$. Also the distances between both s_j, s_{j+1} and t_j, t_{j+1} are 1 for $j = 1, \ldots, k-1$. The example is shown in Figure 3.1.9.

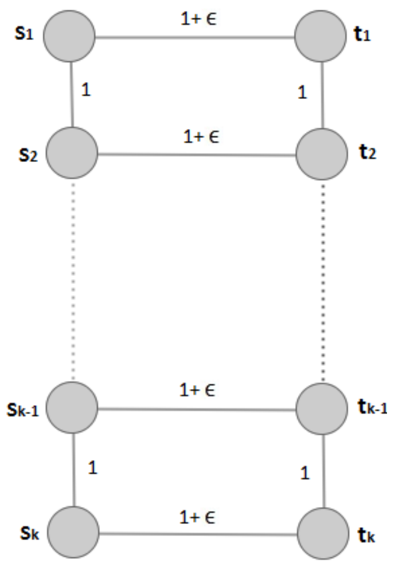

Figure 3.1.9: Bad example for AKR

AKR first takes all the edges with cost of 1, a total of 2k. Then it takes one of the edges whose costs are $1 + \epsilon$. Therefore, the total cost of the solution returned by AKR is $2k + 1 + \epsilon$. However, the cost of the optimum solution is $k(1 + \epsilon)$; where k is the number of terminal pairs and $(1 + \epsilon)$ is the distance between each of them. The solution returned by AKR and the optimum solution is shown in Figure 3.1.10 and Figure 3.1.11, respectively.

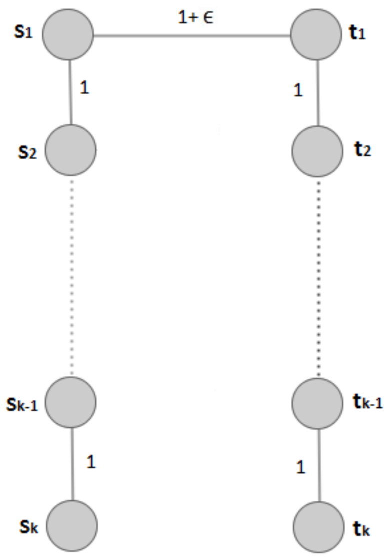

Figure 3.1.10: Illustration of the solution returned by AKR

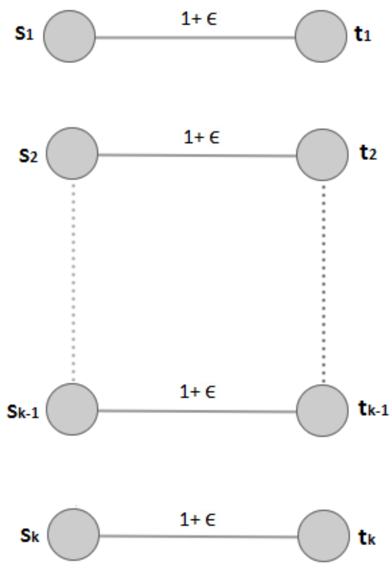

Figure 3.1.11: Illustration of optimum solution

3.2 A Greedy Heuristic

In this section, we give a greedy heuristic for the Steiner Forest problem. This algorithm is a well known algorithm which has 2-approximation factor for the Steiner tree problem. It is fairly simple algorithm compared to AKR. Also, running time of this algorithm is in practice much better than AKR. However, this algorithm is not a 2-approximation algorithm for the Steiner forest problem. We will give a specific example for which the cost of the solution returned by this algorithm is as bad as 4 times the cost of the optimum.

In this algorithm, we first compute the shortest path between s_1 and t_1. In the next iteration, we add all the edges on this shortest path whose weighted lengths are not equal to zero into solution and zero out these edges. Then we compute the shortest path between s_2 and t_2, and so on. The iterations are performed for all terminal pairs from 1 to k.

In what follows, we give an example on which we explain how the algorithm works.

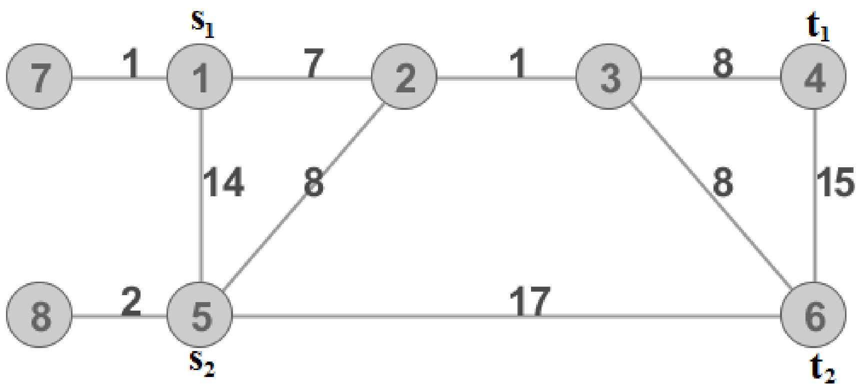

Figure 3.2.1: An instance for Greedy Heuristic

In the first iteration, we compute the shortest path between 1 and 4 (s_1 and t_1). This shortest path includes the edges (1-2), (2-3), and (3-4). In the second iteration, we add the edge (1-2) into the solution since this edge is not equal to zero. In the third iteration, we change the cost of this edge into zero. In the next iterations, we add the edges (2-3) and (3-4) into the solution since the costs of these edges are not equal to zero and we change the costs of these edges into zero. The edges included into the solution up to this point are shown in Figure 3.2.2, with selected edges and the new costs of these edges marked red:

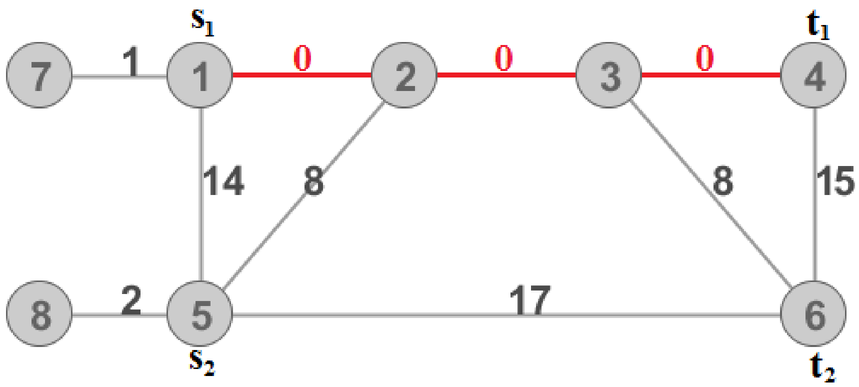

Figure 3.2.2: Illustration of the edges added into solution

In the next iteration, we compute the sorthest path between 5 and 6 (s_2 and t_2). This shortest path includes the edges (5-2), (2-3), and (3-6). In the next iterations, we add all the edges on this shortest path into solution except the edge (2-3) since the cost of this edge is equal to zero and change the costs of these edges into zero as shown in the figure below:

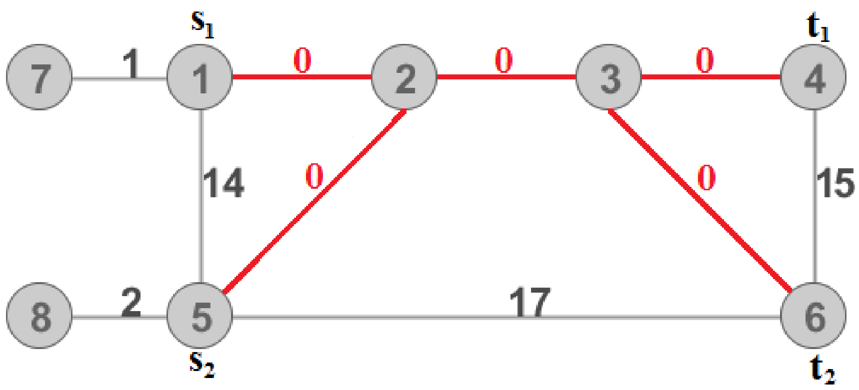

Figure 3.2.3: Illustration of the solution returned by Greedy Heuristic

Finally, we obtain the same solution with optimal which total cost is 32. The solution includes the edges (1-2), (2-3), (3-4), (5-2), and (3-6).

Following is the pseudocode of the algorithm.

Algorithm 3.2.1: Greedy Heuristic for Steiner Forest Problem

1: G=(V, E), $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}, w: E \rightarrow Q^+$
2: F $\leftarrow \emptyset$ //Initialize Forest to empty
3: **for** i ← 1 to k **do**
4: p ← shortest path between s_i and t_i
5: **for** edge e: all the edges on the path p **do**
6: **if** e.cost ≠ 0 **then**
7: F ← F ∪ {e}
8: e ← 0
9: return F

Line 2 initializes the resulting forest F to empty. Each time through the "for" loop of lines 3-8, line 4 computes the shortest path between s_i and t_i and lines 5-8 add all the edges on this path into F if the cost of each edge is not equal to zero and change the cost of the edge into zero. Finally, line 9 returns the solution.

In the following, we provide a specific example which shows that the greedy heuristic we mention above is worse than AKR with respect to approximation ratio. In this example, every terminal pairs are adjacent to each other with a cost of $4-\epsilon$ as shown in Figure 3.2.4. The cost of the solution that greedy algorithm finds is worse than the twice of the cost of the optimum solution.

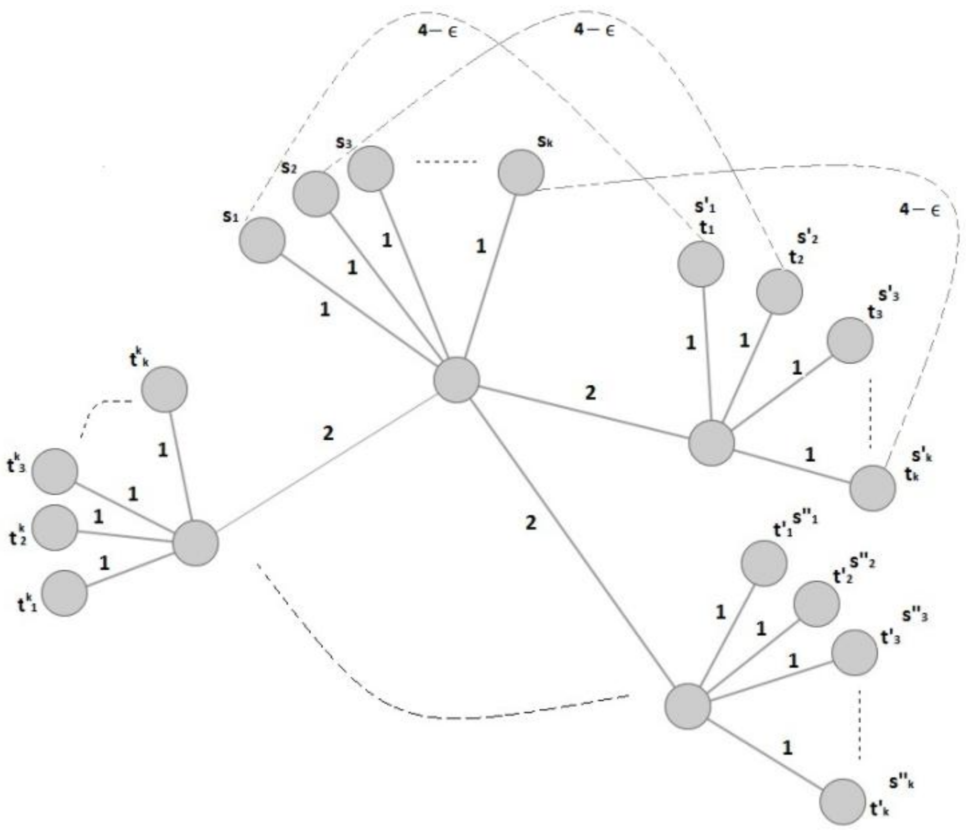

Figure 3.2.4: Bad example for Greedy Heuristic

Greedy algorithm takes all the shortest paths between terminal pairs with distance $4-\epsilon$. The cost of the solution is $(4-\epsilon)k^2$ (see Figure 3.2.5). However, the cost of the optimum solution is k^2+2k (see Figure 3.2.6). Also the cost of the solution given by AKR is $k^2+(k-1)(4-\epsilon)$ as shown in Figure 3.2.7. So, the cost of the solution given by greedy algorithm is 4 times as large as the cost of the optimum solution.

