1998

Experimental study of performance of minimum spanning tree algorithms

Alec Berenbaum

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

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.

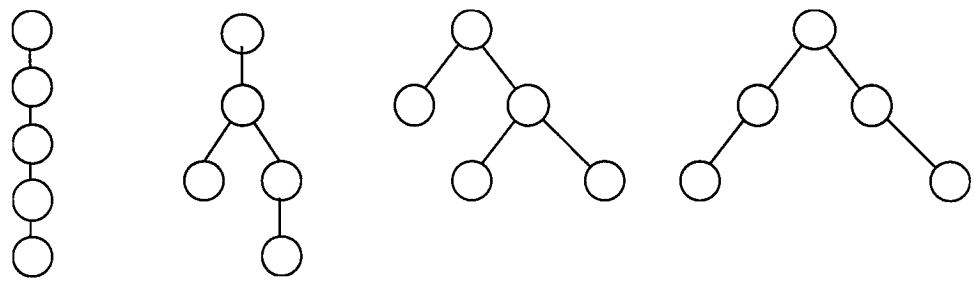

Figure 1 – Rooted trees with 5 nodes (vertices)

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_1. If there exist edges w_1x shorter than vw_1, choose the shortest such edge w_1w_2. Continue in this way, as long as possible, constructing a simple path vw_1, w_1w_2, ... , $w_{k-1}w_k$, 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,..., $q_{l-1}q_l$, with l as large as possible, under the constraint that pq_1, q_1q_2,..., $q_{l-1}q_l$ 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_iq_{i+1}) [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_{u-1}, the solution. An implementation of Borůvka's algorithm would run in time $O(E \lg V)$, 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[i] can be connected to a nearest neighbor [3].
>
> *Principle 2* – Any isolated fragment[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:

Terminal	↔	vertex
Possible Link	↔	edge
Length of Link	↔	length or weight of edge
Connection Network	↔	spanning subgraph
(Without closed loops)	↔	(spanning subtree)
Shortest connection network	↔	shortest spanning subtree

---

[i] Prim refers to the terminal to which at a given stage of the construction, no connections have been made as an *isolated terminal.*

[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 resutls, we first analyze the Kruskal's [2] and Prims'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 $\varnothing$ 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[i] 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.

[i] 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 $x(1)$, $x(2)$, ... , $x(k-1)$ such that
   a) the set $\{x(1), x(2), ..., x(k-1), x(k)\}$ is an independent set
   b) if $\{x(1), x(2), ..., x(k-1), x\}$ 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 an 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 $I = \{e_1, e_2, ..., e_r\}$ is obtained (in a matroid) in which the elements are arranged in nondecreasing order by weight. If $J = \{f_1, f_2, ..., f_r\}$ 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 $i = 1,2,...,(m-1)$. Thus, the proof is required for $i = m$. Suppose $w(f_m) > w(e_m)$. Let $D = \{e_1, e_2, ..., e_{m-1}\}$ 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 $\{f_1, f_2, ..., f_m\}$ 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]:

```
GENERIC-MST(G,w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge (u,v) that is safe for A
4          A ← A Y {(u,v)}
5  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 $A \, Y \{(u,v)\}$ 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.

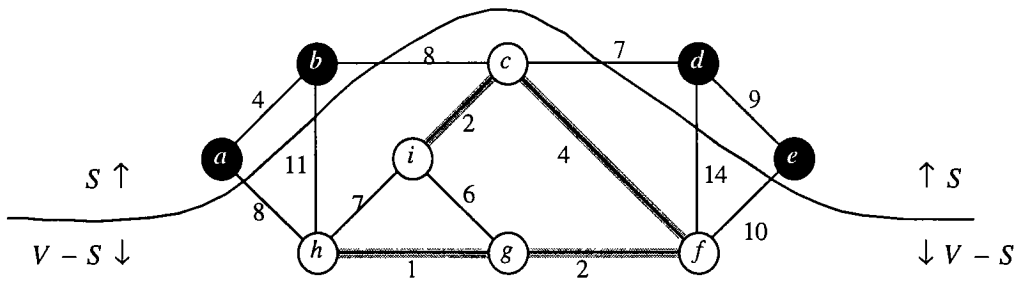

Figure 2 – Cut (*S, V-S*) [7]

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].

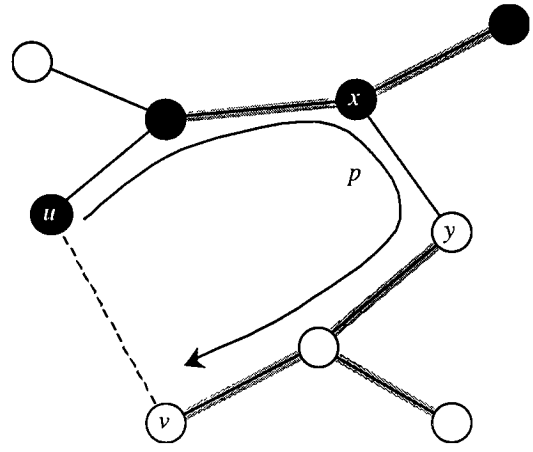

Figure 3 – Proof of Theorem 3

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 $T' = T - \{(x, y)\} \cup \{(u, v)\}$ [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.

$A \subseteq T'$, since $A \subseteq T$ and $(x, y) \notin A$; thus, $A \cup \{(u, v)\} \subseteq T'$. 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 $A = \varnothing$. 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]:

```
MST-KRUSKAL(G,w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3      do MAKE-SET(v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u,v) ∈ E, in order by nondecreasing weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7             then A ← A Y {(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 α 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]:

```
MST-PRIM(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 v, $key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree. If no such edge exists, $key[v] = \infty$. The field $\pi[v]$ specifies the parent of v in the tree [7]. While the algorithm is running, the set of edges A is:

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

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

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

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 $O(m \lg n)$ running time, where $\Omega(n)$ lower bound us 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 $\Theta(n + f \lg n)$ 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:

$$
\begin{aligned}
A(1, j) &= 2^j && \text{for} \quad j \geq 1 \\
A(i,1) &= A(i-1,2) && \text{for} \quad i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) && \text{for} \quad i, j \geq 2
\end{aligned}
$$

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, $O(m \lg^* n)$. They use the aggregate method of amortized analysis to prove the $O(m \lg^* n)$ time bound [7]. CLR present and prove the following theorem:

> A sequence of m MAKE-SET, LINK[i], 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 $O(m \lg^* n)$ [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 $\frac{1}{2}\varepsilon(V^2 - V)$ 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.

---

[i] The LINK procedure is called by 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[i]

$$T = T_M + T_F + T_U + T_S .$$

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:

	MST-KRUSKAL(G,w)	
1	$A \leftarrow \varnothing$	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) $\neq$ FIND-SET(v)	E times
7	**then** $A \leftarrow A \cup \{(u,v)\}$	$V-1$ 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,

[i] T_M – total time of all MAKE-SET operations, T_F – total time of all FIND-SET operations, T_U – total time of all UNION operations, T_S – total time spent sorting edges

$$m = V + 2E + V - 1 = 2V + 2E - 1$$

Given that $E = \frac{V^2 - V}{2} \times \varepsilon$, where $\varepsilon = p$ is the density of the graph G,

$$m = 2V + V^2\varepsilon - V\varepsilon - 1$$
$$= V^2\varepsilon + V(2 - \varepsilon) - 1$$

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

$$m \lg^* V = 4\varepsilon V^2 + (4 - 4\varepsilon)V - 4$$
$$= O(V^2)$$

for fixed density ε. We therefore state that the total running time of ether 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 $i = 1, 2, \ldots, k$. 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 $O(n \lg n)$ 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 $O(\lg n)$. We therefore say that the heapsort runs in $O(n \lg n)$ time. Given that we are sorting the edges of the graph, using the heapsort the sorting time is

$$T_S = O(E \lg E) = O(V^2 \lg V)$$

The total running time of the algorithm is then,

$$\begin{aligned} & O(V^2) + O(V^2 \lg V) \\ & = O(V^2 \lg V) \end{aligned}$$

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[i]

$$T = T_B + \sum_{v \in G} T_E + \sum_{\{u,v\} \in G} T_D = T_B + VT_E + E\varepsilon T_D$$

