Experimental study of performance of minimum spanning tree algorithms

Alec Berenbaum

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Rochester Institute of Technology
Department of Computer Science

Experimental Study of
Performance of Minimum Spanning Tree Algorithms

by

Alec Berenbaum

A thesis, submitted to
the Faculty of the Department of Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

_____________
Professor Stanisław Radziszowski
Department of Computer Science, RIT

_____________
Professor Peter Anderson
Department of Computer Science, RIT

_____________
Professor Rayno Niemi
Department of Information Technology, RIT

August 26, 1998
Experimental Study of
Performance of Minimum Spanning Tree Algorithms

I, Alec Berenbaum prefer to be contacted each time a request for reproduction is made. If permission is granted, any reproduction will not be for commercial use or profit. I can be reached at the following address:

Rochester Institute of Technology
Department of Information Technology
Building 10
1 Lomb Memorial Drive
Rochester, NY 14623
(716) 475 - 7927

Date: 8/4/98
Signature of the Author ____________________
## CONTENTS

ACKNOWLEDGEMENTS. ................................................................. 1

ABSTRACT. .................................................................................. ii

1. **INTRODUCTION.** ................................................................. 1

2. **ALGORITHMS.** ................................................................. 8

3. **EXPERIMENT.** ................................................................. 31

4. **RESULTS.** ........................................................................ 39

5. **ANALYSIS OF RESULTS.** ................................................ 59

6. **CONCLUSIONS.** .............................................................. 73

7. **FURTHER WORK.** ............................................................ 75

8. **BIBLIOGRAPHY.** .............................................................. 76
ACKNOWLEDGEMENTS

I would like to thank Dr. Stanisław Radziszowski for all the help, guidance, and encouragement without which this thesis would not have been finalized. I also wish to thank Dr. Peter Anderson for the help and encouragement he provided throughout my graduate studies in Computer Science at the Rochester Institute of Technology. I extend special thanks to Dr. Rayno Niemi for taking the time out of his busy schedule to serve on my thesis committee.
ABSTRACT

Throughout the study of various theories of algorithms much work has been done in the area of traversal and solving optimization problems on graphs. Some of this work includes studies of finding the Minimal-Cost Spanning Trees (MST) in directed and undirected connected graphs. Several algorithms have been developed for such task. These algorithms tend to differ in performance based on various factors, such as graph density, size of problem spaces, range of weights that can be assigned to the edges of the graphs, edge weight distributions, etc. The data structures used by an algorithm can have a significant impact on algorithm’s performance, for each of the aforementioned factors. This thesis presents the results of the experimental study of the impact the data structures have on performances of Kruskal’s and Prim’s algorithms for finding Minimum-Cost Spanning Trees in connected undirected graphs.

The goal of this study is to compare performance of the practical implementations of Kruskal’s and Prim’s algorithms to their theoretical counterparts, as well as to measure and compare the differences in performances for various implementations of one algorithm, with respect to different implementation of the essential data structures. Performances of different algorithms are studied with respect to each-other for several variations of the types of data. As a result, a table depicting a schedule for use of the various implementations of either of the algorithms, as related to the type of graph used, is presented.

The algorithms are implemented and executed on a single Sun UltraSparc workstation, in order to eliminate the discrepancies, which may result from the differences in the processor speeds and variable CPU loads on multiple test machines. The following implementations are studied:

- Kruskal’s Algorithm with heapsort, and disjoint-sets using union-by-rank and path-compression heuristic
- Kruskal’s Algorithm with counting sort and disjoint-sets using union-by-rank and path-compression heuristic
- Prim’s Algorithm with brute force implementation of priority queues
- Prim’s Algorithm with priority queue implemented using a proper implementation of binary heap with “bubble-up” performed each time a decrease-key operation is performed for a vertex
- Prim’s Algorithm with priority queue implemented using a "lazy" implementation of binary heap with “bubble-up” performed after all decrease-key operations are performed for a vertex
- Prim’s Algorithm with priority queue implemented using a binomial heap
- Prim’s Algorithm with priority queue implemented using a Fibonacci heap

Upon the conclusion of the experiment, the best results were obtained from the implementation of Prim’s algorithm using the “lazy” heap implementation of a priority queue. For sparse graphs, Kruskal’s algorithm with counting sort performed very well, while for higher density graphs, Prim’s algorithm with binomial heap performed very well.
1. INTRODUCTION

The problem of finding minimum spanning trees in connected graphs has a wide range of applications. These include the design of computer and communication networks, power and leased-line telephone networks, wiring connections, links in transportation network, piping in a flow network, network reliability, surface homogeneity tests, image processing, speech recognition, clustering, classification, etc. [1]. The challenge has always been to find the minimum spanning tree (MST) as efficiently as possible in the graphs with a large number of vertices. This thesis studies and compares various methods of accomplishing this task, using variations of the algorithms developed by J.B. Kruskal [2] and R.C. Prim [3] to find the most practical method for solving the minimum spanning tree problem. The graphs vary in densities, sizes, and ranges of the edge weights. Most theoretical implementations of these algorithms disregard some issues that are inherent in the use of a digital computer. Issues such as various overheads associated with memory allocation/deallocation, use of disks, processor speeds, bus speeds, etc. may have significant impact on the expected performance of the theoretical implementation of the algorithm. The goal is to develop a table, which depicts a schedule of various implementations of Kruskal’s and Prim’s algorithms for diverse types of data.

1.1 Basics of Trees

A tree is a non-linear structure that is frequently used in the implementation of computer algorithms. Such structure implies a “branching” relationship between the nodes, much like the branching found in the trees in nature [4]. Donald E. Knuth defines a tree formally as a finite set $T$ of one or more nodes, such that

a) There is one especially designated node called a root of the tree, $root(T)$ [4].

b) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, ..., T_m$, and each of these sets in turn is a tree. The trees $T_1, ..., T_m$, are called subtrees of the root [4].
A tree can be viewed as an acyclic, connected, undirected graph, or likewise defined as an undirected graph in which there exists exactly one path between any two nodes [5]. The three most important properties of trees are:

1. A tree with \( n \) nodes has exactly \( n - 1 \) edges [5].

2. When a single edge is added to a tree, the resulting graph contains exactly one cycle [5], which is a violation of the property 1. Because the graph with \( n \) vertices now contains \( n \) edges, according to the basic property of a tree, that it is an acyclic, connected, undirected graph, introducing a cycle results in the graph no longer holding the tree property.

3. Removal of a single edge from a tree results in a graph that is no longer connected [5], thus resulting in the violation of the property 1, since the graph with \( n - 1 \) edges now contains \( n - 1 \) vertices. According to the basic properties of a tree, that it is an acyclic, connected, undirected graph, detaching one of the vertices from the graph, eliminates the path between that vertex and any other vertex in the graph, resulting in two separate trees.

Figure 1 illustrates several examples of trees of 5 nodes.

![Figure 1 - Rooted trees with 5 nodes (vertices)](image)

### 1.2 Spanning Trees

Seymour Lipschutz and Marc Lipson define a spanning tree of a graph \( G \) as a subgraph \( T \), if \( T \) is a tree and \( T \) includes all the vertices of \( G \) [6]. Thus, every connected graph contains at least one spanning tree. If a graph with \( n \) vertices contains \( E > n - 1 \) edges, then it is possible to remove \( E - (n - 1) \) edges in a manner, such that the connectivity of the graph is preserved, thus eliminating all cycles and resulting in a spanning tree. According to the definition provided by Brassard and Bratley [5], a tree has a property that exactly one unique path exists between any two vertices. If the two adjacent vertices \( v \) and \( u \) are a part of a subgraph, where there exist \( p \) paths from \( v \) to \( u \) and \( p > 1 \), removing the edge \( \{v, u\} \) will result in \( p - 1 \) paths from \( v \) to \( u \). This process can be repeated for
all adjacent vertices, \( u' \) and \( v' \), in the path from \( v \) to \( u \), if more than one path is available from \( u' \) to \( v' \), until exactly one path from \( v \) to \( u \) remains.

### 1.3 Minimum Spanning Trees

We can see that at least one spanning tree can be found in a connected, undirected graph. If the graph is weighted, i.e. a weight \( w(u, v) \) is assigned to every edge \( \{u, v\} \), we then state that the total weight of the spanning tree \( T \) within the graph \( G \) is expressed as the total weight of all edges connecting the vertices of \( T \) [7]

\[
w(T) = \sum_{(u,v) \in T} w(u, v)
\]

The problem of finding a spanning tree in a connected graph with lowest possible \( w(T) \) is known as a minimum spanning tree problem [7].

### 1.4 Historical Perspective

#### 1.4.1 O. Borůvka

The study of minimum spanning tree problem can be dated back as far as 1926, relating to the work of Otakar Borůvka, who became aware of the problem during the rural electrification of Southern Moravia [1]. He has formulated the statement of minimum spanning tree problem as follows [1]:

Given a matrix \( M \) of numbers \( r(x, y) \) \( (x, y = 1, 2, ..., n; n \geq 2) \), all positive and pairwise different, with the exception of \( r(x, x) = 0 \) and \( r(x, y) = r(y, x) \), find a subset of entries, pairwise different and nonzero, such that

1. for any \( p_1, p_2 \), different natural numbers \( \leq n \), the subset contains some \( r(p_1, c_2), r(c_2, c_3), r(c_3, c_4), ..., r(c_{q-2}, c_{q-1}), r(c_{q-1}, p_2) \)

2. the sum of its members is smaller than the sum of members of any other set of numbers pairwise different and nonzero satisfying condition 1 [8].

Borůvka then proceeds with a solution, the summary of which is presented in [1] in modern terms:
1. Choose a vertex $v$ and the shortest incident edge $vw_i$. If there exist edges $w_ix$ shorter than $vw_i$, choose the shortest such edge $w_iw_2$. Continue in this way, as long as possible, constructing a simple path $vw_i, w_iw_2, \ldots, w_{k-1}w_k$, where each $w_iw_{i+1}$ is the shortest edge incident with $w_i$ and is shorter than $w_{i-1}w_i$ [8].

2. Begin at a new vertex $p$ and construct as in 1 another simple path $pq_1, q_1q_2, \ldots, q_{l-1}q_l$, with $l$ as large as possible, under the constraint that $pq_1, q_1q_2, \ldots, q_{l-1}q_l$ are disjoint from the previous path or paths (as well as the constraint that each $q_iq_{i+1}$ is the shortest edge incident with $q_i$ and shorter than $q_{i-1}q_i$) [8].

Repeat until all vertices have been included on some such path [8].

These paths form fragments, and it is easy to see that an edge $ab$ is in the resulting forest $G$ if and only if it is the shortest edge at $a$ or $b$. Hence the forest $G$ is the same as the one obtained by joining each vertex to its nearest neighbor [1].

Graham and Hell [1] summarize Borůvka's method by providing a description of the process. There it is stated, that one forms the distance matrix for the set of fragments of $G$ and repeats the process, producing another forest $G_1$, then $G_2$, and so on, until the forest is just one tree $G_{n-1}$, the solution. An implementation of Borůvka's algorithm would run in time $O(\text{Elg}V)$, where $E$ is the number of edges and $V$ is the number of vertices [1]. Each time the rule, which defines the algorithm is applied, the number of fragments decreases by at least one half [1].

1.4.2 J. B. Kruskal

Kruskal attributes the formulation of the problem to Borůvka. In his paper [2] he considers distinct and positive sets of edge lengths. The primary interest is in establishing uniqueness under these conditions [3]. He provides three different constructions, or algorithms, for finding the minimum spanning trees, which we will discuss further. To summarize Kruskal's algorithm: [1]

1. Sort the edges by weight.

2. Examine each edge in the order of increasing weight.

3. If the edge inclusion does not create a cycle with the edges in the current forest, it is added to the forest; otherwise, it is discarded.
Kruskal provides this in the form of a construction: [2]

**Construction A** – Perform the following step as many times as possible:
Among the edges of \( G \) not yet chosen, choose the shortest edge which does not form any loops [cycles] with those edges already chosen [2].

The efficient implementation of Kruskal’s algorithm can be attributed to the efficiency in sorting of edges by weight, and in finding the fragment containing a given vertex (find-set), and in the merge of two fragments into one(find-union) [1], which is studied in this paper by experimentation. The best implementation of Kruskal’s algorithm is known to run in \( O(E \lg V) \) time [1]. \( O(E) \) time can be achieved if the edge weights are small integers and the radix sorting can be used, or if the edges are in sorted order [9].

In [2], he viewed his construction A as a special case of a more general construction [1]. Kruskal wrote as follows:

**Construction B** – Let \( V \) be an arbitrary but fixed (nonempty) subset of the vertices of \( G \). Then perform the following step as many times as possible:
Among the edges of \( G \) which are not yet chosen, but which are connected either to the vertex of \( V \) or to an edge already chosen, pick the shortest edge which does not form any loops with the edges already chosen [2].

Kruskal states that when \( V \) is a set of all vertices of \( G \), construction B reduces to construction A [1]. When \( V \) consists of a single vertex \( v \) construction B reduces to the algorithm which was later attributed to Prim [3]. Graham and Hell describe Kruskal’s algorithm as follows:

Sort the edges by weight. Given a fragment \( F \) containing \( v \), examine the unused edges in order of increasing weight until an edge is found joining a vertex in \( F \) to the vertex outside of \( F \). Add that edge. At the same time, edges that are found to join two vertices of \( F \) may be discarded [1].

### 1.4.3 R. C. Prim

In 1957 R. C. Prim submitted a manuscript on “Shortest Connection Networks and Some Generalization” for publication in *Bell System Technical Journal*. Prim’s main concern was that problem of inherent interest in the planning of large-scale communication, distribution, and transportation networks also arises in connection with the current rate structure for Bell System leased-line service [3]. Prim gave the following problem statement:
Given a set of (point) terminals, connect them by a network of direct terminal-to-terminal links having the smallest possible total length (sum of the link length). (A set of terminals is “connected”, of course, if and only if there is an unbroken chain of links between every two terminals in the set.) [3]

Prim provides two construction principles for the shortest connection networks:

**Principle 1** – Any isolated terminal\(^\text{i}\) can be connected to a nearest neighbor [3].

**Principle 2** – Any isolated fragment\(^\text{ii}\) can be connected to a nearest available neighbor by a shortest available link [3].

Prim states that since each application of either P1 or P2 reduces the total number of isolated terminals and fragments by one, it is evident that an N – terminal network is connected by \(N - 1\) applications [3]. Prim later provides the validation of principles P1 and P2. He states that the validity of these principles depends on the establishment of two necessary conditions (NC1 and NC2) for a shortest connection network (SCN):

**Necessary Condition 1** – Every terminal in an SCN is directly connected to at least one nearest neighbor [3].

**Necessary Condition 2** – Every fragment in an SCN is connected to at least one nearest neighbor by a shortest available path [3].

In [3] Prim provides the justification of these conditions. He later goes on to generalize the problem statement. Since the initial discussion has been in terms of the points on a distance-true map, the principles P1 and P2 could be based on visual judgements of relative distances [3]. Prim exchanges the visual distances for numerical values. The application of P1 and P2 goes through as before, where the relevant nearest neighbor is determined by a comparison of numerical labels [3]. Prim provides more conventional terminology of the Graph Theory:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal</td>
<td>vertex</td>
</tr>
<tr>
<td>Possible Link</td>
<td>edge</td>
</tr>
<tr>
<td>Length of Link</td>
<td>length or weight of edge</td>
</tr>
<tr>
<td>Connection Network</td>
<td>spanning subgraph</td>
</tr>
<tr>
<td>(Without closed loops)</td>
<td>(spanning subtree)</td>
</tr>
<tr>
<td>Shortest connection network</td>
<td>shortest spanning subtree</td>
</tr>
</tbody>
</table>

\(^{\text{i}}\) Prim refers to the terminal to which at a given stage of the construction, no connections have been made as an isolated terminal.

\(^{\text{ii}}\) Prim refers to the fragment to which, at a given stage of the construction, no external connection has been made as an isolated fragment. According to Prim’s definition, a fragment is a terminal subset connected by direct links, between members of the subset.
Prim then goes on to generalize the original problem by seeking shortest spanning subtrees for arbitrarily connected labeled graphs, thus providing a computational technique, presenting the algorithm to be discussed and studied in greater detail in further sections.
2. ALGORITHMS

In order to proceed with the experiment and detailed analysis of the results, we first analyze the Kruskal's [2] and Prim's [3] algorithms, in order to predict the possible outcomes and to be able to interpret results. To understand and predict a performance of a particular algorithm, we must not only account for the complexities associated with the flow of the algorithm itself; we must account for potential complexities of each step, that may appear less obvious initially. For example, as was mentioned earlier [9], sorting has a significant impact on the performance of Kruskal's algorithm, in some instances. If the implementation of the algorithm is to be targeted for a specific computing platform, we can even take into consideration complexities associated with the implementation of basic operations (steps) on that platform. In this thesis, we examine the complexities associated with the issues that are more abstract than those involved when using a specific computing platform, and thus common among the general majority of the computing environments.

2.1 Greedy Algorithms

The implementations of Kruskal’s and Prim’s algorithm studied in this thesis are presented by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, in their book Introduction to Algorithms [7]. They introduce a notion of a generic algorithm for solving the minimum spanning tree problem, which is a generalization of both, Kruskal’s and Prim’s methods. Both methods are considered to use a “greedy” strategy, which advocates the best possible choice at the time the choice is to be made [7]. One can say that a greedy algorithm does not have an overall strategy, but rather picks the best option possible at any given time.

2.2 Matroids

Cormen, Leiserson, and Rivest (to be referred to as CLR) state that the greedy algorithms yield optimal solutions when a combinatorial structure known as “matroid” is used [7]. The definition presented in [7] is as follows:

A matroid is an ordered pair $M = (S, I)$ such that
1. $S$ is a finite nonempty set [7].
2. $I$ is nonempty family of subsets of $S$, called the independent subsets of $S$, such that if $B \in I$ and $A \subseteq B$, then $A \in I$. $I$ is said to be hereditary if it satisfies this property. The empty set $\emptyset$ is necessarily a member of $I$ [7].
3. If \( A \in I \), \( B \in I \), and \(|A| < |B|\), then there exists some element \( x \in B - A \) such that \( A \cup \{x\} \in I \). Thus, \( M \) satisfies the exchange property [7].

CLR illustrate a graphic matroid \( M_G = (S_G, I_G) \), which they define in terms of an undirected graph \( G = (V, E) \) as follows:

- The set \( S_G \) is defined to be \( E \), the set of edges of \( G \) [7].
- If \( A \) is a subset of \( E \), then \( A \in I_G \) if and only if \( A \) is acyclic. A set of edges is independent if and only if it forms a forest [7].

Thus, we can see that a graphical matroid is closely related to a minimum spanning tree [7].

CLR present the following theorem, which they prove by contradiction [7]:

**Theorem 1:**
All maximal independent subsets in a matroid have the same size.

**Proof:** Suppose to the contrary that \( A \) is a maximal independent subset of \( M \) and there exists another larger maximal independent subset \( B \) of \( M \). Then, the exchange property implies that \( A \) is extendible to a larger independent subset \( A \cup \{x\} \) for some \( x \in B - A \), contradicting the assumption that \( A \) is maximal [7].

Let \( M_G \) be a free\(^1\) tree with exactly \( V - 1 \) edges that connect all vertices of \( G \). If \( M_G \) is a graphic matroid for a connected, undirected graph \( G \), \( M_G \) is a spanning tree of \( G \). If the \( M_G \) is weighted, where there is a weight function \( w(x) \) that assigns a positive weight to each element \( x \in S \), the weight function \( w(x) \) can be extended to subsets of \( S \) by:

\[
w(A) = \sum_{x \in A} w(x)
\]

for any \( A \subseteq S \) [7], which is essential weight function for a spanning tree:

\[
w(T) = \sum_{(u,v) \in T} w(u,v)
\]

where \((u,v)\) is the edge that connects vertices \( u \) and \( v \).

\(^1\) A free tree is a connected graph \( G \), which has no cycles, with no vertex designated as a root of \( G \).
2.2.1 Use of Greedy Approach for Graph Optimization Problems

Studying the optimization problem in a weighted matroid, as defined below, one can see how the greedy approach can be applied to produce optimal results. As demonstrated by V. K. Balakrishnan [10], the greedy approach to solve the matroid optimization problem is reminiscent of the Kruskal’s and Prim’s algorithms to be discussed in the next two sections [10].

Let \( w \) be a nonnegative weight function defined on the ground set \( E \) of an independent system. If \( A \) is a subset of \( E \), the weight of \( A \), denoted by \( w(A) \), is the sum of weights of all elements in \( A \). An optimization problem associated with the independent system is the problem of finding the independent set with the maximum weight [10]:

1. Choose \( x(k) \) distinct from \( x(1), x(2), \ldots, x(k-1) \) such that
   a) the set \{\( x(1), x(2), \ldots, x(k-1), x(k) \)\} is an independent set
   b) if \{\( x(1), x(2), \ldots, x(k-1), x \)\} is an independent set, the weight of \( x \) does not exceed the weight of \( x(k) \).
2. Stop if no such \( x \) exists.

This can be demonstrated by the following theorem [10]:

**Theorem 2:**

A solution of the problem of finding a maximum weight independent set in an independent system can be obtained by using the greedy algorithm for every nonnegative weight function defined on its ground set if and only if the independent system is a matroid [10].

**Proof:**

1. If \( I \) and \( J \) are two independent sets in an independent system, with \( p \) and \( (p+1) \) elements, respectively, let \( w(e) = (p+2) \) for all \( e \) in \( I \), \( w(e) = (p+1) \) for all \( e \) in \( (J-I) \), and \( w(e) = 0 \) for all other nodes \( e \) in the ground set. Then \( w(J) \geq (p+1)(p+1) > p(p+2) = w(I) \); hence, \( I \) is not a solution. By the greedy procedure, \( I \) and then an element is taken from \( (J-I) \). Thus, there exists an element \( e \) in the set \( (J-I) \) such that \( I + e \) is an independent set. Therefore the independent system is a matroid [10].
2. Suppose that by applying the greedy algorithm, an independent set \( I = \{e_1, e_2, \ldots, e_r\} \) is obtained (in a matroid) in which the elements are arranged in nondecreasing order by weight. If \( J = \{f_1, f_2, \ldots, f_r\} \) is an independent set in a matroid, it can be proved by induction that \( w(f_i) \leq w(e_i) \) for every \( i \). It is true for \( i = 1 \). Assumption: the condition
holds for \(i = 1, 2, \ldots, (m-1)\). Thus, the proof is required for \(i = m\). Suppose \(w(f_m) > w(e_m)\). Let \(D = \{e_1, e_2, \ldots, e_{m-1}\}\) and \(A = \{e : w(e) \geq w(f_m)\}\). Then \(D\) is an independent set and, by induction hypothesis, is subset of \(A\). If \(D\) is not maximal in \(A\), there exists \(e\) in \(A\) such that \(D + e\) is independent. But if \(e\) is in \(A\), \(w(e) \geq w(f_m) > w(e_m)\), which implies that after picking \(e_{m-1}\), the greedy algorithm would have selected \(e\) and not \(e_m\).

Thus, \(D\) is maximal in \(A\). Since \(D\) has \((m-1)\) elements, any independent subset of \(A\) cannot have more than \((m-1)\) elements. But \(\{f_1, f_2, \ldots, f_m\}\) is an independent subset of \(A\). The contradiction shows that \(w(f_m) \leq w(e_m)\) [10].

Hence, the greedy approach is the optimal approach for solving the minimal-spanning-tree-problem, since the spanning trees are closely related to matroids, and the greedy approach proves to be the optimal approach for matroid optimization problem.

2.2.2 Greedy Approach for the Minimum spanning tree problem

Cormen, Leiserson, and Rivest present a generalization of Kruskal’s and Prim’s Algorithms, which clearly illustrates the greedy approach. The generalization presented is as follows [7]:

\[
\text{GENERIC-MST}(G,w) \\
1 \quad A \leftarrow \emptyset \\
2 \quad \text{while } A \text{ does not form a spanning tree} \\
3 \quad \text{do} \quad \text{find an edge } (u,v) \text{ that is safe for } A \\
4 \quad \quad A \leftarrow A \cup \{(u,v)\} \\
5 \quad \text{return } A
\]

Algorithm 1 – Generalization of Kruskal’s and Prim’s Algorithms

This algorithm grows the minimum spanning tree one edge at a time. The set \(A\) is always a subset of some minimum spanning tree. With each step a test is performed to ensure that it is “safe” to add an edge \((u,v)\), ensuring that \(A \cup \{(u,v)\}\) is a subset of the minimum spanning tree [7].

The invariant is “trivially” satisfied in line 1 of the algorithm. It is maintained in lines 2 through 4. The challenge is to find the edge that is safe for \(A\), as done in line 3 [7]. The existence of such edge is dictated by the invariant that there is a spanning tree \(T\) such that \(A \subseteq T\); if there exists an edge \((u,v) \in T\) such that \((u,v) \notin A\), then it is safe to add
(u,v) to A [7]. CLR provide a rule, in the form of a theorem, for recognizing safe edges, which they proceed to prove:

**Definitions:**

- **cut:** A cut \((S, V-S)\) of an undirected graph \(G = (V, E)\) is a partition of \(V\) [7], as shown in Figure 2.

- **crossing:** It is said that an edge \((u,v) \in E\) crosses the cut \((S,V-S)\) if one of its endpoints is in \(S\) and the other is in \(S-V\) [7].

- **respect:** A cut respects the set \(A\) of edges if no edge in \(A\) crosses the cut [7].

- **light edge:** An edge that crosses the cut with the weight minimum of any edge crossing the cut [7]. More than one light edge can exist.

**Theorem 3:**

Let \(G = (V, E)\) be a connected, undirected graph with a real-valued weight function \(w\) defined in \(E\). Let \(A\) be the subset of \(E\) that is included in some minimum spanning tree for \(G\), and let \((S,V-S)\) be any cut of \(G\) that respects \(A\), and let \((u,v)\) be a light edge crossing \((S,V-S)\). Then, edge \((u,v)\) is safe for \(A\) [7].

**Proof:**

Let \(T\) be a minimum spanning tree that includes \(A\), and assume that \(T\) does not contain the light edge \((u,v)\). If it does, the proof is complete. Another minimum spanning tree \(T'\) that includes \(A \cup \{(u,v)\}\) is constructed by using the cut-and-paste technique, thus showing that \((u,v)\) is a safe edge for \(A\) [7]. The edge \((u,v)\) forms a cycle with the edges on the path \(p\) from \(u\) to \(v\) in \(T\) as shown below [7].

---

*Figure 2 – Cut \((S, V-S)\) [7]*
Because \( u \) and \( v \) are on the opposite sides of the cut \( (S, S-V) \), there is at least one edge in \( T \) on the path \( p \) that also crosses the cut. Let \( (x, y) \) be such edge. The edge \( (x, y) \) is not in \( A \), because the cut respects \( A \). Since \( (x, y) \) is on the unique path from \( u \) to \( v \) in \( T \), removing \( (x, y) \) breaks \( T \) into two components. Adding \( (u, v) \) reconnects them to form a new spanning tree \( T' = T - \{(x, y)\} \ Y \{(u, v)\} \) [7].

It can now be shown that \( T' \) is a minimum spanning tree. Since \( (u, v) \) is a light edge, crossing \( (S, S-V) \) and \( (x, y) \), also crosses this cut, \( w(u, v) \leq w(x, y) \). Therefore, \( w(T') = w(T) - w(x, y) + w(u, v) \leq w(T) \). Since \( T \) is a minimum spanning tree, so that \( w(T) \leq w(T') \), \( T' \) must be a minimum spanning tree also.

\[ A \subseteq T', \text{ since } A \subseteq T \text{ and } (x, y) \notin A; \text{ thus, } A \ Y \{(u, v)\} \subseteq T'. \] Consequently, since \( T' \) is a minimum spanning tree, \( (u, v) \) is safe for \( A \) [7].