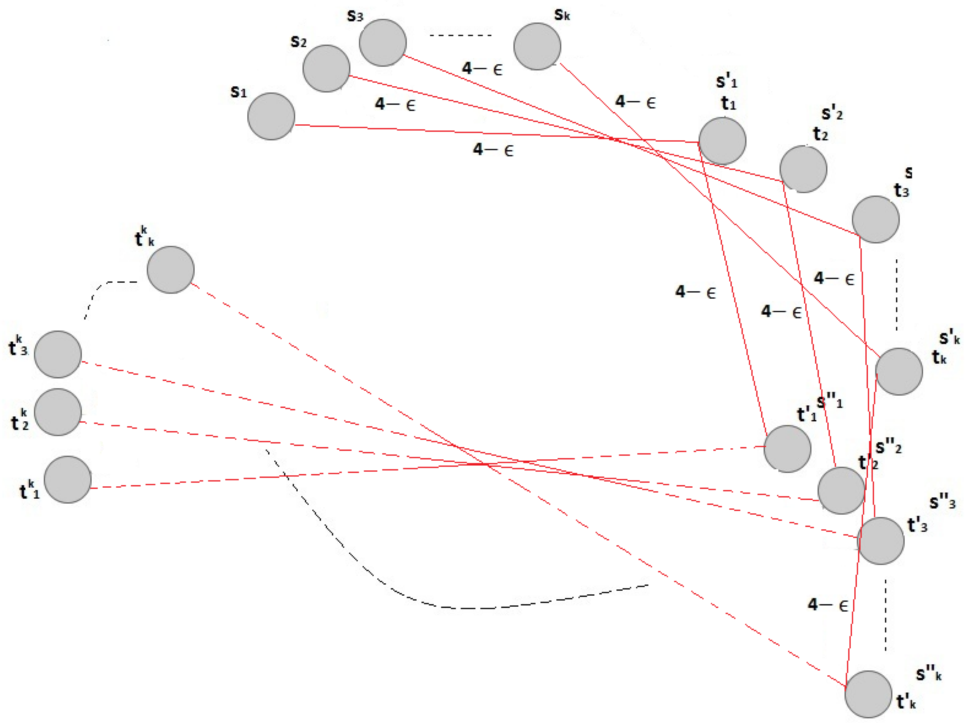

Figure 3.2.5: Forest returned by Greedy Heuristic

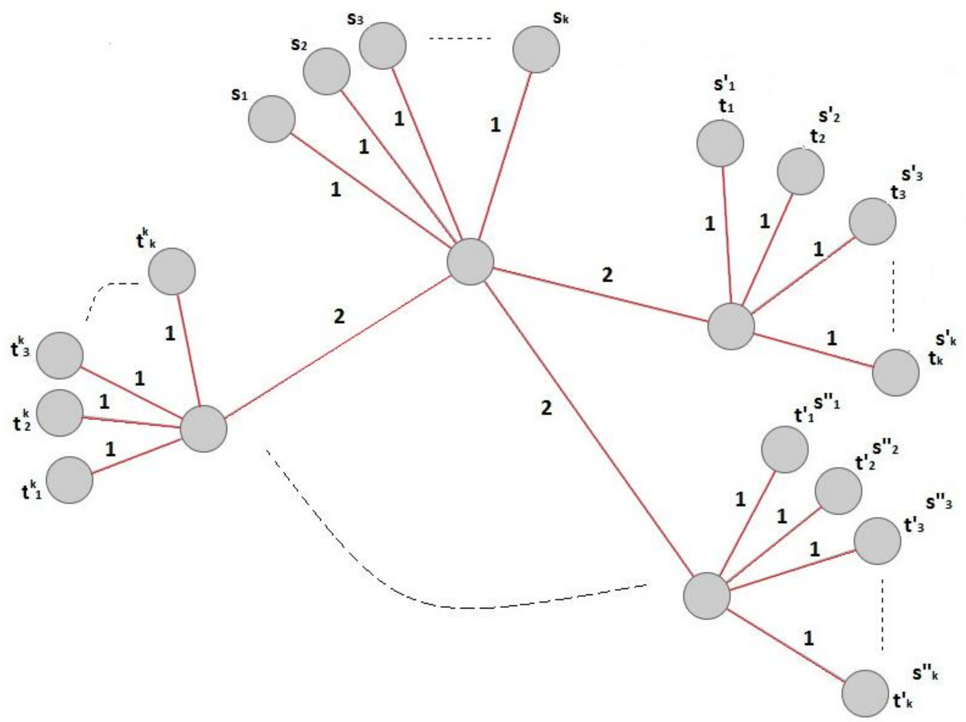

Figure 3.2.6: Optimum solution

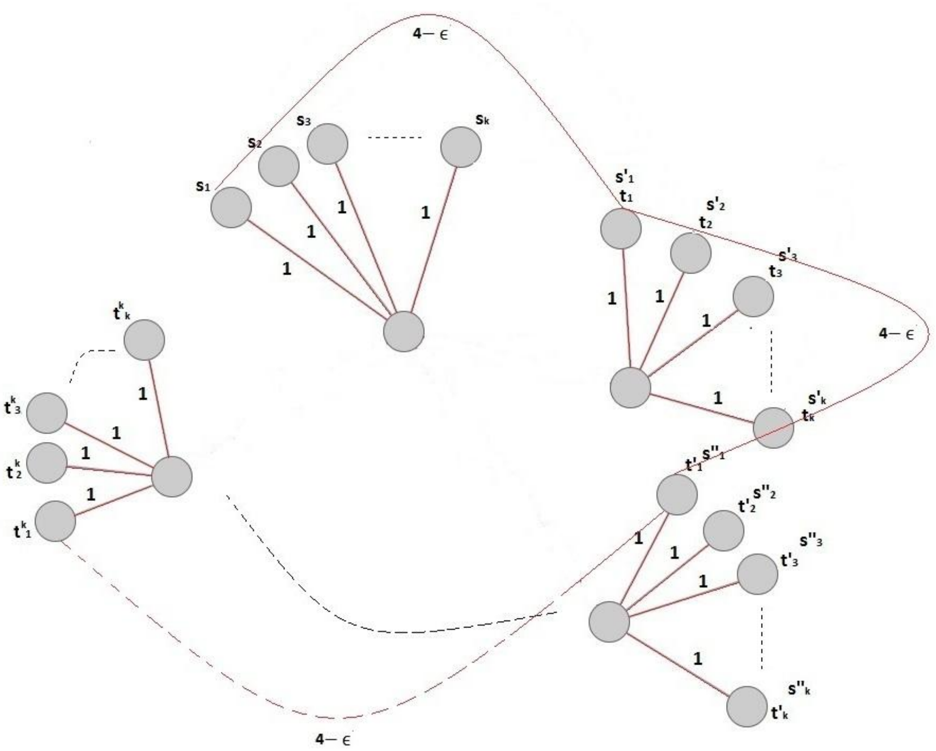

Figure 3.2.7: Forest returned by AKR

CHAPTER 4

NEW ALGORITHMS FOR THE STEINER FOREST PROBLEM

4.1 The New Algorithms

In this section, we provide three equivalent greedy algorithms for the Steiner forest problem, which we appropriately name ÇDK-Kruskal, ÇDK-Prim and ÇDK-Boruvka.

All these algorithms are equivalent and in this section we mainly talk about ÇDK-Kruskal. In this algorithm, firstly, we make a different set for each terminal, a total of $2k$ sets. Then, we compute all the shortest paths between terminals. That is, for the number of k terminal pairs (s_1-t_1, s_2-t_2, … , s_k-t_k) we compute a total of $\binom{2k}{2}$ shortest paths between terminals. Then we sort them in non-decreasing order. After this step, we iteratively check whether the endpoints of the paths are in the same set are or not. If they are not in the same set, we include the path into the solution and combine them into one set by using the union operation. If they are in the same set, we skip this path and continue to the next iteration. This iterations continue until all terminal pairs are connected. After all terminal pairs are connected, we perform a reverse–delete operation. In this operation, if there are unnecessary paths which do not violate the feasibility of the solution, we remove these paths by starting from the last element of array (resulting forest) that contains the selected paths. Following is an example which explains the execution of our algorithms in more detail.

:

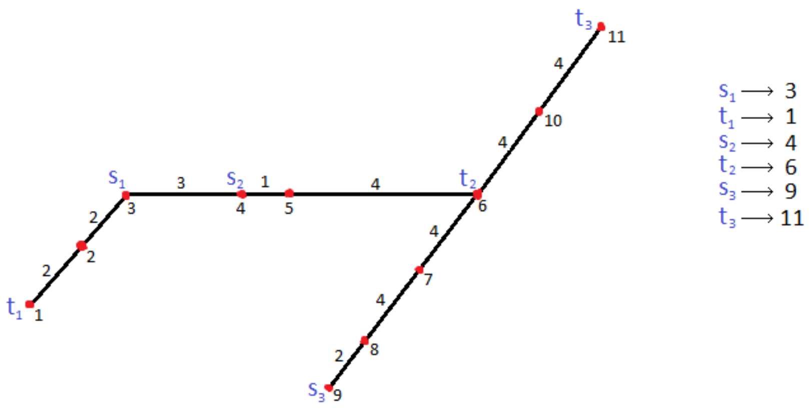

Figure 4.1.1: Illustration of the specific example.

For this example, $k = 3$ and we have terminal pairs (s_1, t_1), (s_2, t_2) and (s_3, t_3).

For each terminal, the algorithm creates a set indicated by the node number as shown in the figure below.

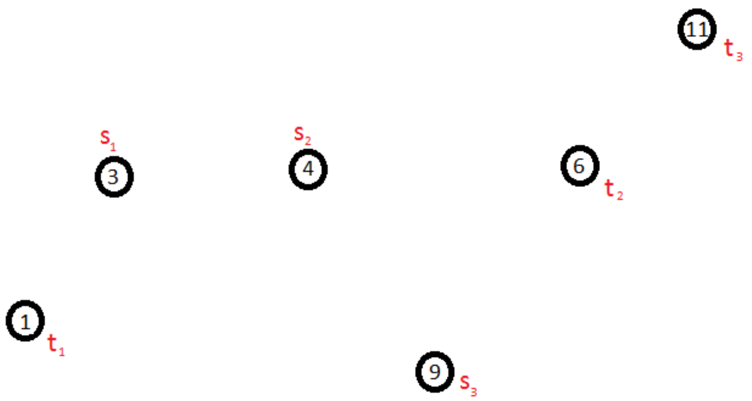

Figure 4.1.2: Illustration of sets.

Then, it computes all the shortest paths between terminals and then sorts them in non-decreasing order. Following is the result of this computation.

	Node1	Node2	Distance
1	3	4	3.0
2	3	1	4.0
3	4	6	5.0
4	4	1	7.0
5	3	6	8.0
6	6	11	8.0
7	6	9	10.0
8	1	6	12.0
9	4	11	13.0
10	4	9	15.0
11	3	11	16.0
12	3	9	18.0
13	9	11	18.0
14	1	11	20.0
15	1	9	22.0

$\binom{6}{2} = 15$

Figure 4.1.3: Sorted shortest paths between terminals

The algorithm first checks whether 3 and 4 sets are in the same set. They are not in the same set, so it adds this path into solution and combines 3 and 4 into one set by using the union operation which can be seen in the following illustration (Figure).

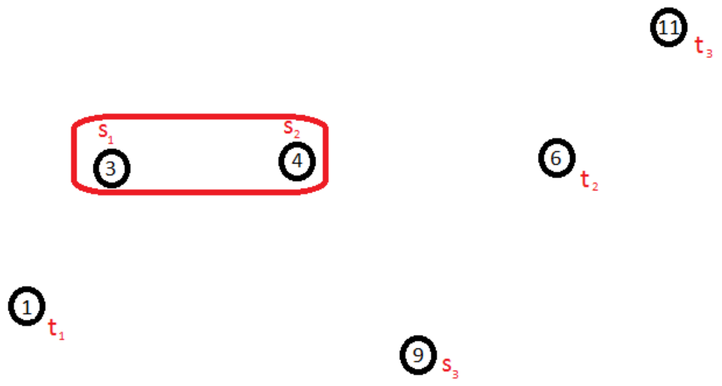

Figure 4.1.4: Illustration of union 3 and 4

Then, it checks whether 3 and 1 sets are in the same set. Since they are not in the same set, it adds this path into solution and combines 3 and 1 into one set by using the union operation as shown in the following figure:

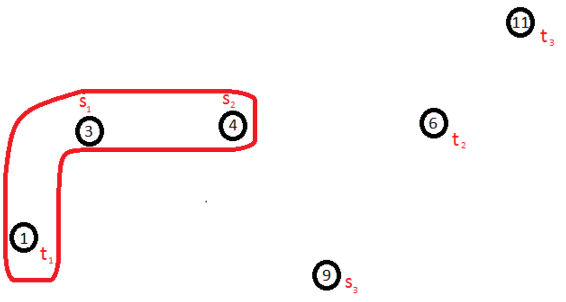

Figure 4.1.5: Illustration of union 1 and 3

Then, it checks 4 and 6. Since they are not in the same set, it adds this path into the solution and combines into one set as follows:

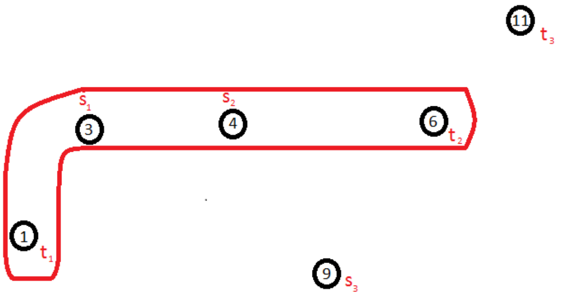

Figure 4.1.6: Illustration of union 4 and 6

Then, it does not add the paths 4-1 and 3-6 since they are already in the same set. Because 6 and 11 are different sets, it adds 6-11 path into the solution and combines them into one set as follows:

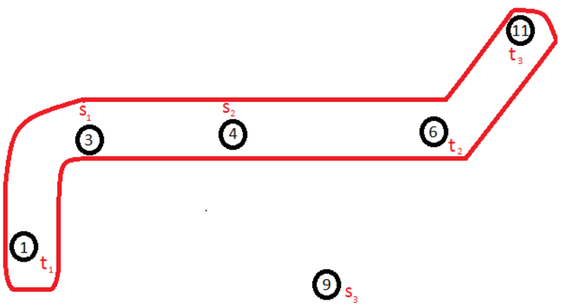

Figure 4.1.7: Illustration of union 6 and 11

Eventually, it adds 6-9 path into the solution since 6 and 9 sets are not in the same set. It then combines them into one set by using the union operation as shown in the figure below:

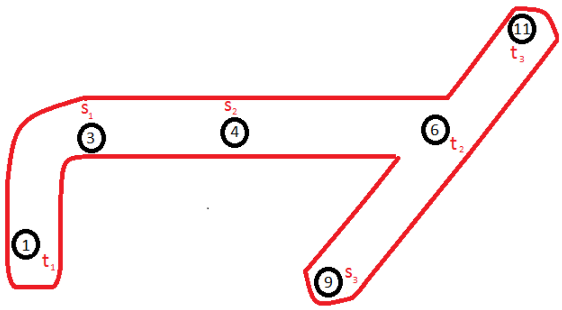

Figure 4.1.8: Union of the sets 6 and 9

Since all terminal pairs are connected at this point, the algorithm terminates. So far, the paths taken by the algorithm are (3-4), (3-1), (4-6), (6-11), (6-9).

Then, in order to exclude unnecessary paths from solution, the algorithm performs a reverse-delete operation. In this operation, it excludes the paths as long as the solution remains feasible, that is, if all terminal pairs are still connected. It tries to remove the paths in the reverse order in which they were added into the solution. We sequentially exclude the paths from the solution and then check whether k pairs (s_1-t_1, s_2-t_2,…, s_k-t_k) are still in the same set. If even only one terminal pair is not in the same set, it means that the excluded path is necessary for the solution so we can not remove this path. If all of the terminal pairs are still connected, it means that the excluded path is unnecessary and we can remove it from the solution.

For this example, we first check 6-9 path which is added last into the solution. If we remove this path, there is no connection between s_3-t_3 so we can't remove this path. We then check whether 6-11 path is removed from the solution and we see that we can not remove this path since there is no connection between s_3-t_3 when we remove this path. We then check whether third path 4-6 is removed, and so on. Only 3-4 path is unnecessary because when we exclude it, each terminal pair s_i-t_i for $1 \leq i \leq 3$ are still connected. So, we remove this path. If any of the remaining paths (3-1, 4-6, 6-11, 6-9) is removed, all of terminal pairs will still not be connected, so we don't remove any of them. For this example, our algorithm finds the solution which includes the paths 3-1, 4-6, 6-11, and 6-9.

We now provide the pseudocodes of the algorithm ÇDK-Kruskal, ÇDK-Prim and ÇDK-Boruvka.

Let H be an adjunct graph derived from the input graph G. H represents all the shortest paths between all terminals. It includes 2k vertices and $\binom{2k}{2}$ edges, that is, it is a complete graph. In the following, we give a procedure named ComputeAdjunct. This procedure forms the adjunct graph H. We will use this procedure in all of our algorithms.

Algorithm 4.1.1: ComputeAdjunct

G=(V, E), R = $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$, $w: E \to Q^+$
H = (V', E', w')
for i ← 1 to k **do**
    for j ← i+1 to k **do**
        c ← compute the cost of the shortest path between s_i *and* s_j
        H.AddEdge(s_i, s_j, c)
        c ← compute the cost of the shortest path between t_i *and* t_j
        H.AddEdge (t_i, t_j, c)
    for j ← 1 to k **do**
        c ← compute the cost of the shortest path between s_i *and* t_j
        H.AddEdge (s_i, t_j, c)
return H

We give pseudocodes of our algorithms ÇDK-Kruskal, ÇDK-Prim and ÇDK-Boruvka as follows:

Algorithm 4.1.2: ÇDK-Kruskal Algorithm

G=(V, E), R = $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$, $w\colon E \to Q^+$
F ← ∅ //Initialize Forest to empty
l ← 0
for each r ∈ R **do**
  MAKE-SET(r)
H = $(V',\ E',\ w')$ ← ComputeAdjunct(G, R, w)
$\sum[1..\binom{2k}{2}]$ ← sort the edges of E' in non-decreasing order by w'
for i ← 1 to $\binom{2k}{2}$ **do**
  p ← $\sum[i]$ //Let $\sum[i]$ be the path between u ∈ R and v ∈ R
  if FIND-SET(p.u) ≠ FIND-SET(p.v) **then**
    $l \leftarrow l + 1$
    F ← F ∪ $\{p_l\}$
    UNION(u,v)
  if all s_i and t_i are connected via F **then**
    break
for $k \leftarrow l$ down to 1 **do** //Reverse-Delete Step
  if $F' - p_k$ is a feasible solution **then**
    Remove p_k from F
return F

Algorithm 4.1.3: ÇDK-Prim Algorithm

G=(V, E), R = $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$, $w\colon E \to Q^+$, r
F ← ∅ //Initialize Forest to empty
l ← 0
S ← MAKE-SET(r)
H = $(V',\ E',\ w')$ ← ComputeAdjunct(G, R, w)
$\sum[1..\binom{2k}{2}]$ ← sort the edges of E' in non-decreasing order by w'
while not all (s_i, t_i) pairs are connected via F **do**
  $F' \leftarrow \emptyset$
  for each vertex u in S **do**
    $F' \leftarrow F'$ ∪ {the cheapest edge (u,v) in $\sum$ such that FIND-SET(u) ≠ FIND-SET(v) }
  $l \leftarrow l + 1$
  p_l ← the cheapest edge (u', v') in F'
  F ← F ∪ $\{p_l\}$
  UNION(u', v')
for $k \leftarrow l$ down to 1 **do** //Reverse-Delete Step
  if $F' - p_k$ is a feasible solution **then**
    Remove p_k from F
return F

Algorithm 4.1.4: ÇDK-Boruvka Algorithm

G=(V, E), R $= \{s_1, t_1, s_2, t_2, \dots, s_k, t_k\}, w: E \to Q^+$
F $\leftarrow \emptyset$ //Initialize Forest to empty
$l \leftarrow 0$
for each r $\in$ R **do**
  MAKE-SET(r)
H $= (V', E', w') \leftarrow$ ComputeAdjunct(G, R, w)
$\sum[1..\binom{2k}{2}] \leftarrow$ sort the edges of E' in non-decreasing order by w'
while not all (s_i, t_i) pairs are connected via F **do**
  for each set S **do**
    $F_S \leftarrow \emptyset$
    $F' \leftarrow F' \cup$ {the cheapest edge (u,v) in $\sum$ such that FIND-SET(u) $\neq$ FIND-SET(v) }
  $l \leftarrow l + 1$
  $p_l \leftarrow$ the cheapest edge (u', v') in F'
  F $\leftarrow$ F $\cup \{p_l\}$
  UNION(u', v')
for $k \leftarrow l$ down to 1 **do** //Reverse-Delete Step
  if $F' - p_k$ is a feasible solution **then**
    Remove p_k from F
return F

4.2 The Equivalence of the Algorithms

In this section, we give a proof that our algorithms are equivalent.