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

[i] T_B – time to build the heap; T_E – time to perform EXTRACT-MIN operation; T_D – time to perform DECREASE-KEY operation.

$$
\begin{aligned}
T &= \Theta(V) + \Theta(V^2) + O(E) \\
&= V + V^2 + \tfrac{1}{2}\varepsilon V^2 + \tfrac{1}{2}\varepsilon V \\
&= (1\tfrac{1}{2}\varepsilon + 1)V^2 + (1\tfrac{1}{2}\varepsilon + 1)V \\
&= \Theta(V^2)
\end{aligned}
$$

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" (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 $\lfloor \lg n \rfloor$ vertices, since the height of a binary heap of n elements is $\lfloor \lg n \rfloor$. 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 $\lfloor \lg n \rfloor$ 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 ∞. This eliminates the requirement of satisfying the heap invariant upon BUILD. Nevertheless, the key value of each element must be set to ∞, 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 ε.

For dense graphs, where ε 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].

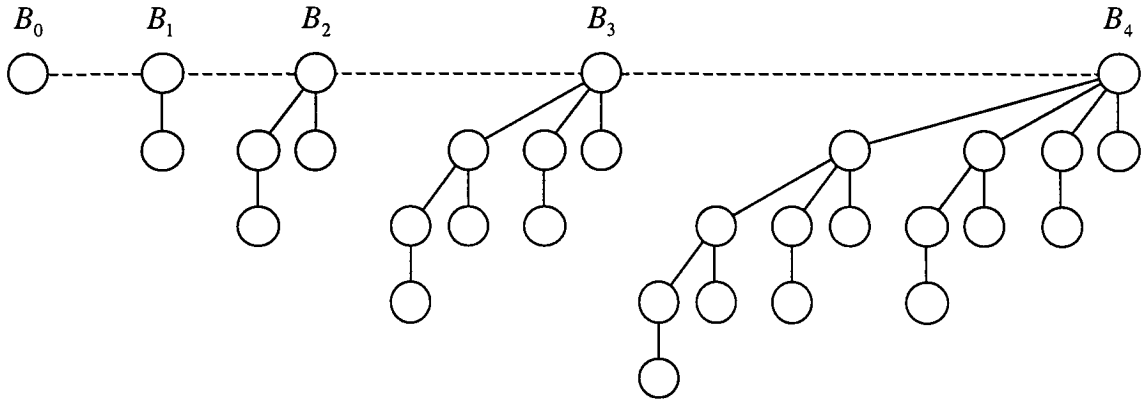

Figure 4 – Binomial Heap consisting of binomial trees B_0 through B_4.

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 $i = 0,1,\text{K},k$.

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 $k-1, k-2, \text{K}, 0$, 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}. B_k, 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}, 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 $B_{k-2}, B_{k-3}, \text{K}, B_0$. When B_{k-1} is linked to B_{k-1}, the resulting root are the roots of $B_{k-1}, B_{k-2}, B_{k-3}, \text{K}, B_0$ [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 $B_0, B_1, \text{K}, B_{k-1}$ 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 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 EXTRACT-MIN operation is not as straight-forward as in the binary heap. When an 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 $\{B_0', B_1', \text{K } B_{k-1}'\}$. Since the trees of the degrees $0,1,\text{K}, k-1$ are already present in the binomial heap, the newly formed trees must be linked with the existing trees. $\{B_0', B_1', \text{K } B_{k-1}'\}$ can be viewed as forming a binomial heap as well. Hence, performing an 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 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:

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

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

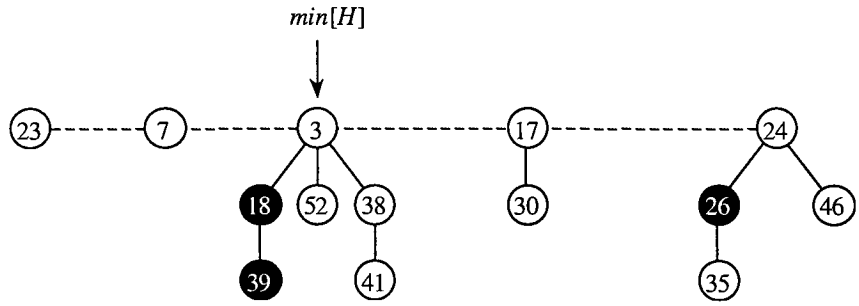

Figure 5 – Fibonacci Heap

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 $O(1)$. FIND-MIN is performed in $O(1)$ as well, since there is a pointer to the root node with the minimum key value. 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 CUT is performed, followed by a CASCADING-CUT. The 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 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 CUT is performed on the parent, followed by a 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 CASCADING-CUTS are completed, the $min[H]$ pointer is updated, if necessary. The time to perform DECREASE-KEY operation is amortized to $O(1)$ [13]. The actual cost to perform the DECREASE-KEY operation is $O(1)$ plus the time to perform the CASCADING-CUTs. If CASCADING-CUT is recursively called c times, given that the cost of decreasing the key is $O(1)$, the