Because each of the \( V-1 \) edges is successfully determined, the loop in lines 2–4 of the GENERIC-MST is executed \( V-1 \) times. Initially \( A = \emptyset \). There are \( V \) trees in \( G_A \). This number is reduced by 1 during each iteration. The algorithm terminates when the forest contains a single tree [7].
2.3 Kruskal’s Algorithm

Kruskal’s algorithm falls perfectly into the greedy paradigm. Since Kruskal’s algorithm is essentially a specialization of the GENERIC-MST [7] presented by CLR, it proceeds with the same approach. The approach taken by Kruskal is to pick the edge with the smallest weight value such that both vertices of the edge are not present in the set $T$ and add that edge to the set $T$. The process of picking the edge with the smallest weight value is characteristic of the greedy approach taken by the algorithm. The following implementation is provided by CLR [7]:

$$\text{MST-KRUSKAL}(G,w)$$
1. $A \leftarrow \emptyset$
2. for each vertex $v \in V[G]$
3. do MAKE-SET($v$)
4. sort the edges of $E$ by nondecreasing weight $w$
5. for each edge $(u,v) \in E$, in order by nondecreasing weight
6. do if $\text{FIND-SET}(u) \neq \text{FIND-SET}(v)$
7. then $A \leftarrow A \cup \{(u,v)\}$
8. UNION($u,v$)
9. return $A$

Algorithm 2 – Implementation of Kruskal’s Algorithm

CLR [7] claim that this implementation is the asymptotically fastest implementation known today. They attribute this to the use of the disjoint-set data structures to maintain several disjoint-sets of elements (trees in the forest), which results in the running time $O(E\alpha(E,V))$, excluding the sorting time, where $\alpha$ is the functional inverse of the Ackerman’s function [7]. We will study this implementation in more detail. By simple examination of the algorithm, we can easily see that the complexity of the algorithm itself, excluding the MAKE-SET, FIND-SET, UNION, and sorting operations is $O(E)$ where $E$ is the number of edges in the graph $G$. The implementation of these operations may have a significant impact on the performance of the algorithm. We will study the results produced by variations of implementation of sorting. The two implementations to be studied here are the heapsort and the counting sort. To observe the effect of sorting on the performance, we can observe the change in the performance by eliminating the sorting entirely. This can be achieved by measuring the performance of the algorithm minus the time required for sorting the edges. Hence, one can observe the performance of the algorithm based strictly on the implementation of the disjoint-set data structure. We will present the experiment setup and the results in the next chapter. Since the use of the disjoint-set data structure yields the most efficient implementation of Kruskal’s algorithm known today, the main focus of the experiment is on sorting, which can result in severe degradation of performance.
2.4 Prim’s Algorithm

Prim’s algorithm is a specialization of the GENERIC-MST [7], similarly to Kruskal’s algorithm. Unlike Kruskal, where multiple trees in the forest are joined until a single spanning tree is formed, Prim maintains a single tree in the set $A$. The tree starts from an arbitrary root vertex $r$ and grows until it spans all vertices in the set $V$. With each iteration, a light edge connecting a vertex in $A$ to a vertex in $V - A$ is added to the tree; adding only the edges that are safe for $A$. When the algorithm terminates, the edges in $A$ form a minimum spanning tree [7]. The augmentation of the tree with each step with the edge that has a minimum weight renders this strategy “greedy”. The efficiency of the algorithm depends on the strategy used for selecting a new edge to be added to the tree.

CLR provide the implementation of the algorithm which uses the priority queue $Q$ which is based on a key field [7]:

```plaintext
MST-PR1M(G,w,r)
1   Q ← V[G]
2   for each u ∈ Q
3     do key[u] ← ∞
4     key[r] ← 0
5     π[r] ← NIL
6   while Q ≠ ∅
7     do u ← EXTRACT-MIN(Q)
8       for each v ∈ Adj[u]
9         do if v ∈ Q and w(u,v) < key[v]
10            then π[r] ← u
11            key[v] ← w(u,v)
```

Algorithm 3 - Implementation of Prim’s Algorithm

For each vertex $u$, $key[u]$ is the minimum weight of any edge connecting $u$ to a vertex in the tree. If no such edge exists, $key[u] = ∞$. The field $π[v]$ specifies the parent of $v$ in the tree [7]. While the algorithm is running, the set of edges $A$ is:

$$A = \{(v,π[v]) : v ∈ V - \{r\} - Q\}$$

Upon termination, when the priority queue is empty, the set of edges $A$ is [7]:

$$A = \{(v,π[v]) : v ∈ V - \{r\}\}.$$
To reiterate, the performance of Prim’s algorithm depends on the implementation of the priority queue. Next chapter describes the study of the performance obtained as the result of experimentation with various implementations of the priority queue $Q$.

2.5 Areas of “Weakness”

The main goal of the thesis is to test the performance of the Kruskal’s and Prim’s algorithms using various implementations of the data structures that can have a significant effect on the algorithm’s performance. The basic analysis of the algorithms can reveal the areas where the implementation of the data structure can have a significant impact.

2.5.1 Areas of “weakness” in Kruskal’s Algorithm

Examining the implementation of Kruskal’s Algorithm shown in Algorithm 2 one can easily identify the complexity of the overall algorithm, excluding the complexities inherent in some of the basic steps. One can easily see that the lines 1, 4, and 9 are executed once; the lines 2 and 3 are executed $\Theta(V)$ times; and the lines 5 – 8 are executed $O(E)$ times. Thus, ignoring the time complexities, which might be inherent in some of the basic steps, the initial observation suggests that the algorithm runs in $O(E)$ time. However, lines 3, 4, and 6 – 8 involve calls to other procedures. Hence, the implementation of these procedures may significantly affect the overall performance of the algorithm. Since the implementation of the Kruskal’s algorithm in this experiment uses the disjoint-set data structure with path-compression heuristic, it shall remain a constant factor, and thus will not be considered an area of “weakness”. It is, however necessary to examine the running time of the Kruskal’s Algorithm with disjoint-sets and path-compression heuristic in order to understand their contribution to the overall complexity of the algorithm. The disjoint-set data structure used in this implementation of the Kruskal’s Algorithm supports the following operations: MAKE-SET($x$), UNION($x,y$) and FIND-SET($x$), where $x$ and $y$ denote the objects that are the members of the sets. The MAKE-SET($x$) operation creates a set where $x$ is the only object. FIND-SET($x$) returns a pointer to some representative of the set which contains $x$. UNION($x,y$) merges two sets, each containing $x$ and $y$ objects into one set containing both $x$ and $y$ objects. The implementation used in this experiment is provided by CLR [7], which uses the union by rank with path compression. The union by rank yields the $O(m \log n)$ running time, where $\Omega(n)$ lower bound is denoted by $m$ [11]. The path compression heuristic yields the worst-case running time of $\Theta(f \log_{(1+f/n)} n)$ if $f \geq n$ and $\Theta(n + f \log n)$ if $f < n$, where $f$ is the number of FIND-SET operations [11]. The worst case running time when both, union by rank and path compression are used is $O(m \alpha(m,n))$. The $\alpha(m,n)$ is the inverse of the Ackerman’s function [11], and is defined as follows:
\[
A(l,j) = 2^j \quad \text{for} \quad j \geq 1 \\
A(i,1) = A(i-1,2) \quad \text{for} \quad i \geq 2 \\
A(i,j) = A(i-1,A(i,j-1)) \quad \text{for} \quad i, j \geq 2
\]

According to CLR [7] in any conceivable application of the disjoint-set data structure, the running time can be viewed as \( m \) in most practical situations, since \( \alpha(m,n) \leq 4 \) for \((m,n)\). CLR introduce a slightly weaker upper bound on the running time, \( O(m \log^* n) \). They use the aggregate method of amortized analysis to prove the \( O(m \log^* n) \) time bound [7]. CLR present and prove the following theorem:

A sequence of \( m \) MAKE-SET, LINK, and FIND-SET operations, \( n \) of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time \( O(m \log^* n) \) [7].

The proof is provided in [7] on pp. 455 – 457.

The concern still lies with sorting. Since the disjoint-set operations with path compression heuristics allow the algorithm to run in practically linear time, it appears that most of the complexity is attributed to sorting. Hence this is the area studied in greater detail through experimentation.

### 2.5.2 Areas of “weakness” in Prim’s Algorithm

Examining the implementation of Prim’s Algorithm shown in Algorithm 3 following complexities of basic operations become obvious. Line 1 is executed in \( \Theta(V) \) time, since every vertex in the graph has to be enqueued. Lines 2 and 3 are executed in \( \Theta(V) \) time as well. Lines 4 and 5 are executed once. The lines 6 and 7 are executed \( V \) times; the lines 8 and 9 are executed \( \frac{1}{2}(V^2 - V) \) times\(^{ii} \); and lines 10 and 11 are executed \( \frac{1}{2}e(V^2 - V) \) times. This yields the \( O(E) \) running time. The area of concern is the implementation of the priority queue \( Q \). It is therefore anticipated that it is a major factor in the overall performance of the algorithm. Thus, the implementation of the priority queue to be used in Prim’s algorithm is studied in a great detail through experimentation.

---

\(^{ii}\) The \textbf{LINK} procedure is called by \textbf{UNION} procedure. It takes to two roots as inputs, and links them such that the root with the higher rank becomes a parent of the root with the lower rank. If both ranks are equal, then the rank of the parent is incremented by 1.

\(^{ii}\) The adjacent edges are sought in the adjacency matrix, which was provided as input data to the function. The edges are located by iterating from the edge \( e(x, y) \) of the matrix, until \( x = y \), for each row.
2.6 Expected Results

Prior to performing the experiment, we conduct a detailed analysis of all implementations used, in order to be able to predict the performance of each. The complexities associated with the "areas of weakness" of each implementation are expected to be the major contributors to the running time of the algorithms, and therefore are candidates for optimization. To predict the performance of each implementation studied in this thesis, we proceed with thorough analysis of the implementation of each algorithm.

2.6.1 Analysis of Kruskal's Algorithm

Both implementations of Kruskal's algorithm used in this experiment consist of two phases, sorting and growing a forest. Each phase can be potentially a dominating factor in the total running time of the algorithm. We therefore need to identify and predict the area where most of the time will be spent. The total running time of the algorithm can be expressed as

\[ T = T_M + T_F + T_U + T_S. \]

We first determine the contribution that the operations associated with disjoint-set data structure make to the total running time of the algorithm. Looking at the implementation of the algorithm below, we note the number of times each line is expected to execute:

```plaintext
MST-KRUSKAL(G,w)
 1  A <- 0                      1 time
 2  for each vertex v in V[G]    V times
 3    do MAKE-SET(v)            V times
 4  sort the edges of E by nondecreasing weight w       1 time
 5  for each edge (u,v) in E, in order by nondecreasing weight  E times
 6    do if FIND-SET(u) != FIND-SET(v)    E times
 7    then  A <- A U {(u,v)}     V-1 times
 8    UNION(u,v)               V-1 times
 9  return A
```

Since all disjoint-set operations with union by rank and path compression run in \( O(m \lg^* n) \) time, we first determine the values of \( n \) and \( m \). Since \( m \) is the total of all MAKE-SET, FIND-SET, and UNION operations, \( m = T_M + T_F + T_U \). Thus,

\[ T_M - \text{total time of all MAKE-SET operations}, \quad T_F - \text{total time of all FIND-SET operations}, \quad T_U - \text{total time of all UNION operations}, \quad T_S - \text{total time spent sorting edges} \]
\[ m = V + 2E + V - 1 = 2V + 2E - 1 \]

Given that \[ E = \frac{V^2 - V}{2} \times \epsilon \], where \( \epsilon = p \) is the density of the graph \( G \),

\[
m = 2V + V^2 \epsilon - V \epsilon - 1 = V^2 \epsilon + V(2 - \epsilon) - 1
\]

and given that \( \lg^* V \equiv 4 \), because the number of MAKE-SET operations is equal to \( V \) and \( 100 \leq V \leq 1700 \)[7],

\[
m \lg^* V = 4 \epsilon V^2 + (4 - 4 \epsilon)V - 4 = O(V^2)
\]

for fixed density \( \epsilon \). We therefore state that the total running time of each implementation is

\[ O(V^2) + O(T_s) \]

The implementations of Kruskal’s algorithm used in this experiment differ only in the method used to sort the edges of the graph. We therefore look at the time required by each sorting method, and use that to determine which is the dominating factor in the algorithm’s total running time.

**Counting Sort (kcs).**

The counting sort runs in linear time \( O(n) \)[7]. The sort determines for each input element \( x \), the number of elements less than \( x \). Then each element is placed directly into its position in the output array. The input is an array \( A[1..n] \), of length \( n \). Two other arrays are used, \( B[1..n] \) to store the sorted output, and \( C[1..k] \) to be used as working storage, where \( k \) is the value of the largest element \( x \) in \( A \). All values in \( C \) are initially 0. The values of each input element are then inspected. If the value of the input element is \( i \), the \( C[i] \) is incremented, resulting in \( C[i] \) containing the number of input elements equal to \( i \) for each integer value of \( i = 1, 2, ..., k \). Then the number of elements less than or equal to \( i \) is determined, by keeping a running sum of the array \( C \). Finally, each element of \( A \) is placed in its correct sorted position in the output array \( B \)[12]. We can therefore see that the counting sort runs in \( O(E) = O(V^2) \) time. Hence, the total running time of the algorithm is:

\[
O(V^2) + O(V^2) = O(V^2)
\]
for fixed density ε. From this we cannot conclude which phase of the implementation dominates the total running time. Thus, we make an assumption that both phases contribute equally. This is to be verified through experimentation.

**Heapsort (khp).**

The running time of the heapsort is $O(n \log n)$ steps. The sort requires one call to the procedure to build the heap, which is done on $O(n)$ time, and $n - 1$ calls to procedure to restore the heap property, each taking $O(\log n)$. We therefore say that the heapsort runs in $O(n \log n)$ time. Given that we are sorting the edges of the graph, using the heapsort the sorting time is

$$T_s = O(E \log E) = O(V^2 \log V)$$

The total running time of the algorithm is then,

$$O(V^2) + O(V^2 \log V) = O(V^2 \log V)$$

Hence, for the implementation of Kruskal’s algorithm using the heapsort, sorting is the dominating factor in the total running time. We can therefore anticipate the implementation of Kruskal’s algorithm that uses the heapsort to perform worse than the implementation of Kruskal’s algorithm that uses the counting sort.

Kruskal’s algorithm is performed in two phases. In the first phase the edges are sorted in a nondecreasing order. In the second phase, a forest is grown until $V$ vertices are connected to form a minimum spanning tree. The “areas of weakness” are self-contained, which renders the analysis of the running times attributed to the implementations of various sections of the algorithm simple.

### 2.6.2 Analysis of Prim’s Algorithm

In Prim’s algorithm, the “area of weakness” is in the implementation of the priority queue, which is tightly integrated into several steps of the algorithm. We therefore concentrate on the overall running time of the algorithm, rather than looking at it in phases, as was done in Kruskal’s algorithm. This algorithm uses total of three operations, **Build-Queue**, **Extract-Min**, and **Decrease-Key**, times for which we will denote as
\( T_B, T_E, \) and \( T_D, \) respectively. The worst case total running time of Prim’s algorithm can be expressed as

\[
T = T_B + \sum_{v \in G} T_E + \sum_{(u,v) \in G} T_D = T_B + VT_E + E\varepsilon T_D
\]

MST-PRIM(G,w,r)
1 \( Q \leftarrow V[G] \) \hspace{3cm} V times
2 \text{for each} \ u \in Q \hspace{2cm} V \text{times}
3 \hspace{0.5cm} \text{do} \ key[u] \leftarrow \infty \hspace{2cm} V \text{times}
4 \ key[r] \leftarrow 0 \hspace{2cm} 1 \text{time}
5 \pi[r] \leftarrow \text{NIL} \hspace{1cm} 1 \text{time}
6 \text{while} \ Q \neq \emptyset \hspace{1cm} V \text{times}
7 \hspace{0.5cm} \text{do} \ u \leftarrow \text{EXTRACT-MIN}(Q) \hspace{1cm} V \text{times}
8 \hspace{1cm} \text{for each} \ v \in \text{Adj}[u] \hspace{1cm} \frac{1}{2} (V^2 - V) \text{times}
9 \hspace{1.5cm} \text{do if} \ v \in Q \text{ and} \ w(u,v) < key[v] \hspace{1cm} \frac{1}{2} (V^2 - V) \text{times}
10 \hspace{2.5cm} \text{then} \ \pi[r] \leftarrow u \hspace{1cm} \frac{1}{2} \varepsilon (V^2 - V) \text{times}
11 \hspace{3.5cm} key[v] \leftarrow w(u,v) \hspace{1cm} \frac{1}{2} \varepsilon (V^2 - V) \text{times}

We can see that the performance of the algorithm is affected by the running time of the priority queue operations. We now look at each of the implementations of Prim’s algorithm studied in this thesis.

**Brute Force (pbf).**

This implementation does not have a build cost associated with the use of priority queue. The edges and their keys are stored in an array. To perform an \text{EXTRACT-MIN} operation, the array is searched for the element with minimum key, which is then returned. The location where the element with the minimum key is stored, is set to \( \infty \). \text{EXTRACT-MIN} operation will always be done in \( \Theta(V) \) time. Since no heap is used in this implementation, we do not need to worry about maintaining any heap properties. The \text{DECREASE-KEY} operation will therefore be always done in \( \Theta(1) \) time. The total running time of the “brute force” implementation of Prim’s algorithm is

\[ T_B - \text{time to build the heap; } T_E - \text{time to perform } \text{EXTRACT-MIN} \text{ operation; } T_D - \text{time to perform } \text{DECREASE-KEY} \text{ operation.} \]
\[ T = \Theta(V) + \Theta(V^2) + O(E) \]
\[ = V + V^2 + \frac{1}{2} \varepsilon V^2 + \frac{1}{2} \varepsilon V \]
\[ = (1 + \frac{1}{2} \varepsilon + 1)V^2 + (1 + \frac{1}{2} \varepsilon + 1)V \]
\[ = \Theta(V^2) \]

Thus, the complexity of the implementation of Prim’s algorithm using “brute force” for priority queues appears to be very tightly bound. This tight bound is attributed to the asymptotic complexity of the EXTRACT-MIN operation. Since precisely \( V \) EXTRACT-MIN operations are to be performed, and given that each operation takes \( \Theta(V) \) time, we can expect a very consistent performance of this algorithm. In general, the performance is expected to be worse than that of other implementation, however for dense graphs, it may perform as well as the other implementations studied in this experiment.

**Binary Heap – “Proper Implementation” (\( \text{php}_a \)).**

When analyzing the performance of algorithms that make use of heaps, we consider the cost of maintaining a heap property in EXTRACT-MIN and DECREASE-KEY operations. Each time a minimum element is extracted the last element on the heap is moved in the place of the minimum element. The element that was placed in the place of the minimum element is “sifted down”, until the heap invariant is restored. The element that is “sifted down” will “travel” the maximum length of \( \lceil \lg n \rceil \) vertices, since the height of a binary heap of \( n \) elements is \( \lceil \lg n \rceil \). We then conclude that the EXTRACT-MIN operation is performed in \( O(\lg V) \) time. The DECREASE-KEY operation (Algorithm 3, line 11, page 15) requires that heap invariant is maintained. When a key is decreased, the new value is “percolated up” until the heap invariant is satisfied. In the worst case, this percolation is done in \( \lceil \lg n \rceil \) swaps, for the heap of \( n \) elements. The DECREASE-KEY operation is therefore performed in \( O(\lg V) \) time. Another factor that we need to consider is the cost to build the heap upon initialization. In this implementation of Prim’s algorithm, the heap is build with all elements having the key value, which is equal to \( \infty \). This eliminates the requirement of satisfying the heap invariant upon BUILD. Nevertheless, the key value of each element must be set to \( \infty \), thus requiring the iteration over the entire array on which the heap is implemented. The running time of the build operation is therefore \( \Theta(V) \) for the graph with \( V \) vertices. We can therefore say that \( T_B = \Theta(V) \), \( T_E = O(\lg n) \), and \( T_D = O(\lg n) \). The total running time of the implementation of Kruskal’s algorithm with priority queue implemented on a binary heap is
\[ T = \Theta(V) + O(V \lg V) + O(E \lg V) \]
\[ = \Theta(V) + O(V \lg V) + O(V^2 \lg V) \]
\[ = O(E \lg V) \]

for fixed density \( \varepsilon \).

For dense graphs, where \( \varepsilon \) approaches 1, we can expect the asymptotic bound to approach \( O(V^2 \lg V) \), given that \( E = \frac{1}{2} \varepsilon (V^2 - V) \).

From this we can expect performance better than that of the implementation that uses the "brute force" implementation of the priority queue.

**Binomial Heap (pbinh).**

Binomial heap is a collection of binomial trees. The \( i \)-th binomial tree \( B_i \), \( i \geq 0 \) is defined recursively. It consists of a root node and \( i \) children. The \( j \)-th child, \( 1 \leq j \leq i \), is the root of the binomial tree \( B_{j-1} \) [5].

![Figure 4 - Binomial Heap consisting of binomial trees \( B_0 \) through \( B_4 \).](image)

Figure 4 illustrates the binomial heap consisting of five binomial trees, \( B_0 \) through \( B_4 \). CLR provide the following lemma:

**Lemma:**

1. The binomial tree \( B_k \) contains \( 2^k \) nodes.
2. \( B_k \) has a height \( k \).
3. There are exactly \( \binom{k}{i} \) nodes of depth \( i = 0,1,K,k \).
4. The root has a degree \( k \), which is greater than that of any other node. If the children of that root are numbered from left to right, by \( k-1, k-2, K, 0 \), child \( i \) is the root of the subtree \( B_i \) [7].

**Proof:**

The basis of the proof is \( B_0 \); the inductive step is \( B_{k-1} \).

1. Binomial tree \( B_k \) consists of two trees \( B_{k-1} \), therefore, has \( 2^{k-1} + 2^{k-1} = 2^k \) nodes [7].
2. Because of the way two \( B_{k-1} \) trees are linked to form \( B_k \), the maximum depth of a node in \( B_k \) is one greater than the maximum depth of a node in \( B_{k-1} \). By inductive hypothesis, the maximum depth is \( (k-1) + 1 = k \) [7].
3. Let \( D(k, i) \) be the number of nodes at depth \( i \) of a binomial tree \( B_k \). Because \( B_k \) is two linked \( B_{k-1} \) trees, a node at the depth \( i \) in \( B_{k-1} \) appears in \( B_k \) once at depth \( i \), and once at depth \( i + 1 \). Thus, the number of nodes at depth \( i \) in \( B_k \) is the number of nodes at the depth \( i \) in \( B_{k-1} \) plus the number of nodes at depth \( i - 1 \) in \( B_{k-1} \) [7]. Then,

\[
D(k, i) = D(k-1, i) + D(k-1, i-1)
\]

\[
= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}
\]

4. The only node with greater degree in \( B_k \) than \( B_{k-1} \) is the root, which has one more child than \( B_{k-1} \). Because the root of \( B_{k-1} \) has a degree \( k-1 \), the root of \( B_k \) has a degree \( k \). By inductive hypothesis, the children of \( B_{k-1} \) are \( B_{k-2}, B_{k-3}, K, B_0 \). When \( B_{k-1} \) is linked to \( B_{k-1} \), the resulting root are the roots of \( B_{k-1}, B_{k-2}, B_{k-3}, K, B_0 \) [7].

**Corollary:**

The maximum degree of any node in an \( n \)-node binomial tree is \( \lg n \) [7].

**Proof:**

Properties 1 and 4 of lemma.

A binomial heap is a collection of binomial trees, where each tree must be different in size. Each tree must also satisfy the heap invariant, thus ensuring that the element with the minimum key value is stored in one of the roots of the trees. The trees are stored in the order of increasing size. The trees may be organized as a linked list, however in the implementation used in this experiment, the trees were stored in an array of pointers.
Given that the size of a binomial tree $B_k$ is the sum of sizes of all trees $B_0, B_1, K, B_{k-1}$ plus 1, thus $|B_k| = 2^k$, the total number of elements in a binomial heap with $k$ trees is $\sum_{i=0}^{k} 2^i$. Since there is at most one binomial tree with a given root in a binomial heap, we can see that there is at most $\lfloor \lg n \rfloor + 1$ binomial trees in any binomial heap.

Thus, from these properties we see that the minimum element can be located in the worst case in $O(\lg n)$ time. The \texttt{DECREASE-KEY} operation would also be performed in $O(\lg n)$ time, since only the tree in which the key has been decreased needs to have the heap invariant restored. The \texttt{EXTRACT-MIN} operation is not as straight-forward as in the binary heap. When an \texttt{EXTRACT-MIN} is performed, one of the roots is detached. Detaching a root from a binomial tree $B_k$ results in a formation of a set of binomial trees $\{B'_0, B'_1, K B'_{k-1}\}$. Since the trees of the degrees 0,1,K,k−1 are already present in the binomial heap, the newly formed trees must be linked with the existing trees. \{B'_0, B'_1, K B'_{k-1}\} can be viewed as forming a binomial heap as well. Hence, performing an \texttt{EXTRACT-MIN} operation on a binomial heap $H$, results in the formation of two binomial heaps, $H$ and $H'$, which must be merged to form a single heap. Prior to merging the two heaps, the trees of $H'$ must be joined to form the binomial heap, which is done in $O(\lg n)$ time. Once we have two heaps, we link the trees of equal degrees with one another. The link is done in $O(1)$ by making the root of $B_{k-1}$ the child of $B'_{k-1}$, thus resulting in $B_k$. However, this may result in the formation of more than one tree of the same degree. The trees with the same degrees are linked again, to form the tree with a degree one higher. This is repeated for all pairs of trees in both heaps. Since this process is analogous to the bitwise addition with carry, the process of merging two binomial heaps is done in $O(\lg n)$ time. We can therefore conclude that the \texttt{EXTRACT-MIN} operation takes $O(\lg n)$ time. We also need to consider the time to build the binomial heap. This is done in once. However, when used in Prim's algorithm we need to start with the binomial heap containing $V$ elements. The insert operation involves merging of two heaps, thus taking $O(\lg n)$. Thus, insertion of $n$ elements is done in $O(n \lg n)$, which we will consider the build time for our purposes. We then conclude, that

\[
T_B = O(V \lg V) \\
T_E = O(\lg V) \\
T_D = O(\lg V)
\]

The total running time of Prim's algorithm with the priority queue implemented as a binomial heap is
\[ T = O(V \lg V) + O(V \lg V) + O(E \lg V) \\
= O(E \lg V) \]

We then expect performance similar to that of binary heap with proper implementation. For the dense graphs, as with proper heap implementation, we can expect the asymptotic upper bound approach \( O(V^2 \lg V) \). However, given the nature of the data structure used to implement the binomial heap, an array of multiple trees, we can expect higher overhead, hence worse performance.

**Fibonacci Heap (pfibh).**

Like binomial heap, Fibonacci heap is a collection of heap-ordered trees. The sizes of these trees grow according to the Fibonacci sequence defined by the recurrence:

\[
F_k = \begin{cases} 
0 & \text{if } k = 0 \\
1 & \text{if } k = 1 \\
F_{k-1} + F_{k+1} & \text{if } k \geq 2 
\end{cases}
\]

The trees in the Fibonacci heap are not constrained like the trees with binomial heap. They are rooted, but not ordered by size.