Let $E' = \{e_1, e_2, \dots, e_{\binom{2k}{2}}\}$ be the set of edges in increasing order. We argue by induction on l, the number of edges selected by ÇDK-Kruskal throughout its execution. For $l = 1$, ÇDK-Kruskal selects e_1. Let $e_1 = (u, v)$. Consider ÇDK-Boruvka and ÇDK-Prim at a stage in which u and v are not in the same set, and the edges that are adjacent to u and v are considered. Since e_1 is the smallest weight edge, it will be included in the forests that the aforementioned two algorithms compute by their very definition. This settles the base case of the induction. Assume, as the induction hypothesis that, before ÇDK-Kruskal selects the $(l + 1)$st edge, it has already selected the set $F_l = \{e_1, e_{i_2}, \dots, e_{i_l}\}$ and all the other edges up to e_{i_l} excluding F_l are not selected, and the set of selected edges and unselected edges are the same for ÇDK-Boruvka and ÇDK-Prim. If there is an edge between e_{i_l} and $e_{i_{l+1}}$ in the ordering, say $e = (u, v)$, then this edge is not selected by ÇDK-Kruskal because it creates a cycle, i.e. u and v are in the same set.

Consider the execution of ÇDK-Boruvka and ÇDK-Prim when this edge is considered. Since we know by induction hypothesis that u and v will be in the same set for these algorithms via the edges in F_l, e will not be selected. Otherwise, we get that one of the edges in F_l will not be selected by these algorithm which contradicts the induction hypothesis. All that remains is to show that $e_{i_{l+1}}$ is selected ÇDK-Boruvka and ÇDK-Prim. Let $e_{i_{l+1}} = (u, v)$ and consider the stage in which these algorithms consider the edges adjacent to u and v (clearly, u and v are not in the same set). Since, $e_{i_{l+1}}$ is the smallest weight edge which does not create a cycle by the choice of ÇDK-Kruskal, it will also be selected by the other two algorithms. Finally, note that the termination condition of all the algorithms is equivalent: all the terminal pairs are connected.

4.3 A Bad Example for Our Algorithms

In this section, we give a particular example on which our algorithms do not even give a constant factor approximation. Suppose that we have a graph with nodes $u_1, u_2, u_3, \dots, u_k$. The distances between u_i and u_{i+1} are 1 for $i = 1, \dots, k-2$ and u_k has distance 2 from all other nodes. Then, replace $u_1, u_2, u_3, \dots, u_k$ with $s_i - t_i$ pairs with distance epsilon between them except u_k which should be replaced with $s_k - t_k$ with distance 3. The example is shown below:

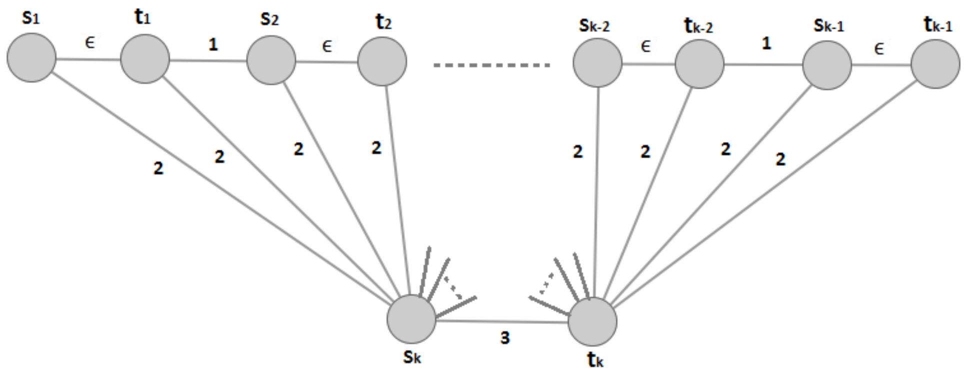

Figure 4.3.1: Bad example for ÇDK-Kruskal

For this example, our algorithms take all paths (k-1 paths) with distance epsilon. Then it takes all paths (k-2 paths) with distance 1. Then, it takes 2 paths with distance 2. The solution found by our algorithms is shown below:

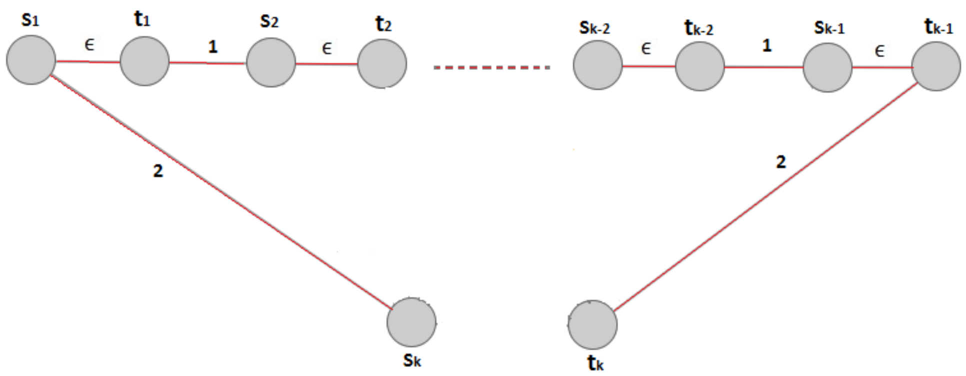

Figure 4.3.2: Illustration of the forest returned by ÇDK-Kruskal

The cost of the solution found by our algorithms is $(k-1) * \epsilon + (k-2) + 2 * 2 = (k-1) * \epsilon + (k+2)$. However; the optimum solution is $3 + (k-1) * \epsilon$. The optimum solution is shown below:

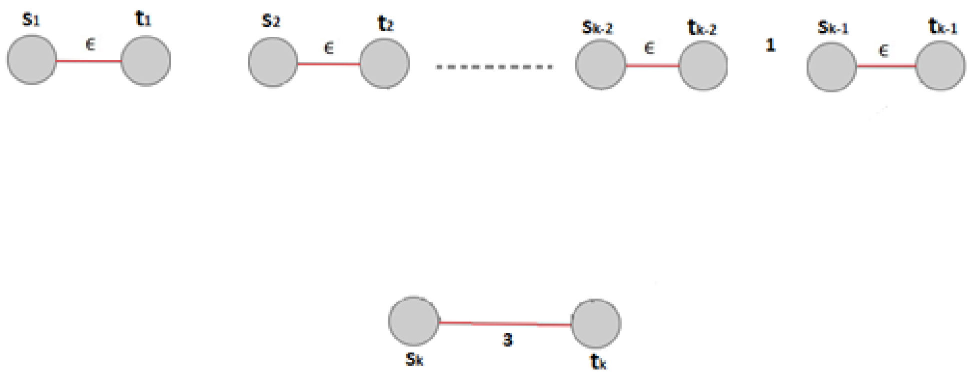

Figure 4.3.3: Optimum forest

CHAPTER 5

EXPERIMENTAL RESULTS

In this section, we discuss experiments that we have conducted by running Greedy Heuristic, ÇDK-Kruskal and AKR. The experiments are performed on a computer with specifications as listed below:

Processor	: Intel Core i5=3470 CPU @ 3.20GHz
Ram	: 4 GB
System type	: 64 bit operating system
Operating System	: Windows 7 64 bit

The implementations of the algorithms are done on Java NetBeans IDE 7.4 platform. In the following, we first summarize these three algorithms and then discuss the condition in which AKR work slower.

How does ÇDK-Kruskal work?

Firstly, it makes a set for each terminal and then calculates the shortest paths between all the terminals. There are $C\binom{2k}{2}$ shortest paths that need to be calculated. Then it sequentially checks all the paths from the smallest weight to the largest. If the endpoints of the path checked are not in the same set, it includes the path into the forest and combines these two sets into one set. This process is continuously done until all s-t pairs are connected.

How does AKR work?

At the beginning, it forms active sets for all the terminals. Then it determines the shortest edge that comes out from the active sets since the smallest edge will become tight first. If nodes that are at the endpoints of this edge are not in the same set, the edge is included to the forest and the nodes of this edge is included to the active set. Then, the edges are updated by substracting the weight of the selected edge from the weights of all the edges. This process is performed iteratively until all s-t pairs are connected.

How does Greedy Heuristic works?

In the first iteration, it calculates the shortest path between s_1 and t_1. In the next iteration, the cost of the edges that are not zero are included to the resulting forest. Then, it changes the value of the selected edges to zero, that is, it contracts the s_1-t_1 pair. In the next iteration, it applies the same procedure to the s_2-t_2 pair. These iterations are performed up to $s_k - t_k$.

5.1 Experiment 1

In all of our experiments, we compare the solution costs and the running times of the three algorithms, showing them on a table and also providing a figure for ease of resresentation.

In this specific experiment

- We created a random graph in the Erdös-Renyi model with 1000 nodes.
- The probabilities of having an edge between two nodes are 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9. For each probability value, a graph with 1000 nodes is created. Hence, using the probability values, we created 9 different random graphs.
- Edge weights are integers and randomly choosen between 1 and 10000.
- We formed k random s-t pairs for k={2, 3, 5, 10, 20, 30, 40 and 50}. For instance, for k=2, we selected 2 s-t pairs (4 terminals) randomly from the nodes in the graph.

Greedy Heuristic, ÇDK-Kruskal, and AKR have fairly similar costs with AKR being slightly better. However, if we take a look at their running time, we can see that Greedy Heuristic and ÇDK-Kruskal are closer to each other than they are to AKR. Greedy Heuristic is faster than ÇDK-Kruskal since it does not compute an adjunct graph. As for AKR, it works much slower than these two algorithms. The reason is that, when edge weights span on wide interval, which in this case 1-10000, causes AKR to execute too many iterations. Recall that, AKR updates every node and edge by substacting the weight of the newly added edge from their weights. So, the more varied the edge weights, the larger the number of updates, which slows down the algorithm.

Table 5.1.1: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.1 and Edge Weights 1-10000

k	Probability of edge existance: 0.1 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1058	1091	1086	156	103	1290
3	2653	3687	3313	59	97	727
5	4072	5258	5041	83	195	3521
10	6870	7626	7127	162	324	2822
20	13758	14423	13373	208	681	18516
30	18341	19494	17946	429	1135	3660
40	24283	26415	24655	427	1312	18225
50	25585	25810	23821	722	1646	6631

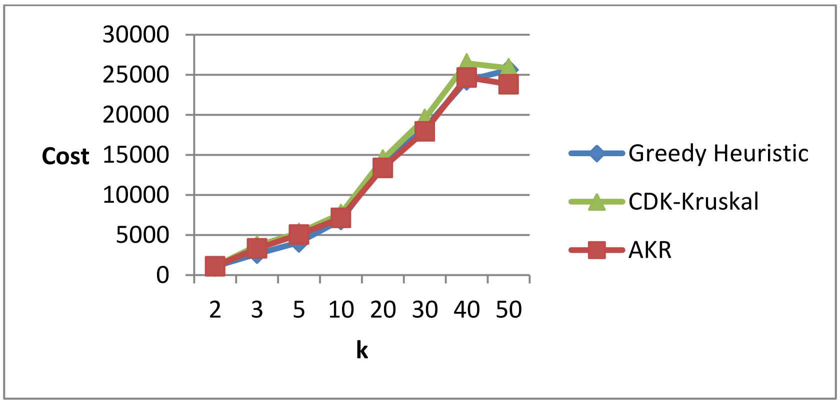

Figure 5.1.1: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.1

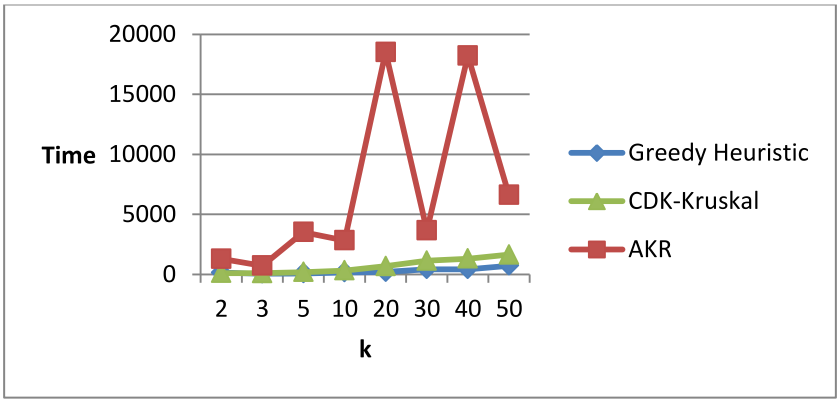

Figure 5.1.2: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.1

Table 5.1.2: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.2 and Edge Weights 1-10000

k	Probability of edge existance: 0.2 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	506	506	506	53	76	123
3	1186	1273	1483	91	214	582
5	1926	2179	1975	122	361	3588
10	3385	3779	3509	298	600	1756
20	7614	7855	7169	353	975	11391
30	8412	9132	8355	552	2011	4939
40	10176	10221	9505	769	2509	11042
50	11842	12443	11718	965	2919	30617

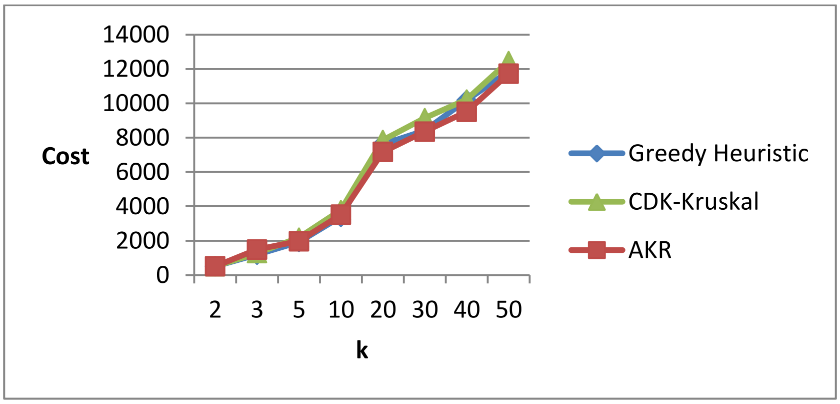

Figure 5.1.3: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.2

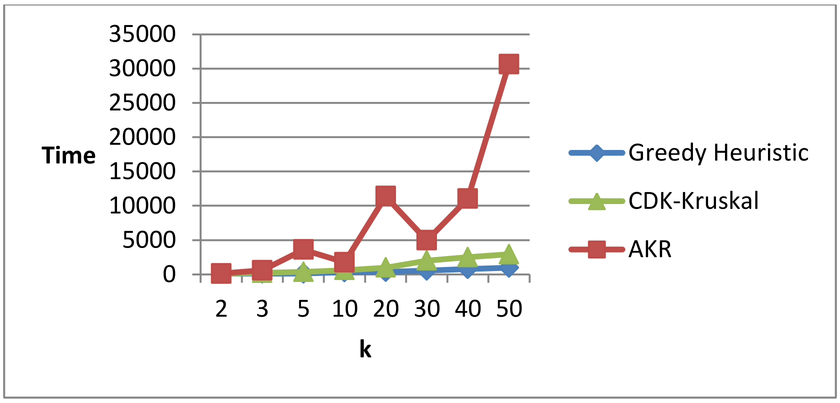

Figure 5.1.4: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.2

Table 5.1.3: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.3 and Edge Weights 1-10000

k	Probability of edge existance: 0.3 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	466	471	466	68	140	4181
3	849	1134	1134	125	241	787
5	1216	1560	1341	203	352	4179
10	2988	3464	3024	265	912	4184
20	4611	4683	4325	560	2090	4946
30	6263	6720	6243	689	2848	14917
40	7174	7285	6709	1109	3170	6153
50	10031	10234	9572	1273	3967	31722