![Figure 5 - Fibonacci Heap](image)

The Fibonacci heap is implemented as a circular doubly linked list of roots. Each root points to only one child. The children of each root are also linked in a circular doubly linked list. Each child has a pointer to its parent. The advantages of using the circular doubly linked lists is that a node can be removed in \( O(1) \) time, and two such lists can be concatenated "spliced" together in \( O(1) \) time. Each node has two attributes, a degree, and a marked flag. The degree indicates the number of children the node has; the marked flag indicate whether a given node has lost a child, since the last time that node was made a child of another node. The node becomes unmarked whenever it is made a child of another node, or it is newly created. The pointer \( min[H] \) indicates the tree whose root has
the minimum key value of all other roots in the heap. When the Fibonacci heap is created, like binomial heap it is created empty. To store all vertices on the heap, they must be inserted one at a time. When a node is inserted into the heap, it is added to the root list of the heap. Since this is a concatenation operation, it is done in $O(1)$. \textsc{Find-Min} is performed in $O(1)$ as well, since there is a pointer to the root node with the minimum key value. \textsc{Decrease-Key} operation results in assignment of a new key value to a node. If the new key value of the node is not less than that of its parent, no changes are made to the heap. If the new value is less than the parent, a \textsc{Cut} is performed, followed by a \textsc{Cascading-Cut}. The \textsc{Cut} operation “cuts” the node from its parent and adds it to the root list. The parent’s degree is decremented, and the node that has been cut is “unmarked”. The \textsc{Cascading-Cut} checks if the parent of the node cut is marked. If it is not, the parent is marked. If the parent is marked, a \textsc{Cut} is performed on the parent, followed by a \textsc{Cascading-Cut}. This process is performed recursively, moving the nodes out into the root list, until an unmarked node or a root is reached. When all \textsc{Cascading-Cuts} are completed, the $\min[H]$ pointer is updated, if necessary. The time to perform \textsc{Decrease-Key} operation is amortized to $O(1)$ [13]. The actual cost to perform the \textsc{Decrease-Key} operation is $O(1)$ plus the time to perform the \textsc{Cascading-Cuts}. If \textsc{Cascading-Cut} is recursively called $c$ times, given that the cost of decreasing the key is $O(1)$, the actual cost of performing the \textsc{Decrease-Key} plus the cost of cascading cuts is $O(c)$. \textsc{Decrease-Key} operation does not result in the ordered list of trees. The ordered list property is restored when the \textsc{Extract-Min} operation is performed. The cost of the \textsc{Decrease-Key} operation is therefore amortized. To determine the amortized cost of the \textsc{Decrease-Key} operation, we perform the amortized analysis. We first compute the change in potential. $H$ denotes the Fibonacci heap prior to the \textsc{Decrease-Key} operation. Each recursive call to the \textsc{Cascading-Cut}, except for the last one, cuts a marked node and clears its mark flag. This results in $c$ more trees in the root list than there was initially. If $t(H)$ denotes the number of trees in the root list prior to \textsc{Decrease-Min} operation, then upon completion there are $t(H) + c$ trees in the root list upon completion. There are at most $m(H) - c + 2$ marked nodes. $c - 1$ nodes are unmarked by cascading cuts, and the last call to \textsc{Cascading-Cut} may have marked a node. The change in potential, is then,

\[
(t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H)) = 4 - c
\]

The amortized cost of \textsc{Decrease-Key} is at most

\[
O(c) + 4 - c = O(1)
\]

\textsc{Extract-Min} operation is more complicated. This is where the trees of the root list are consolidated, the work that was delayed when \textsc{Insert} and \textsc{Extract-Min} operations were performed. When the root with the minimum key value is extracted, each one of its
children is made a root. The minimum node is removed from the root list. The CONSOLIDATE operation links the roots of equal degrees until at most, one root of each degree remains. The LINK operation removes a root from the list of roots, and then making it a child of another node, unmarking the node removed. The amortized time for extracting a node with the minimum key value from the Fibonacci heap is $O(D(n))$. The minimum node has at most $D(n)$ children. When CONSOLIDATE is called, the size of the root list is at most $D(n) + t(H) - 1$. $t(H)$ denotes the original root list of $H$. Since one node is extracted, we subtract 1. We also consider $D(n)$ children of the extracted node. Thus, the total actual work is $O(D(n) + t(H))$. The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterwards is at most $(D(n) + 1) + 2m(H)$. The amortized cost is at most

$$O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) = O(D(n)) + O(t(H)) - t(H) = O(D(n))$$

The cost of performing each link is defrayed by reduction in potential when the link reduces the number of roots by one. These operations do not preserve the property that all trees in the Fibonacci heap are unordered binomial trees. But these trees are close enough that the maximum degree $D(n)$ can be bound by $O(\lg n)$. Thus, EXTRACT-MIN operation is performed in $O(\lg n)$ amortized time.

We therefore conclude the time to build the heap to be used in the implementation of Prim's algorithm with priority queue implemented as a Fibonacci heap is

$$T_B = O(V),$$

which accounts for the time to build the heap, which is $O(1)$, and the time to insert $V$ elements onto the heap, each done in $O(1)$. The time to DECREASE-KEY is

$$T_D = O(1)$$

amortized time, and the time to EXTRACT-MIN is

$$T_E = O(\lg V)$$

amortized time. We can therefore expect the total running time of the implementation of Prim's algorithm that uses the Fibonacci heap, to be
\[ T = O(V) + O(\log V) + O(E) \\
    = O(E) \]

amortized time, which for dense graphs can approach \( O(V^2) \). The Fibonacci heap implementation promises a very good performance because the priority queue operations execute in the amortized time.

**Binary Heap – “lazy implementation” (php).**

This implementation of the binary heap is somewhat similar to the implementation of Fibonacci heap in the sense that the maintenance of the heap invariant is delayed until absolutely necessary. In the proper implementation of the binary heap, the heap property is restored immediately after the DECREASE-KEY operation has been performed on one of the nodes. In “lazy” implementation, the heap property is restored prior to performing the EXTRACT-MIN operation. “Lazy” implementation provides an “unstable” heap. This heap can only be used in the implementation of Prim’s algorithm used in this experiment. For more general purpose, the “lazy” implementation may not be suitable. We have taken the advantage that between two EXTRACT-MIN operations, a key of any node will be decreased at most once. Hence, no keys are percolated until the EXTRACT-MIN is to be performed. Prior to performing EXTRACT-MIN, all nodes whose keys have been decreased are “percolated up” if necessary. We now use this information to determine the running time of the implementation of Prim’s algorithm with priority queue implemented using the “lazy” heap. Like in the proper implementation, the time to build the heap is \( T_B = \Theta(V) \). The time to perform EXTRACT-MIN now requires the heap property to be restored, prior to performing the operation. We also restore the heap immediately after the EXTRACT-MIN is performed. We then say that EXTRACT-MIN is performed in \( 2 \log V \) time, hence \( T_E = O(\log V) \). The time to perform DECREASE-KEY operation is then done in \( \Theta(1) \) time. The running time of this implementation is:

\[ T = \Theta(V) + O(V \log V) + O(V_1 \cdot V_2) \\
    = O(E) \]

where \( V_1 \) is the number of operations required to perform EXTRACT-MIN operation, and \( V_2 \) is the number of operations required to rebuild the heap to restore its property after the DECREASE-KEY operation has been performed.

As with Fibonacci heap, \( O(E) \) will approach \( O(V^2) \) for the higher density graphs.
3. EXPERIMENT

The main goal of the experiment is to test the performance of the Kruskal’s and Prim’s algorithms using various implementations of the data structures that can have a significant effect on the algorithm’s performance. Seven implementations were tested:

- Kruskal’s Algorithm with heapsort, and with Path-Compression algorithms
- Kruskal's Algorithm with counting sort and with Path-Compression algorithms
- Prim’s Algorithm with brute force implementation of priority queues
- Prim’s Algorithm with priority queue implemented using a proper implementation of binary heap with “bubble-up” performed each time a decrease-key operation is performed for a vertex
- Prim’s Algorithm with priority queue implemented using a "lazy" implementation of binary heap with “bubble-up” performed after all decrease-key operations are performed for a vertex
- Prim’s Algorithm with priority queue implemented using a binomial heap
- Prim’s Algorithm with priority queue implemented using a Fibonacci heap

The experiment was conducted on a Sun Ultra 1 SBus (UltraSPARC 143 MHz) workstation. Each implementation was tested on nine different types of graphs. The graphs used for the experiment ranged from 100 to 1700 vertices and were generated at random. Hence nine graphs of each size were used. The graphs types varied in density, three were used for each, $p = 0.2$, $p = 0.5$, and $p = 0.8$, where $p$ is the probability of an edge being found between any two vertices. For each density $p$ three ranges of the edge weight $w$ were used, $1 - 10$, $1 - 100$, and $1020 - 1022$, with all weights $w$ having positive integer values.

3.1 Graph Representation

The graphs used in the experiment are stored in compressed text files. Each file contains an adjacency matrix representation of the graph, preceded by an integer, which specifies the number of vertices in the input graph. Each row and each column position represent a vertex of the graph. The values in the adjacency matrix represent the weights of the edges joining the two vertices represented by the row and the column of each value; 0 indicates no edge. Since the graphs used in this experiment are bi-directional, the adjacency matrices used to represent these graphs have a reflexive and symmetric property with respect to the $(0,0)\ldots(n,n)$ diagonal, where $n$ is the number of vertices of the graph. Figure 6 illustrates such representation. No edges originating from, and ending at the same vertex are to be present in the input graphs, therefore the diagonal is always contains zero values. The grayed area of the input matrix is a reflection of the white area.
Hence, only one half of the matrix is needed to represent the connected graph to be used in the experiment.

The adjacency matrix is represented by an array of arrays of integers. An array of integers is allocated to store the values of each row of the matrix with pointers to these arrays stored in a pointer array. Thus, the array indexes of the integer arrays represent the matrix columns, and the array indexes of the pointer array represent the matrix rows.

Upon the startup of the program that implements the algorithm being tested, the first integer in the data file, which indicates the number of vertices in the graph, is immediately read. This value is used to determine the size of memory to be allocated to accommodate the adjacency matrix to be read. Once the memory is allocated the rest of the data file is read. The values are stored in the corresponding locations in the matrix representation described above. When the entire adjacency matrix is read, a function that implements the minimum spanning tree algorithm is invoked, with the adjacency matrix as one of its input arguments. Upon exit, one of the output arguments from the function is a similar adjacency matrix, which contains the representation of the minimum spanning tree. The output matrix is written to a specified output file, the memory is released, and the program is terminated.

The data files used in the experiment were generated using the genegraph program, written specifically for this purpose. The genegraph program accepts as command line arguments the number of vertices, graph density, minimum edge weight, maximum edge weight, and the name of file in which the graph is to be stored. The genegraph program was used to generate the random graphs that were used in the experiment. Each graph generated, was used with all, seven, algorithm implementations studied, in order to compare the performances of the algorithms in one set of data. Table 1 summarizes the types of data used in this experiment. The column that contains the values for the number of vertices indicates a range. A separate data file was generated for every 100 vertices in that range, with the total of 153 data files.
Table 1 – Summary of data files used in the experiment.

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>Density</th>
<th>Minimum Edge Weight</th>
<th>Maximum Edge Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 – 1700</td>
<td>0.2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.2</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.2</td>
<td>1020</td>
<td>1022</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.5</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.5</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.5</td>
<td>1020</td>
<td>1022</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.8</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.8</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>100 – 1700</td>
<td>0.8</td>
<td>1020</td>
<td>1022</td>
</tr>
</tbody>
</table>

3.2 Program

The program was written in C and ran on the UltraSPARC with a 143 MHz processor. Eight algorithms are implemented, although only seven are used in this experiment. The algorithms implemented. The one algorithm that is not used, is one of the worst implementations of Kruskal’s algorithm, which uses the insertion sort and a recursive heuristic to determine if adding an edge will result in a cycle. This implementation was used to demonstrate the severity of the effect of using an inefficient data structure in the areas of “weakness”. The references to the algorithm implementations used in the experiment are summarized in the Table 2. Hence, the khs reference indicates the Kruskal’s with heap sort.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Algorithm Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>khs</td>
<td>Kruskal with Heap Sort</td>
</tr>
<tr>
<td>kcs</td>
<td>Kruskal with Counting Sort</td>
</tr>
<tr>
<td>pbf</td>
<td>Prim Using Brute Force for Queue Implementation</td>
</tr>
<tr>
<td>php_a</td>
<td>Prim with Queue Implemented on Heap (proper)</td>
</tr>
<tr>
<td>php</td>
<td>Prim with Queue Implemented on Heap (lazy)</td>
</tr>
<tr>
<td>pbinh</td>
<td>Prim with Queue Implemented on Binomial Heap</td>
</tr>
<tr>
<td>pfibh</td>
<td>Prim with Queue Implemented on Fibonacci Heap</td>
</tr>
</tbody>
</table>

Table 2 – References to Algorithms Used in the Experiment
Each algorithm is implemented as an individual function, which is invoked by the main program. The main program reads the data file, sets up the input and the output matrices, and runs the algorithms based on the parameters specified by the user. The user of the program specifies the algorithm to use for obtaining a minimum spanning tree and the number of iterations of the algorithm. The multiple iterations of the algorithm allow for more accurate timing results, and thus, for higher precision of measurements. Five iterations were used for each algorithm. The average of the times produced was used as the final running time of the algorithm.

3.2.1 Time Measurements

The time measurements were taken using the `getrusage()` system call. This system call returns a structure `struct rusage`, from the members of which the time in seconds is assembled. The running time was measured prior to the beginning of the algorithm’s execution, and immediately upon termination. The timings were taken in the algorithm function. Only the execution of the code, which implements the algorithm, was timed. I/O, allocation of data structures, and matrix manipulations were ignored. Hence, the running time of the algorithm tested that was obtained using this method represents the actual algorithm execution time as accurately as possible. When the function, which implements the algorithm exits, the time is returned to the main program.

3.2.2 Data Representation

One of the goals in implementing the algorithms studied in this experiment is to use the most efficient, yet “fair” representation of data. The use of pointers and dynamic memory allocation was avoided as much as possible, unless the actual data structure used by the algorithm dictated such. Much effort was placed into maintaining a high degree of consistency when implementing the data structures for all algorithms, in order to prevent the results from being obscured by various overheads that might be associated with maintaining the data structures. Much use was made of the arrays, however, the implementation of the binomial heap and Fibonacci heap does require the use of pointers and dynamic allocation. To defray the cost of the overhead that might be associated with dynamic allocation, all heap nodes are allocated at once, prior to start of the algorithm’s execution, and thus prior to initial timing. The pointers to the nodes allocated are stored on the stack, which is implemented as an array. The pointers to the nodes are dispensed upon request, thus reducing the memory allocation overhead to a minimum. The disjoint-sets and binary heaps are implemented as arrays of records, eliminating the use of pointers entirely.

For Kruskal’s algorithm, the set of edges is representing using an array of structures, each containing the vertex u, the vertex v, and the weight w of the edge. The following structure definition is used to represent such:
struct _tagEdges_
{
    int u;        /* Vertex u */
    int v;        /* Vertex v */
    int weight;   /* Weight w */
}

The sets of the disjoint-set structure are implemented as arrays of integers, where each integer represents a vertex. The sorting is performed on the array of edges represented by the struct _tagEdges_.

The priority queue used in Prim's algorithm is implemented as a structure which contains an array to store the vertices, a member for a head of the queue, a member for a tail of the queue, and the size length counters. Following structure definition implements such:

struct _tagPriQueue_
{
    int* vertices;     /* Array of vertices */
    int head;          /* Head of the queue */
    int tail;          /* Tail of the queue */
    int size;          /* Capacity of the queue */
    int length;        /* Number of elements */
}

The key and the parent of the vertex are each stored in the array of integers, where each cell stores a vertex. Each vertex stored on the priority queue can locate reference its parent \(\pi\) or its key using the index reference. For example, the entry for the parent \(\pi\) of vertex 6, is located in the \(\Pi[6]\), where \(\Pi\) is the variable that contains the array of \(\pi\).

The implementation of the priority queue using "brute force", the EXTRACT-MIN operation is performed by iterating over the entire, thus finding the vertex with the minimum key value. Once such vertex is found, a NIL value is inserted in its place, and the vertex is returned. Hence, the vertex is extracted in \(\Theta(\sqrt{V})\) time.

The binary heap operations, used with Prim's algorithm, are performed on the array field \texttt{vertices} of the struct _tagPriQueue_. The parent, left, and right references are implemented using macros, thus reducing the overhead associated with the function calls.

The vertex representation for the implementation of the priority queue using the binomial and Fibonacci heaps has been altered. Since both implementations require the representation of vertices in multiple ordered trees, much of the vertex information was moved from arrays into the nodes. Binomial heap is a collection of binomial trees, each
containing at least one key value. The following structure defines a single node of the binomial tree:

```c
struct _tagBinNode_
{
    /***
    *** Binomial Tree attributes
    ***/
    struct _tagBinNode_* pParent; /* Pointer to parent */
    struct _tagBinNode_* pLeftChild;
    struct _tagBinNode_* pSibling;

    /***
    *** Graph vertex attributes
    ***/
    int nKey;
    int unVert;
}
```

To allow direct access to any of the nodes in the binomial heap, an array of pointers is used, where the index of each array cell represents the vertex. In addition an array of Boolean values is used to indicate if the vertex is in the queue. These attributes are stored in the structure that represents the entire binomial queue. The definition of the structure is as follows:

```c
struct __tagCollection__
{
    int nCapacity;
    int nTrees; /* Number of trees */
    int nCurrentSize; /* Vertex count */
    struct _tagBinNode_* aTrees; /* Array of trees */
    int* aInQueue; /* Array of flags */
    /* to indicate if */
    /* vertex is in the*/
    /* queue */
    struct _tagBinNode_* apVertexNodes;/* Node references */
}
```

All nodes are allocated ahead of time, and are stored in an array. The pointers to the nodes are dispensed upon request. This process replaces the dynamic allocation to reduce the overhead, as mentioned earlier. The following structure implements the store containing the nodes of the binomial heap:

```c
struct __tagBinHeapMemMgt__
{
    struct _tagBinNode_* aNodes; /* Array of Nodes */
    struct _tagBinNode_** apNodes; /* Array of pointers to*/
```
unsigned unUnused; /* Nodes available */
unsigned unTop; /* Stack Pointer */

Similar representation is used for the priority queue implementation using the Fibonacci heap. The memory management and the collection are implemented in the same manner. The information about each vertex is stored in a node. The key for the vertex is stored in the node, as well. In addition, nodes contain the attributes of the Fibonacci heap. The following structure defines a single node of the Fibonacci heap:

```c
struct __tagFibNode__
{
    /*
     ** Tree attributes
     */
    struct __tagFibNode__* pParent; /* Parent of the node */
    struct __tagFibNode__* pChild; /* One child of node */
    struct __tagFibNode__* pLeft; /* Left sibling */
    struct __tagFibNode__* pRight; /* Right sibling */
    int nDegree; /* Degree of the node */
    int bMark; /* Marks the node if */
    /* node loses a child */

    /*
     ** Vertex attributes
     */
    int nKey; /* Key of the vertex */
    unsigned int unVert; /* Vertex */
}
```

The collection is implemented as follows:

```c
struct __tagFibCollection__
{
    struct __tagFibNode__* pRootList; /* List of roots */
    int nCurrentSize;
    struct __tagFibNode__* pMin; /* Pointer to node with the minimum key value */
    struct __tagFibNode__* aInQueue; /* Array of flags to indicate if vertex is in the queue */
    struct __tagFibNode__* apVertexNodes; /* Node references */
    int nMaxSize;
}```
struct __tagFibNode__* A; /* Consolidation array*/
int Dn; /* Number of trees */
}

The memory management for the Fibonacci heap is implemented in the manner identical to that of the binomial heap. The following structure is used for the memory store:

struct __tagFibHeapMemMgt__
{
    struct __tagFibNode__* aNodes; /* Array of Nodes */
    struct __tagFibNode__* apNodes; /* Array of pointers to nodes */
    unsigned unUnused; /* Nodes available */
    unsigned unTop; /* Stack Pointer */
}

The memory required to implement the data structure for all implementations is allocated ahead of time and is not considered in the time measurements. Since the memory allocation may contribute significantly to the amount of time required to run the algorithm, the entire process is performed in one step, thus yielding the run time of \( \Theta(1) \), which is eliminated from the test entirely.

3.3 Running the Experiment

The experiment was run using the computing facilities of the Computer Science department of Rochester Institute of Technology. Nine UltraSPARC\(s \) were used simultaneously. The graphs were generated on the local disks in order not to use space on the file server. A shell script was used to launch the program multiple times in order to test each algorithm with all variations of the data. The resulting timings were captured and written to a file using Unix's standard I/O redirection. Upon completion, shell scripts were used to remove the graphs from the local disks.
4. Results

To reiterate, the experiment was conducted using three sets of graphs; each set is of different graph density. The densities of graphs used are defined in terms of the probability \( p \) of an edge existing between any two vertices \( u \) and \( v \). The densities of the graphs are \( p = 0.2 \), \( p = 0.5 \), and \( p = 0.8 \). Each set of graphs consists of three subsets. Each subset contains 17 graphs, which range in the number of vertices from 100 to 1700. The subsets are categorized by the range of weights \( w \) that are assigned to an existing edge \((u,v)\) at random. The weight ranges used are 1 through 10, 1 through 100, and 1020 through 1022. Hence, total of 153 graphs was used in the experiment. The data set can be represented graphically as shown in Figure 7.

After the results have been tabulated, it became apparent that the implementation of Prim's algorithm with priority queue implemented on a heap, using the "lazy" implementation of the heap (khs), has outperformed all other implementations in all tests with consistent results with respect to the range of weights \( w \). A slight degradation in performance was observed with the increase of graph density. Relatively consistent performance, with respect to range of the edge weights, was also observed for Prim's algorithm with priority queue implemented on a heap with the "proper" heap implementation (php_a), with slight degradation in performance observed with the increase of graph density. Prim's algorithm with priority queue implemented on a binomial heap (pbinh), displayed similar consistency to the other two implementations php and php_a, although its overall performance was relatively poor. Prim's algorithm with priority queue implemented using brute force (pbf), which was originally intended to be used as an example of a pathological case, provided the same consistency as the other
implementations of the Prim's algorithm, although it has outperformed the pbinh implementation in every case, as well as the implementation of Kruskal's algorithm, where the heapsort is used to sort the edges (khs). For the higher density graphs, pbf has even outperformed the implementation of Kruskal's algorithm that uses the counting sort (kcs). Hence, the observations reveal that the range of values for the range of weights has little or no impact on the performance of the implementations of Prim's algorithm used in this experiment. The slight degradation in performance that was observed with the increase in graph density, has shown a high degree of consistency for all implementations of Prim's algorithm.

The implementations of Kruskal's algorithm have shown far less consistency than the implementations of Prim's algorithm. Kruskal's algorithm with heapsort (khs) had the worst performance in every case. Some degree of degradation was observed with the increase in the range of weights $w$. Severe degradation was observed with the increase of density of the graphs. Kruskal's algorithm with counting sort (kcs) displayed a level of consistency similar to that of khs. Although higher than that of Prim's algorithm's implementation's, the degradation with respect to the graph density appears to be far less severe than that of the khs implementation.

Overall, the best performance in all cases was observed in php. kcs produced the results close to those of php for sparse graphs. In the medium to high density graphs, pfibh produced the results that are close to the results produced by php. php_a in all cases ran slower than pfibh, however the difference in performance of the two appears to be consistent. pbf performed worse than php_a, but likewise, maintained the same consistency in the difference in run times. pbinh maintained the same consistency in performance, although its runtime is substantially slower that that of the rest of the implementations of Prim's algorithm. khs had the worst performance of all, in every case. For dense graphs with $w = 0.8$, kcs ran slightly faster than khs. For the sparse graphs with $w = 0.2$, kcs ran slightly slower than php.

These results are summarized in the Table 3 through Table 11 and accompanying charts in Figure 8 through Figure 16 on the following pages:
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0147</td>
<td>0.0031</td>
<td>0.0066</td>
<td>0.0058</td>
<td>0.0038</td>
<td>0.0141</td>
<td>0.0066</td>
</tr>
<tr>
<td>200</td>
<td>0.0709</td>
<td>0.0114</td>
<td>0.0258</td>
<td>0.0205</td>
<td>0.0117</td>
<td>0.0432</td>
<td>0.0212</td>
</tr>
<tr>
<td>300</td>
<td>0.1831</td>
<td>0.0254</td>
<td>0.0595</td>
<td>0.0454</td>
<td>0.0241</td>
<td>0.0892</td>
<td>0.0388</td>
</tr>
<tr>
<td>400</td>
<td>0.3467</td>
<td>0.0451</td>
<td>0.1030</td>
<td>0.0833</td>
<td>0.0458</td>
<td>0.1484</td>
<td>0.0656</td>
</tr>
<tr>
<td>500</td>
<td>0.5619</td>
<td>0.0706</td>
<td>0.1612</td>
<td>0.1194</td>
<td>0.0611</td>
<td>0.2214</td>
<td>0.0963</td>
</tr>
<tr>
<td>600</td>
<td>0.8487</td>
<td>0.1021</td>
<td>0.2322</td>
<td>0.1713</td>
<td>0.0867</td>
<td>0.3170</td>
<td>0.1338</td>
</tr>
<tr>
<td>700</td>
<td>1.1899</td>
<td>0.1395</td>
<td>0.3190</td>
<td>0.2333</td>
<td>0.1164</td>
<td>0.4199</td>
<td>0.1772</td>
</tr>
<tr>
<td>800</td>
<td>1.5936</td>
<td>0.1823</td>
<td>0.4121</td>
<td>0.3015</td>
<td>0.1494</td>
<td>0.5362</td>
<td>0.2283</td>
</tr>
<tr>
<td>900</td>
<td>2.0922</td>
<td>0.2320</td>
<td>0.5220</td>
<td>0.3809</td>
<td>0.1878</td>
<td>0.6708</td>
<td>0.2853</td>
</tr>
<tr>
<td>1000</td>
<td>2.6412</td>
<td>0.2870</td>
<td>0.6439</td>
<td>0.4692</td>
<td>0.2308</td>
<td>0.8428</td>
<td>0.3517</td>
</tr>
<tr>
<td>1100</td>
<td>3.2705</td>
<td>0.3466</td>
<td>0.7803</td>
<td>0.5680</td>
<td>0.2782</td>
<td>0.9863</td>
<td>0.4190</td>
</tr>
<tr>
<td>1200</td>
<td>4.0365</td>
<td>0.4132</td>
<td>0.9299</td>
<td>0.6803</td>
<td>0.3304</td>
<td>1.1661</td>
<td>0.4969</td>
</tr>
<tr>
<td>1300</td>
<td>4.7130</td>
<td>0.4851</td>
<td>1.0939</td>
<td>0.7945</td>
<td>0.3879</td>
<td>1.3594</td>
<td>0.5801</td>
</tr>
<tr>
<td>1400</td>
<td>5.5150</td>
<td>0.5627</td>
<td>1.2700</td>
<td>0.9225</td>
<td>0.4479</td>
<td>1.5735</td>
<td>0.6650</td>
</tr>
<tr>
<td>1500</td>
<td>6.4591</td>
<td>0.6467</td>
<td>1.4749</td>
<td>1.0734</td>
<td>0.5143</td>
<td>1.7837</td>
<td>0.7591</td>
</tr>
<tr>
<td>1600</td>
<td>7.4359</td>
<td>0.7387</td>
<td>1.6651</td>
<td>1.2249</td>
<td>0.5863</td>
<td>2.0561</td>
<td>0.8660</td>
</tr>
<tr>
<td>1700</td>
<td>8.6536</td>
<td>0.8346</td>
<td>1.8706</td>
<td>1.3646</td>
<td>0.6575</td>
<td>2.2849</td>
<td>0.9370</td>
</tr>
</tbody>
</table>

Table 3 – Performances for sparse graphs with $1 \leq w \leq 10$, with times in CPU seconds
Figure 8 – Performances for sparse graphs with $1 \leq w \leq 10$ (see Table 3)
Graph Density: 0.2
Range of Edge Weights: 1 – 100

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0158</td>
<td>0.0031</td>
<td>0.0065</td>
<td>0.0061</td>
<td>0.0043</td>
<td>0.0140</td>
<td>0.0068</td>
</tr>
<tr>
<td>200</td>
<td>0.0771</td>
<td>0.0115</td>
<td>0.0251</td>
<td>0.0218</td>
<td>0.0137</td>
<td>0.0437</td>
<td>0.0203</td>
</tr>
<tr>
<td>300</td>
<td>0.2038</td>
<td>0.0254</td>
<td>0.0580</td>
<td>0.0485</td>
<td>0.0283</td>
<td>0.0909</td>
<td>0.0407</td>
</tr>
<tr>
<td>400</td>
<td>0.3733</td>
<td>0.0452</td>
<td>0.1006</td>
<td>0.0816</td>
<td>0.0466</td>
<td>0.1516</td>
<td>0.0671</td>
</tr>
<tr>
<td>500</td>
<td>0.6168</td>
<td>0.0710</td>
<td>0.1574</td>
<td>0.1262</td>
<td>0.0709</td>
<td>0.2288</td>
<td>0.1003</td>
</tr>
<tr>
<td>600</td>
<td>0.9265</td>
<td>0.1021</td>
<td>0.2255</td>
<td>0.1784</td>
<td>0.0977</td>
<td>0.3265</td>
<td>0.1404</td>
</tr>
<tr>
<td>700</td>
<td>1.3015</td>
<td>0.1415</td>
<td>0.3099</td>
<td>0.2430</td>
<td>0.1308</td>
<td>0.4323</td>
<td>0.1860</td>
</tr>
<tr>
<td>800</td>
<td>1.7522</td>
<td>0.1897</td>
<td>0.4002</td>
<td>0.3111</td>
<td>0.1651</td>
<td>0.5526</td>
<td>0.2385</td>
</tr>
<tr>
<td>900</td>
<td>2.2851</td>
<td>0.2443</td>
<td>0.5059</td>
<td>0.3928</td>
<td>0.2064</td>
<td>0.7412</td>
<td>0.2971</td>
</tr>
<tr>
<td>1000</td>
<td>2.9086</td>
<td>0.3010</td>
<td>0.6263</td>
<td>0.4823</td>
<td>0.2509</td>
<td>0.8465</td>
<td>0.3699</td>
</tr>
<tr>
<td>1100</td>
<td>3.5995</td>
<td>0.3647</td>
<td>0.7598</td>
<td>0.5837</td>
<td>0.2996</td>
<td>1.0157</td>
<td>0.4345</td>
</tr>
<tr>
<td>1200</td>
<td>4.5098</td>
<td>0.4345</td>
<td>0.9069</td>
<td>0.6943</td>
<td>0.3529</td>
<td>1.1996</td>
<td>0.5151</td>
</tr>
<tr>
<td>1300</td>
<td>5.2311</td>
<td>0.5110</td>
<td>1.0658</td>
<td>0.8095</td>
<td>0.4124</td>
<td>1.4260</td>
<td>0.5999</td>
</tr>
<tr>
<td>1400</td>
<td>6.1445</td>
<td>0.5923</td>
<td>1.2423</td>
<td>0.9344</td>
<td>0.4739</td>
<td>1.6142</td>
<td>0.6886</td>
</tr>
<tr>
<td>1500</td>
<td>7.1988</td>
<td>0.6805</td>
<td>1.4202</td>
<td>1.0760</td>
<td>0.5400</td>
<td>1.8346</td>
<td>0.7829</td>
</tr>
<tr>
<td>1600</td>
<td>8.2506</td>
<td>0.7768</td>
<td>1.6287</td>
<td>1.2313</td>
<td>0.6140</td>
<td>2.0870</td>
<td>0.8917</td>
</tr>
<tr>
<td>1700</td>
<td>9.6819</td>
<td>0.8765</td>
<td>1.8304</td>
<td>1.3793</td>
<td>0.6866</td>
<td>2.3454</td>
<td>1.0021</td>
</tr>
</tbody>
</table>

Table 4 – Performances for sparse graphs with 1 ≤ \( w \) ≤ 100, with times in CPU seconds
Figure 9– Performances for sparse graphs with $1 \leq w \leq 100$ (see Table 4)
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbindh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0115</td>
<td>0.0035</td>
<td>0.0068</td>
<td>0.0053</td>
<td>0.0032</td>
<td>0.0137</td>
<td>0.0063</td>
</tr>
<tr>
<td>200</td>
<td>0.0543</td>
<td>0.0119</td>
<td>0.0262</td>
<td>0.0195</td>
<td>0.1020</td>
<td>0.0423</td>
<td>0.0183</td>
</tr>
<tr>
<td>300</td>
<td>0.1386</td>
<td>0.0257</td>
<td>0.0603</td>
<td>0.0442</td>
<td>0.0224</td>
<td>0.0875</td>
<td>0.0366</td>
</tr>
<tr>
<td>400</td>
<td>0.2578</td>
<td>0.0455</td>
<td>0.1044</td>
<td>0.0767</td>
<td>0.0382</td>
<td>0.1450</td>
<td>0.0620</td>
</tr>
<tr>
<td>500</td>
<td>0.4263</td>
<td>0.0711</td>
<td>0.1625</td>
<td>0.1190</td>
<td>0.0586</td>
<td>0.2194</td>
<td>0.0934</td>
</tr>
<tr>
<td>600</td>
<td>0.6282</td>
<td>0.1023</td>
<td>0.2334</td>
<td>0.1694</td>
<td>0.0837</td>
<td>0.3164</td>
<td>0.1310</td>
</tr>
<tr>
<td>700</td>
<td>0.8994</td>
<td>0.1397</td>
<td>0.3211</td>
<td>0.2312</td>
<td>0.1134</td>
<td>0.4359</td>
<td>0.1737</td>
</tr>
<tr>
<td>800</td>
<td>1.1921</td>
<td>0.1824</td>
<td>0.4147</td>
<td>0.2989</td>
<td>0.1461</td>
<td>0.5331</td>
<td>0.2244</td>
</tr>
<tr>
<td>900</td>
<td>1.5563</td>
<td>0.2320</td>
<td>0.5249</td>
<td>0.3803</td>
<td>0.1843</td>
<td>0.6742</td>
<td>0.2795</td>
</tr>
<tr>
<td>1000</td>
<td>1.9937</td>
<td>0.2866</td>
<td>0.6472</td>
<td>0.4687</td>
<td>0.2269</td>
<td>0.8197</td>
<td>0.3445</td>
</tr>
<tr>
<td>1100</td>
<td>2.4258</td>
<td>0.3468</td>
<td>0.7832</td>
<td>0.5652</td>
<td>0.2736</td>
<td>0.9816</td>
<td>0.4136</td>
</tr>
<tr>
<td>1200</td>
<td>2.9752</td>
<td>0.4133</td>
<td>0.9339</td>
<td>0.6768</td>
<td>0.3259</td>
<td>1.1618</td>
<td>0.4890</td>
</tr>
<tr>
<td>1300</td>
<td>3.4941</td>
<td>0.4851</td>
<td>1.0971</td>
<td>0.7920</td>
<td>0.3823</td>
<td>1.4264</td>
<td>0.5729</td>
</tr>
<tr>
<td>1400</td>
<td>4.1360</td>
<td>0.5622</td>
<td>1.2743</td>
<td>0.9168</td>
<td>0.4430</td>
<td>1.5676</td>
<td>0.6563</td>
</tr>
<tr>
<td>1500</td>
<td>4.7967</td>
<td>0.6461</td>
<td>1.4588</td>
<td>1.0966</td>
<td>0.5126</td>
<td>1.7764</td>
<td>0.7538</td>
</tr>
<tr>
<td>1600</td>
<td>5.4759</td>
<td>0.7384</td>
<td>1.6698</td>
<td>1.2138</td>
<td>0.5809</td>
<td>2.0251</td>
<td>0.8568</td>
</tr>
<tr>
<td>1700</td>
<td>6.3664</td>
<td>0.8335</td>
<td>1.8733</td>
<td>1.3595</td>
<td>0.6520</td>
<td>2.3088</td>
<td>0.9662</td>
</tr>
</tbody>
</table>

Table 5 – Performances for sparse graphs with $1020 \leq w \leq 1022$, with times in CPU seconds
Figure 10 – Performances for sparse graphs with $1020 \leq w \leq 1022$ (see Table 5)
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pblbh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0401</td>
<td>0.0070</td>
<td>0.0069</td>
<td>0.0058</td>
<td>0.0038</td>
<td>0.00139</td>
</tr>
<tr>
<td>200</td>
<td>0.1903</td>
<td>0.0272</td>
<td>0.0270</td>
<td>0.0257</td>
<td>0.0249</td>
<td>0.01919</td>
</tr>
<tr>
<td>300</td>
<td>0.5006</td>
<td>0.1097</td>
<td>0.1072</td>
<td>0.1088</td>
<td>0.1043</td>
<td>0.09003</td>
</tr>
<tr>
<td>400</td>
<td>0.8109</td>
<td>0.1715</td>
<td>0.1676</td>
<td>0.1768</td>
<td>0.1843</td>
<td>0.14933</td>
</tr>
<tr>
<td>500</td>
<td>1.5010</td>
<td>0.2480</td>
<td>0.2410</td>
<td>0.2310</td>
<td>0.2428</td>
<td>0.21627</td>
</tr>
<tr>
<td>600</td>
<td>2.6352</td>
<td>0.3379</td>
<td>0.3330</td>
<td>0.3210</td>
<td>0.3263</td>
<td>0.28630</td>
</tr>
<tr>
<td>700</td>
<td>3.6263</td>
<td>0.4143</td>
<td>0.4120</td>
<td>0.4056</td>
<td>0.4056</td>
<td>0.35841</td>
</tr>
<tr>
<td>800</td>
<td>5.0571</td>
<td>0.5598</td>
<td>0.5425</td>
<td>0.5369</td>
<td>0.5396</td>
<td>0.48685</td>
</tr>
<tr>
<td>900</td>
<td>6.9574</td>
<td>0.8619</td>
<td>0.8565</td>
<td>0.8686</td>
<td>0.8696</td>
<td>0.80501</td>
</tr>
<tr>
<td>1000</td>
<td>10.0000</td>
<td>1.4260</td>
<td>1.4120</td>
<td>1.4210</td>
<td>1.4263</td>
<td>1.38853</td>
</tr>
<tr>
<td>1100</td>
<td>14.5237</td>
<td>1.3661</td>
<td>1.3530</td>
<td>1.3659</td>
<td>1.3569</td>
<td>1.3530</td>
</tr>
<tr>
<td>1200</td>
<td>18.5526</td>
<td>1.1961</td>
<td>1.1859</td>
<td>1.2126</td>
<td>1.2162</td>
<td>1.2126</td>
</tr>
<tr>
<td>1300</td>
<td>24.8104</td>
<td>1.0756</td>
<td>1.0676</td>
<td>1.0805</td>
<td>1.1056</td>
<td>1.0605</td>
</tr>
<tr>
<td>1400</td>
<td>29.8104</td>
<td>1.0194</td>
<td>1.0120</td>
<td>1.0256</td>
<td>1.0431</td>
<td>1.0256</td>
</tr>
<tr>
<td>1500</td>
<td>34.9526</td>
<td>0.9396</td>
<td>0.9330</td>
<td>0.9426</td>
<td>0.9426</td>
<td>0.9396</td>
</tr>
<tr>
<td>1600</td>
<td>40.5526</td>
<td>0.8194</td>
<td>0.8120</td>
<td>0.8256</td>
<td>0.8256</td>
<td>0.8194</td>
</tr>
<tr>
<td>1700</td>
<td>46.1526</td>
<td>0.7288</td>
<td>0.7230</td>
<td>0.7356</td>
<td>0.7356</td>
<td>0.7288</td>
</tr>
</tbody>
</table>

Table 6 - Performances for medium density graphs with $1 \leq w \leq 10$, with times in CPU seconds.
Figure 11 – Performances for medium density graphs with $1 \leq w \leq 10$ (see Table 6)
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0436</td>
<td>0.0070</td>
<td>0.0068</td>
<td>0.0063</td>
<td>0.0047</td>
<td>0.0165</td>
<td>0.0086</td>
</tr>
<tr>
<td>200</td>
<td>0.2147</td>
<td>0.0273</td>
<td>0.0261</td>
<td>0.0225</td>
<td>0.0147</td>
<td>0.0449</td>
<td>0.0217</td>
</tr>
<tr>
<td>300</td>
<td>0.5585</td>
<td>0.0616</td>
<td>0.0603</td>
<td>0.0499</td>
<td>0.0301</td>
<td>0.0930</td>
<td>0.0436</td>
</tr>
<tr>
<td>400</td>
<td>0.9993</td>
<td>0.1091</td>
<td>0.1041</td>
<td>0.0837</td>
<td>0.0491</td>
<td>0.1556</td>
<td>0.0731</td>
</tr>
<tr>
<td>500</td>
<td>1.6661</td>
<td>0.1780</td>
<td>0.1631</td>
<td>0.1306</td>
<td>0.0747</td>
<td>0.2398</td>
<td>0.1202</td>
</tr>
<tr>
<td>600</td>
<td>2.4838</td>
<td>0.2621</td>
<td>0.2349</td>
<td>0.1840</td>
<td>0.1022</td>
<td>0.3339</td>
<td>0.1514</td>
</tr>
<tr>
<td>700</td>
<td>3.4783</td>
<td>0.3577</td>
<td>0.3239</td>
<td>0.2504</td>
<td>0.1373</td>
<td>0.4437</td>
<td>0.2012</td>
</tr>
<tr>
<td>800</td>
<td>4.6982</td>
<td>0.4661</td>
<td>0.4190</td>
<td>0.3220</td>
<td>0.1747</td>
<td>0.5683</td>
<td>0.2580</td>
</tr>
<tr>
<td>900</td>
<td>6.1086</td>
<td>0.5903</td>
<td>0.5308</td>
<td>0.4065</td>
<td>0.2174</td>
<td>0.7081</td>
<td>0.3310</td>
</tr>
<tr>
<td>1000</td>
<td>7.7067</td>
<td>0.7303</td>
<td>0.6568</td>
<td>0.4995</td>
<td>0.2656</td>
<td>0.8679</td>
<td>0.3971</td>
</tr>
<tr>
<td>1100</td>
<td>9.5028</td>
<td>0.8841</td>
<td>0.7977</td>
<td>0.6034</td>
<td>0.3189</td>
<td>1.0441</td>
<td>0.4741</td>
</tr>
<tr>
<td>1200</td>
<td>11.9055</td>
<td>1.0768</td>
<td>0.9501</td>
<td>0.7232</td>
<td>0.3776</td>
<td>1.2346</td>
<td>0.5627</td>
</tr>
<tr>
<td>1300</td>
<td>13.6841</td>
<td>1.2354</td>
<td>1.1190</td>
<td>0.8427</td>
<td>0.4413</td>
<td>1.4683</td>
<td>0.6559</td>
</tr>
<tr>
<td>1400</td>
<td>16.0854</td>
<td>1.4419</td>
<td>1.2987</td>
<td>0.9749</td>
<td>0.5098</td>
<td>1.6574</td>
<td>0.7534</td>
</tr>
<tr>
<td>1500</td>
<td>18.8288</td>
<td>1.6544</td>
<td>1.4870</td>
<td>1.1228</td>
<td>0.5841</td>
<td>1.8760</td>
<td>0.8591</td>
</tr>
<tr>
<td>1600</td>
<td>21.5607</td>
<td>1.8884</td>
<td>1.7049</td>
<td>1.2850</td>
<td>0.6647</td>
<td>2.1367</td>
<td>0.9801</td>
</tr>
<tr>
<td>1700</td>
<td>25.2302</td>
<td>2.1301</td>
<td>1.9153</td>
<td>1.4441</td>
<td>0.7508</td>
<td>2.4498</td>
<td>1.1613</td>
</tr>
</tbody>
</table>

Table 7 – Performances for medium density graphs with $1 \leq w \leq 100$, with times in CPU seconds
Medium density graphs (p = 0.5) with edge weights 1 - 100

Figure 12 – Performances for medium density graphs with 1 ≤ w ≤ 100 (see Table 7)
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfbh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0313</td>
<td>0.0073</td>
<td>0.0070</td>
<td>0.0053</td>
<td>0.0032</td>
<td>0.0135</td>
<td>0.0063</td>
</tr>
<tr>
<td>200</td>
<td>0.1490</td>
<td>0.0278</td>
<td>0.0272</td>
<td>0.0201</td>
<td>0.0109</td>
<td>0.0426</td>
<td>0.0191</td>
</tr>
<tr>
<td>300</td>
<td>0.3857</td>
<td>0.0618</td>
<td>0.0717</td>
<td>0.0461</td>
<td>0.0240</td>
<td>0.0915</td>
<td>0.0390</td>
</tr>
<tr>
<td>400</td>
<td>0.6841</td>
<td>0.1009</td>
<td>0.1078</td>
<td>0.0796</td>
<td>0.0411</td>
<td>0.1482</td>
<td>0.0657</td>
</tr>
<tr>
<td>500</td>
<td>1.1420</td>
<td>0.1721</td>
<td>0.1680</td>
<td>0.1234</td>
<td>0.0637</td>
<td>0.2259</td>
<td>0.1000</td>
</tr>
<tr>
<td>600</td>
<td>1.6992</td>
<td>0.2492</td>
<td>0.2415</td>
<td>0.1764</td>
<td>0.0908</td>
<td>0.3223</td>
<td>0.1404</td>
</tr>
<tr>
<td>700</td>
<td>2.3797</td>
<td>0.3390</td>
<td>0.3321</td>
<td>0.2414</td>
<td>0.1237</td>
<td>0.4299</td>
<td>0.1883</td>
</tr>
<tr>
<td>800</td>
<td>3.1604</td>
<td>0.4430</td>
<td>0.4325</td>
<td>0.3132</td>
<td>0.1603</td>
<td>0.5653</td>
<td>0.2436</td>
</tr>
<tr>
<td>900</td>
<td>4.1703</td>
<td>0.5614</td>
<td>0.5636</td>
<td>0.3978</td>
<td>0.2025</td>
<td>0.6927</td>
<td>0.3058</td>
</tr>
<tr>
<td>1000</td>
<td>5.2069</td>
<td>0.6934</td>
<td>0.6694</td>
<td>0.4899</td>
<td>0.2492</td>
<td>0.8463</td>
<td>0.3747</td>
</tr>
<tr>
<td>1100</td>
<td>6.3499</td>
<td>0.8401</td>
<td>0.8099</td>
<td>0.5934</td>
<td>0.3013</td>
<td>1.0249</td>
<td>0.4611</td>
</tr>
<tr>
<td>1200</td>
<td>7.9240</td>
<td>0.9995</td>
<td>0.9663</td>
<td>0.7106</td>
<td>0.3596</td>
<td>1.2236</td>
<td>0.5635</td>
</tr>
<tr>
<td>1300</td>
<td>9.3117</td>
<td>1.1728</td>
<td>1.1354</td>
<td>0.8324</td>
<td>0.4219</td>
<td>1.4489</td>
<td>0.6269</td>
</tr>
<tr>
<td>1400</td>
<td>10.7743</td>
<td>1.3621</td>
<td>1.3199</td>
<td>0.9657</td>
<td>0.4904</td>
<td>1.6257</td>
<td>0.7244</td>
</tr>
<tr>
<td>1500</td>
<td>12.4001</td>
<td>1.5644</td>
<td>1.5121</td>
<td>1.1116</td>
<td>0.5621</td>
<td>1.8579</td>
<td>0.8392</td>
</tr>
<tr>
<td>1600</td>
<td>14.3191</td>
<td>1.7913</td>
<td>1.7267</td>
<td>1.2743</td>
<td>0.6427</td>
<td>2.1243</td>
<td>0.9525</td>
</tr>
<tr>
<td>1700</td>
<td>16.7918</td>
<td>2.0175</td>
<td>1.9417</td>
<td>1.4319</td>
<td>0.7251</td>
<td>2.3949</td>
<td>1.0694</td>
</tr>
</tbody>
</table>

Table 8 – Performances for medium density graphs with $1020 \leq w \leq 1022$, with times in CPU seconds
Medium density graphs (p = 0.5) with edge weights 1020 - 1022

Figure 13 – Performances for medium density graphs with 1020 ≤ w ≤ 1022 (see Table 8)
### Graph Density: 0.8
*Range of Edge Weights: 1 – 10*

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0682</td>
<td>0.0108</td>
<td>0.0070</td>
<td>0.0059</td>
<td>0.0038</td>
<td>0.0140</td>
<td>0.0069</td>
</tr>
<tr>
<td>200</td>
<td>0.3294</td>
<td>0.0429</td>
<td>0.0277</td>
<td>0.0210</td>
<td>0.0121</td>
<td>0.0438</td>
<td>0.0207</td>
</tr>
<tr>
<td>300</td>
<td>0.8385</td>
<td>0.0968</td>
<td>0.0639</td>
<td>0.0477</td>
<td>0.0265</td>
<td>0.0917</td>
<td>0.0421</td>
</tr>
<tr>
<td>400</td>
<td>1.5404</td>
<td>0.1746</td>
<td>0.1101</td>
<td>0.0828</td>
<td>0.0450</td>
<td>0.1525</td>
<td>0.0720</td>
</tr>
<tr>
<td>500</td>
<td>2.5484</td>
<td>0.2741</td>
<td>0.1716</td>
<td>0.1275</td>
<td>0.0694</td>
<td>0.2340</td>
<td>0.1084</td>
</tr>
<tr>
<td>600</td>
<td>3.7630</td>
<td>0.3946</td>
<td>0.2469</td>
<td>0.1825</td>
<td>0.0986</td>
<td>0.3328</td>
<td>0.1547</td>
</tr>
<tr>
<td>700</td>
<td>5.2957</td>
<td>0.5371</td>
<td>0.3396</td>
<td>0.2496</td>
<td>0.1338</td>
<td>0.4434</td>
<td>0.2049</td>
</tr>
<tr>
<td>800</td>
<td>7.0743</td>
<td>0.7012</td>
<td>0.4389</td>
<td>0.3241</td>
<td>0.1733</td>
<td>0.5696</td>
<td>0.2654</td>
</tr>
<tr>
<td>900</td>
<td>9.2075</td>
<td>0.8895</td>
<td>0.5555</td>
<td>0.4104</td>
<td>0.2186</td>
<td>0.7184</td>
<td>0.3337</td>
</tr>
<tr>
<td>1000</td>
<td>11.5917</td>
<td>1.0985</td>
<td>0.6846</td>
<td>0.5054</td>
<td>0.2693</td>
<td>0.8783</td>
<td>0.4108</td>
</tr>
<tr>
<td>1100</td>
<td>14.2646</td>
<td>1.3391</td>
<td>0.8291</td>
<td>0.6645</td>
<td>0.3282</td>
<td>1.0556</td>
<td>0.4957</td>
</tr>
<tr>
<td>1200</td>
<td>17.6639</td>
<td>1.5826</td>
<td>0.9891</td>
<td>0.7329</td>
<td>0.3882</td>
<td>1.2538</td>
<td>0.5808</td>
</tr>
<tr>
<td>1300</td>
<td>20.6639</td>
<td>1.8575</td>
<td>1.1623</td>
<td>0.8591</td>
<td>0.4546</td>
<td>1.4923</td>
<td>0.6869</td>
</tr>
<tr>
<td>1400</td>
<td>23.9797</td>
<td>2.1588</td>
<td>1.3493</td>
<td>0.9953</td>
<td>0.5275</td>
<td>1.6834</td>
<td>0.7978</td>
</tr>
<tr>
<td>1500</td>
<td>27.8348</td>
<td>2.4826</td>
<td>1.5447</td>
<td>1.1479</td>
<td>0.6058</td>
<td>1.9195</td>
<td>0.9337</td>
</tr>
<tr>
<td>1600</td>
<td>32.0384</td>
<td>2.8346</td>
<td>1.7676</td>
<td>1.3158</td>
<td>0.6924</td>
<td>2.2692</td>
<td>1.0410</td>
</tr>
<tr>
<td>1700</td>
<td>37.4164</td>
<td>3.1890</td>
<td>1.9893</td>
<td>1.4811</td>
<td>0.7829</td>
<td>2.4799</td>
<td>1.1652</td>
</tr>
</tbody>
</table>

Table 9 – Performances for dense graphs with $1 \leq \nu \leq 10$, with times in CPU seconds
Figure 14 – Performances for dense graphs with $1 \leq w \leq 10$ (see Table 9)
<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0738</td>
<td>0.0108</td>
<td>0.0069</td>
<td>0.0064</td>
<td>0.0048</td>
<td>0.0146</td>
<td>0.0076</td>
</tr>
<tr>
<td>200</td>
<td>0.2359</td>
<td>0.0430</td>
<td>0.0266</td>
<td>0.0229</td>
<td>0.0151</td>
<td>0.0465</td>
<td>0.0226</td>
</tr>
<tr>
<td>300</td>
<td>0.2928</td>
<td>0.0970</td>
<td>0.0621</td>
<td>0.0507</td>
<td>0.0311</td>
<td>0.0947</td>
<td>0.0451</td>
</tr>
<tr>
<td>400</td>
<td>1.6901</td>
<td>0.1811</td>
<td>0.1075</td>
<td>0.0859</td>
<td>0.5080</td>
<td>0.1659</td>
<td>0.0770</td>
</tr>
<tr>
<td>500</td>
<td>2.7794</td>
<td>0.2688</td>
<td>0.1684</td>
<td>0.1320</td>
<td>0.0759</td>
<td>0.2394</td>
<td>0.1143</td>
</tr>
<tr>
<td>600</td>
<td>4.2202</td>
<td>0.4176</td>
<td>0.2425</td>
<td>0.1871</td>
<td>0.1055</td>
<td>0.3455</td>
<td>0.1603</td>
</tr>
<tr>
<td>700</td>
<td>5.8406</td>
<td>0.5668</td>
<td>0.3339</td>
<td>0.2560</td>
<td>0.1428</td>
<td>0.4540</td>
<td>0.2162</td>
</tr>
<tr>
<td>800</td>
<td>7.8653</td>
<td>0.7396</td>
<td>0.4324</td>
<td>0.3297</td>
<td>0.1816</td>
<td>0.5950</td>
<td>0.2758</td>
</tr>
<tr>
<td>900</td>
<td>10.2333</td>
<td>0.9385</td>
<td>0.5476</td>
<td>0.4168</td>
<td>0.2278</td>
<td>0.7305</td>
<td>0.3441</td>
</tr>
<tr>
<td>1000</td>
<td>12.8628</td>
<td>1.1590</td>
<td>0.6759</td>
<td>0.5134</td>
<td>0.2783</td>
<td>0.8665</td>
<td>0.4253</td>
</tr>
<tr>
<td>1100</td>
<td>15.7993</td>
<td>1.4029</td>
<td>0.8196</td>
<td>0.6191</td>
<td>0.3356</td>
<td>1.0770</td>
<td>0.5096</td>
</tr>
<tr>
<td>1200</td>
<td>19.7110</td>
<td>1.6724</td>
<td>0.9767</td>
<td>0.7412</td>
<td>0.3994</td>
<td>1.2881</td>
<td>0.6162</td>
</tr>
<tr>
<td>1300</td>
<td>22.9228</td>
<td>1.9649</td>
<td>1.1531</td>
<td>0.8676</td>
<td>0.4682</td>
<td>1.5077</td>
<td>0.7033</td>
</tr>
<tr>
<td>1400</td>
<td>26.6568</td>
<td>2.2788</td>
<td>1.3379</td>
<td>1.0060</td>
<td>0.5401</td>
<td>1.7072</td>
<td>0.8096</td>
</tr>
<tr>
<td>1500</td>
<td>30.8877</td>
<td>2.6208</td>
<td>1.5322</td>
<td>1.1597</td>
<td>0.6193</td>
<td>1.9407</td>
<td>0.9321</td>
</tr>
<tr>
<td>1600</td>
<td>35.8009</td>
<td>2.9895</td>
<td>1.8616</td>
<td>1.3248</td>
<td>0.7103</td>
<td>2.2323</td>
<td>1.0511</td>
</tr>
<tr>
<td>1700</td>
<td>41.5835</td>
<td>3.3666</td>
<td>1.9668</td>
<td>1.4885</td>
<td>0.7938</td>
<td>2.5511</td>
<td>1.2263</td>
</tr>
</tbody>
</table>

Table 10 – Performances for dense graphs with $1 \leq w \leq 100$, with times in CPU seconds
Dense graphs (p = 0.8) with edge weights 1 - 100

Figure 15 – Performances for dense graphs with 1 ≤ w ≤ 100 (see Table 10), with times in CPU seconds
Graph Density: 0.8
Range of Edge Weights: 1020 – 1022

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs</th>
<th>kcs</th>
<th>pbf</th>
<th>php_a</th>
<th>php</th>
<th>pbinh</th>
<th>pfibh</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0525</td>
<td>0.0562</td>
<td>0.0071</td>
<td>0.0054</td>
<td>0.0032</td>
<td>0.0138</td>
<td>0.0065</td>
</tr>
<tr>
<td>200</td>
<td>0.2452</td>
<td>0.2170</td>
<td>0.0277</td>
<td>0.0209</td>
<td>0.0113</td>
<td>0.0436</td>
<td>0.0198</td>
</tr>
<tr>
<td>300</td>
<td>0.6237</td>
<td>0.4881</td>
<td>0.0639</td>
<td>0.0473</td>
<td>0.0252</td>
<td>0.0914</td>
<td>0.0403</td>
</tr>
<tr>
<td>400</td>
<td>1.1482</td>
<td>0.8768</td>
<td>0.1101</td>
<td>0.0815</td>
<td>0.0436</td>
<td>0.1518</td>
<td>0.0694</td>
</tr>
<tr>
<td>500</td>
<td>1.9403</td>
<td>1.3727</td>
<td>0.1718</td>
<td>0.1269</td>
<td>0.0674</td>
<td>0.2324</td>
<td>0.1053</td>
</tr>
<tr>
<td>600</td>
<td>2.7999</td>
<td>1.9819</td>
<td>0.2475</td>
<td>0.1824</td>
<td>0.0963</td>
<td>0.3381</td>
<td>0.1475</td>
</tr>
<tr>
<td>700</td>
<td>3.9653</td>
<td>2.7812</td>
<td>0.3394</td>
<td>0.2494</td>
<td>0.1316</td>
<td>0.4442</td>
<td>0.2016</td>
</tr>
<tr>
<td>800</td>
<td>5.2624</td>
<td>3.5189</td>
<td>0.4387</td>
<td>0.3239</td>
<td>0.1702</td>
<td>0.5688</td>
<td>0.2595</td>
</tr>
<tr>
<td>900</td>
<td>6.8018</td>
<td>4.4565</td>
<td>0.5551</td>
<td>0.4101</td>
<td>0.2154</td>
<td>0.7154</td>
<td>0.3272</td>
</tr>
<tr>
<td>1000</td>
<td>8.6712</td>
<td>5.5038</td>
<td>0.6849</td>
<td>0.5059</td>
<td>0.2651</td>
<td>0.8739</td>
<td>0.4026</td>
</tr>
<tr>
<td>1100</td>
<td>10.5708</td>
<td>6.6648</td>
<td>0.8303</td>
<td>0.6134</td>
<td>0.3209</td>
<td>1.0603</td>
<td>0.4818</td>
</tr>
<tr>
<td>1200</td>
<td>12.9411</td>
<td>7.9342</td>
<td>0.9886</td>
<td>0.7350</td>
<td>0.3829</td>
<td>1.2528</td>
<td>0.5730</td>
</tr>
<tr>
<td>1300</td>
<td>15.0497</td>
<td>9.3148</td>
<td>1.1636</td>
<td>0.8613</td>
<td>0.4504</td>
<td>1.4629</td>
<td>0.6703</td>
</tr>
<tr>
<td>1400</td>
<td>17.9377</td>
<td>10.8058</td>
<td>1.3494</td>
<td>0.9968</td>
<td>0.5216</td>
<td>1.7059</td>
<td>0.7770</td>
</tr>
<tr>
<td>1500</td>
<td>20.7476</td>
<td>12.4265</td>
<td>1.5450</td>
<td>1.1484</td>
<td>0.6003</td>
<td>1.9234</td>
<td>0.8810</td>
</tr>
<tr>
<td>1600</td>
<td>23.6403</td>
<td>14.2014</td>
<td>1.7680</td>
<td>1.3169</td>
<td>0.6853</td>
<td>2.1915</td>
<td>1.0054</td>
</tr>
<tr>
<td>1700</td>
<td>27.3645</td>
<td>15.9829</td>
<td>2.0573</td>
<td>1.4789</td>
<td>0.7751</td>
<td>2.4779</td>
<td>1.1341</td>
</tr>
</tbody>
</table>

Table 11 – Performances for dense graphs with 1020 ≤ w ≤ 1022, with times in CPU seconds
Dense graphs \((p = 0.8)\) with edge weights 1020 - 1022

![Graph showing running time vs number of vertices for different algorithms.]

Figure 16 – Performances for dense graphs with \(1020 \leq w \leq 1022\) (see Table 11)
5. **Analysis of Results**

Three sets of graphs were used in the experiment. The densities of graphs used are defined in terms of the probability $p$ of an edge existing between any two vertices $u$ and $v$, hence $p = \varepsilon$. Three densities used in this experiment, one for each set of graphs, are $p = 0.2$, $p = 0.5$, and $p = 0.8$. Each set of graphs consists of three subsets. Each subset contains 17 graphs, which range in the number of vertices from 100 to 1700, within one of the tree ranges of edge weights, 0 – 10, 0 – 100, and 1020 – 1022. The weights are randomly assigned to any existing edge $(u,v)$. The total of 153 graphs with uniform distribution in these intervals was used in the experiment. The data set can be represented graphically as shown in Figure 7.

### 5.1 Kruskal’s Algorithm

As we showed earlier, sorting edges by weight can be the most contributing factor to the run time of implementation of Kruskal’s algorithm used in this experiment. To confirm that, a performance measurement of Kruskal’s algorithm was made on seven graphs with 100 – 700 vertices and density $p = 0.2$, for which the algorithm performed well. The results are displayed Table 12.

Figure 17 displays the comparison of the Kruskal’s algorithm with sorting not timed versus the khs ad kcs implementations with sorting timed.

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>khs (Heapsort)</th>
<th>kcs (Counting Sort)</th>
<th>sort not timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0525</td>
<td>0.0562</td>
<td>0.0026</td>
</tr>
<tr>
<td>200</td>
<td>0.2452</td>
<td>0.2170</td>
<td>0.0094</td>
</tr>
<tr>
<td>300</td>
<td>0.6237</td>
<td>0.4881</td>
<td>0.0210</td>
</tr>
<tr>
<td>400</td>
<td>1.1482</td>
<td>0.8768</td>
<td>0.0350</td>
</tr>
<tr>
<td>500</td>
<td>1.9403</td>
<td>1.3727</td>
<td>0.0570</td>
</tr>
<tr>
<td>600</td>
<td>2.7999</td>
<td>1.9819</td>
<td>0.0818</td>
</tr>
<tr>
<td>700</td>
<td>3.9653</td>
<td>2.7812</td>
<td>0.1124</td>
</tr>
</tbody>
</table>

**Table 12** – Performance of Kruskal’s Algorithm excluding the sorting time
We can see the differences in growth rates of each implementation by finding the ratios of performance for several graphs with a number of vertices $V$. Table 13 contains the ratios of time spent by Kruskal’s algorithm performing the disjoint-set operations and the time spent sorting. We arrive at the results by dividing time spent sorting by the time spent performing disjoint set operations $R = \frac{T - T_S}{T_S}$.

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>$R_{HS}$</th>
<th>$R_{CS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>19.1923</td>
<td>20.6154</td>
</tr>
<tr>
<td>200</td>
<td>25.0851</td>
<td>22.0851</td>
</tr>
<tr>
<td>300</td>
<td>28.7000</td>
<td>22.2429</td>
</tr>
<tr>
<td>400</td>
<td>31.8057</td>
<td>24.0514</td>
</tr>
<tr>
<td>500</td>
<td>33.0404</td>
<td>23.0825</td>
</tr>
<tr>
<td>600</td>