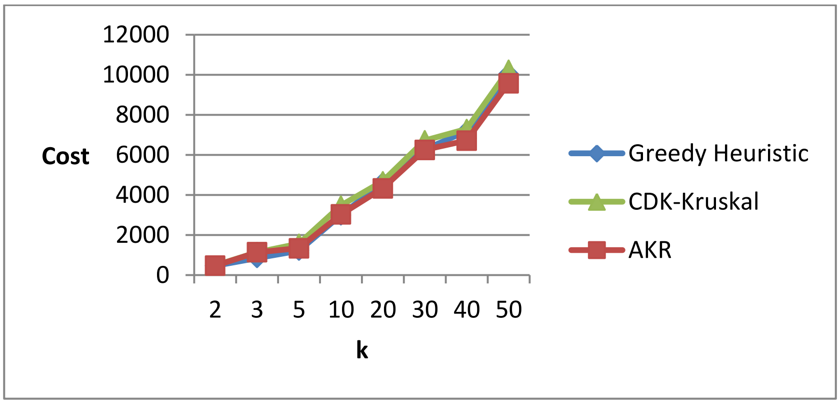

Figure 5.1.5: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.3

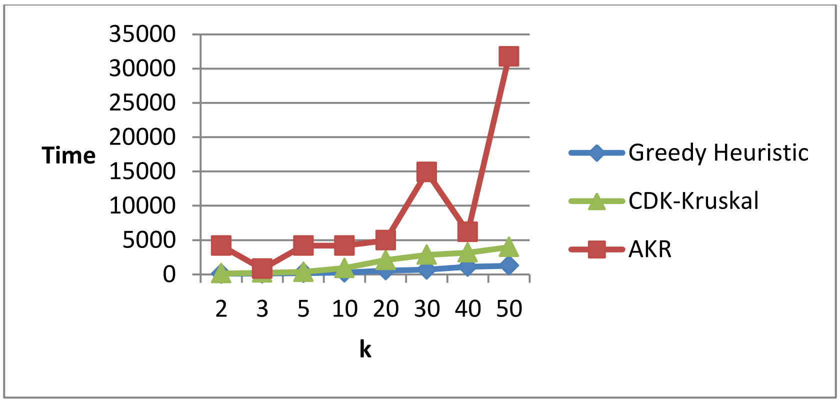

Figure 5.1.6: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.3

Table 5.1.4: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.4 and Edge Weights 1-10000

k	Probability of edge existance: 0.4 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	327	353	325	208	188	590
3	502	628	502	144	359	804
5	976	1338	1198	174	529	3653
10	2128	2561	2323	327	1206	5216
20	2983	3114	2803	655	2406	4479
30	4504	5114	4532	1745	3278	6892
40	5632	5715	5330	1315	3922	20575
50	6396	6490	6028	1669	5228	11751

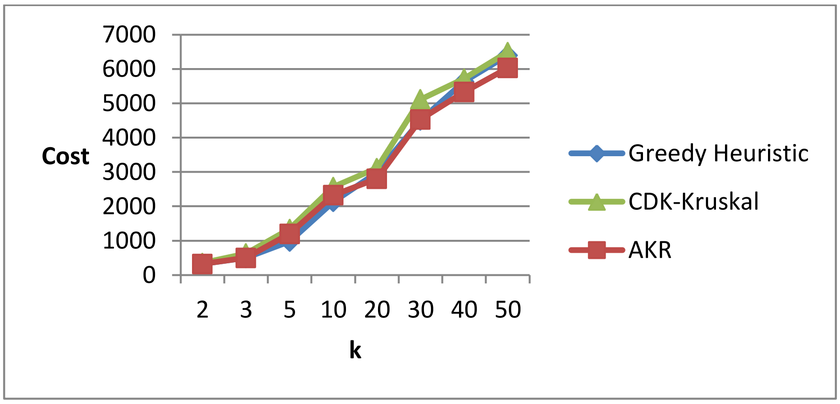

Figure 5.1.7: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.4

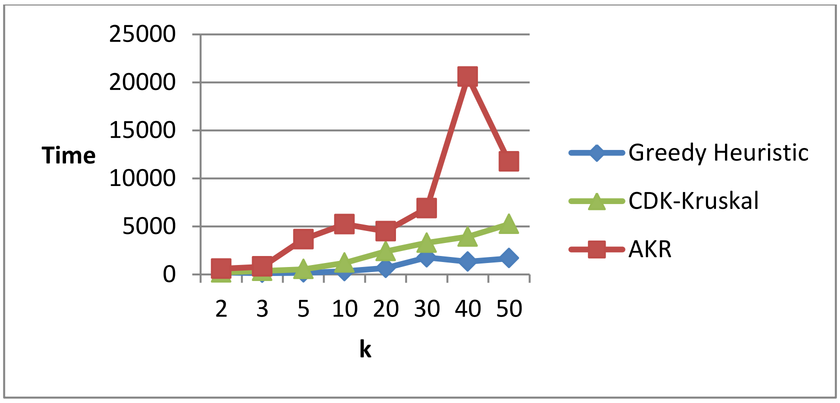

Figure 5.1.8: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.4

Table 5.1.5: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.5 and Edge Weights 1-10000

k	Probability of edge existance: 0.5 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	324	399	351	111	289	460
3	510	583	503	146	429	633
5	719	864	759	213	664	3332
10	1687	2092	1904	380	1332	6559
20	3032	3068	2819	903	2889	7527
30	4157	4513	4226	1863	3684	18653
40	4819	5071	4722	1331	4956	24492
50	5851	5989	5470	1818	6414	17499

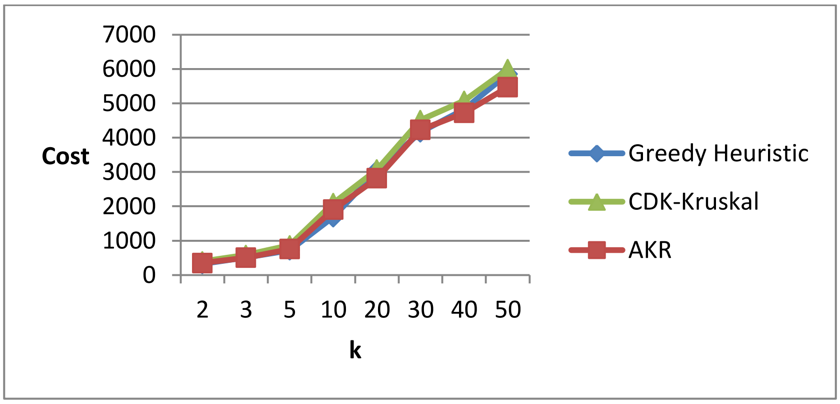

Figure 5.1.9: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.5

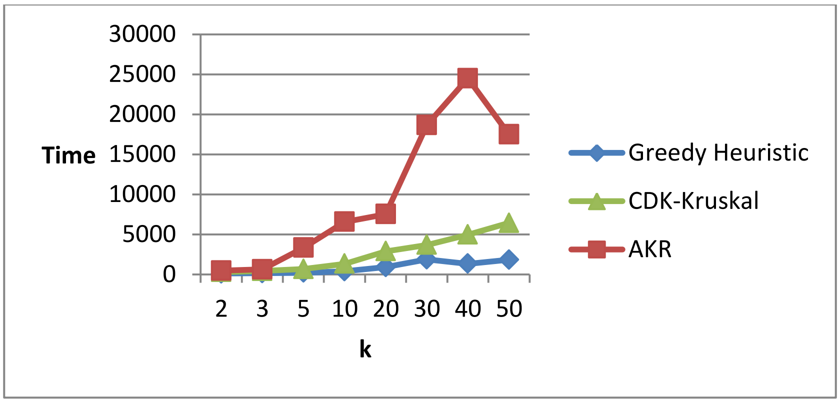

Figure 5.1.10: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.5

Table 5.1.6: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.6 and Edge Weights 1-10000

k	Probability of edge existance: 0.6 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	294	435	405	117	289	1058
3	349	460	418	163	480	1119
5	607	725	688	342	743	2877
10	1322	1362	1230	493	1797	3439
20	2185	2327	2151	1034	3792	6977
30	3138	3493	3172	1467	5180	9271
40	3955	4157	3696	1804	7193	23761
50	4257	4311	3994	2519	7946	6069

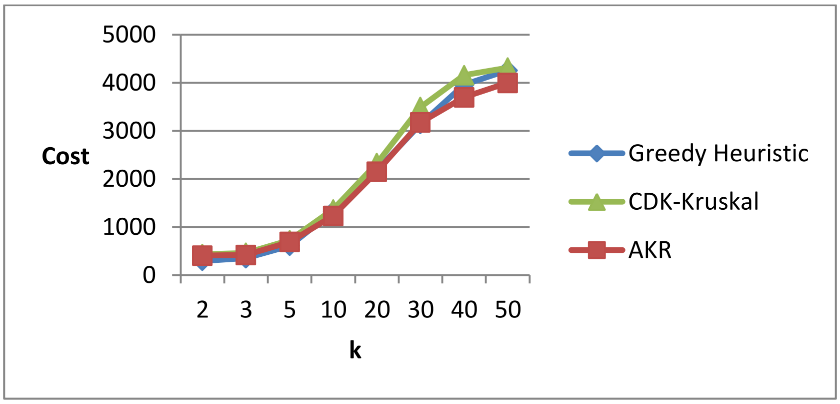

Figure 5.1.11: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.6

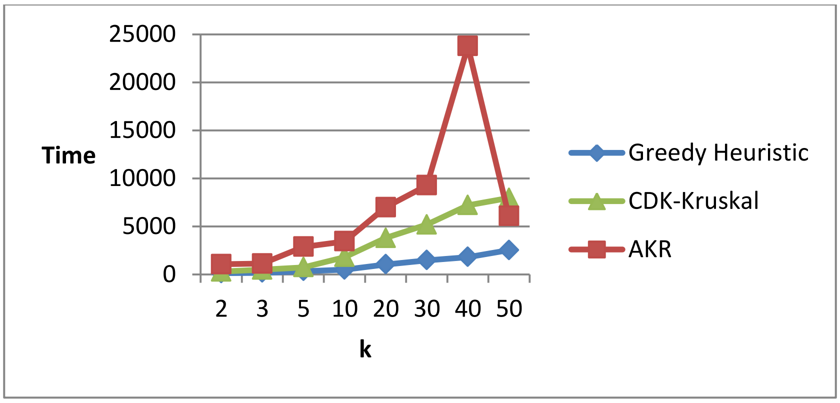

Figure 5.1.12: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.6

Table 5.1.7: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.7 and Edge Weights 1-10000

k	Probability of edge existance: 0.7 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	169	169	169	1382	291	858
3	372	426	394	176	493	1164
5	406	546	482	339	858	699
10	1004	1199	1133	585	2156	2511
20	1796	1888	1702	1085	3782	4483
30	2645	2870	2581	1618	5828	8980
40	3386	3485	3214	1951	7237	16816
50	4203	4335	4004	2736	9424	20525

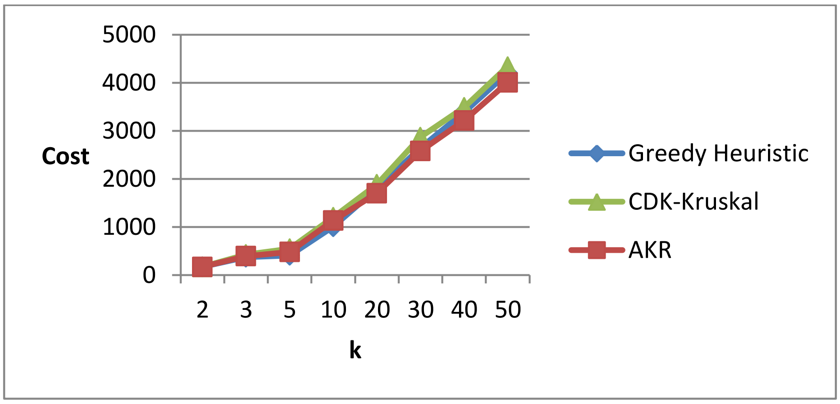

Figure 5.1.13: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.7

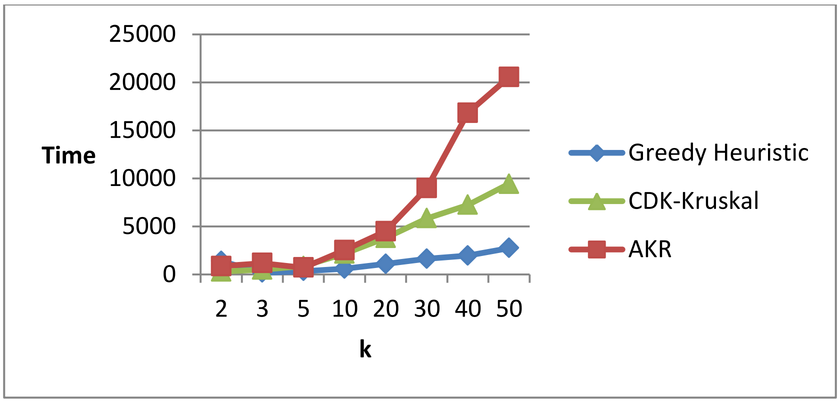

Figure 5.1.14: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.7

Table 5.1.8: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.8 and Edge Weights 1-10000

k	Probability of edge existance: 0.8 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	204	272	261	147	415	1064
3	233	288	282	274	467	2327
5	507	585	557	402	993	2224
10	849	978	920	663	2377	7722
20	1874	2048	1861	1275	4714	6753
30	2300	2434	2261	1868	6502	27047
40	2882	2915	2641	2470	8607	9482
50	3458	3423	3199	2894	10606	9738

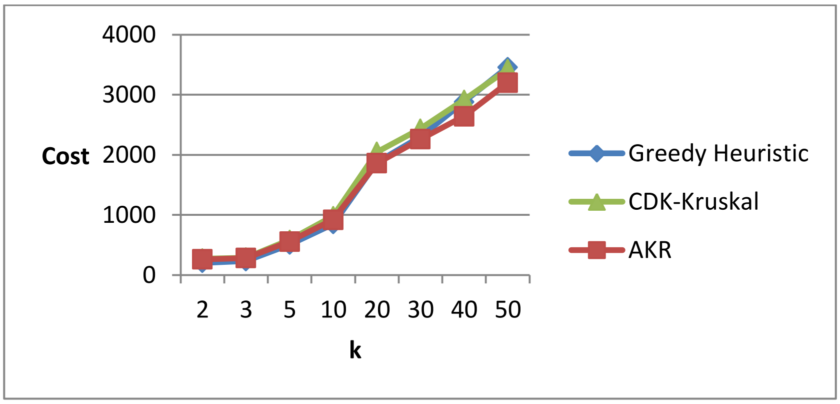

Figure 5.1.15: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.8

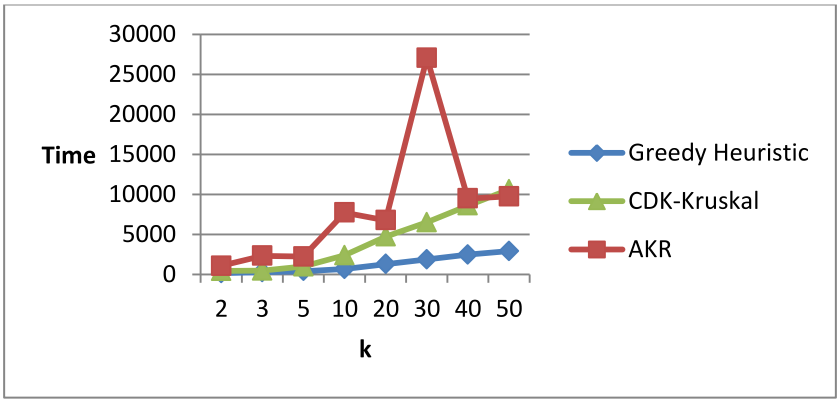

Figure 5.1.16: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.8

Table 5.1.9: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.9 and Edge Weights 1-10000

k	Probability of edge existance: 0.9 Edge weights: 1-10000					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	136	201	136	147	410	551
3	240	324	254	194	608	252
5	479	659	593	359	1127	3858
10	890	950	887	632	2348	6145
20	1498	1658	1496	1279	4796	14068
30	1967	2203	2005	1684	6464	14657
40	2428	2528	2335	2373	9074	22726
50	2472	2638	2447	3082	11750	15493

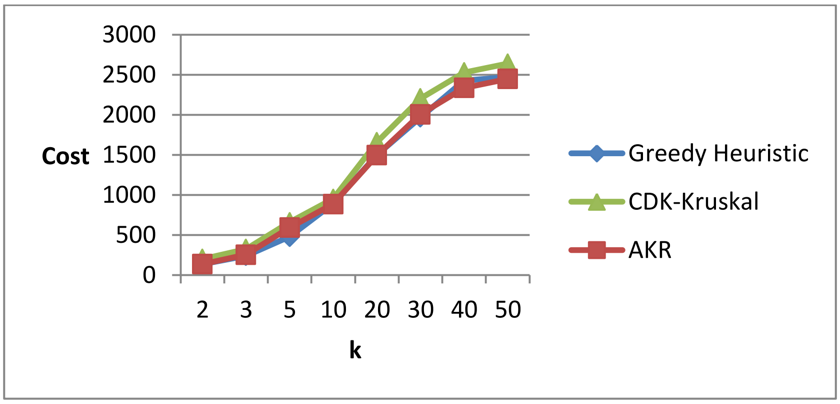

Figure 5.1.17: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.9

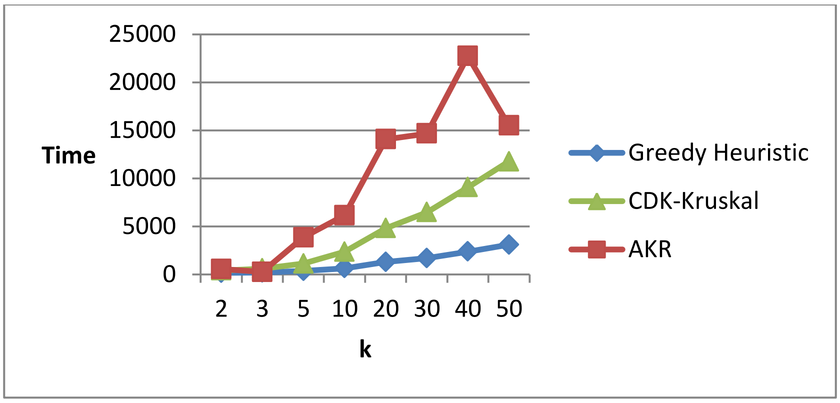

Figure 5.1.18: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.9

5.2 Experiment 2

In this experiment

- We created a random graph in the Erdös-Renyi model with 1000 nodes.
- The probabilities of having an edge between two nodes are 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 and 0.9. For each probability value, a graph with 1000 nodes is created. Hence, using the probability values, we created 9 different random graphs.
- Edge weights are integers and randomly choosen between 1 and 100.
- We formed k random s-t pairs for k={2, 3, 5, 10, 20, 30, 40 and 50. For instance, for k=2, we selected 2 s-t pairs (4 terminals) randomly from the nodes in the graph.