<td>33.2286</td>
<td>23.2286</td>
</tr>
<tr>
<td>700</td>
<td>34.2785</td>
<td>23.7438</td>
</tr>
</tbody>
</table>

Table 13 – Growth ratio of two implementations of Kruskal’s algorithm

We can see that the rate of growth of the ratio for the implementation of Kruskal’s algorithm using the heapsort is higher than that of the implementation using the counting sort. Figure 18 illustrates the same graphically. The ratio obtained from the heapsort
implementation appears to be logarithmic, while the ratio obtained from the counting sort implementation appears to be linear.

![Rate of growth of ratio of time spent sorting and time spent performing disjoint-set operations](image)

**Figure 18 – Growth ratio of two implementations of Kruskal’s algorithm**

Hence, the complexity of khs is determined by the heapsort. The complexity of kcs is determined by both, the counting sort and the disjoint-set operations. Since the counting sort runs in linear time, the kcs outperformed the khs, which uses the heapsort that runs in logarithmic time.

Looking at the results, we observe that the performance of Kruskal’s algorithm is strongly affected by the density of the graph. Figure 19 through Figure 21 illustrate the differences in growth, of the running time as a function of number of vertices. Heapsort appears to be affected the most by the increase in density. This is due to the complexity of the implementation of Kruskal’s algorithm, which uses the heapsort. Since the time spent sorting edges is \( O(E \log E) \), given the implication of higher density meaning a larger number of edges, a drop in performance is a result. The performance of the implementation of Kruskal’s algorithm that uses a counting sort does not appear to be affected too severely by the increase in graph density. However, it appears that the magnitude of the edge weights had a strong impact on the performance of the implementation that uses the counting sort. The counting sort determines the size of the array that stores the intermediate information (Page 19) to be the magnitude of the largest value to be sorted. Given that the counting sort runs in linear time, the performance of the sort is determined by the largest of the values to be sorted.
We observe that when $1020 \leq w \leq 1022$. In this weight range, the implementation that uses the heapsort even outperforms the implementation that uses the counting sort for the sparse graphs.

We can therefore conclude that in the cases when the highest edge weight value is in the low numbers, we can expect a very good performance from the implementation of Kruskal’s algorithm where the counting sort is used.

Figure 19 – Effect of density on performance of Kruskal’s algorithm for graphs with $1 \leq w \leq 10$. 

![Comparison of performances of implementations of Kruskal’s algorithm for graphs of various densities with a weight range 1 - 10](image)
Comparison of performances of implementations of Kruskal’s algorithm for graphs of various densities with a weight range $w = 1 - 100$

![Graph](chart1.png)

Figure 20 – Effect of density on performance of Kruskal's algorithm for graphs with $1 \leq w \leq 100$.

Comparison of performances of implementations of Kruskal's algorithm for graphs of various densities with a weight range $w = 1020 - 1022$

![Graph](chart2.png)

Figure 21 – Effect of density on performance of Kruskal's algorithm for graphs with $1020 \leq w \leq 1022$. 
5.2 Prim’s Algorithm

When analyzing the implementations of Prim’s algorithm, we obtained three different asymptotic run-time bounds. The implementation with priority queue implemented using “brute force” was estimated to run in \( O(EV) = O(V^2) \) time. The implementation that uses a proper implementation of binary heap for the priority queue and the implementation that uses a binomial heap to implement the priority queue were estimated to run in \( O(E \lg V) = O(V^2 \lg V) \) time. The implementation that uses the “lazy” binary heap and the implementation that uses the Fibonacci heap were estimated to run in \( O(E) = O(V^2) \) time for the fixed density \( \varepsilon \).

The first step is to verify that the ratio of the performance curves of the implementations that are within the same time bound is constant. We will use the implementation of Prim’s algorithm where the priority queue is implemented using the proper binary heap as the base, against which all other implementations are to be compared. The ratio is calculated using

\[
R_I = \frac{T_I}{T_{php_a}}
\]

where \( I \) is the implementation whose ratio is measured against the php_a implementation. Table 14 through Table 16 show the results obtained for all densities and the vertex ranges:

<table>
<thead>
<tr>
<th>( V )</th>
<th>( p = 0.2 )</th>
<th>( p = 0.5 )</th>
<th>( p = 0.8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_{php} )</td>
<td>( R_{php} )</td>
<td>( R_{php} )</td>
<td>( R_{php} )</td>
</tr>
<tr>
<td>100</td>
<td>1.1379</td>
<td>0.6552</td>
<td>2.4310</td>
</tr>
<tr>
<td>200</td>
<td>1.2585</td>
<td>0.5707</td>
<td>2.1073</td>
</tr>
<tr>
<td>300</td>
<td>1.3106</td>
<td>0.5308</td>
<td>1.9648</td>
</tr>
<tr>
<td>400</td>
<td>1.3265</td>
<td>0.5498</td>
<td>1.7815</td>
</tr>
<tr>
<td>500</td>
<td>1.3501</td>
<td>0.5117</td>
<td>1.6543</td>
</tr>
<tr>
<td>600</td>
<td>1.3552</td>
<td>0.5061</td>
<td>1.5806</td>
</tr>
<tr>
<td>700</td>
<td>1.3673</td>
<td>0.4889</td>
<td>1.7968</td>
</tr>
<tr>
<td>800</td>
<td>1.3668</td>
<td>0.4955</td>
<td>1.7784</td>
</tr>
<tr>
<td>900</td>
<td>1.3704</td>
<td>0.4930</td>
<td>1.7611</td>
</tr>
<tr>
<td>1000</td>
<td>1.3723</td>
<td>0.4919</td>
<td>1.7962</td>
</tr>
<tr>
<td>1100</td>
<td>1.3738</td>
<td>0.4896</td>
<td>1.7364</td>
</tr>
<tr>
<td>1200</td>
<td>1.3699</td>
<td>0.4857</td>
<td>1.7141</td>
</tr>
<tr>
<td>1300</td>
<td>1.3798</td>
<td>0.4882</td>
<td>1.7110</td>
</tr>
<tr>
<td>1400</td>
<td>1.3757</td>
<td>0.4855</td>
<td>1.7057</td>
</tr>
<tr>
<td>1500</td>
<td>1.3740</td>
<td>0.4791</td>
<td>1.6817</td>
</tr>
<tr>
<td>1600</td>
<td>1.3594</td>
<td>0.4787</td>
<td>1.6786</td>
</tr>
<tr>
<td>1700</td>
<td>1.3708</td>
<td>0.4818</td>
<td>1.6744</td>
</tr>
</tbody>
</table>

Table 14 – Ratios obtained for all implementations of Prim’s algorithm and the implementation that uses a proper implementation of binary heap (php_a) for \( 1 \leq w \leq 10 \)
### Table 15 – Ratios obtained for all implementations of Prim’s algorithm and the implementation that uses a proper implementation of binary heap (php_a) for $1 \leq w \leq 100$

<table>
<thead>
<tr>
<th>$V$</th>
<th>$p = 0.2$</th>
<th>$p = 0.5$</th>
<th>$p = 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
</tbody>
</table>

### Table 16 – Ratios obtained for all implementations of Prim’s algorithm and the implementation that uses a proper implementation of binary heap (php_a) for $1020 \leq w \leq 1022$

<table>
<thead>
<tr>
<th>$V$</th>
<th>$p = 0.2$</th>
<th>$p = 0.5$</th>
<th>$p = 0.8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
<tr>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
<td>$R_{php}$</td>
</tr>
</tbody>
</table>

Page 65
Running Time ratios of implementations of Prim's algorithm and implementation that uses proper implementation of binary heap for weights 1 - 10

Figure 22 – Ratios obtained for all implementations of Prim's algorithm and the implementation that uses a proper implementation of binary heap (php_a) for 1 \leq w \leq 10

Running Time ratios of implementations of Prim's algorithm and implementation that uses proper implementation of binary heap for weights 1 - 100

Figure 23 – Ratios obtained for all implementations of Prim's algorithm and the implementation that uses a proper implementation of binary heap (php_a) for 1 \leq w \leq 100

Page 66
Running Time ratios of Implementations of Prim’s algorithm and implementation that uses proper implementation of binary heap for weights 1020 - 1022

![Graph showing running time ratios](image)

**Figure 24** – Ratios obtained for all implementations of Prim’s algorithm and the implementation that uses a proper implementation of binary heap (php_a) for 1020 ≤ w ≤ 1022

**Brute Force Implementation (pbf)**

Figure 22 through Figure 24 show that the ratios obtained for the graphs of different densities for all ranges of edge weights are fairly consistent. The ratio for the “brute force” implementation displays a slight growth in magnitude for all graphs used. Looking at the numbers we can see that the ratio is fairly constant, although it displays a fairly small growth. The asymptotic upper bound for the “brute force” implementation of Prim’s algorithm is estimated to remain consistent regardless of the value of e. The running time of the algorithm is dominated by the EXTRACT-MIN operation, which iterates over the entire array of vertices precisely V times for very iteration of the algorithm. Each implementation has a potential to perform $V^2$ iterations over all vertices for each adjacent edge, however in the case of the “brute force” force implementation of Prim’s algorithm, $V^2$ complexity is assured, in order to perform all EXTRACT-MIN operations necessary to complete the task.

Figure 25 shows that regardless of the density of the graph, or the range of weights w, the “brute force” implementation of Prim’s algorithm will provide very consistent performance. According to Figure 14, Figure 15, and Figure 16, the implementation of Prim’s algorithm with priority queue implemented using “brute force” has outperformed the implementation with priority queue implemented using binomial heap. This can be attributed to the results of the analysis of the binomial heap, which revealed that as the density of the graph approaches 1, the asymptotic complexity of the implementation of
Prim’s algorithm that uses the binomial heap approaches $O(V^2)$, the asymptotic bound of the “brute force” implementation of the algorithm.

![Change in performance of Prim's algorithm which uses "brute force" implementation of priority queue](image)

**Figure 25** – Change in performance of “brute force” implementation of Prim’s algorithm for different data sets.

**Proper Binary Heap Implementation (php_a)**

Because we use the performance of the implementation of Prim’s algorithm that uses a proper binary heap as the value against which performances of other implementations are measured, we examine the change density and change in the range of weights $w$, that affect the performance of that implementation.

Figure 26 displays a change in performance similar to that shown in Figure 25. The curves in Figure 26 display a higher degree of divergence with respect to graph density, as the number of vertices increases. Thus, the performance appeared to drop slightly for the more dense graphs. This suggests that for the graphs with number of vertices larger than 1700 the change in performance will be more visible for more dense graphs. We can also speculate that if very sparse graphs, $0 < p \leq 0.2$, had been used in the experiment, we would have been able to see a more significant change in performance with the change in graph density. Nevertheless, according the performance ratio obtained, we see that the implementation of Prim’s algorithm, which uses a binary heap implementation of priority
queue outperforms the implementation of where the priority queue is implemented using "brute force".

\[
\begin{array}{c}
\text{Change in performance of Prim's algorithm which uses proper "binary heap" implementation of priority queue} \\
\text{Figure 26 – Change in performance of “binary heap” implementation of Prim's algorithm for different data sets.}
\end{array}
\]

**Binomial Heap Implementation (pbinh)**

The implementation where the priority queue is implemented as a binomial heap had the worst performance of all implementations of Prim's algorithm. Although its asymptotic upper bound is the same as the binary heap implementation, the poor performance can be attributed to the implementation of the binomial heap data structure. The binomial heap is a list of binomial trees, where every tree is implemented as a list of individually allocated nodes linked using pointers. EXTRACT-MIN, which results in a large number of MERGE and UNION operations, performs a significantly more work than do other implementations used in this experiment. The overhead associated with the use of binomial heap results in a constant value large enough to result in a performance that is worse than some of the implementations with higher upper bound produced in this experiment. We speculate that if the number of vertices is large enough, the implementation that uses the binomial heap will eventually perform as well as the other implementations with the same asymptotic upper bound. Looking at Figure 27 we see that for the most part the performance of the algorithm appears to be fairly consistent, regardless of the graph density or the range of edge weights. As with other
implementation, the curves begin to diverge as the number of vertices becomes large. We observe a slight drop in performance as the number of vertices increases. As with the implementation, where the binary heap is used for the priority queue implementations, we can speculate here that at some large $V$ the drop in performance will be very significant. We can also speculate that we can observe drastic shifts in performance for very sparse graphs, where $0 < p \leq 0.2$.

![Change in performance of Prim's algorithm which uses proper "binomial heap" implementation of priority queue](image)

**Figure 27** – Change in performance of “binomial heap” implementation of Prim’s algorithm for different data sets.

**Fibonacci Heap Implementation (pfibh).**

The implementation where the priority queue is implemented as a Fibonacci heap, has shown a performance better than other implementations, with the exception of the implementation where a “lazy” version of binary heap was used. The ratio of the performance of Fibonacci heap implementation and that of the implementation where the proper binary heap is used is fairly constant, although we can still observe the slight decrease in the curve. As with the other implementations, we speculate that more visible results can be seen with graphs with larger number of vertices and lower graph densities. According to Figure 28 we see that the performance curves begin to diverge much earlier than the performance curves produced by other implementations. This is attributed to the density of the graph. Since the estimated asymptotic complexity of the implementation is $O(E)$ in amortized time, $\varepsilon$ becomes a significant factor in algorithm’s performance. Hence, for low values of $\varepsilon$, the performance of the algorithm that uses a Fibonacci heap
as a priority queue is significantly better. We also note that the graphs with a narrower range of edge weights yield slightly better performance than the graphs with the wider range of weights. We can attribute this to the fact that the running time is amortized. If the range of weights is small, the potential for spending time to restore the heap invariant of the trees within the Fibonacci heap is lower than it would be if the range of weights is wide. We can therefore conclude, that this implementation provides a very good performance for the sparse graphs with small range of edge weights. We can therefore speculate that for sparse graphs with a very large number of vertices the implementation of Prim’s algorithm that uses the Fibonacci heap implementation of the priority queue will be one of the best performers.

![Change in performance of Prim’s algorithm which uses proper "Fibonacci heap" implementation of priority queue](image)

Figure 28 – Change in performance of “Fibonacci heap” implementation of Prim’s algorithm for different data sets.

"Lazy" Binary Heap Implementation (php)

The ratio of the “lazy” implementation of binary heap and the proper implementation of the binary heap, appears fairly constant as shown in Figure 22, Figure 23, and Figure 24. Looking at the corresponding tables, we can see a slight decrease in the ratio with respect to the number of vertices, for every graph density and the range of edge weights w. Thus, we can see a slight difference in the asymptotic complexity of the “lazy” heap and the
proper heap implementations of Prim’s algorithm. Since the “lazy” heap implementation runs in $O(E)$, we can expect to see better performance for sparse graphs. The performance divergence based on the graph density can be seen in Figure 29. We can also note that the performance tends to drop slightly for the graphs with wider range of edge weights. This behavior is analogous to that of the Fibonacci heap. By delaying the process of restoring the heap invariant until the EXTRACT-MIN operation is to be performed, we are amortizing the running time of the algorithm. Thus, we obtain the results similar to those of the Fibonacci heap. However, the “lazy” heap implementation outperforms the Fibonacci heap implementation. We will attribute this to the lower overhead involved in implementing a binary heap. Like binomial heap, Fibonacci heap is a collection of trees. Each three is implemented as individually allocated nodes linked together using pointers. Hence the run-time constant is expected to be higher for the Fibonacci heap implementation than that of the “lazy” heap implementation.

![Change in performance of Prim’s algorithm which uses lazy “binary heap” implementation of priority queue](image)

**Figure 29** – Change in performance of “lazy heap” implementation of Prim’s algorithm for different data sets.

Given the obtained results, we can state that the implementation of Prim’s algorithm that uses the “lazy” heap as a priority queue demonstrated the best performance in all cases. Although its performance tends to fluctuate slightly with the graph density and the range of edge weights, this fluctuation is not significant enough for the graph sizes used in this experiment. For larger graphs, we speculate that the performance curves will display more divergence.
6. CONCLUSIONS

From the results we obtained we can conclude that the best performance is obtained from the use of Prim’s algorithm with priority queue implemented as a “lazy” heap. This version of Prim’s algorithm outperformed all other algorithms used in this experiment because the operations that maintain the heap invariant were delayed until the EXTRACT-MIN operation. Thus, the cost associated with maintaining the heap invariant was reduced from $O(E \lg V)$ to $O(V \lg V)$. This resulted in the lower amortized run time with respect to its counterpart, where the heap is implemented properly. In the case of lazy heap, the dominating term in the equation is $E$, hence the running time of the algorithm is $O(E)$. The good performance is also attributed to the simplicity of the data structure used to implement the “lazy” heap. Since it was implemented as an array, there was very little overhead associated with the implementation. For that reason the Prim’s algorithm with priority queue implemented as a “lazy” heap has outperformed its counterpart where the priority queue is implemented as a Fibonacci heap.

One word of caution, however. Although the implementation of Prim’s algorithm has outperformed all other implementations of Prim’s algorithm and all implementations of Kruskal’s algorithm, “lazy” heap implementation violates the heap data type abstraction. It is only useful as a priority queue used in Prim’s algorithm or in other areas where the DECREASE-KEY operation is performed more frequently than other operations. In some instance the “lazy” heap can even be expected to offer worse performance than its proper counterpart. For example, if the application that uses the binary heap performs many PEEK (look at the minimum value, but don’t extract it) operations, the time to perform such operation will be increased from $\Theta(1)$ to $O(\lg n)$, since the heap invariant must be ensured. We can therefore look at the priority queue being tightly integrated into the algorithm implementation where the “lazy” heap is used.

Overall, we can conclude that the implementations of algorithms perform differently on different sets of data. Kruskal’s algorithm that sorts the edges using the counting sort performs exceptionally well on sparse graphs. It showed very good performance, second to the implementation of Prim’s algorithm using the “lazy” heap, on all graphs where $p = 0.2$. In some of the real world applications such as the telephone line routing problems, where $p$ is expected to be much lower than 0.2, Kruskal’s algorithm with the counting sort will be more than adequate. For the graphs with $p > 0.2$ Prim’s algorithm with priority queue implemented as a Fibonacci heap, performed second to the implementation of Prim” algorithm that uses the “lazy” heap implementation of priority queue.
We can therefore conclude making the following recommendation of the best algorithm implementations for various types of data, see Table 17.

<table>
<thead>
<tr>
<th>Weight range</th>
<th>$0 &lt; p \leq 0.2$</th>
<th>$p \equiv 0.5$</th>
<th>$0.8 \leq p &lt; 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrow weight</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
</tr>
<tr>
<td></td>
<td>2. Kruskal with counting sort</td>
<td>2. Prim with Fibonacci heap</td>
<td>2. Prim with Fibonacci heap</td>
</tr>
<tr>
<td>Medium weight</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
</tr>
<tr>
<td></td>
<td>2. Kruskal with counting sort</td>
<td>2. Prim with Fibonacci heap</td>
<td>2. Prim with Fibonacci heap</td>
</tr>
<tr>
<td>High weight</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
<td>1. Prim with &quot;lazy&quot; heap</td>
</tr>
<tr>
<td></td>
<td>2. Kruskal with counting sort</td>
<td>2. Prim with Fibonacci heap</td>
<td>2. Prim with Fibonacci heap</td>
</tr>
</tbody>
</table>

Table 17 – Recommended algorithms for finding minimum spanning trees in different types of graphs.

Although these recommendations have been made, the results are not as conclusive as were intended. It appears that using graphs with 1700 vertices are not large enough to see a significant shift in performance for various implementations of Prim’s algorithm. Also the density of 0.2 appeared to be too high to show its impact on the performance of the implementations of Prim’s algorithm used in this experiment. However, we can easily eliminate the poor performers, Kruskal’s algorithm with heap sort and Prim’s algorithm with “brute force” implementation of the priority queue. The implementation of Prim’s algorithm using the binomial heap, displayed very poor performance, however we will not discard it until we test on graphs with the number of vertices far larger than 1700.
7. FURTHER WORK

The further work of will involve repeating the tests on graphs with much larger number of vertices and much lower density. Thus we will be able to tell better if the performance of the algorithms reflects their asymptotic bounds, since number of speculations was made, and several questions, such as where will binomial heap implementation of Prim's algorithm converge with the proper binary heap implementation, are unanswered.

The implementation of the binomial heap is to be reevaluated. Some overhead might be reduced with elimination of some use of pointers.

We believe that performance of the implementation of Prim's algorithm that uses the "lazy" implementation of binary heap can be improved even further with the use of flag to indicate whether any keys have been decreased, and whether the heap invariant was broken. The further work will involve the implementation of this algorithm and tested against the data obtained in this experiment.
BIBLIOGRAPHY


MST.mak

BinHeap.o: MST.h BinHeap.h
primh.o: MST.h PrimHeap.h
PrimHeap.o: MST.h PrimHeap.h

# Housekeeping

Archive: $(SOURCEFILES) Makefile
ls $(SOURCEFILES) Makefile | cpio -o cv | compress > Archive

clean:
/bin/rm -rf $(OBJFILES) MSt.o gengraph.o ptreepository Templates.DB .sh core

# Main targets

all: MST gengraph

MST: MSt.o $(OBJFILES)
$(CC) $(CFLAGS) -o MSt.o $(OBJFILES) $(LIBFLAGS)

gengraph: gengraph.o $(OBJFILES)
$(CC) $(CFLAGS) -o gengraph gengraph.o $(OBJFILES) $(LIBFLAGS)

# Dependencies

MSt.o: MSt.h
gengraph.o: MSt.h
heap.o: MSt.h
#heapest.o: MSt.h
mtx.o: MSt.h
primh.o: MSt.h
prinb.o: MSt.h
krusk_un.o: MSt.h
krusk_cs.o: MSt.h
krusk_bf.o: MSt.h
primb.o: MSt.h BinHeap.h
/*
#define __DEBUG__

typedef enum { FALSE = 0, TRUE = 1 } BOOL;

**** Definitions for method specification
****
#define KRUSKAL_INSRT "kis"
#define KRUSKAL_HEAPRT "knr"
#define KRUSKAL_COUNTSORT "kc"
#define PRIM_RF "prf"
#define PRIM_HEAP "php"
#define PRIM_HEAP_1 "php,a"
#define PRIM_BINHEAP "pbinh"
#define PRIM_MAXHEAP "pmaxh"

**** Generic Definitions
****
#define INFINITY 0xffffffff
#define NIL -1

**** Error codes
****
#define CANNOT_ALLOCATE_MEMORY -10
#define CANNOT_CREATE_HEAP -11
#define NOT_ENOUGH_CONNECTED_VERTICES -100

**** STRUCTURES USED IN KRUSKAL'S ALGORITHM WHICH DOESN'T USE HEAPSORT, BUT ****
**** USES THE INSERTION SORT
*******************************************************************************/

**** Graph type definition
****
typedef int* GraphRow;

*************************************************************************************

**** PRIORITY QUEUE DEFINITION FOR USE IN PRIM'S ALGORITHM
*******************************************************************************/

****
typedef struct tagPriorityQueue
{
 int* vertices; /* Array of vertices */
 int head; /* Head of the queue */
 int tail; /* Tail of the queue */
 int size; /* Total size of queue */
 int length; /* Total length of que*/
} PBBQueue;

typedef int* KeyArray; /* Array of keys */
typedef int* FIAarray; /* Array of PIs */

****
#define IS_QUEUE_EMPTY(Q) ((Q).length <= 0)

MST.h

/****
**** HEAP DEFINITIONS FOR USE IN PRIM'S ALGORITHM
*******************************************************************************/

****
#define PARENT(I) (((I) - 1) / 2)
#define LEFT(I) (((I) * 2) + 1)
#define RIGHT(I) (((I) * 2) + 2)

typedef PBHashTable HeapStruct; /* Heap structure */

****
#define VERT_LIST_IMPL_TO_BE_USED_WITH_BINOMIAL_HEAP
****
typedef struct tagVertList
{
 int u; /* u values */
 BOOLEAN bRemoved;
} VertList, *FVertList;

/****
**** RETURN CODE DEFINITIONS
****
#define HEAP_ERR_UNDERFLOW -100;

/****
**** DEFINITIONS FOR KRUSKAL'S ALGORITHM
*******************************************************************************/

typedef struct tagEdges
{
 int u;
 int v;
 int weight;
} Edge, *FedgeList;

/****
**** CPU TIMINGS
****
*******************************************************************************/

#include <sys/time.h>
#include <sys/resource.h>
extern int get_usage();
#define CPU_TIMES(get_usage(RUSAGE_SELF,&ruse),
 ruse.ru_utime.tv_usec + ruse.ru_stime.tv_usec +
 le-6 * (ruse.ru_utime.tv_usec + ruse.ru_stime.tv_usec))
#ifndef __BINHEAP_H__

/*
** FILE: BinHeap.h
**
** CONTENTS: Definition of data types and function prototypes which
** implement a Binomial Heap.
**
** Structure to define a node of the heap tree. The definition
** of this structure is also used for the declaration of the
** binomial trees, as well as the binomial queue
**
** typedef struct _tagBinNode_
** {
**     *** Binomial Tree attributes
**     ***
**     struct _tagBinNode_ * pParent; /* parent node */
**     struct _tagBinNode_ * pLeftChild; /* children node */
**     struct _tagBinNode_ * pSibling; /* sibling node */
**     int nDegree; /* degree */
**     int nKey; /* key value (must be integer) */
**     unsigned int unVert; /* Pointer to index of vertex */
** } BinNode, *PBinNode, *Position, *BinTree, **BinKeyPtrArray;

** Structure to provide the memory store of all nodes -- to avoid the
** dynamic allocation, which may be costly.
**
**
*/

typedef struct _tagBinHeapMemMgr_
{
    BinNode* aNodes; /* Array of nodes */
    PBinNode* apNodes; /* Array of pointers */
    unsigned numNodes; /* Total nodes */
    unsigned top; /* top of the stack */
} NodeStore;

/*
** Prototypes of functions to perform binomial heap operations
**
** BOOL NodeStore_destroy(NodeStore* pStore);
** PBinNode NodeStore_alloc(NodeStore* pStore);
** void NodeStore_free(PBinNode* node, NodeStore* pStore);
**
** PBinQueue BinQueue_make(int nVertices);
** void BinQueue_Destroy(PBinQueue* ppQueue);
** BinTree BinQueue_combineTrees(BinTree t1, BinTree t2);
** PBinQueue BinQueue_merge(PBinQueue h1, PBinQueue h2, int nCapacity,
**                          int* pnErrCode);
** PBinNode BinQueue_extractMin(PBinQueue h);
** PBinQueue BinQueue_insert(PBinQueue h, PBinNode n, int* pnErrCode);
** int BinQueue_decreaseKey(PBinQueue h, int unVert, int nKey);
** int BinQueue_min(PBinQueue h);

** ERROR CODE DEFINITIONS
**
*/
define BINQUEUE_SIZE_OVERFLOW  -10
#define BINQUEUE_VERTEX_DUPLICATION -11
#define BINQUEUE_VERTEX_NOT_IN_QUEUE -12
#define BINQUEUE_QUEUE_IS_EMPTY -13
#define BINQUEUE_DECREASE_TO_GREATER_VALUE -14
#define BINQUEUE_OK 0

#endif

BinHeap.h
#ifndef __FIBHEAP_H__

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

/**
 *** Structure to define a node of the heap tree. The definition
 *** of this structure is also used for the declaration of the
 *** trees which comprise the Fibonacci heap.
 */
typedef struct __tagFibNode
{
  /**
   *** Tree Attributes
   */
  struct __tagFibNode* pParent; /* parent pointer */
  struct __tagFibNode* pChild;  /* child pointer */
  struct __tagFibNode* pLeft;   /* left sibling */
  struct __tagFibNode* pRight;  /* right sibling */
  int nDegree; /* number of children the node */
  /* has in the child list */
  BOOL bMark; /* Indicates if the node has lost*/
  /* a child since the last time */;
  /* the node was made a child of*/
  /* another list */
  int nKey; /* key value (must be integer) */
  unsigned int unVert; /* Pointer to index of vertex */
} FibNode, *PFibNode, *FibTree, **FibKeyPtrArray;

/**
 *** Structure to define an entire heap. This heap contains a pointer to
 *** the list of nodes, a counter to maintain the number of trees in the
 *** heap, the current size of the heap (total number of nodes), a pointer
 *** to a root of the heap nodes, an array of pointers to insert/extract
 *** pointers to nodes, where an index of the array represents a vertex of
 *** the connected graph with which the Fibonacci heap is used.
 **/
typedef struct __tagFibCollection_
{
  PFibNode pRootList; /* List of roots */
  int nCurrentSize;
  PFibNode pMin; /* Pointer to node with minimum */
} FibCollection, *PFibCollection;

#define __FIBHEAP_H__
#endif

/**
 *** Key value
 */

BOOL* aInQueue; /* Array of flags to flag if a*/
/* node is in the queue */

PFibNode* apVertexNodes; /* Array of pointers to nodes */

int nMaxSize; /* Maximum size of the heap */

PFibNode* A; /* Auxiliary array A for*/
/* consolidation */

int Dm; /* D(n) */

} FibQueue, *PFibQueue;

/**
 *** Structure to provide the memory store of all nodes -- to avoid the
 *** dynamic allocation, which may be costly.
 **/

typedef struct __tagFibHeapMemMgr_
{
  /*
   * FibNode* aNodes; /* Array of nodes */
   * PFibNode* apNodes; /* Array of pointers */
   * unsigned numNodes; /* Total nodes */
   * unsigned top; /* top of the stack */
   */
  FibNodeStore* pStore;
} FibHeapMemManager, *PFibHeapMemManager;

/**
 *** PROTOTYPES
 **/

BOOL FibNodeStore_create(int size, FibNodeStore* pStore);
void FibNodeStore_destroy(FibNodeStore* pStore);
PFibNode FibNodeStore_alloc(FibNodeStore* pStore);
void FibNodeStore_free(FibNodeStore* pStore);

PFibQueue FibHeap_make(int nMaxSize);
void FibHeap_destroy(PFibQueue pQue);

PFibNode FibHeap_concatLists(PFibNode pL1, PFibNode pL2);
PFibNode FibHeap_insert(PFibQueue h, PFibNode pNode);
PFibQueue FibHeap_union(PFibQueue h1, PFibQueue h2);
PFibNode FibHeap_extractMin(PFibQueue h);
PFibNode FibHeap_consolidate(PFibQueue h);
PFibQueue FibHeap_link(PFibQueue h, PFibNode pY, PFibNode pX);
PFibQueue FibHeap_decreaseKey(PFibQueue h, unsigned int unVert, int nKey);
PFibQueue FibHeap_cut(PFibQueue h, PFibNode pX, PFibNode pY);
PFibQueue FibHeap_cascadingCut(PFibQueue h, PFibNode pY);

#} FibQueue, *PFibQueue;

/*
 *** FibHeap.h
 */

/******
 *** FILE: FibHeap.h
 ****
 *** CONTENTS: Definition of data types and function prototypes which
 *** implement a Fibonacci Heaps.
 ****

***************************************************************************/

*/

07/14/98
18:41:14
# gengraph.c

This function with connections