Specifications of this experiment are the same with first experiment except the edge weight interval. In this case, while the cost results of the three algorithms are still similar, the running times are significantly different. In this experiment AKR is the fastest algorithm.While the edge weight interval is smaller, AKR does a smaller number of updates, which makes its running time better compared to the previous experiment. In general, we see that ÇDK-Kruskal is not well suited for these type of graphs.

Table 5.2.1: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.1 and Edge Weights 1-100

k	Probability of edge existance: 0.1 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	22	26	24	163	99	141
3	35	40	37	57	97	464
5	39	48	43	101	126	105
10	106	108	105	126	297	229
20	176	198	179	230	592	146
30	271	306	280	272	818	182
40	331	356	327	483	1053	333
50	399	387	365	565	1302	219

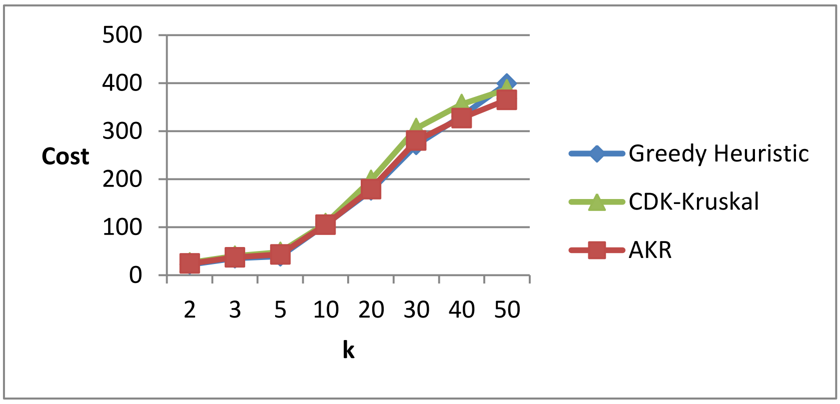

Figure 5.2.1: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.1

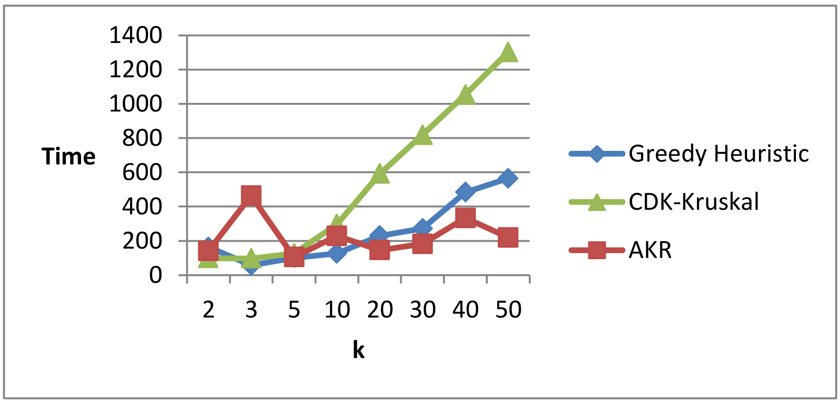

Figure 5.2.2: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.1

Table 5.2.2: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.2 and Edge Weights 1-100

k	Probability of edge existance: 0.2 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	14	14	14	118	77	87
3	21	31	27	97	259	135
5	31	37	35	136	411	114
10	64	71	66	218	707	182
20	112	118	112	380	1350	182
30	168	173	159	594	2006	289
40	227	232	217	1040	1950	282
50	270	257	241	869	2917	209

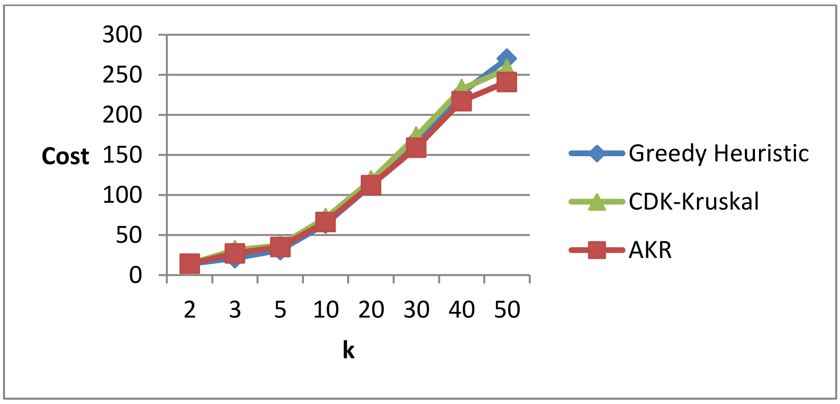

Figure 5.2.3: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.2

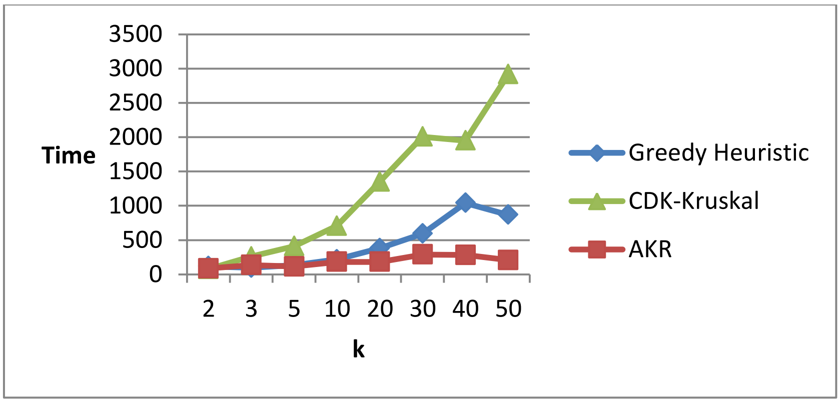

Figure 5.2.4: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.2

Table 5.2.3: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0. and Edge Weights 1-100

k	Probability of edge existance: 0.3 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	9	14	14	75	163	67
3	14	17	14	99	232	83
5	29	34	32	159	487	96
10	51	55	50	324	1147	166
20	103	107	99	409	1481	298
30	146	148	137	593	2134	293
40	176	164	153	745	2779	336
50	223	212	202	1317	3413	419

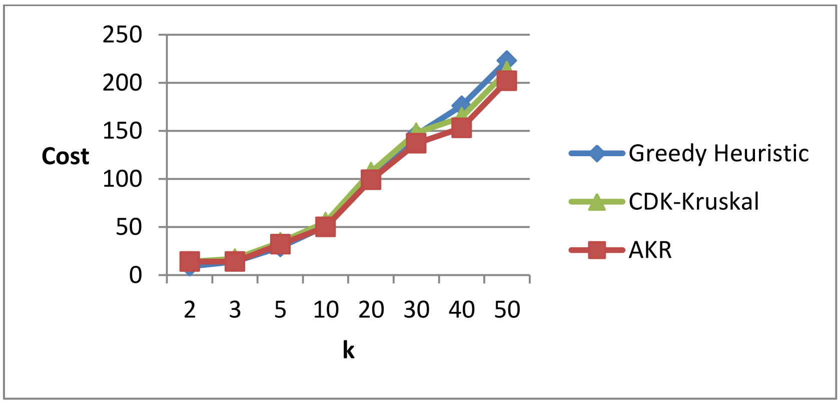

Figure 5.2.5: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.3

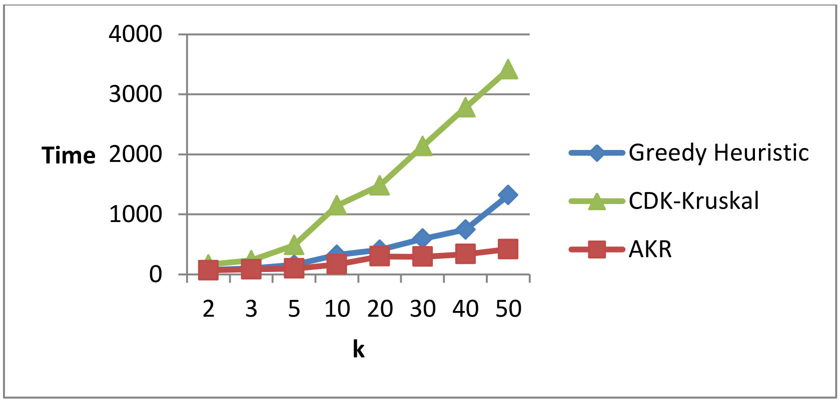

Figure 5.2.6: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.3

Table 5.2.4: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.4 and Edge Weights 1-100

k	Probability of edge existance: 0.4 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	11	12	11	104	263	538
3	14	17	14	144	413	685
5	20	31	30	207	676	194
10	46	54	51	413	1313	89
20	95	98	90	1096	2042	279
30	124	132	127	980	3401	179
40	155	148	140	1167	4364	211
50	209	188	181	1270	4959	543

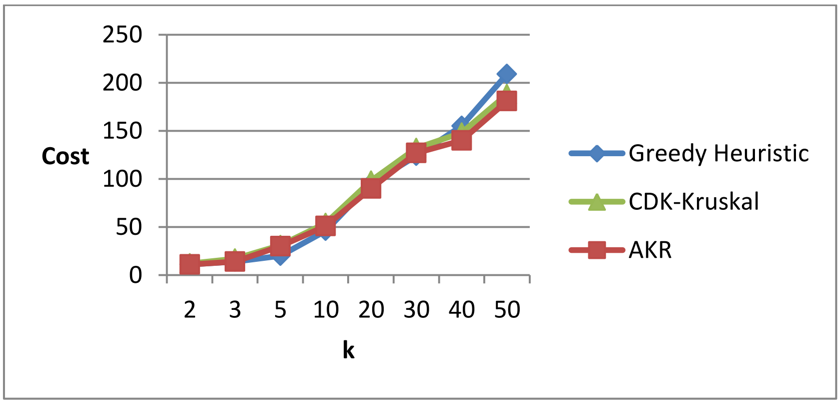

Figure 5.2.7: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.4

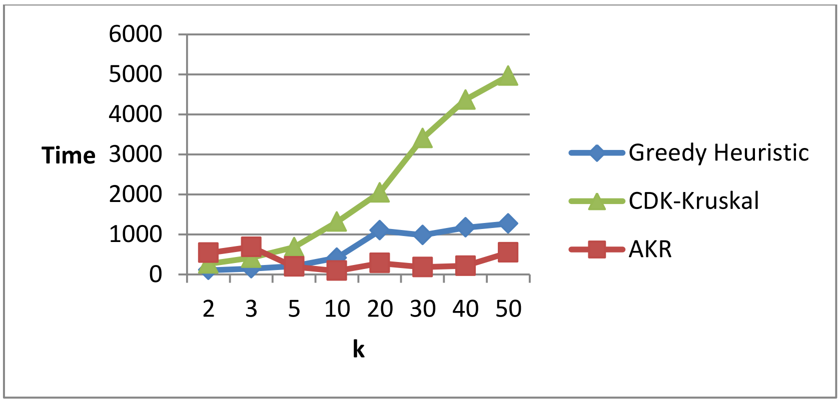

Figure 5.2.8: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.4

Table 5.2.5: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.5 and Edge Weights 1-100

k	Probability of edge existance: 0.5 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	9	11	11	103	254	95
3	14	16	16	170	504	195
5	18	27	26	222	724	85
10	38	44	45	438	1447	423
20	80	90	88	923	3385	149
30	121	117	111	1393	4577	213
40	160	159	154	1620	4940	364
50	195	179	173	2188	7126	395

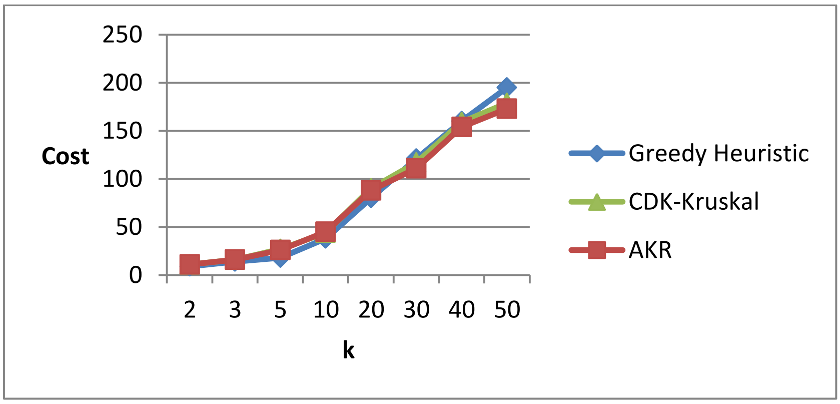

Figure 5.2.9: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.5

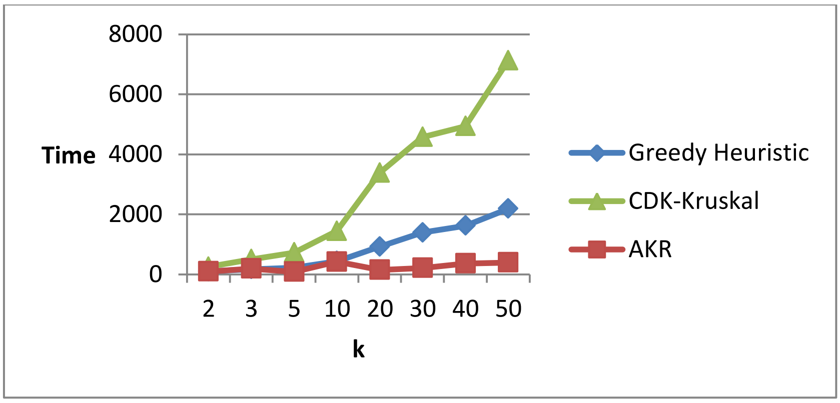

Figure 5.2.10: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.5

Table 5.2.6: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.6 and Edge Weights 1-100

k	Probability of edge existance: 0.6 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	8	12	12	112	287	111
3	11	14	14	169	511	162
5	22	26	23	292	997	296
10	33	43	37	431	1576	113
20	76	77	76	958	3565	302
30	109	113	108	1292	4658	297
40	135	141	132	1592	5491	410
50	176	157	153	2302	8643	415

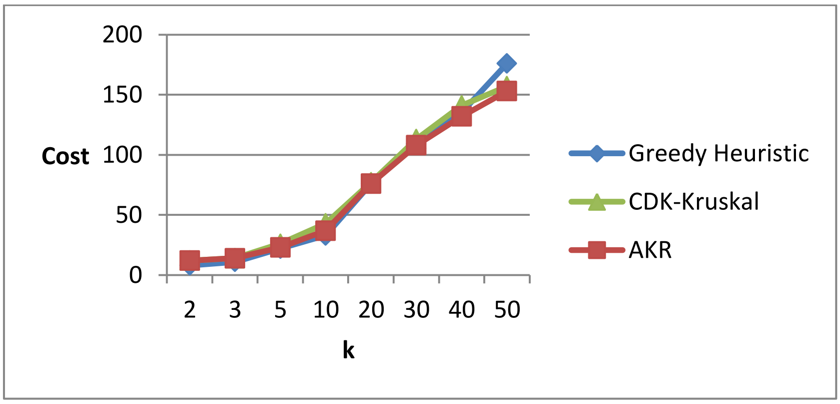

Figure 5.2.11: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.6

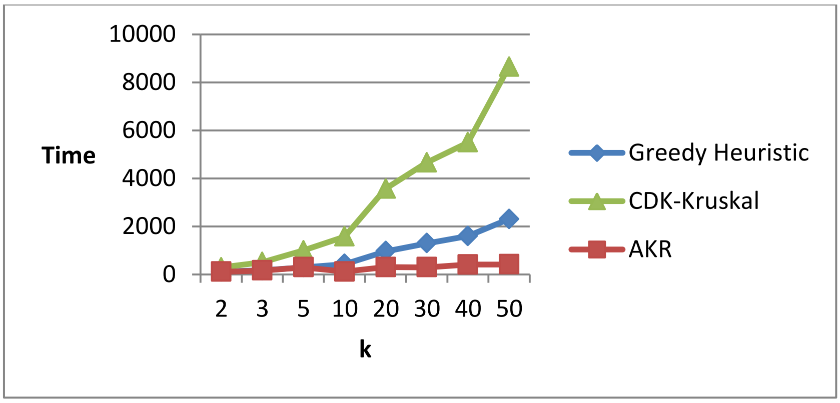

Figure 5.2.12: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.6

Table 5.2.7: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.7 and Edge Weights 1-100

k	Probability of edge existance: 0.7 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	8	11	8	128	346	127
3	11	17	16	172	522	186
5	17	23	21	342	1093	107
10	31	37	37	606	2257	275
20	66	75	74	1077	3974	362
30	100	107	102	1605	6223	451
40	135	133	129	2311	8056	453
50	159	154	149	2462	9240	534

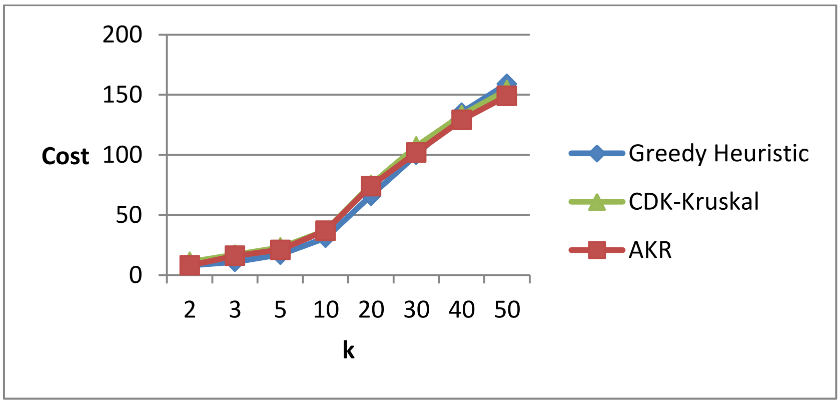

Figure 5.2.13: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.7

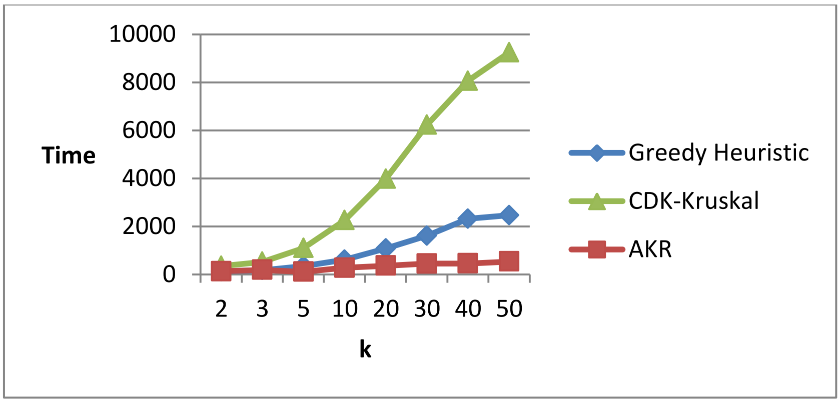

Figure 5.2.14: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.7

Table 5.2.8: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.8 and Edge Weights 1-100

k	Probability of edge existance: 0.8 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	7	7	6	128	346	76
3	10	12	14	218	616	336
5	16	27	25	302	1031	107
10	35	46	42	620	2208	164
20	68	75	73	1380	3615	304
30	98	106	101	1991	5914	401
40	125	126	120	2012	7123	394
50	153	146	145	2982	10982	399

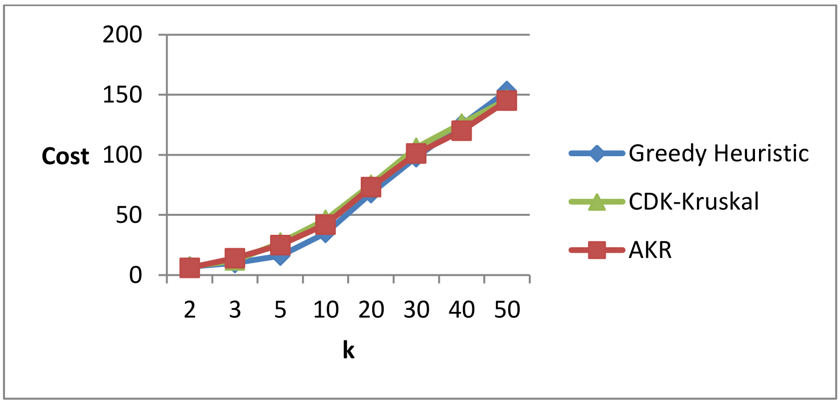

Figure 5.2.15: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.8

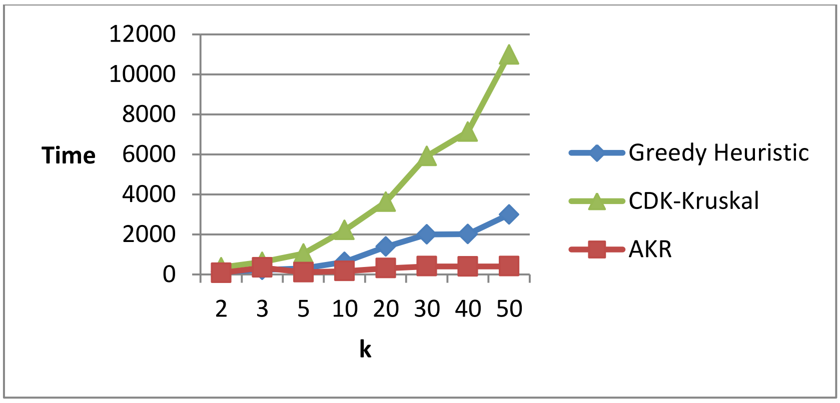

Figure 5.2.16: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.8

Table 5.2.9: Results of Cost and Running Time of Three Algorithms with Probability of Edge Existence 0.9 and Edge Weights 1-100

k	Probability of edge existance: 0.9 Edge weights: 1-100					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	8	10	9	148	413	224
3	11	15	15	204	636	266
5	13	19	19	305	1047	129
10	32	43	40	720	2175	203
20	65	78	76	1292	4380	346
30	90	100	97	1766	6816	1200
40	126	122	117	2429	8984	476
50	157	147	143	3490	10872	510

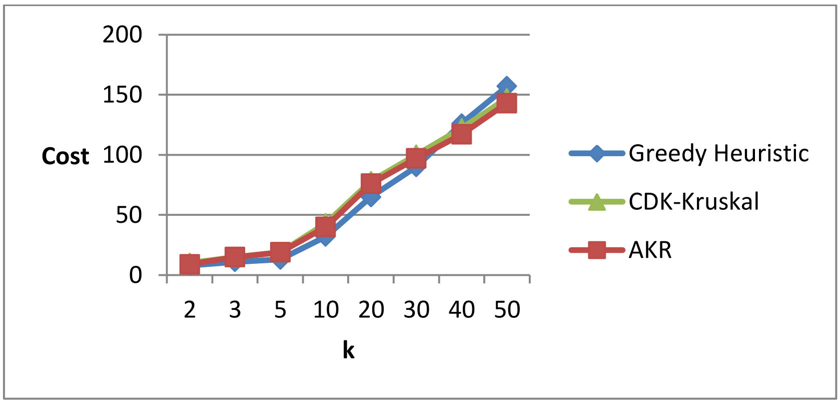

Figure 5.2.17: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.9

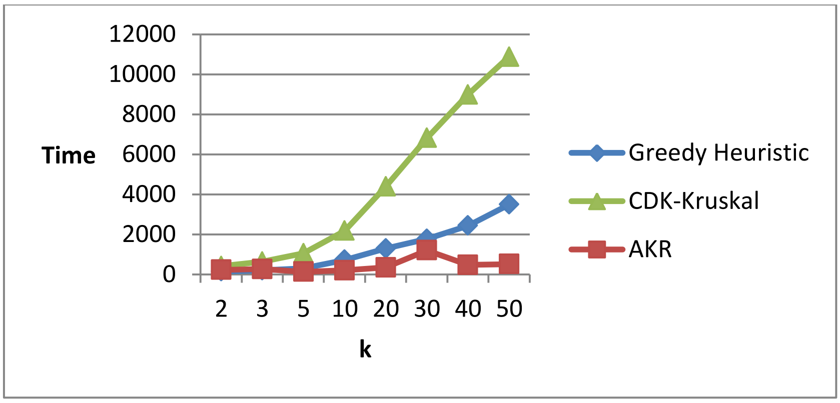

Figure 5.2.18: Running Time Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on Random Graph with Probability of Edge Existance 0.9

5.3 Experiment 3

In this experiment, we test the algorithms on a real-world geometric graph.

- We have obtained our data from the National TSP Collection website [19]. In this website, there exist TSP data of 25 different countries. Among these countries, we have chosen Uruguay that has 734 cities. We would like to note that the results that we have derived from several other countries are similar to the one we present here. So, we have decided that it is sufficient to give results for a single country.
- We created a geometric graph in the usual sense. First, we calculated the maximum distance between any two cities. Then, we multiply the maximum distance with 0.1, 0.2, 0.4, 0.6, 0.8, and 1.0. If the distance between two cities is less than the result of this multiplication, we include the edge into the graph. Note that for multiplication with 1.0, we have a complete graph.
- The data that we get from the website is fractional, but here while creating the graph, we round them to the nearest whole number.

We see from the results that our algorithm ÇDK-Kruskal has a very good running time compared to AKR albeit they have almost identical costs. Also ÇDK-Kruskal and AKR give better cost results than Greedy Heuristic.

The reason of high running time of AKR is the same as the reason we have mentioned in Experiment 1: the edge weights span a wide interval in real-world geometric graphs. In general, we see that ÇDK-Kruskal is a very good alternative to AKR in real-world geometric graphs since it gives comparable solution costs, but its running time is much better.

Figure 5.3.1: The Map of Uruguay [20]

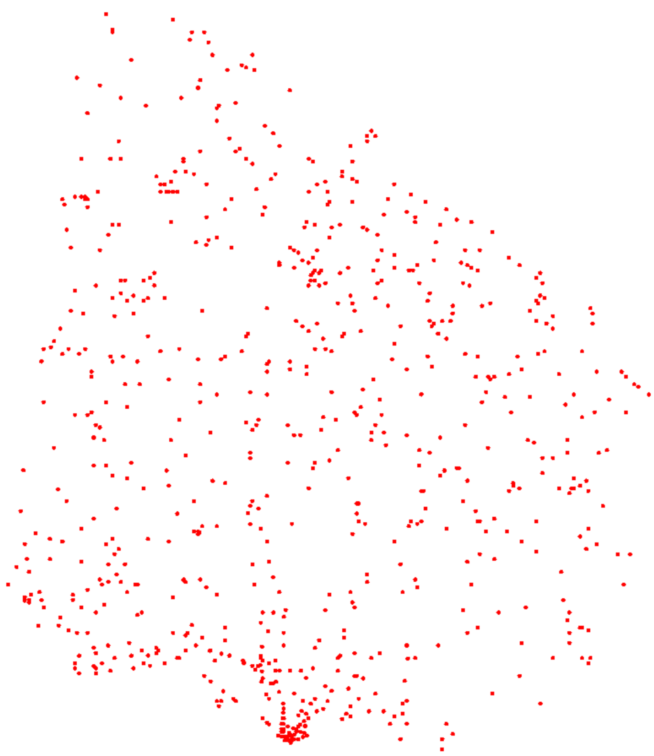

Figure 5.3.2: The Point Set of Uruguay [21]

Table 5.3.1: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 0,1 times maximum distances

k	The biggest weight x 0.1					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	3525	3525	3525	286	72	2302
3	6387	7309	7309	31	51	4081
5	7065	6710	6710	38	85	1756
10	15520	14632	14632	71	167	6035
20	20360	17729	17729	101	292	3677
30	25529	21116	21116	135	474	3300
40	31431	25409	25409	196	718	3420
50	34753	27183	27183	217	635	5219

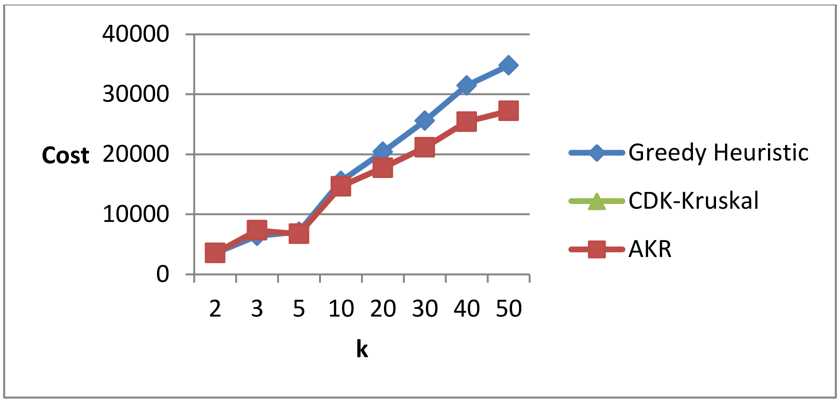

Figure 5.3.3: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,1

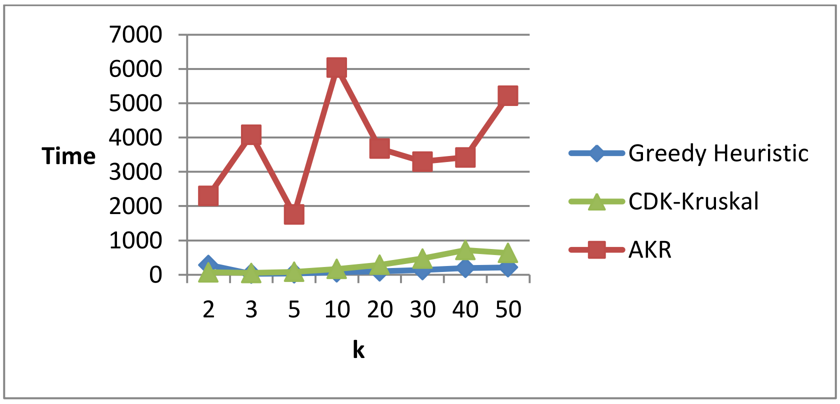

Figure 5.3.4: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,1

Table 5.3.2: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 0,2 times maximum distances

k	The biggest weight x 0.2					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	6687	6845	6845	44	99	27969
3	4932	4973	4973	61	172	10533
5	8219	6897	6897	86	263	11889
10	14913	11964	11964	164	585	22107
20	20798	16641	16641	308	1092	26068
30	25405	19186	19186	443	1685	14505
40	34036	26815	26815	602	2259	24731
50	34858	27091	27091	784	2970	27661