```c
int nVertices - number of vertices the graph is to have.
double dProb - probability of an edge being generated.
int nMinWeight - minimum edge weight
int nMaxWeight - maximum edge weight
```

**RETURNS:**

```
NOTHING
```

**DESCRIPTION:**

This function fills the graph with connections based on the probability and the weight range passed via the input arguments.

```c
void GenGraph(GraphRow* pGraph, int nVertices, double dProb, int nMinWeight, int nMaxWeight)
{
    int i, j; /* Indexes into the graph matrix */
    double dNumGenerated; /* Random number generated */
    int nWeight; /* Weight generated */
    int nRandVal;

    srand48(8376761); /* Seed the random number generator */
    for (i = 0; i < nVertices; i++)
        for (j = i + 1; j < nVertices; j++)
        {
            rand();
            nRandVal = rand();
            nRandVal = nRandVal % 100;
            dNumGenerated = (double)nRandVal / (double)90;
            if (dNumGenerated > dProb)
                pGraph[i][j] = 0;
            else
            {
                /* At this point, we will have to generate the edge */
                nWeight = rand();
                nWeight = nWeight % (nMaxWeight - nMinWeight);
                nWeight += nMinWeight;
                pGraph[i][j] = nWeight;
            }
            pGraph[j][i] = pGraph[i][j];
        }
}
```

**FUNCTION:**

GenGraph

**INPUT ARGUMENTS:**

```
GraphRow* sGraph - graph buffer to be filled up by
```
---

**07/07/98**

**15:14:49**

gengraph.c

```c
... 55 RETURNS: NOTHING
... 60 *** DESCRIPTION: Main function of the program. This function checks
... 61 *** the command line arguments to make sure that the ***
... 62 *** parameters specified for the graph generation are ***
... 63 *** valid. Then this function initiates the graph ***
... 64 *** generation sequence.
... 65 ***
void main(int argc, char** argv)
{
    char c;  /* Character extracted from arguments */
    char* sFileName = NULL;  /* Name of the file where graph is stored */
    int nVertices = -1;  /* Number of vertices the graph is to have*/
    double dProbability = 1.0;  /* Probability of edge being generated */
    int nMinWeight = -1;  /* Minimum edge weight */
    int nMaxWeight = -1;  /* Maximum edge weight */
    GraphRow* sGraph = NULL;  /* Pointer to the array that stores graph */
    int rc;  /* Return code from the function calls */

    /***
    **** Extract the command line arguments with options:
    **** n = number of vertices
    **** p = probability of the edge being present
    **** w = minimum weight
    **** W = maximum weight
    ****
    while ( (c = getopt(argc, argv, "n:p:w:W:f:")) != -1 )
    switch (c)
    {
        case 'n': if (nVertices > -1)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else nVertices = atoi(optarg);
        break;
        case 'p': if (dProbability > -1)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else dProbability = atof(optarg);
        break;
        case 'w': if (nMinWeight > -1)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else nMinWeight = atoi(optarg);
        break;
        case 'W': if (nMaxWeight > -1)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else nMaxWeight = atoi(optarg);
        break;
        case 'f': if (sFileName != NULL)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else
            nMaxWeight = atoi(optarg);
            break;
        case 'f': if (sFileName != NULL)
        {
            PrintUsageMessage(stderr, argv[0],
                "Illegal use of command line arguments");
            if (sFileName != NULL) free(sFileName);
            exit(1);
        }
        else
            sFileName = (char *)malloc(strlen(optarg));
            strcpy(sFileName, optarg);
            break;
            
            /* Make sure that all required command line argumnets have been entered */
            if ( (nVertices < 0) || (dProbability < 0.0) || (nMinWeight < 0) ||
                (nMaxWeight < 0) )
            {
                PrintUsageMessage(stderr, argv[0],
                    "Illegal use of command line arguments");
                if (sFileName != NULL) free(sFileName);
                exit(1);
            }
            
            /***
            **** We are now ready to create a graph.
            ****
            if ( ((rc = CreateGraph(nVertices, &sGraph)) == 0)
            {
                fprintf(stderr, "$s: Cannot allocate memory needed to create graph.
                    
                if (sFileName != NULL) free(sFileName);
                exit(0);
            }
            else if (rc == -1)
            {
                fprintf(stderr, "$s: Illegal number of vertices specified. $s.
                    
                if (sFileName != NULL) free(sFileName);
                exit(1);
            }
            else
                GenGraph(sGraph, nVertices, dProbability, nMinWeight, nMaxWeight);
            
            /***
            **** We can now fill the graph with edges
            ****
            GenGraph(sGraph, nVertices, dProbability, nMinWeight, nMaxWeight);
            
            /***
            **** Save the graph to the file
            ****
            if ( ((rc = SaveGraph(sGraph, nVertices, sFileName)) == 0)
            {
```
07/07/98
15:14:49

```c
    fprintf(stderr, "%s: Cannot open file %s to save the graph.\n", 
            argv[0], sFileName);
    }

    /***
    **** Cleanup before exiting the suste
    ****/
    if (sFileName != NULL) free(sFileName);
    DestroyGraph(&aGraph, nVertices);
```
```c
// Function: CreateGraph
void CreateGraph(int nVertices, GraphRow** pGraph)
{*
    int i, j;
    /* Indexes into the graph matrix */

    **** Make sure the number of vertices specified is legal
    ****
    if (nVertices < 1)
        return -1;

    **** Try to allocate memory
    ****
    *pGraph = (GraphRow*)malloc(sizeof(int) * nVertices);
    if (*pGraph == NULL)
        return 0;
    for (i = 0; i < nVertices; i++)
    {
        (*pGraph)[i] = (GraphRow*)malloc(nVertices * sizeof(int));
        if ((*pGraph)[i] == NULL)
            return -1;
        for (j = 0; j < nVertices; j++)
        {*
            (*pGraph)[i][j] = 0;
            return 1;
        }
    }

    **** Initialize the graph to all 0's == no edges
    ****
    for (i = 0; i < nVertices; i++)
        for (j = 0; j < nVertices; j++)
        {*
            (*pGraph)[i][j] = 0;
            return 1;
        }
}**
```
else
    if ((pOutFile = fopen(sFileName, "w")) == NULL)
        return 0;

    /****
    /** Write the graph to the file
    ****/
    fprintf(pOutFile, "%d\n", nVertices);
    for (i = 0; i < nVertices; i++)
    {
        for (j = 0; j < nVertices; j++)
            fprintf(pOutFile, "%d", pGraph[i][j]);
        fprintf(pOutFile, "\n");
    }
    if (sFileName != NULL)
        fclose(pOutFile);
    return 1;
}

/** ...

*** FUNCTION: ReadGraph
***
*** WRITTEN BY: Alec Berenbaum
***
*** INPUT ARGUMENTS:
*** char* sFileName - name of the file from
*** which the graph data is
*** read. IF THIS FALSE IS
*** NULL, stdin is used.
***
*** OUTPUT ARGUMENTS:
*** int* nVertices - number of vertices in the
*** graph
***
*** int** pGraph - pointer to the array where
*** the graph data is to be held.
***
*** RETURNS:
*** int - 1 = Read successfully
*** 0 = couldn't open file
*** -1 = Error on read and/or not enough
*** data
*** -2 = Couldn't allocate memory
***
*** DESCRIPTION:
*** This function reads a graph from a specified
*** file and writes it into the supplied matrix.
*/

int ReadGraph(char* sFileName, int* nVertices, int** pGraph, int* pnBig)
{
    int i = 0, j;
    FILE* pFile;
    int stat;
    int rc;

    /****
    /** See if the stdin is to be used for input
    ****/
    if (sFileName == NULL)
        pFile = stdin;
    else
        if ((pFile = fopen(sFileName, "r")) == NULL)
            return 0;

        /****
        /** Now that we got this far, start reading.
        ****/
        if ((stat = fscanf(pFile, "%d", nVertices)) == EOF)
            {
                if (sFileName != NULL)
                    fclose(pFile);
                return -1;
            }

        /****
        /** Allocate space for graph
        ****/
        if ( (rc = CreateGraph("nVertices, pGraph)) == 0)
            return -2;
        else if (rc == -1)
            return -1;

        *pnBig = 0;
        while (i < *nVertices) && (stat != EOF))
        {
            j = 0;
            while ((j < *nVertices) &&
                   ((stat = fscanf(pFile, "%d", &(*pGraph[i][j])) == EOF))
                   {
                if ((*pGraph)[] > *pnBig)
                    *pnBig = (*pGraph)[];
                j++;
            }
            if (stat != EOF) i++;
        }

        /****
        /** Now that we are out of the loop, let's see what threw us out
        ****/
        if ((i < *nVertices) || (j < *nVertices))
            {
                /****
                /** Didn't read all of it
                ****/
                if (sFileName != NULL)
                    fclose(pFile);
                DestroyGraph(pGraph, *nVertices);
                printf("didn't read all of it: i = %d j = %d, nVertices = %d\n", i, j, *nVertices);
                return -1;
            }

        /****
        /** SUCCESS!!!!! = Return 1
        ****/
        if (sFileName != NULL)
            fclose(pFile);
        return 1;
}

//***********************************************************************/
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MST.h"

extern char* optarg;
extern int optind;

void PrintUsageMessage(FILE* outFile, char* progName)
{
    fprintf(outFile, "USAGE: %s -g <graph file> -m <method> [-o <output file>]", progName);
}

void main(int argc, char** argv)
{
    char* sGraphFileName = NULL;
    char* sMethod = NULL;
    char* sOutputFileName = NULL;
    int bGotO = 0, bGotM = 0, bGotI = 0;
    char c;
    int nVertices, nEdges, row, col, nTime;
    double dfTime, dfAccTime = 0;
    int nTotalCost;
    int nTimesToIterate = 1;

    int i, j;

    /**** Extract the command line arguments ****/

    while ((c = getopt(argc, argv, "g:m:o:i:")) != -1)
    switch (c)
    {
        case 'i':
            if (bGotI)
            {
                PrintUsageMessage(stderr, argv[0]);
                if (sGraphFileName != NULL) free(sGraphFileName);
                if (sMethod != NULL) free(sMethod);
                if (sOutputFileName != NULL) free(sOutputFileName);
                exit(1);
            }
            break;
        case 'g':
            if (bGotG)
            {
                PrintUsageMessage(stderr, argv[0]);
                if (sGraphFileName != NULL) free(sGraphFileName);
                if (sMethod != NULL) free(sMethod);
                if (sOutputFileName != NULL) free(sOutputFileName);
                exit(1);
            }
            break;
        case 'm':
            if (bGotM)
            {
                PrintUsageMessage(stderr, argv[0]);
                if (sGraphFileName != NULL) free(sGraphFileName);
                if (sMethod != NULL) free(sMethod);
                if (sOutputFileName != NULL) free(sOutputFileName);
                exit(1);
            }
            break;
        case 'o':
            if (bGotO)
            {
                PrintUsageMessage(stderr, argv[0]);
                if (sGraphFileName != NULL) free(sGraphFileName);
                if (sMethod != NULL) free(sMethod);
                if (sOutputFileName != NULL) free(sOutputFileName);
                exit(0);
            }
            break;
    }

    /**** Make sure that all required command line arguments have been entered *****/

    if (! bGotM)
    {
        PrintUsageMessage(stderr, argv[0]);
        if (sGraphFileName != NULL) free(sGraphFileName);
        if (sMethod != NULL) free(sMethod);
        if (sOutputFileName != NULL) free(sOutputFileName);
        exit(1);
    }

    /**** Validate the method entered *****/

    if ((strcmp(sMethod, KRUSKAL_INSTSRT) != 0) &&
        (strcmp(sMethod, KRUSKAL_HEAPSR) != 0) &&
        (strcmp(sMethod, KRUSKAL_COUNTSORT) != 0) &&
        (strcmp(sMethod, PRIM_DEF) != 0))
    {
```
```c

void CreateMST(void)
{
    int i, j, nVertices, nEdges, n1, n2, n3;
    double distances[1000][1000];
    int adjacency[1000][1000];

    // Load graph data
    loadGraphData(&nVertices, &nEdges, distances, adjacency);

    // Sort edges by weight
    sortEdges(distances, nEdges);

    // Find MST using Prim's algorithm
    findMST(distances, adjacency, nVertices, &n1, &n2, &n3);

    // Print MST
    printMST(n1, n2, n3);
}
```

MST.c

```c

else if (strcmp(sMethod, PRIM_HEAP) == 0)
    nTotCost = php_mst(sGraph, nVertices, aOutGraph, &dftime);
else if (strcmp(sMethod, PRIM_HEAP_1) == 0)
    nTotCost = php_mst_1(sGraph, nVertices, aOutGraph, &dftime);
else if (strcmp(sMethod, PRIM_HEAP_2) == 0)
    nTotCost = php_mst_2(sGraph, nVertices, aOutGraph, &dftime);
else if (strcmp(sMethod, PRIM_BF) == 0)
    nTotCost = pb_mst(sGraph, nVertices, aOutGraph, &dftime);
else if (strcmp(sMethod, PRIM_BF_1) == 0)
    nTotCost = pb_mst_1(sGraph, nVertices, aOutGraph, &dftime);
else if (strcmp(sMethod, PRIM_BF_2) == 0)
    nTotCost = pb_mst_2(sGraph, nVertices, aOutGraph, &dftime);
```
# krusk_bf.c

```c
int nAlertVertex - vertex that signals a cycle
int nVertices - number of vertices
GraphRow* pGraph - graph

OUTPUT ARGUMENTS: NONE

RETURNS: int - 1 if there is a cycle / 0 if there is no cycle

DESCRIPTION: This function performs a DFS on a specified vertex to determine if the edge, of which it is part, forms a cycle if added to the graph.
```

```c
int kis_findCycle(int nStartVert, int nPrevVert, int nAlertVertex, int nVertices, GraphRow* pGraph)
{
    int i; /* Index into the graph */
    int bFound = 0; /* Flags if the cycle has been found */

    /* See if the "alert" vertex has been found */
    if (nStartVert == nAlertVertex)
        return i;

    /* See if any out-going vertices lead to the alert vertex */
    i = 0;
    while((i < nVertices) && (!bFound))
    {
        if (((pGraph[nStartVert][i] > 0) && (i != nPrevVert))
            bFound = kis_findCycle(i, nStartVert, nAlertVertex, nVertices, pGraph);
        i++;
    }
    return bFound;
}
```

```c
/**
   **** Begin the sort
   ****/
for (j = 1; j < nEdgeCount; j++)
{
    key.v = E[j].v;
    key.u = E[j].u;
    key.weight = E[j].weight;
    i = j - 1;
    while((i >= 0) && (E[i].weight > key.weight))
    {
        E[i + 1].v = E[i].v;
        E[i + 1].u = E[i].u;
        E[i + 1].weight = E[i].weight;
        i--;
    }
    E[i + 1].v = key.v;
    E[i + 1].u = key.u;
    E[i + 1].weight = key.weight;
}
```
kruskal_bf.c

```c
int kis_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
            double* pdTime)
{
    int nCost = 0; /* Cost of the tree */
    int u, v, k; /* Vertices examined at one time */
    EdgeList E; /* Edges of the graph */
    int nEdgeCount = 0; /* Number of edges in the graph */
    unsigned short unStartTime; /* Start time milliseconds part */
    unsigned short unEndTime; /* End time milliseconds part */
    time_t tStartStartime; /* Start time seconds part */
    time_t tEndtime; /* End time seconds part */
    struct timeb tp; /* Time structure */

    /*************************************************************************/
    /** INITIALIZATION **/  
    /*************************************************************************/

    /*************************************************************************/
    /** Count edges in the graph and allocate space for E **/  
    /*************************************************************************/

    for (v = 0; v < nVertices; v++)
        for (u = v + 1; u < nVertices; u++)
            if (pGraph[v][u] != 0) nEdgeCount++;
    E = (EdgeList)malloc(nEdgeCount * sizeof(Edge));

    /*************************************************************************/
    /** Collect all edges into the list **/  
    /*************************************************************************/

    k = 0;
    for (v = 0; v < nVertices; v++)
        for (u = v + 1; u < nVertices; u++)
            if (pGraph[v][u] != 0)
                {  
                    E[k].u = u;  
                    E[k].v = v;  
                    E[k].weight = pGraph[v][u];
                    k++;
                }

    /*************************************************************************/
    /** Assuming that the output graph is already empty, we will proceed **/  
    /** with the algorithm **/  
    /*************************************************************************/

    /*************************************************************************/
    /** Start the timer **/  
    /*************************************************************************/

    ftime(&tp);
    tStartStartime = tp.time; /* Get seconds */
    unStartStartime = tp.milli.; /* Get milliseconds */

    /*************************************************************************/
    /** Sort the edges of E by non-decreasing weight v **/  
    /*************************************************************************/

    kis_insSort(E, nEdgeCount);

    /*************************************************************************/
    /** Iterate over the edge list adding edges to the graph such that there **/  
    /** are no cycles - BRUTE force **/  
    /*************************************************************************/

    k = 0;
    while (k < nEdgeCount)
    {  
        if (! kis_findCycle(E[k].v, E[k].v, E[k].u, nVertices, pOutGraph))
            {  
                pOutGraph[E[k].u][E[k].v] = E[k].weight;
                pOutGraph[E[k].v][E[k].u] = E[k].weight;
                nCost += E[k].weight;
            }
        k++;
    }

    /*************************************************************************/

    /*************************************************************************
```c
// PATH COMPRESSION FUNCTIONS

*** FUNCTION: make_set
***.
*** INPUT ARGUMENTS: int x - vertex for which the set is made
*** int* p - array of pointing values
*** int* rank - array of ranks
*** OUTPUT ARGUMENTS: NONE
*** RETURNS: NOTHING
*** DESCRIPTION: Makes a set of pointers for the vertex of the graph.

void make_set(int x, int* p, int* rank)
{
    p[x] = x;
    rank[x] = 0;
}

*** FUNCTION: find_set
***.
*** INPUT ARGUMENTS: int x - vertex for which the set is made
*** int* p - array of pointing values
*** OUTPUT ARGUMENTS: int - pointing value at the X
*** RETURNS: int
*** DESCRIPTION: Finds a value pointing at the X

int find_set(int x, int* p)
{
    if (x != p[x])
        p[x] = find_set(p[x], p);
    return p[x];
}

*** FUNCTION: link
***.
*** INPUT ARGUMENTS: int x - vertex being linked
*** int y - vertex being linked
*** int* p - array of pointing values
*** int* rank - array of ranks for each vertex
*** OUTPUT ARGUMENTS: NONE
*** RETURNS: NOTHING
*** DESCRIPTION: Links two sets together into one.