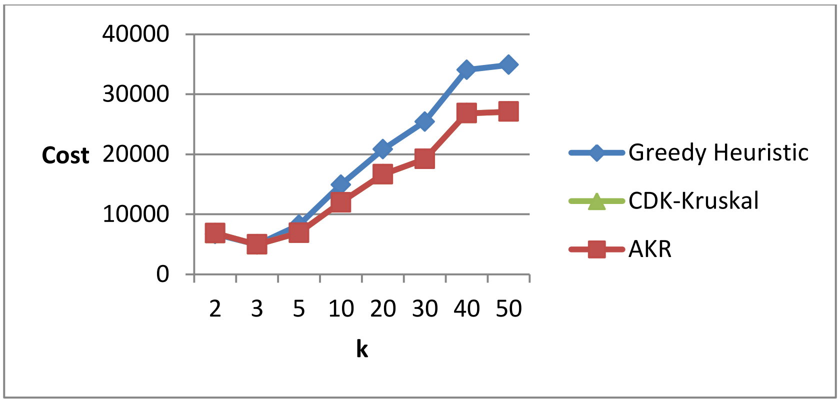

Figure 5.3.5: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,2

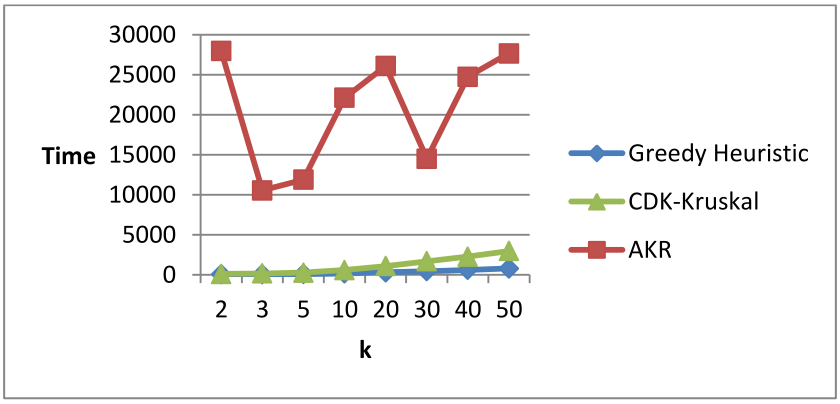

Figure 5.3.6: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,2

Table 5.3.3: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 0,4 times maximum distances

k	The biggest weight x 0.4					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	4876	5410	5410	66	159	155856
3	6185	6283	6467	107	310	90351
5	7745	6748	6748	173	535	147420
10	13681	11756	11756	250	915	67835
20	19037	16151	16151	684	2515	78827
30	27353	22401	22401	643	2159	88209
40	33068	24000	24000	1520	3652	49592
50	35003	26569	26569	1359	4681	84602

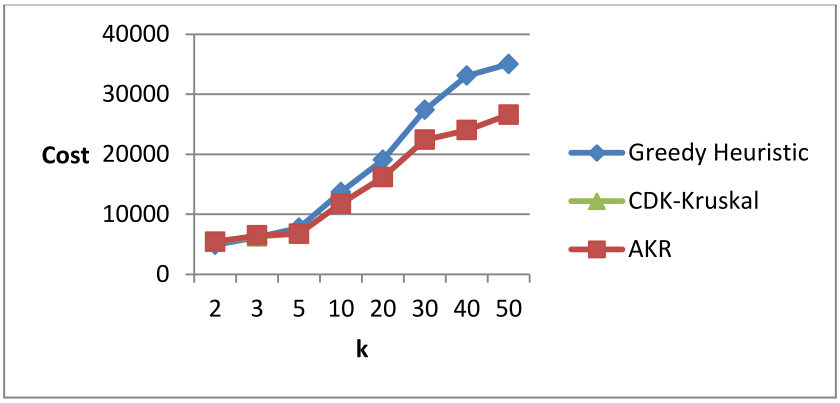

Figure 5.3.4: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,4

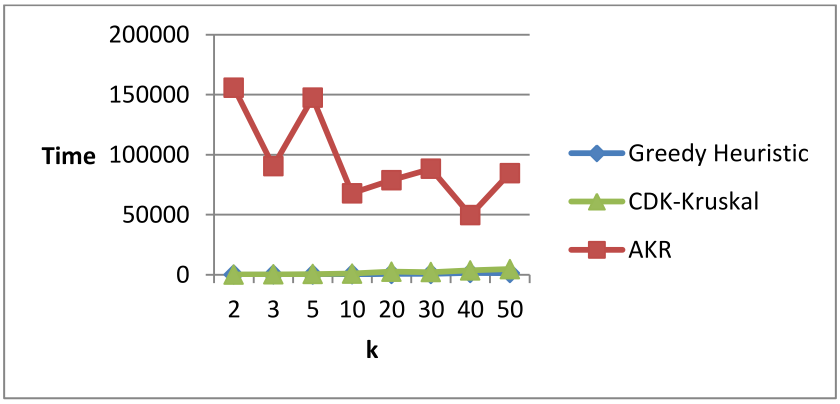

Figure 5.3.8: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,4

Table 5.3.4: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 0,6 times maximum distances

k	The biggest weight x 0.6					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1651	1707	1707	99	253	2405
3	5153	5124	5124	206	693	56888
5	7739	7286	7286	259	892	170808
10	15296	11857	11857	612	2284	69025
20	22411	17693	17693	655	2137	124553
30	27970	19848	19848	1294	5163	80048
40	29842	22701	22701	2765	9873	116657
50	35897	26800	26800	3344	12775	110849

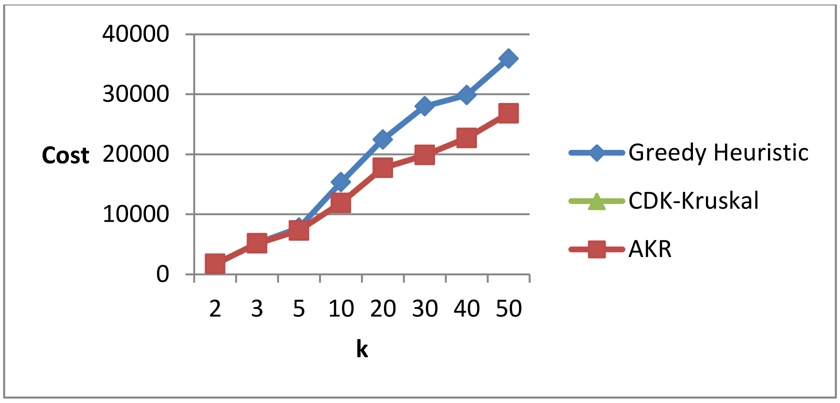

Figure 5.3.9: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,6

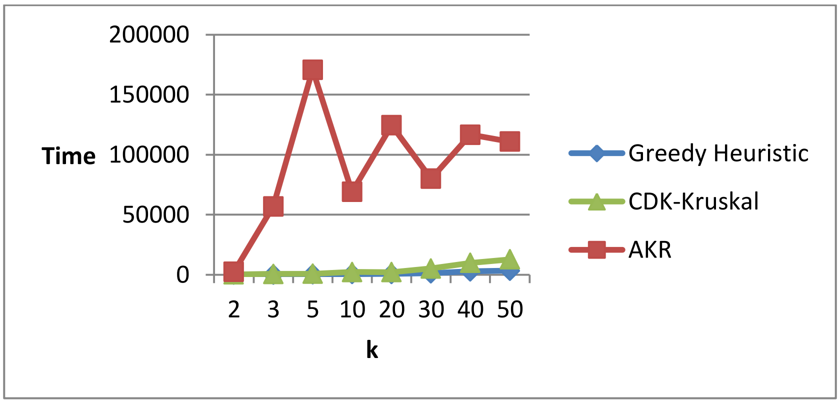

Figure 5.3.10: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,6

Table 5.3.5: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 0,8 times maximum distances

k	The biggest weight x 0.8					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	4140	4487	4487	101	286	98610
3	5305	5879	4936	245	839	61061
5	9323	8414	8414	169	612	268368
10	16664	13736	13736	657	2476	210610
20	22708	18032	17638	1219	4511	238231
30	27719	21100	21100	1767	6802	166786
40	38726	26709	26709	2443	7611	140340
50	37633	25992	26027	2717	9969	133018

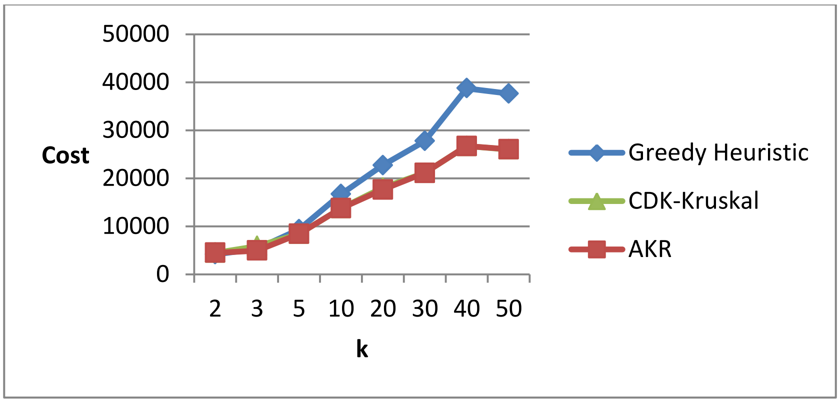

Figure 5.3.11: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,8

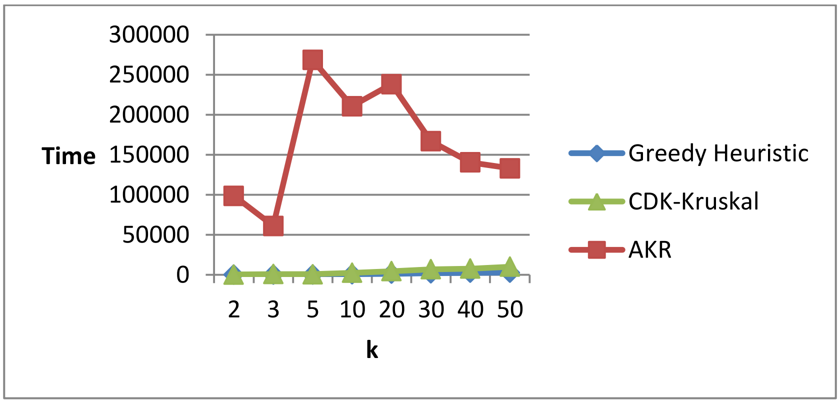

Figure 5.3.12: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 0,8

Table 5.3.6: Results of Cost and Running Time of Three Algorithms on TSP-Uruguay Graph with Edge Weights less than 1,0 times maximum distances

k	The biggest weight x 1.0					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	2489	3205	3205	141	447	47499
3	6248	5875	5875	151	462	159850
5	10608	10750	10750	240	867	205106
10	15976	14331	14331	553	2143	124357
20	22304	18269	18269	1310	4917	113558
30	29026	22790	22790	1364	4806	142498
40	33763	23600	23600	2892	6681	125293
50	40540	26979	26979	2944	9480	120993

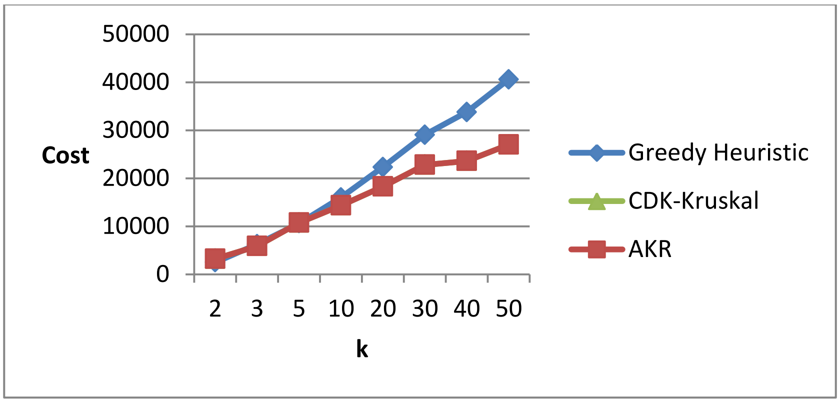

Figure 5.3.13: Cost Comparison of AKR, Greedy and ÇDK-Kruskal Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 1,0

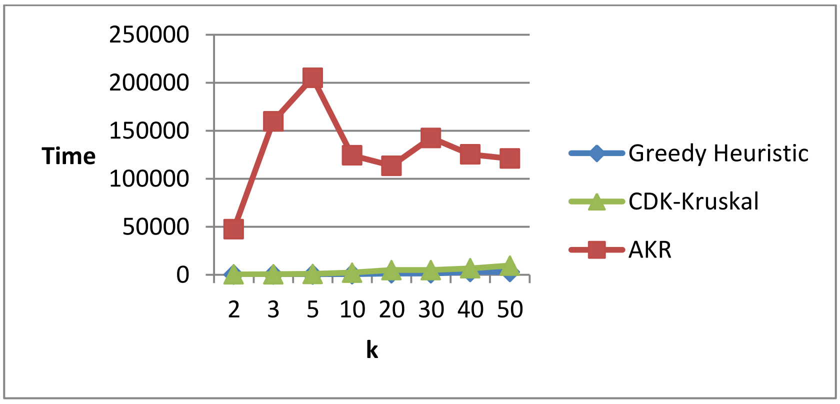

Figure 5.3.14: Running Time Comparison of the three Algorithms on TSP Uruguay Graph with edge weights which are smaller than the biggest distance x 1,0

5.4 Experiment 4

In this experiment, we test the algorithms on a random geometric graph.

- We form a graph with 1000 nodes.
- We determine by selecting from the interval 0-1000 x and y coordinates of a node which the graph will include.
- The distance between two nodes is specified by their Euclidean distance.
- To determine whether an edge exist between two nodes, first we calculate the maximum distance between all the nodes in the graph. Then, we multiply the maximum distance with 0.1, 0.2, 0.4, 0.6, 0.8 and 1.0. If the distance of two nodes is less than the result of this multiplication, we include the edge into the graph.

The result of this section are very similar to those of previous section except the fact that for the multiplication factor 1 (where we have a complete graph), the running time of AKR gets closer to ÇDK-Kruskal. Overall, ÇDK-Kruskal turns out to be a very good alternative to AKR especially on sparse random geometric graphs and real-world geometric graphs.

Table 5.4.1: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 0,1 times maximum distances

k	The biggest weight x 0.1					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1078	1099	1099	184	61	2924
3	1690	1611	1611	60	105	1589
5	1939	1624	1624	75	165	1764
10	3169	2730	2730	112	328	1335
20	5078	4217	4217	204	633	1795
30	6386	5379	5379	296	745	2076
40	7815	6086	6086	375	703	3007
50	8200	6639	6639	443	1493	1969

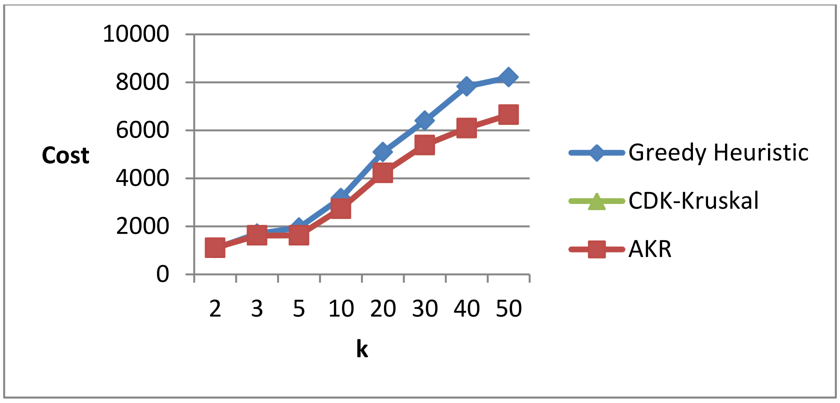

Figure 5.4.1: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,1

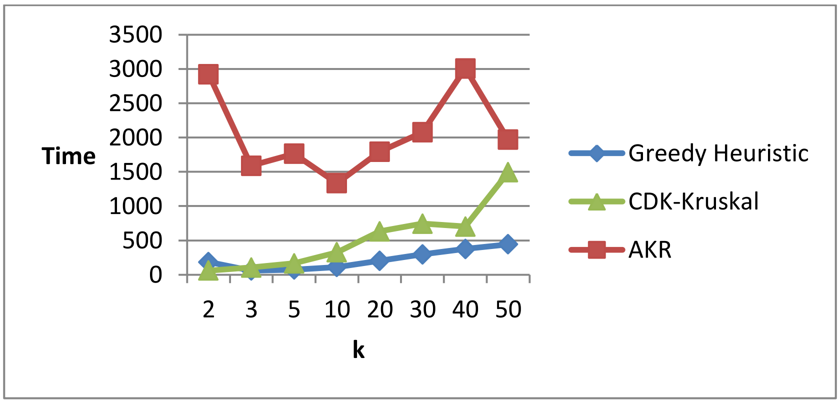

Figure 5.4.2: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,1

Table 5.4.2: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 0,2 times maximum distances

k	The biggest weight x 0.2					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1068	1147	1147	90	212	11014
3	1848	1848	1848	126	354	18951
5	2073	1828	1828	187	541	12366
10	3053	2568	2568	332	1184	8221
20	5692	4215	4215	404	1376	11204
30	6753	5269	5269	941	1789	12350
40	7470	5930	5930	1274	3422	7805
50	9078	6655	6655	1525	3807	8323

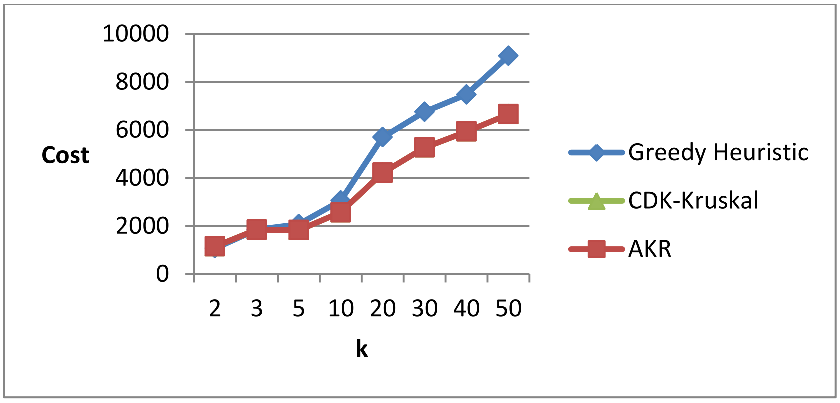

Figure 5.4.3: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,2

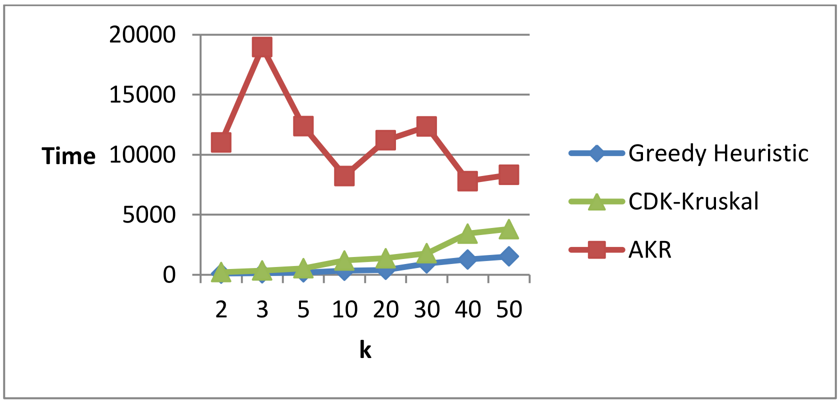

Figure 5.4.4: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,2

Table 5.4.3: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 0,4 times maximum distances

k	The biggest weight x 0.4					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1100	1100	1100	145	376	25464
3	1241	1251	1251	154	509	32494
5	1940	2013	1946	291	899	66867
10	3865	3244	3183	732	2912	41165
20	4694	4017	4017	893	2694	26140
30	6889	5134	5134	2019	3941	24370
40	7808	6101	6101	1546	6423	21898
50	8558	6257	6257	2745	10181	28652

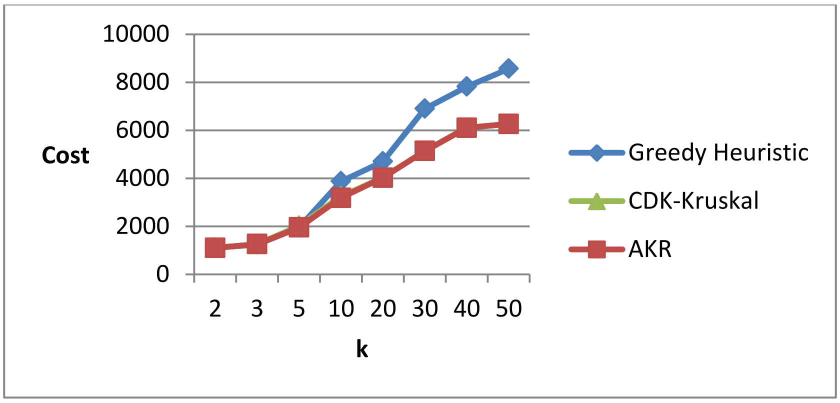

Figure 5.4.5: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,4

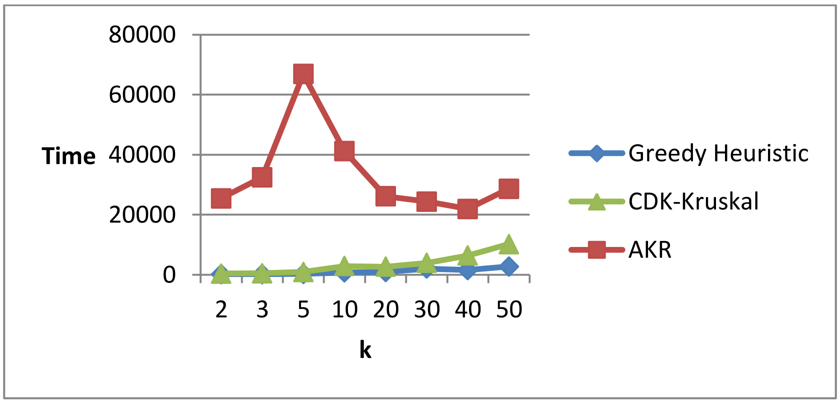

Figure 5.4.6: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,4

Table 5.4.4: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 0,6 times maximum distances

k	The biggest weight x 0.6					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1199	1225	1225	176	503	113857
3	1803	1466	1466	252	848	36524
5	2269	2156	2156	508	1278	73105
10	3455	2727	2727	946	3803	91906
20	5397	4003	4003	1439	5440	63906
30	6970	5342	5342	2216	9040	32943
40	7884	5993	5993	3029	9141	31322
50	8259	6157	6157	3863	14517	82427

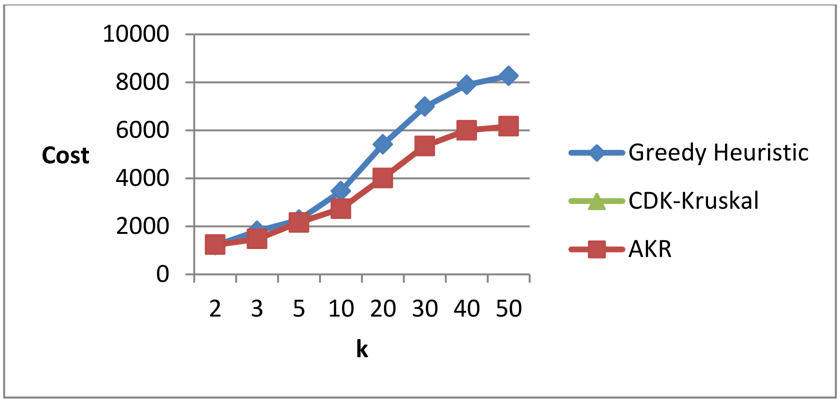

Figure 5.4.7: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,6

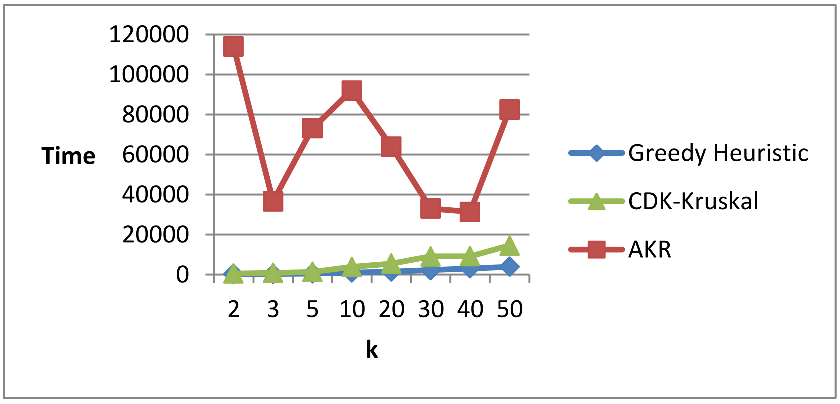

Figure 5.4.8: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,6

Table 5.4.5: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 0,8 times maximum distances

k	The biggest weight x 0.8					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	262	262	262	208	514	1732
3	1722	1695	1695	288	910	65194
5	2424	2376	2376	496	1766	143164
10	3299	3054	3054	991	4024	83026
20	5330	4374	4374	1620	6625	38187
30	6296	5145	5155	2226	8600	55763
40	7653	5956	5956	4594	13307	35233
50	8597	6313	6313	4448	12292	49798

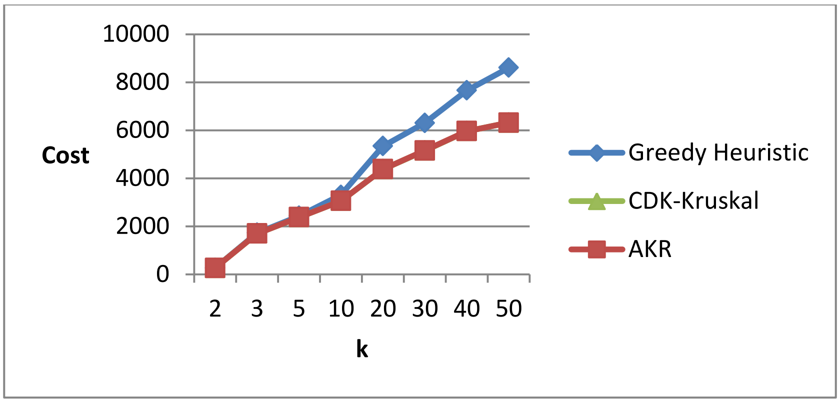

Figure 5.4.9: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,8

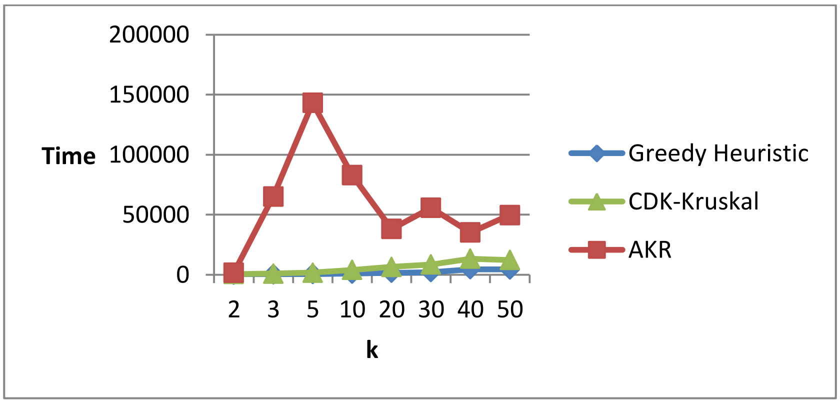

Figure 5.4.10: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 0,8

Table 5.4.6: Results of Cost and Running Time of Three Algorithms on Geometric Random Graph with Edge Weights less than 1,0 times maximum distances

k	The biggest weight x 1.0					
	COST			RUNNING TIME(ms)		
	Greedy Heuristic	ÇDK-Kruskal	AKR	Greedy Heuristic	ÇDK-Kruskal	AKR
2	1134	1412	1412	265	805	54279
3	2105	2153	2153	306	1084	174682
5	3012	2840	2840	607	2139	127667
10	3366	2978	2978	849	3233	66796
20	5082	4298	4298	1712	6174	48771
30	6225	4711	4711	2708	8953	59389
40	8366	6326	6326	4132	14385	52029
50	8797	6743	6743	4060	15759	29929

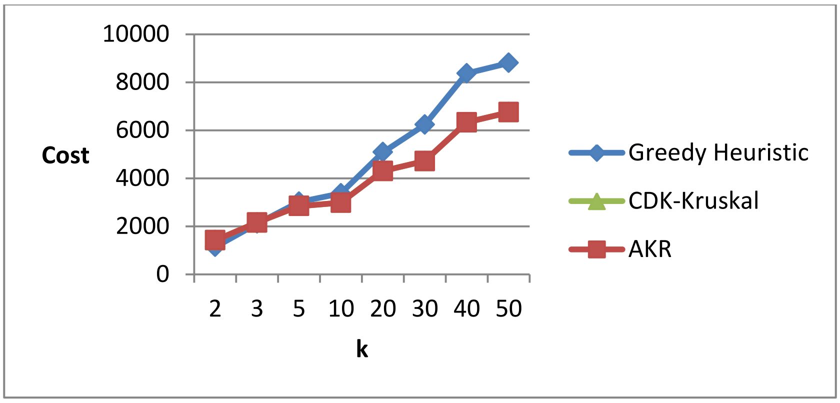

Figure 5.4.11: Cost Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 1,0

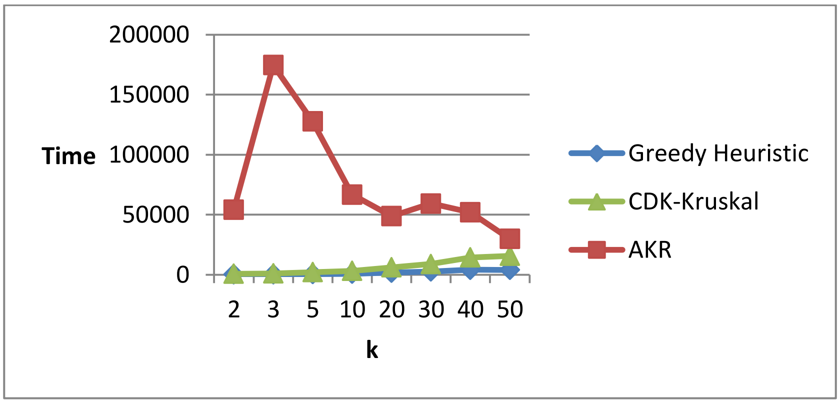

Figure 5.4.12: Running Time Comparison of the three Algorithms on Geometric Random Graph with edge weights which are smaller than the biggest distance x 1,0

REFERENCES

[1] David P. Williamson and David B. Shmoys, *The Design of Approximation Algorithm*, Cambridge, 2011.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

[3] A. Agrawal, P. N. Klein, and R. Ravi, "When trees collide: An approximation algorithm for the generalized steiner problem on networks", *SIAM J. Comput*., Vol. 24, pp. 440-456, 1995.

[4] M. X. Goemans and D. P. Williamson, "A general approximation technique for constrained forest problems", *SIAM J. Comput*., Vol. 24, pp. 296-317, 1995.

[5] J. Könemann, S. Leonardi, G. Schafer, and S. H. M. Van Zwam, A group-strategy proof cost sharing mechanism for the Steiner forest game, SIAM J. Comput., 37 (2008), pp. 1319-1341.

[6] A. Kumar, A. Gupta, And T. Roughgarden, A constant-factor approximation algorithm for the multicommodity rent-or-buy problem, in FOCS, 2002, pp. 333-344

[7] A. Gupta, A. Kumar, M. Pal, And T. Roughgarden, Approximation via cost-sharing: A simple approximation algorithm for the multicommodity rent-or-buy problem, in FOCS, 2003, pp. 606-615

[8] A. Gupta, A. Kumar, M. Pal, And T. Roughgarden, Approximation via cost sharing: Simpler and better approximation algorithms for network design, J. ACM, 54 (2007), p. 11.

[9] L. Becchetti, J. Könemann, S. Leonardi, And M. Pal, Sharing the cost more efficiently: Improved approximation for multicommodity rent-or-buy, ACM Transactions on Algorithms, 3 (2007).

[10] L. Fleischer, J. Könemann, S. Leonardi, And G. Schafer, Strict cost sharing schemes for steiner forest, SIAM J. Comput., 39 (2010), pp. 3616-3632.

[11] D. P. Williamson, M.X. Goemans, M. Mihail, And V. V. Vazirani, A primal-dual approximation algorithm for generalized steiner network problems, Combinatorica, 15 (1995), pp. 435-454.

[12] M. X. Goemans, A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, E. TARDOS, AND D. P. WILLIAMSON, Improved approximation algorithms for network design problems, in SODA, 1994, pp. 223-232.

[13] K. Jain, A factor 2 approximation algorithm for the generalized Steiner network problem, Combinatorica, 21 (2001), pp. 39-60.

[14] A. Levitin, Introduction to the Design & Analysis of Algorithms (3. ed.), Pearson Press, 2012.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms (3. ed.), MIT Press, 2009.,

[16] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, Proc. AMS, 7 (1956), pp. 48-50.

[17] R. C. Prim, Shortest connection networks and some generalizations, Bell Syst. Tech. Journal, 36 (1957), pp. 1389-1401.

[18] J. Nesetril, E. Milkov, and H. Nesetrilov, Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comment, history, Discrete Mathematics, 233 (2001), pp. 3-36.

[19] http://www.math.uwaterloo.ca/tsp/world/countries.html

[20] http://www.math.uwaterloo.ca/tsp/world/uymap.html

[21] http://www.math.uwaterloo.ca/tsp/world/uypoints.html