void link(int x, int y, int* p, int* rank)
{
    if (rank[x] > rank[y])
        p[y] = x;
    else
    {
        p[x] = y;
        if (rank[x] == rank[y])
            rank[y]++;
    }
}
```

```c
*** FUNCTION: kr_union
***.
*** INPUT ARGUMENTS: int x - vertex being linked
*** int y - vertex being linked
*** int* p - array of pointing values
*** int* rank - array of ranks for each vertex
*** OUTPUT ARGUMENTS: NONE
*** RETURNS: NOTHING
*** DESCRIPTION: Joins two vertices to form an edge.

void kr_union(int x, int y, int* p, int* rank)
{
    link(find_set(x, p), find_set(y, p), p, rank);
}
```

```c
*** HEAPSORT IMPLEMENTATION
***.
#define KRLEFT(I) (((I) * 2) + 1)
#define KRRIGHT(I) (((I) * 2) + 2)
```
void edge_swap(Edge* e1, Edge* e2)
{
  Edge tmp;
  tmp.v = e1->v;
  tmp.u = e1->u;
  tmp.weight = e1->weight;
  e1->v = e2->v;
  e1->u = e2->u;
  e1->weight = e2->weight;
  e2->v = tmp.v;
  e2->u = tmp.u;
  e2->weight = tmp.weight;
}

 gboolean Kr_Heapsort(EdgeList E, int nEdgeCount)
{
  int i;
  int size;
  kr_build_heap(E, nEdgeCount);
  size = nEdgeCount - 1;
  for (i = size; i > 0; i--)
    edge_swap(&E[0], &E[i]);
  kr_heapify(E, 0, --size);
  return TRUE;
}

 gboolean Kr_Heapify(EdgeList E, int i, int nNumEdges)
{
  int l; /* index of the left value of node */
  int r; /* index of the right value of node*/
  int largest; /* index of the largest weight edge*/
  int nVertices = E->nVertices;
  int nNumEdges = E->nNumEdges;
  int p = i;
  int k = (i < nNumEdges) && (E[i].weight > E[largest].weight) ? i : largest;

  if ((p < nNumEdges) && (E[p].weight > E[k].weight))
    largest = p;
  else
    largest = k;

  if ((largest != i) && edge_swap(&E[largest], &E[i]));
  kr_heapify(E, largest, nNumEdges);

 gboolean kpc_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
                  double* pdfTime)
{
  int* p; /* Array of vertices for compression */
  int* rank; /* Ranks of these vertices */
  int nCost = 0; /* Cost of the tree */
  int u, v, k; /* Vertices examined at one time */
  int flag = TRUE; /* Edge list is not empty */
  int nEdgeCount = 0; /* Number of edges in the graph */

  for (int i = 0; i < nVertices; i++)
    for (int j = i; j < nVertices; j++)
      if (pGraph->adj[i][j] > 0)
  
  while (flag)
  
  return TRUE;
}
```c
/**
 **** Allocate space for p and rank
 *****/
p = (int*)malloc(nVertices * sizeof(int));
r = (int*)malloc(nVertices * sizeof(int));

/**
 **** Count edges in the graph and allocate space for E
 *****/
for (v = 0; v < nVertices; v++)
  for (u = v + 1; u < nVertices; u++)
    if (pGraph[v][u] != 0) nEdgeCount++;
E = (EdgeList*)malloc(nEdgeCount * sizeof(Edge));

/**
 **** Collect all edges into the list
 *****/
k = 0;
for (v = 0; v < nVertices; v++)
  for (u = v + 1; u < nVertices; u++)
    if (pGraph[v][u] != 0)
      { 
        E[k].u = u;
        E[k].v = v;
        E[k].weight = pGraph[v][u];
        k++;
      }

/*** Assuming that the output graph is already empty, we will proceed ***/
**** with the algorithm
******************************************************************************

/**
 **** Take start time reading
 *****/
ftime(&tp);
tStart = tp.time;
unStart = tp.millisec;
/*
dfStartCPUtime = CPU_TIME;
*/
for (v = 0; v < nVertices; v++)
  make_set(v, p, rank); /* Make set for every vertex */

/*** Sort the edges by non-decreasing weight
*****/
kr_heapsort(E, nEdgeCount);
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include "krusk.h"

/**
 * COUNTING SORT IMPLEMENTATION
 */

void edge_copy(Edge* e1, Edge* e2)
{
    e1->u = e2->u;
    e1->v = e2->v;
    e1->weight = e2->weight;
}

void countSort(Edge A[], Edge B[], int nLength, int C[], int k)
{
    int i, j;
    for (i = 0; i < k; i++)
        C[i] = 0;
    for (j = 0; j < nLength; j++)
        C[A[j].weight - 1]++;

    /**
     * C now contains the number of elements equal to i
     */
    for (i = 1; i < k; i++)
        C[i] += C[i - 1];

    /**
     * C now contains the number of elements less than or equal to i
     */
    for (j = nLength - 1; j >= 0; j--)
          C[A[j].weight - 1]--;
        }

}

/**
 * kcs_mst
 */

int kcs_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph, int nLargest,
            double* pdfTime)
{
    int* p;
    int* rank;
    int nCost = 0;
    int u, v, k;
    int nEdgeCount = 0;

    EdgeList E;
    EdgeList *Etemp;

    unsigned short unStarttime;
    unsigned short unEndtime;
    struct timeb tp;
    struct tms ruse;
    double dStartCPUtime;
    double dEndCPUtime;

    int* C;

    /**
     * Allocate space for p and rank
     */
    p = (int*)malloc(nVertices * sizeof(int));
    rank = (int*)malloc(nVertices * sizeof(int));

    /**
     * Count edges in the graph and allocate space for E
     */
    for (v = 0; v < nVertices; v++)
        for (u = v + 1; u < nVertices; u++)
            if (pGraph[v][u] != 0)
                E[nEdgeCount++] = 0;
        Etemp = (EdgeList*)malloc(nEdgeCount * sizeof(Edge));
        E = (EdgeList*)malloc(nEdgeCount * sizeof(Edge));

    C = (int*)malloc(nLargest * sizeof(int));

    /**
     * Collect all edges into the list
     */
    k = 0;
    for (v = 0; v < nVertices; v++)
        for (u = v + 1; u < nVertices; u++)
            if (pGraph[v][u] != 0)
                { Etemp[k].u = u;
                  Etemp[k].v = v;
                  Etemp[k].weight = pGraph[v][u];
                  k++;
                }

    /**
     * Assuming that the output graph is already empty, we will proceed
     */

}
### krusk_cs.c

```c
**** with the algorithm
*****************************/

/****
**** Take start time reading
****/
/*
ftime(&tp);
tStartTime = tp.time;
unStartTime = tp.millitm;
*/
dfStartCPUTime = CPUTIME;

for (v = 0; v < nVertices; v++)
  make_set(v, p, rank); /* Make set for every vertex */

/****
**** Sort the edges by non-decreasing weight
*****/
countSort(E, nEdgeCount, C, nLargest);

/****
**** Iterate over the edges of E by non-decreasing weight
*****/
for (k = 0; k < nEdgeCount; k++)
{
  if (find_set(E[k].u, p) != find_set(E[k].v, p))
  {
    nCost += E[k].weight;
    pOutGraph[E[k].u][E[k].v] = E[k].weight;
    pOutGraph[E[k].v][E[k].u] = E[k].weight;
    kr_union(E[k].u, E[k].v, p, rank);
  }
}

/****
**** MST is done - stop the timer
*****/
/*
ftime(&tp);
tEndTime = tp.time;
unEndTime = tp.millitm;
*pdfTime = ((double)EndTime - (double)tStartTime) +
           ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
dfEndCPUTime = CPUTIME;
*pdfTime = dfEndCPUTime - dfStartCPUTime;

/****
**** We now completed the MST - release the allocated resources
*****/
free(p);
free(rank);
free(E);
free(C);
free(Rtp);

return nCost;

******************************************************************************/
#include "MST.h"

******************************************************************************
*** HEAP FUNCTIONS
******************************************************************************

FUNCTION: heaphify

*** INPUT ARGUMENTS: HeapStruct* pHeap - pointer to the heap structure

*** int i - index for which the heap is heaphified

*** KeyArray key - array of keys for every vertex on the heap

*** OUTPUT ARGUMENTS: NONE

*** DESCRIPTION: This function ensures the heap property to be maintained on the heap.

******************************************************************************

void heaphify(HeapStruct* pHeap, int i, KeyArray key)
{
    int l;
    /* left index */
    int r;
    /* right index */
    int smallest;
    /* vertex index with the smallest key */
    int tmp;
    /* emporary variable used in swapping values */

    l = LEFT(i);         
    r = RIGHT(i);

    /* Find the vertex with the smallest value around i so it can be made a parent
    */
    smallest = (l < pHeap->size &&
                (key[pHeap->vertices[l]] < key[pHeap->vertices[i]])) ? l : i;
    if ((r < pHeap->size) &&
        (key[pHeap->vertices[r]] < key[pHeap->vertices[smallest]]))
        smallest = r;

    /* Make the smalles a parent */
    if (smallest != i)
    {
        tmp = pHeap->vertices[i];
        pHeap->vertices[i] = pHeap->vertices[smallest];
        pHeap->vertices[smallest] = tmp;

        /* Heaphify around the smallest to propogate */
        heaphify(pHeap, smallest, key);
    }
}

******************************************************************************

FUNCTION: heap_sort

*** INPUT ARGUMENTS: HeapStruct* pHeap - pointer to the heap structure

*** KeyArray key - array of keys for every vertex on the heap

*** OUTPUT ARGUMENTS: NONE

*** DESCRIPTION: This function sorts the values of the heap using the heapsort

******************************************************************************

void heap_sort(HeapStruct* pHeap, KeyArray key)
{
    int i;
    /* index into the heap value array */
    int tmp;
    /* temporary variable used in swapping values */

    heap_build(pHeap, key);   /* build the heap fiirts */

    for (i = (pHeap->length - 1); i >= 1; i--)
    {
        tmp = pHeap->vertices[0];
        pHeap->vertices[0] = pHeap->vertices[i];
        pHeap->vertices[i] = tmp;
        heaphify(pHeap, 0, key);
    }
}
/* PRIORITY QUEUE FUNCTIONS */

** FUNCTION: ** heap_extract_min

** INPUT ARGUMENTS: **
HeapStruct* pHeap - pointer to the heap structure
KeyArray key - array of keys for every vertex on the heap

** OUTPUT ARGUMENTS: **
NONE

** RETURNS: **
int - the value with minimal key extracted or
HEAP_ERR_UNDERFLOW if the heap is empty

** DESCRIPTION: **
This function extracts the minimal value from the heap and ensures that the heap property is maintained.

int heap_extract_min(HeapStruct* pHeap, KeyArray key)
{
    int min; /* minimal value being extracted */

    if (pHeap->size < 1)
        return HEAP_ERR_UNDERFLOW;

    min = pHeap->vertices[0];
    pHeap->vertices[0] = pHeap->vertices[pHeap->size - 1];
    (pHeap->size)--;
    heapify(pHeap, 0, key);

    return min;
}

/* PRIORITY QUEUE FUNCTIONS */
#include <stdio.h>
#include <sys/types.h>
// *include <sys/time.h>*/
#include <sys/timeb.h>
#include "MST.h"

/***
** FUNCTION: pbf_alloc_data_structs
** INPUT ARGUMENTS: int nVertices - number of vertices
** OUTPUT ARGUMENTS: PBFQueue* pQ - structure for priority queue
** KeyArray* pKey - array of keys
** PIArray* pPi - array of PI values
** RETURNS: NOTHING
** DESCRIPTION: This function allocates resources for the Q, PI, and Key arrays used by Prim's algorithm. The initial values are set by the algorithm itself. So the timings consider them as well.
**
** void pbf_alloc_data_structs(int nVertices, PBFQueue* pQ, KeyArray* pKey, PIArray* pPi)
**
** Allocate space for the priority queue
**
/**
* (pQ) = (PBFQueue *)malloc(sizeof(PBFQueue));
* (pQ)->nVertices = (int *)malloc(nVertices * sizeof(int));
/**
** Allocate space for the KEY array
**
/**
* (pKey) = (int *)malloc(nVertices * sizeof(int));
/**
** Allocate space for the PI array
**
/**
* (pPi) = (int *)malloc(nVertices * sizeof(int));
** for (i = 0; i < nVertices; i++)
** { ((pQ)->nVertices)[i] = NIL;
** (pPi)[i] = NIL
** (pKey)[i] = INFINITY;
** }
**
**/******
***/

/***
** FUNCTION: pbf_is_in_que
** INPUT ARGUMENTS: int vertex - vertex sought in queue
** PBFQueue* Q - queue in which vertex is sought
** OUTPUT ARGUMENTS: NONE
** RETURNS: int - 1 if the specified vertex is in queue; 0 otherwise
** DESCRIPTION: This function determines if the vertex specified via the argument 'vertex' is in queue specified via the argument 'Q'.
**
** int pbf_is_in_que(int vertex, PBFQueue* Q)
**
** Index into the queue */

/******
*/

```c
void pbf_alloc_data_structs(int nVertices, PBFQueue* pQ, KeyArray* pKey, PIArray* pPi)
{
    int i;
    /* Index into arrays for initialization */

    /* Allocate space for the priority queue */
    (*pQ) = (PBFQueue *)malloc(sizeof(PBFQueue));
    (*pQ)->nVertices = (int *)malloc(nVertices * sizeof(int));

    /* Allocate space for the KEY array */
    (*pKey) = (int *)malloc(nVertices * sizeof(int));

    /* Allocate space for the PI array */
    (*pPi) = (int *)malloc(nVertices * sizeof(int));
    for (i = 0; i < nVertices; i++)
    {
        ((pQ)->nVertices)[i] = NIL;
        (pPi)[i] = NIL
        (pKey)[i] = INFINITY;
    }

    /* Index into the queue */
}
```

```c
int pbf_is_in_que(int vertex, PBFQueue* Q)
{
    int i = 0;
    /* Index into the queue */

    /* This function determines if the vertex specified via the argument 'vertex' is in queue specified via the argument 'Q'. */
```
# prmbf.c

**** We have now found the minimal vertex. Extract it ****

```c
if (j := NIL)
  
  min = Q->vertices[j];
  Q->vertices[j] = NIL;
  Q->size--;)
  return min;
```

****

```c

/**********************************************************************

*** MST

/**********************************************************************

****

```
08/02/98
19:13:31

primef.c

****
**** Take start time reading
****/
/*
ftime(&tp);
tStart = tp.time;
unStart = tp.mallitm;
*/
dfStartCPUtime = CPUtime;

****
**** put all vertices onto the queue - Use only connected ones. The size
**** member of the queue is used to account for the number of vertices
**** put onto the queue
****/
Q->head = 0;
Q->size = 0;
u = 0;
while (u < nVertices)
{
    /*
    **** Search for the first connected vertex to determine if it goes onto
    **** the queue
    ****/
    v = 0;
    while ((v < nVertices) && (pGraph[u][v] == 0))
        v++;
    /*
    **** See what stopped the loop
    ****/
    if (v < nVertices)
    {
        /*
        **** The vertex is connected. put it onto the queue
        ****/
        Q->vertices[Q->head] = u;
        (Q->size)+=
        (Q->head)+=
    }
    u++;
}

****
**** Make sure that the number of connected vertices is at least 2.
****/
if (Q->size < 2)
{
    pbf_free_data_structs(&Q, &key, &Pi);
    return NOT_ENOUGH_CONNECTED_VERTICES;
} else
    Q->length = Q->size;

****
**** We now have all vertices on the queue - set the key for each vertex to
**** INFINITY
****/
for (Q->head = 0; Q->head < Q->size; (Q->head)+=)
    { u = Q->vertices[Q->head];
      key[u] = INFINITY;
    }

****
**** Assume r is the first entry in the Queue - set its key to 0 and its PI
****
#include <stdio.h>
#include <sys/types.h>
/*#include <sys/time.h>*/
#include <sys/timeb.h>
#include "MST.h"

***** PRIORITY QUEUE FUNCTIONS *****


 **** FUNCTION: php_alloc_data_structs ****
 **** INPUT ARGUMENTS: int nVertices - number of vertices ****
 **** OUTPUT ARGUMENTS: PBFQueue* pQ - structure for priority queue ****
 **** RETURNS: NOTHING ****
 **** DESCRIPTION: This function allocates resources for the Q, PI, and Key arrays used by Prims algorithm. The initial values are set by the algorithm itself, so the timings consider them as well.

void php_alloc_data_structs(int nVertices, PBFQueue* pQ, KeyArray* pKey, PIArray* pPI)
{
    int i;
    /* Index into arrays for initialization */

    /* Allocate space for the priority queue */
    (*pQ) = (PBFQueue *)malloc(sizeof(PBFQueue));
    (*pQ)->vertices = (int *)malloc(nVertices * sizeof(int));

    /* Allocate space for the KEY array */
    (*pKey) = (int *)malloc(nVertices * sizeof(int));

    /* Allocate space for the PI array */
    (*pPI) = (int *)malloc(nVertices * sizeof(int));
    for (i = 0; i < nVertices; i++)
    {
        (*pQ)->vertices[i] = NIL;
        (*pPI)[i] = NIL;
        (*pKey)[i] = INFINITY;
    }
}


 **** FUNCTION: php_is_in_que ****
 **** INPUT ARGUMENTS: int vertex - vertex sought in queue ****
 **** OUTPUT ARGUMENTS: PBFQueue* Q - queue in which vertex is sought ****
 **** RETURNS: int - 1 if the specified vertex is in queue 0 otherwise ****
 **** DESCRIPTION: This function determines if the vertex specified via the argument 'vertex' is in queue specified via the argument 'Q'.

int php_is_in_que(int vertex, PBFQueue* Q)
{
    int i = 0;
    /* Index into the queue */

/**
 * Search exhaustively until the vertex is either found or not
 */
if (Q->size <= 0) 
return 0;
while ((i < Q->size) && (vertex != Q->vertices[i])) i++;
/**
 * See what stopped the loop and return accordingly
 */
if (i >= Q->size) 
return 0;
return (Q->vertices[i] == vertex);
*/

primhp.c

double dfEndCPUtil;

/*******************************/

MST

FUNCTION:

**
 ***
 *** INPUT ARGUMENTS:
 ***
 *** int nVertices - number of vertices
 ***
 *** OUTPUT ARGUMENTS:
 ***
 *** double* pdfTime - time it took to get MST
 ***
 *** RETURNS:
 ***
 *** DESCRIPTION:
 ***
 *** This function finds a MST in the graph supplied via the pGraph argument. The
 *** Prim's algorithm is used using the BRUTE ***
 *** force to extract the minimum vertex from ***
 *** the priority queue.
 ***

int php_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph, 
double* pdfTime)
{
    PBQueue* Q; /* Priority Queue */
    KeyArray key; /* Keys */
    PAArray p; /* P */
    int u, v, k; /* Vertices examined at one time */
    int nCost = 0; /* Cost of the tree */
    int tmp; /* Temporary storage for swapping */
    int curNode; /* Node currently looked at */
    int i;

    unsigned short unStartTime; /* Start time milliseconds part */
    unsigned short unEndTime; /* End time milliseconds part */
    time_t tStart; /* Start time seconds part */
    time_t tEnd; /* End time seconds part */
    struct timeb tp; /* Time structure */
    struct rusage ruse;
    double dfStartCPUtil;

    double dfEndCPUtil;
    /* allocate resources for the queue, pi, and key arrays */
    php_alloc_data_structs(nVertices, &Q, &key, &p);
    /*
    * take start time reading
    */
    ftime(&tp);
    tStartTime = tp.time;
    unStartTime = tp.millisim;
    /*
    * dfStartCPUtil = CPUtil;
    */
    /**
    * put all vertices onto the queue - use only connected ones. the size
    * member of the queue is used to account for the number of vertices
    * put onto the queue
    */
    Q->head = 0;
    Q->size = 0;
    u = 0;
    while (u < nVertices)
    {
        /**
        * search for the first connected vertex to determine if it goes onto
        * the queue
        */
        v = 0;
        while ((v < nVertices) && (pGraph[u][v] == 0)) 
        v++;
        /**
        * see what stopped the loop
        */
        if (v < nVertices)
        {
            /**
            * the vertex is connected, put it onto the queue
            */
            Q->vertices[Q->size] = u;
            (Q->size)++;
            (Q->head)++;
        }
        u++;
    }
    /**
    * make sure that the number of connected vertices is at least 2.
    * if (Q->size < 2)
    {
        php_free_data_structs(&Q, &key, &p);
        return NOT_ENOUGH_CONNECTED_VERTICES;
    }
    else
    Q->length = Q->size;
    /**
    * we now have all vertices on the queue - set the key for each vertex to
    * infinity
    */
for (Q->head = 0; Q->head < Q->size; (Q->head)++)
{
    u = Q->vertices[Q->head];
    key[u] = INFINITY;
}

/**
 ** Assume r is the first entry in the Queue - set its key to 0 and its PI
to NIL:
**/

u = Q->vertices[0];
key[u] = 0;
Pi[u] = NIL;

heap_build(Q, key);

/>**** Begin iteration over the queue looking for edges
**** /
while (Q->size > 0)
{
    /*
    ** If there is a retained edge, add it to the graph
    **/
    u = heap_extract_min(Q, key);
    /*
    ** Iterate over all vertices adjacent to u
    **/
    for (k = 0; k < Q->size; k++)
    {
        v = Q->vertices[k];
        if ((u == v) && (pGraph[u][v] != 0) && (pGraph[u][v] < key[v]))
        {
            Pi[v] = u;
            key[v] = pGraph[u][v];
            /*
            ** Fix the heap
            **/
            curNode = k;
            while ( (curNode != 0) && (key[Q->vertices[PARENT[curNode]]]) <
                key[Q->vertices[curNode]])
            {
                tmp = Q->vertices[curNode];
                Q->vertices[curNode] = Q->vertices[PARENT[curNode]];"
                Q->vertices[PARENT[curNode]] = tmp;
                curNode = PARENT(curNode);
            }
        }
    }
    /*
    ** for (i = (Q->size / 2) - 1; i >= 0; i--)
    ** heapify(Q, i, key);
    */
    /*
    ** heapify(Q, 0, key), */
    }
unsigned short unEndTime; /* End time milliseconds part */
time_t tStartTime; /* Start time seconds part */
time_t tEndTime; /* End time seconds part */
struct timeb tp; /* Time structure */
double dfStartCPUTime;
double dfEndCPUTime;
struct rusage ruse;

/**
 *** Allocate resources for the queue, pi, and key arrays
 /***/
php_alloc_data_structs(nVertices, &Q, &key, &pi);

/**
 *** Take start time reading
 /***/
/*
tStart_time = tp_time;
unStart_time = tp.milliitm;
*/
/**
 *** Put all vertices onto the queue - Use only connected ones. The size
 *** member of the queue is used to account for the number of vertices
 *** put onto the queue
 /***/
Q->head = 0;
Q->size = 0;
u = 0;
while (u < nVertices)
{
    /*
    *** Search for the first connected vertex to determine if it goes onto
    *** the queue
    /***/
    v = 0;
    while (((v < nVertices) && (pGraph[u][v] == 0))
        v++;
    /*
    *** See what stopped the loop
    /***/
    if (v < nVertices)
    {
        /*
        *** The vertex is connected, put it onto the queue
        /***/
        Q->vertices[Q->head] = u;
        (Q->size)++;
        (Q->head)++;
    }
    u++;
}

/**
 *** Make sure that the number of connected vertices is at least 2.
 ***/
if (Q->size < 2)
{
    php_free_data_structs(&Q, &key, &pi);
    return NOT_ENOUGH_CONNECTED_VERTICES;
} else
    Q->length = Q->size;

primhp.c

/**
 *** We Now have all vertices on the queue - set the key for each vertex to
 *** INFINITY
 ***/
for (Q->head = 0; Q->head < Q->size; (Q->head)++)
{
    u = Q->vertices[Q->head];
    key[u] = INFINITY;
}

/**
 *** Assume r is the first entry in the Queue - set its key to 0 and its PI
 *** to NIL
 ***/
u = Q->vertices[0];
key[u] = 0;
pi[u] = NIL;
heap_build(Q, key);

/**
 *** Begin iteration over the queue looking for edges
 ***/
while (Q->size > 0)
{
    /*
    *** If there is a retained edge, add it to the graph
    ***/
    u = heap_extract_min(Q, key);
    /*
    *** Iterate over all vertices adjacent to u
    ***/
    for (k = 0; k < Q->size; k++)
    {
        v = Q->vertices[k];
        if (((u != v) && (pGraph[u][v] != 0)) && (pGraph[u][v] < key[v]))
        {
            Pi[v] = u;
            key[v] = pGraph[u][v];
        }
    }
    for (i = (Q->size / 2) - 1; i >= 0; i--)
       heapify(Q, i, key);
    /*
    heapify(Q, 0, key); /*
}

/**
 *** We should now have the MST!!!!
 *** Take the end time reading
 ***/
/*
tStart_time = tp_time;
unEnd_time = tp.milliitm;
*pdfTime = (double)tEnd_time - (double)tStart_time +
((double)unEnd_time - (double)unStart_time) / 1000.0;
*/
dfEndCPUTime = CfUTIME;
pdfTime = dfEndCPUTime - dfStartCPUTime;

/**
 *** Reconstruct the output tree
 ***/
for (u = 0; u < nVertices; u++)
if (Pi[u] != NIL)
    nCost += pOutGraph[Pi[u]][u] = pOutGraph[u][Pi[u]] = pGraph[u][Pi[u]];

/*
 **** Clean up
 *****/
php_free_data_structs(&Q, &key, &Pl);
return nCost;
}*/

primhp.c
```c
/** DATA STRUCTURE MANAGEMENT FUNCTIONS */

FUNCTION: pbh_alloc_data_structs
INPUT ARGUMENTS: int nVertices - number of vertices
OUTPUT ARGUMENTS: short** aInQueFlags - flags if the vertex is in the queue
BinKeyPtrArray* pKey - array of pointers which contain nodes to heaps
PIArray* pPi - array of PI values
NodeStore* pStore - store for the nodes that are to be used for the binomial heap implementation

RETURNS: NOTHING

DESCRIPTION: This function allocates resources for the Binomial Heap, an array of flags which indicates whether the vertex is in the queue, and array of parent nodes, and the actual node store.

void pbh_alloc_data_structs(int nVertices, short** aInQueFlags, BinKeyPtrArray* pKey, PIArray* pPi, NodeStore* pStore)
{
    int i; /* Index into arrays for initialization */
    /* Allocate space for the array of flags */
    *aInQueFlags = (short *)malloc(nVertices * sizeof(short));
    /* Allocate space for the KEY array */
    *pKey = (BinNode*)malloc(nVertices * sizeof(BinNode));
    /* Create the nodes for the binomial heap usage */
    *pStore = (NodeStore*)malloc(sizeof(NodeStore));
    NodeStore_create(nVertices, *pStore);
    /* 
*/
}

FUNCTION: pbh_free_data_structs
INPUT ARGUMENTS: NONE
OUTPUT ARGUMENTS: short** aInQueFlags
BinKeyPtrArray* pKey - array of pointers which contain nodes to heaps
PIArray* pPi - array of PI values
NodeStore* pStore - store for the nodes that are to be used for the binomial heap implementation

RETURNS: NOTHING

DESCRIPTION: This function releases resources used by the datastructures in the Prim's algorithm.

void pbh_free_data_structs(short** aInQueFlags, BinKeyPtrArray* pKey, PIArray* pPi, NodeStore* pStore)
{
    /* Release all array resources */
    free(*aInQueFlags);
    *aInQueFlags = NULL;
    /* 
*/
    /* Destroy the node store */
    NodeStore_destroy(*pStore);
    free(*pStore);
    /* 
*/
    /* Release PI and Key resources */
    free(*pKey);
    free(*pPi);
    *pKey = NULL;
    *pPi = NULL;
}
```
primbh.c

NodeStore nodeStore; /* Store of pointers */
PBinQueue pQueue; /* Pointer to binomial queue */
unsigned short unStartTime; /* Start time milliseconds part */
unsigned short unEndTime; /* End time milliseconds part */
time_t tStart, tEnd; /* Start and end time seconds part */
struct timeb tp; /* Time structure */
int nErrCode; /* User for debugging */

struct usage ruse;
double dfStartCPUtime;
double dfEndCPUtime;

/*** Allocates data structures */
NodeStore_create(nVertices, &nodeStore);
pQueue = BinQueue_make(nVertices);
PI = (PIArray)malloc(nVertices * sizeof(int));

/*** Take start time reading */

*time(&tp);
tStart = tp.time;
unStart = tp.milliTime;

/*** dfStartCPUtime = CPUtime; */

/*** Put all vertices onto the queue - Use only connected ones. The size of the queue is used to account for the number of vertices put onto the queue */

for (u = 0; u < nVertices; u++)
{
    /* Search for the first connected vertex to determine if it goes onto the queue */
    v = 0;
    while ((v < nVertices) & (pGraph[u][v] == 0))
    {
        v++;
    }
    /* See what stopped the loop */
    if (v < nVertices)
    {
        /* The vertex is connected, queue it up */
        pNode = NodeStore_alloc(&nodeStore);
        pNode->unVert = u;
        if (nVertCount == 0)
        {
            pNode->nKey = 0;
            pRootVertex = pNode;
        }
        pQueue = BinQueue_insert(pQueue, pNode, &nErrCode);
        nVertCount++;
    }

/***/

**** FUNCTION: pnh_mst

**** INPUT ARGUMENTS:
GraphRow* pGraph - input graph
***
int nVertices - number of vertices
***

**** OUTPUT ARGUMENTS:
GraphRow* pOutGraph - output graph
***

**** RETURNS:
int - Total cost of the MST
***

**** DESCRIPTION:
This function finds a MST in the graph supplied via the pGraph argument. The Prim's algorithm is used using the BRUTE force to extract the minimum vertex from the priority queue.

****

int pnh_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph, double* pdfTime)
{
    PIArray PI; /* PI's */

    int u, v, k; /* Vertices examined at one time */
    int nCost = 0; /* Cost of the tree */
    int tmp; /* Temporary storage for swapping */
    int curNode; /* Node currently looked at */
    int nVertCount = 0; /* Counter of connected vertices */
    PBinNode pNode; /* Node that was extracted */
    PBinNode pRootVertex; /* Root vertex */
/**
 * Make sure that the number of connected vertices is at least 2.
 */
if (nVertCount < 2)
{
    NodeStore_destroy(&nodeStore);
    BinQueue_destroy(&apQueue);
    free(Pi);
    return NOT_ENOUGH_CONNECTED_VERTICES;
}
/**
 * Assume r is the first entry in the Queue - set its key to 0 and its PI
to NIL
 */
u = pRootVertex->unVert;
P[0] = NIL;
/**
 * Begin iteration over the queue looking for edges
 */
while ((pNode = BinQueue_extractMin(pQueue)) != NULL)
{
    u = pNode->unVert;
    /**
     * Iterate over all vertices adjacent to u
     */
    for (v = 0; v < nVertices; v++)
    {
        if (v != u && (pQueue->lnQueue[v] && (pGraph[u][v] != 0) &&
          (pGraph[u][v] < pQueue->apVertexNodes[v]->nKey))
        {
            P[v] = u;
            BinQueue_decreaseKey(pQueue, v, pGraph[u][v]);
        }
    }
    /**
     * We should now have the MST!!!
     * Take the end time reading
     */
    //
    tStart = tp.time;
    unEndTime = tp.millisecond;
    *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
    //
    dfEndCPUTime = CPUTCUTIME;
    *pdfTime = dfEndCPUTime - dfStartCPUTime;

    /**
     * Reconstruct the output tree
     */
    for (u = 0; u < nVertices; u++)
    {
        if (P[u] != NIL)
        {
            nCost = pOutGraph[P[u]][u] = pOutGraph[u][P[u]] = pGraph[u][P[u]];
        }
    }
    /**
     * Clean up
     */
#include "BinHeap.h"

FUNCTION: NodeStore_create
***
*** INPUT ARGUMENTS: int size - size of the store to be created
***
*** OUTPUT ARGUMENTS: NodeStore* pStore - node store created
***
*** RETURNS: BOOL - TRUE if successful, FALSE if failed
***
*** DESCRIPTION: This function allocates memory necessary for the
*** nodes of the binomial heap. The allocated
*** memory is to be used for the "allocation" of
*** nodes.
***
BOOL NodeStore_create(int size, NodeStore* pStore)
{
int i; /* Array index */

   *** First, allocate space for the actual nodes.
   ***
   if ((pStore->aNodes = (PBinNode)malloc(size * sizeof(BinNode))) == NULL)
      return FALSE;

   *** Now that the nodes are allocated, we allocate space to hold pointers
   *** to each
   ***
   if ((pStore->aNodes = (PBinNode *)malloc(size * sizeof(PBinNode))) == NULL)
   {
      free(pStore->aNodes);
      return FALSE;
   }

   *** Everything is allocated. Now we set pointers and initialize values
   ***
   for (i = 0; i < size; i++)
   {
      pStore->aNodes[i] = &pStore->aNodes[i]; /* set the pointer */
      pStore->aNodes[i].nKey = INFINITY;
      pStore->aNodes[i].pParent = NULL;
      pStore->aNodes[i].pSibling = NULL;
      pStore->aNodes[i].pLeftChild = NULL;
   }

   pStore->numNodes = size;
   pStore->top = 0;

   *** Done
   return TRUE;
}

FUNCTION: NodeStore_destroy
***
*** INPUT ARGUMENTS: NodeStore* pStore - node store destroyed
***
*** OUTPUT ARGUMENTS: NONE
***
*** RETURNS: NOTHING
***
*** DESCRIPTION: This function releases memory allocated for the
*** binomial heaps.
***
void NodeStore_destroy(NodeStore* pStore)
{
   free(pStore->aNodes);
   free(pStore->aNodes);
}

FUNCTION: NodeStore_alloc
***
*** INPUT ARGUMENTS: NodeStore* pStore - node store from which the
*** node is to be allocated
***
*** OUTPUT ARGUMENTS: NONE
***
*** RETURNS: PBinNode - pointer to binomial heap node. NULL
***   if this operation fails.
***
*** DESCRIPTION: This function allocates a node from the store and
*** returns a pointer to it. If this operation
*** fails, NULL is returned
***
PBinNode NodeStore_alloc(NodeStore* pStore)
{
   *** Check if nodes are available for allocation
   ***
   if (pStore->top >= pStore->numNodes)
      return NULL;

   *** Get the next node and return a pointer to it, move the top down
   ***
   return pStore->aNodes[(pStore->top++)];
}
BinHeap.c

/*
 * FUNCTION:    NodeStore_free
 * INPUT ARGUMENTS:  PBinNode* node - pointer to the pointer that
 *                     points to node to be freed
 * OUTPUT ARGUMENTS:  NONE
 * DESCRIPTION: This function returns the allocated node back to
 *                the store.
 */

void NodeStore_free(PBinNode* node, NodeStore* pStore)
{
    pStore->top--;  
    pStore->pNodes[pStore->top] = *node;  
    *node = NULL;
}

/*
 * FUNCTION:    BinQueue_make
 * INPUT ARGUMENTS:  int nVertices
 * OUTPUT ARGUMENTS:  PBinQueue pQueue
 * DESCRIPTION: This function creates a new queue.
 */

PBinQueue BinQueue_make(int nVertices)
{
    int i;
    PBinQueue pQueue;  
    pQueue = (PBinQueue)malloc(sizeof(BinQueue));  
    pQueue->nTrees = floor(log2((double)nVertices)) + 1;  
    pQueue->nTrees = (BinTree*)malloc(pQueue->nTrees * sizeof(BinTree));  
    for (i = 0; i < pQueue->nTrees; i++)  
        pQueue->aTrees[i] = NULL;
    pQueue->aTrees = (PBinNode**)malloc(nVertices * sizeof(PBinNode));  
    pQueue->aQueue = (BOOL*)malloc(nVertices * sizeof(BOOL));  
    for (i = 0; i < nVertices; i++)  
        pQueue->aQueue[i] = FALSE;  
    pQueue->aQueue = (PBinNode**)malloc(nVertices * sizeof(PBinNode));

    pQueue->nCapacity = nVertices;  
    pQueue->nCurrentSize = 0;  
    pQueue->nMin = BINQUEUE_QUEUE_IS_EMPTY;
    return pQueue;
}

void BinQueue_destroy(PBinQueue* ppQueue)
{
    free(pQueue->aTrees);
    free(pQueue->aQueue);
    free(pQueue->aVertices);
    free(pQueue);
}

BinTree BinQueue_combineTrees(BinTree t1, BinTree t2)
{
    if (t1->nKey > t2->nKey)  
        return BinQueue_combineTrees(t2, t1);
    t2->pSibling = t1->pLeftChild;
    t2->pParent = t1;
    t1->pLeftChild = t2;
    return t1;
}

PBinQueue BinQueue_merge(PBinQueue h1, PBinQueue h2, int nCapacity,  
                          int* pnErrCode)
{
    BinTree t1,  
    BinTree t2,  
    BinTree carry = NULL;
    int i, j;
    if ((*pnErrCode = (h1->nCurrentSize + h2->nCurrentSize > nCapacity)  
         ? BINQUEUE_SIZE_OVERFLOW  
         : BINQUEUE_OK) != BINQUEUE_OK)
        return h1;
    h1->nCurrentSize += h2->nCurrentSize;
    for (i = 0; j = 1; j <<= h1->nCurrentSize; i++, j *= 2)  
        t1 = h1->aTrees[i];  
        t2 = h2->aTrees[i];
        switch (!!t1 + 2 * !!t2 + 4 * !!carry)  
        {  
            case 0:  
            /* No Trees */  
            break;
            case 1:  
            /* Only h1 */  
            break;
            case 2:  
            /* Only h2 */  
            if ((h1->nMin == -1)  
                 && (h1->aTrees[h1->nMin]->_key > t2->_key))  
                h1->nMin = i;
                break;
            case 3:  
            /* h1 and h2 */  
            carry = BinQueue_combineTrees(t1, t2);
            h1->aTrees[i] = carry;
            break;
            case 4:  
            /* Only carry */  
            if ((h1->nMin == -1)  
                 && (h1->aTrees[h1->nMin]->_key > carry->_key))  
                h1->nMin = i;
                break;
        }
        return h1;
}
BinHeap.c

for (j = nMinInd - 1; j >= 0; j--)
{
    pDeletedQueue->aTrees[j] = pDeletedTree;
    pDeletedTree = pDeletedTree->pSibling;
    pDeletedQueue->aTrees[j]->pSibling = NULL;
}

h->aTrees[nMinInd] = NULL;

h->mCurrentSize += pDeletedQueue->mCurrentSize + 1;

h = BinQueue_merge(h, pDeletedQueue, h->mCapacity, &nErrCode);

BinQueue_destroy(&pDeletedQueue);

h->aInQueue[p0ldRoot->unVert] = FALSE;
h->apVertexNodes[p0ldRoot->unVert] = NULL;

return p0ldRoot;

}

PBinQueue BinQueue_insert(PBinQueue h, PBinNode n, int* pnErrCode)
{
    PBinQueue pTmpQueue;

    /**
    *** Make sure that the node is not already in the tree
    *** /
    if (h->aInQueue[n->unVert])
    {
        *pnErrCode = BINQUEU_VERTEX_DUPLICATION;
        else
        {
            pTmpQueue = BinQueue_make(h->mCapacity);
            pTmpQueue->mCurrentSize = 1;
            pTmpQueue->aTrees[0] = n;
            pTmpQueue->apVertexNodes[n->unVert] = n;
            pTmpQueue->aInQueue[n->unVert] = TRUE;
            pTmpQueue->mMin = 0;

            h = BinQueue_merge(h, pTmpQueue, h->mCapacity, pnErrCode);
            BinQueue_destroy(&pTmpQueue);
            h->aInQueue[n->unVert] = TRUE;
        }
        return h;
    }

    int BinQueue_min(PBinQueue h)
    {
        int i;
        int nMinInd = BINQUEU_QUEUE_IS_EMPTY;
        int nMin = INFINITY;

        for (i = 0; i < h->mTrees; i++)
        {
            if ((h->aTrees[i] != NULL) && (h->aTrees[i]->nKey < nMin))
            {
                nMin = h->aTrees[i]->nKey;
                nMinInd = i;
            }
        }
        return nMin;
    }

PBInode BinQueue_extractMin(PBinQueue h)
{
    int i, j;
    int nMinTree;
    PBinQueue pDeletedQueue;
    Position pDeletedTree;
    Position pOldRoot;
    Position pCurSib;
    PBinNode pMinNode;
    int nErrCode;
    int nMinInd;

    if (h->mCurrentSize == 0)
    return NULL;

    nMinInd = BinQueue_min(h);

    pDeletedTree = h->aTrees[nMinInd];
    pOldRoot = pDeletedTree;
    pDeletedTree = pDeletedTree->pLeftChild;

    /**
    *** Make sure the parent pointers of the nodes descending from the node
    *** deleted are all NULL
    *** /
    pCurSib = pDeletedTree;
    while (pCurSib != NULL)
    {
        pCurSib->pParent = NULL;
        pCurSib = pCurSib->pSibling;
    }

    pDeletedQueue = BinQueue_make(h->mCapacity);
    pDeletedQueue->mCurrentSize = (1 << nMinInd) - 1;
```c
int BinQueue_decreaseKey(BinQueue h, int unVert, int nKey)
{
    BinNode pCurNode; /* Node being decreased */
    BinNode pParentNode; /* Parent node of node being decreased */
    BinNode pTmp; /* Temporary pointer to nodes */
    int nTmpKey;
    unsigned int unTmpVert;
    int nTmpDegree;

    *** Make sure we are working with a valid vertex
    ***
    if (((h->aInQueue[unVert]) || (h->apVertexNodes[unVert] == NULL))
        return BINQUEUE_VERTEX_NOT_IN_QUEUE;
    else if (nKey > h->apVertexNodes[unVert]->nKey)
        return BINQUEUE_DECREASE_TO_GREATER_VALUE;

    *** Begin decrease procedure
    ***
    pCurNode = h->apVertexNodes[unVert]; /* Get pointer to node decreased */
    pParentNode = pCurNode->pParent; /* Get pointer to the parent node */
    pCurNode->nKey = nKey; /* Assign new key value to that node */

    *** 'Bubble Up'
    ***
    while ((pParentNode != NULL) && (pCurNode->nKey < pParentNode->nKey))
    {
        *** Exchange the key and all satellite fields
        ***
        nTmpKey = pCurNode->nKey;
        pCurNode->nKey = pParentNode->nKey;
        pParentNode->nKey = nTmpKey;

        unTmpVert = pCurNode->unVert;
        pCurNode->unVert = pParentNode->unVert;
        pParentNode->unVert = unTmpVert;

        nTmpDegree = pCurNode->nDegree;
        pCurNode->nDegree = pParentNode->nDegree;
        pParentNode->nDegree = nTmpDegree;

        *** Update the list of vertex pointers
        ***

        *** 'Bubble up'
        ***
        pCurNode = pParentNode->pParent;
        pParentNode = pParentNode->pParent;
    }
    return BINQUEUE_OK;
}
```
```c
08/02/98
19:19:02

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include "FibHeap.h"

primfh.c

/**
 *** Allocate data structures
 ***
 FibNodeStore_create(nVertices, &nodeStore);
PQueue = FibHeap_make(nVertices);
Pii = (PIArray)malloc(nVertices * sizeof(int));
/**
 *** Take start time reading
 */
ftime(&tp);
tStart = tp.time;
unStart = tp.millisecond;
/**
daStartCPUtime = CPUtime;
/**
 *** Put all vertices onto the queue - Use only connected ones. The size
 *** member of the queue is used to account for the number of vertices
 *** put onto the queue
 */
for (u = 0; u < nVertices; u++)
{
 *** Search for the first connected vertex to determine if it goes onto
 *** the queue
 ***
v = 0;
while ((v < nVertices) && (pGraph[u][v] == 0))
 v++;
/**
 *** See what stopped the loop
 ***/
if (v < nVertices)
{
 *** The vertex is connected, queue it up
 ***
pNode = FibNodeStore_alloc(&nodeStore);
pNode->xVert = u;
if (nVertCount == 0)
{
 pNode->xKey = 0;
pRootVertex = pNode;
}
PQueue = FibHeap_insert(PQueue, pNode);
nVertCount++;
}
/**
 *** Make sure that the number of connected vertices is at least 2.
 ***/
if (nVertCount < 2)
{
 FibNodeStore_destroy(&nodeStore);
FibHeap_destroy(PQueue);
free(Pii);
return NOTEnoughCONNECTED_VERTEXES;
}
/**
 *** Assume r is the first entry in the Queue - set its key to 0 and its PI
 ***
...
08/02/98
19:19:02

primfh.c

/**
 * u = pRootVertex->unVert;
 * Pi[u] = NIL;
 */

/*** Begin iteration over the queue looking for edges
 ***/
while ((pNode = FibHeap_extractMin(pQueue)) != NULL) {
  u = pNode->unVert;

  /**
   * Iterate over all vertices adjacent to u
   ***/
  for (v = 0; v < nVertices; v++) {
    if ((v != u) && (pQueue->aInQueue[v]) && (pGraph[u][v] == 0) &&
      (pGraph[u][v] < pQueue->aVertices[v]->nKey)) {
      Pi[v] = u;
      FibHeap_decreaseKey(pQueue, v, pGraph[u][v]);
    }
  }

  /**
   * We should now have the MST!!!!
   * Take the end time reading
   ***/

  /*
   * ftime(&tp);
   * tEndTime = tp.time;
   * unEndTime = tp.millis;
   * pdtTime = ((double)tEndTime - (double)tStartTime) +
             ((double)unEndTime - (double)unStartTime) / 1000.0;
   */
  dEndCPuTime = CPUTime;
  pdtTime = dEndCPuTime - dStartCPuTime;

  /**
   * Reconstruct the output tree
   ***/
  for (u = 0; u < nVertices; u++) {
    if (Pi[u] != NIL)
      nCost += pOutGraph[Pi[u]][u] * pOutGraph[u][Pi[u]] = pGraph[u][Pi[u]];
  }

  /**
   * Clean up
   ***/
  FibNodeStore_destroy(&nodeStore);
  FibHeap_destroy(&pQueue);
  free(Pi);

  return nCost;
}

**************************************************************************/
#include "FibHeap.h"

FUNCTION: FibNodeStore_create

***

INPUT ARGUMENTS: int size - size of the store to be created

OUTPUT ARGUMENTS: FibNodeStore* pStore - node store created

RETURNS: BOOL - TRUE if successful, FALSE if failed

DESCRIPTION: This function allocates memory necessary for the nodes of the binomial heap. The allocated memory is to be used for the "allocation" of nodes.

BOOL FibNodeStore_create(int size, FibNodeStore* pStore)
{
    int i; /* Array index */

    *** First, allocate space for the actual nodes.
    if ((pStore->aNodes = (FFibNode*)malloc(size * sizeof(FFibNode))) == NULL)
        return FALSE;

    *** Now that the nodes are allocated, we allocate space to hold pointers to each
    if ((pStore->apNodes = (FFibNode*)malloc(size * sizeof(FFibNode))) == NULL)
    {
        free(pStore->aNodes);
        return FALSE;
    }

    *** Everything is allocated. Now we set pointers and initialize values
    for (i = 0; i < size; i++)
    {
        pStore->aNodes[i] = &pStore->aNodes[i]; /* set the pointer */
        pStore->aNodes[i].pLeft = NULL;
        pStore->aNodes[i].pRight = NULL;
        pStore->aNodes[i].pParent = NULL;
        pStore->aNodes[i].pChild = NULL;
        pStore->aNodes[i].nKey = INFINITY;
        pStore->aNodes[i].nVert = 0;
        pStore->aNodes[i].nDegree = 0;
        pStore->aNodes[i].nMark = FALSE;
    }

    pStore->numNodes = size;
    pStore->top = 0;

    *** Done
    return TRUE;
}

FUNCTION: FibNodeStore_destroy

***

INPUT ARGUMENTS: FibNodeStore* pStore - node store destroyed

OUTPUT ARGUMENTS: NONE

RETURNS: NOTHING

DESCRIPTION: This function releases memory allocated for the binomial heaps.

void FibNodeStore_destroy(FibNodeStore* pStore)
{
    free(pStore->aNodes);
    free(pStore->apNodes);
}

FUNCTION: FibNodeStore_alloc

***

INPUT ARGUMENTS: FibNodeStore* pStore - node store from which the node is to be allocated

OUTPUT ARGUMENTS: NONE

RETURNS: FFibNode - pointer to binomial heap node. NULL if this operation fails.

DESCRIPTION: This function allocates a node from the store and returns a pointer to it. If this operation fails, NULL is returned.

FFibNode FibNodeStore_alloc(FibNodeStore* pStore)
{
    *** Check if nodes are available for allocation
    if (pStore->top >= pStore->numNodes)
        return NULL;

    *** Get the next node and return a pointer to it, move the top down
    return pStore->apNodes[pStore->top++];
}
FibHeap.c

/**
 *** Function:  FibNodeStore_free
 ***
 *** Input Arguments:  PFibNode* node - pointer to the pointer that
 ***  points to node to be freed
 ***
 *** Output Arguments:  FibNodeStore* pStore - node store to which the
 ***  node is to be returned.
 ***
 *** Returns:  None
 ***
 *** Description:  This function returns the allocated node back to
 ***  the store.
 ***
 */

void FibNodeStore_free(PFibNode* node, FibNodeStore* pStore)
{
    pStore->top--;  
    pStore->apNodes[pStore->top] = *node;
    *node = NULL;
}

PFibQueue FibHeap_make(int nMaxSize)
{
    PFibQueue pRetVal;  
    int i;  
    /* Return value */
    /* Index into the arrays */

    /* Allocate the queue itself and set its rootlist to NULL */
    pRetVal = (PFibQueue)malloc(sizeof(PFibQueue));
    pRetVal->pRootList = NULL;
    pRetVal->nCurrentSize = 0;
    pRetVal->pMin = NULL;
    pRetVal->nMaxSize = nMaxSize;

    /* Allocate space for the arrays and initialize them */
    pRetVal->apVertexNodes = (PFibNode*)malloc(nMaxSize * sizeof(PFibNode));
    pRetVal->aInQueue = (BOOL*)malloc(nMaxSize * sizeof(BOOL));
    /r
    pRetVal->A = (PFibNode*)malloc(nMaxSize * sizeof(PFibNode));
    /r
    for (i = 0; i < nMaxSize; i++)
    { /r
    }
FibHeap

*** Create and initialize the resulting heap
***
\[ h = \text{FibHeap}\_\text{make}(h1->nMaxSize); \quad / \text{Create the resulting heap} *\]
\[ h->pMin = h1->pMin; \quad / \text{Initialize the pointer to min} *\]

*** Concatenate the root list of H2 with the root list of H
***
\[ h->pRootList = \text{FibHeap\_concatLists}(h1->pRootList, h2->pRootList); \]

*** Adjust the minimum pointer to reflect the min
***
if \((h1->pMin == \text{NULL}) || (h2->pMin != \text{NULL}) \&\& (h2->pMin->nKey < h1->pMin->nKey))
\[ h->pMin = h2->pMin; \]
\[ h->nCurrentSize = h1->nCurrentSize + h2->nCurrentSize; \]
\[ return h; \]

FibNode

\[ \text{FibHeap\_extractMin}(\text{FibQueue} h) \]
\{ \text{FibNode pMinNode = NULL}; \quad / \text{Pointer to node with min key} */
\text{FibNode pCurChild; } \quad / \text{Iteration over child nodes} */
\text{BOOL bMadeFullCycle = FALSE; } \quad / \text{Flags full cycle over list} */
\}

if \((h->nCurrentSize > 0) \&\& (h->pRootList != \text{NULL}) \&\&
\text{((pMinNode = h->pMin) != \text{NULL}))
\{ \quad / \text{Add all children of the node pointed to by pMinNode to the root list} */
\text{if }\text{pMinNode->pChild} != \text{NULL} \}
\{ \quad / \text{Make sure that the parent of every child is NULL} */
\text{pCurChild = pMinNode->pChild; }
while (!bMadeFullCycle)
\{ \text{pCurChild->pParent = NULL; }
\text{pCurChild = pCurChild->pRight; }
\text{if (pCurChild = pMinNode->pChild)
\text{bMadeFullCycle = TRUE; }}
\}
\}

*** Now for the node itself
***
\text{pCurChild = pMinNode->pChild; }
\text{pCurChild->pParent = NULL; }
\text{pMinNode->pChild = NULL; }

*** Concatenate the child list with the heap’s root list and set new
*** minimum

FibQueue

\[ \text{FibHeap\_union}(\text{FibQueue} h1, \text{FibQueue} h2) \]
\{ \text{FibQueue h; } \quad / \text{Pointer to the resulting heap} */
\}

/**/
```c
FibHeap.c

*** Iterate over the nodes in the root
***/

/**
 * Break the circle to stop
 */
if (h->nCurrentSize > 1)
{
    pMinNode->pRight = pMinNode->pLeft;
    pMinNode->pLeft->pRight = pMinNode->pRight;
    if (h->pRootList == pMinNode)
        h->pRootList = pMinNode->pRight;
    h->pMin = pMinNode->pRight;
    FibHeap_consolidate(h);
}
else
{
    h->pRootList = NULL;
    h->pMin = NULL;
}

/**
 * h->nCurrentSize--; 
 * h->aInQueue[pMinNode->unVert] = NULL;
 * h->aInQueue[pMinNode->unVert] = FALSE;
 */

return pMinNode;

FPFibNode FibHeap_consolidate(FPFibQueue h)
{
    int i; /* Index into the array */
    int D; /* Size of the auxiliary array */
    PPFibNode pw; /* Pointer to the current node */
    PPFibNode px, py; /* Nodes X and Y */
    PPFibNode ptmp; /* Swap pointer */
    int d; /* Degree of the current node */
    BOOL bFullCycle = FALSE; /* Made a full cycle over the list*/

    *** Allocate and initialize the auxiliary array A
    ***/
    /* D = (int) ceil(log2((double) h->nCurrentSize)) + 1; */
    /* D = h->nCurrentSize; */ /* For debugging on Borland with no log2 */
    for (i = 0; i < h->D; i++)
        h->A[i] = NULL;

    ***
    }
    return h;
}
FibHeap.c

if (!!(h->inQueue[unVert]) ||
    ((pX = h->vertexNodes[unVert]) == NULL) ||
    (nKey == h->vertexNodes[unVert]->nKey))
    return h;

    /*
    * Set the new key to the vertex
    */
P->nKey = nKey;
P = pX->parent;

if ((P != NULL) && (P->nKey < P->nKey))
    FibHeap_cut(h, P, P->nKey);
    FibHeap_cascadingCut(h, P);

if (P->nKey < h->pMin->nKey)
    h->pMin = P;
return h;

FibHeapQueue FibHeap_decreaseKey(FibHeapQueue h, unsigned int unVert, int nKey)
{
    FibNode pY;
    FibNode pX;

    /*
    * Check if the key can be decreased. If not, exit without doing anything
    */
    /*
    * Remove node Y from the root list
    */
    if (P->right != NULL)
        if (P->right->pLeft == P->right)
            if (P->right->pRight == P->right)
                P->right->pLeft = P->right;
                P->right = P->right;
                P->pChild = P->right;
                P->pRight = P->right;
                P->pParent = P;
                else
                    P->pRight = P;
                    P->pRight->pLeft = P->pLeft;
                    P->pRight->pRight = P->pRight;
                    P->pRight->pChild = P;
                    P->pRight->pDegree = P->pRight;
                    P->pRight->pMark = FALSE;
                    return h;
    
    P->pDegree = P;
    P->pDegree = P->pDegree;
    P->pDegree = P->pDegree;
    P->pDegree = P->pDegree;
    return h;
}
FibHeap.c

PPfibQueue FibHeap_cascadingCut (PPfibQueue h, PPfibNode pY)
{
    PPfibNode pParent;    /* Pointer to parent of node Y    */
    if ((pParent = pY->pParent) != NULL)
        if (!pY->bMark)
            pY->bMark = TRUE;
        else
            { FibHeap_cut (h, pY, pParent);
               FibHeap_cascadingCut (h, pParent);
            }

    return h;
}