actual cost of performing the DECREASE-KEY plus the cost of cascading cuts is $O(c)$. DECREASE-KEY operation does not result in the ordered list of trees. The ordered list property is restored when the EXTRACT-MIN operation is performed. The cost of the DECREASE-KEY operation is therefore amortized. To determine the amortized cost of the DECREASE-KEY operation, we perform the amortized analysis. We first compute the change in potential. H denotes the Fibonacci heap prior to the DECREASE-KEY operation. Each recursive call to the 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 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 CASCADING-CUT may have marked a node. The change in potential, is then,

$$\begin{aligned}&((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))\\&=4-c\end{aligned}$$

The amortized cost of DECREASE-KEY is at most

$$O(c)+4-c=O(1)$$

EXTRACT-MIN operation is more complicated. This is where the trees of the root list are consolidated, the work that was delayed when INSERT and 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

$$\begin{aligned} &O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) \\ &= O(D(n))+O(t(H))-t(H) \\ &= O(D(n)) \end{aligned}$$

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 dome 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

$$T = O(V) + O(\lg V) + O(E)$$
$$= O(E)$$

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 $2\lg V$ time, hence $T_E = O(\lg V)$. The time to perform DECREASE-KEY operation is then done in $\Theta(1)$ time. The running time of this implementation is:

$$T = \Theta(V) + O(V \lg V) + O(V_1 \cdot V_2)$$
$$= O(E)$$

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 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 $(0,0)...(n,n)$ 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.

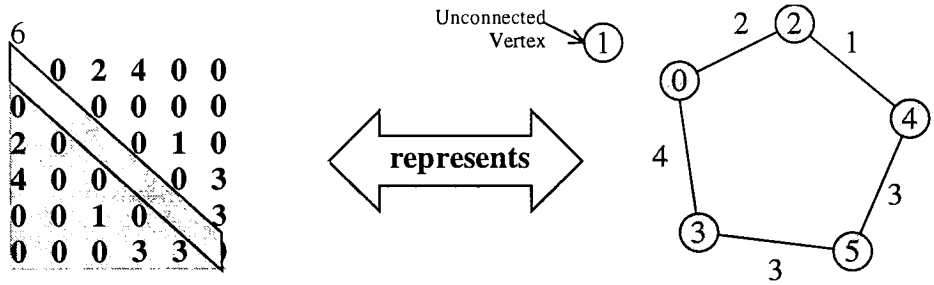

Figure 6 – Data File Format: Adjacency Matrix Representation of Graphs

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 *gengraph* program, written specifically for this purpose. The *gengraph* 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 *gengraph* 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.

Number of Vertices (1 file for each)	Density	Minimum Edge Weight	Maximum Edge Weight
100 – 1700	0.2	0	10
100 – 1700	0.2	0	100
100 – 1700	0.2	1020	1022
100 – 1700	0.5	0	10
100 – 1700	0.5	0	100
100 – 1700	0.5	1020	1022
100 – 1700	0.8	0	10
100 – 1700	0.8	0	100
100 – 1700	0.8	1020	1022

Table 1 – Summary of data files used in the experiment.

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.

Reference	Algorithm Implementation
khs	Kruskal with Heap Sort
kcs	Kruskal with Counting Sort
pbf	Prim Using Brute Force for Queue Implementation
php_a	Prim with Queue Implemented on Heap (proper)
php	Prim with Queue Implemented on Heap (lazy)
pbinh	Prim with Queue Implemented on Binomial Heap
pfibh	Prim with Queue Implemented on Fibonacci Heap

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 ruse`, 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 π or its *key* using the index reference. For example, the entry for the parent π of vertex 6, is located in the `Pi[6]`, where `Pi` is the variable that contains the array of π.

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 $\Theta(|V|)$ time.

The binary heap operations, used with Prim's algorithm, are performed on the array field `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:

```
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:

```
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:

```
struct __tagBinHeapMemMgt__
{
   struct _tagBinNode_*  aNodes;  /* Array of Nodes      */
   struct _tagBinNode_** apNodes; /* Array of pointers to*/
```

```
                                    /*                nodes */
   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:

```
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:

```
struct __tagFibCollection__
{
   struct __tagFibNode__* pRootList; /* List of roots    */
   int                    nCurrentSize;
   struct __tagFibNode__* pMin;      /* Pointer to node  */
                                     /* with the minimum */
                                     /* key value        */
   int                    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 UltraSPARCs 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.

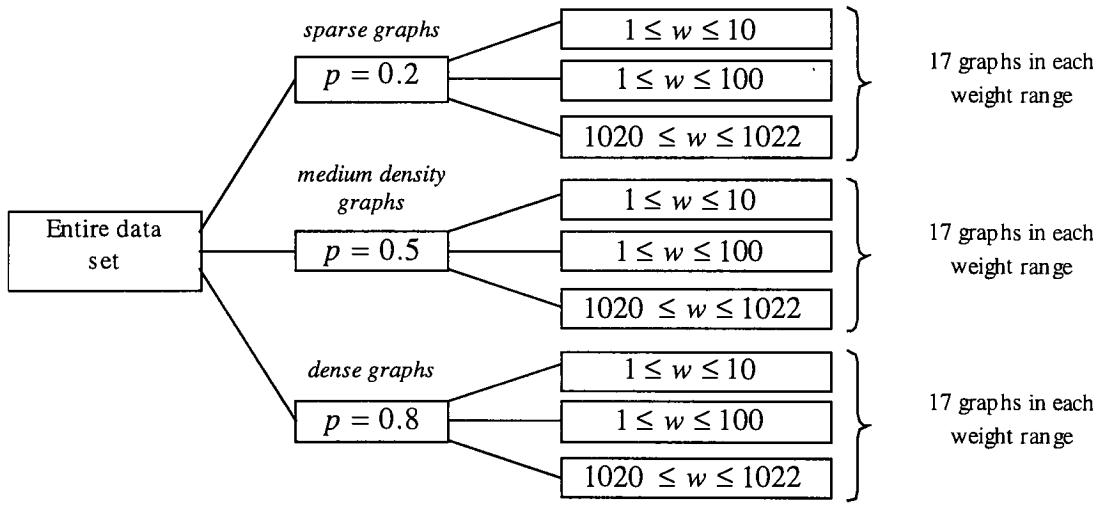

Figure 7 – Categorization of the data sets used in the experiment

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:

Graph Density: 0.2 Range of Edge Weights: 1 - 10							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0147	0.0031	0.0066	0.0058	0.0038	0.0141	0.0066
200	0.0709	0.0114	0.0258	0.0205	0.0117	0.0432	0.0212
300	0.1831	0.0254	0.0595	0.0454	0.0241	0.0892	0.0388
400	0.3467	0.0451	0.1030	0.0833	0.0458	0.1484	0.0656
500	0.5619	0.0706	0.1612	0.1194	0.0611	0.2214	0.0963
600	0.8487	0.1021	0.2322	0.1713	0.0867	0.3170	0.1338
700	1.1899	0.1395	0.3190	0.2333	0.1164	0.4199	0.1772
800	1.5936	0.1823	0.4121	0.3015	0.1494	0.5362	0.2283
900	2.0922	0.2320	0.5220	0.3809	0.1878	0.6708	0.2853
1000	2.6412	0.2870	0.6439	0.4692	0.2308	0.8428	0.3517
1100	3.2705	0.3466	0.7803	0.5680	0.2782	0.9863	0.4190
1200	4.0365	0.4132	0.9299	0.6803	0.3304	1.1661	0.4969
1300	4.7130	0.4851	1.0939	0.7945	0.3879	1.3594	0.5801
1400	5.5150	0.5627	1.2700	0.9225	0.4479	1.5735	0.6650
1500	6.4591	0.6467	1.4749	1.0734	0.5143	1.7837	0.7591
1600	7.4359	0.7387	1.6651	1.2249	0.5863	2.0561	0.8660
1700	8.6536	0.8346	1.8706	1.3646	0.6575	2.2849	0.9370

Table 3 – Performances for sparse graphs with $1 \leq w \leq 10$, with times in CPU seconds

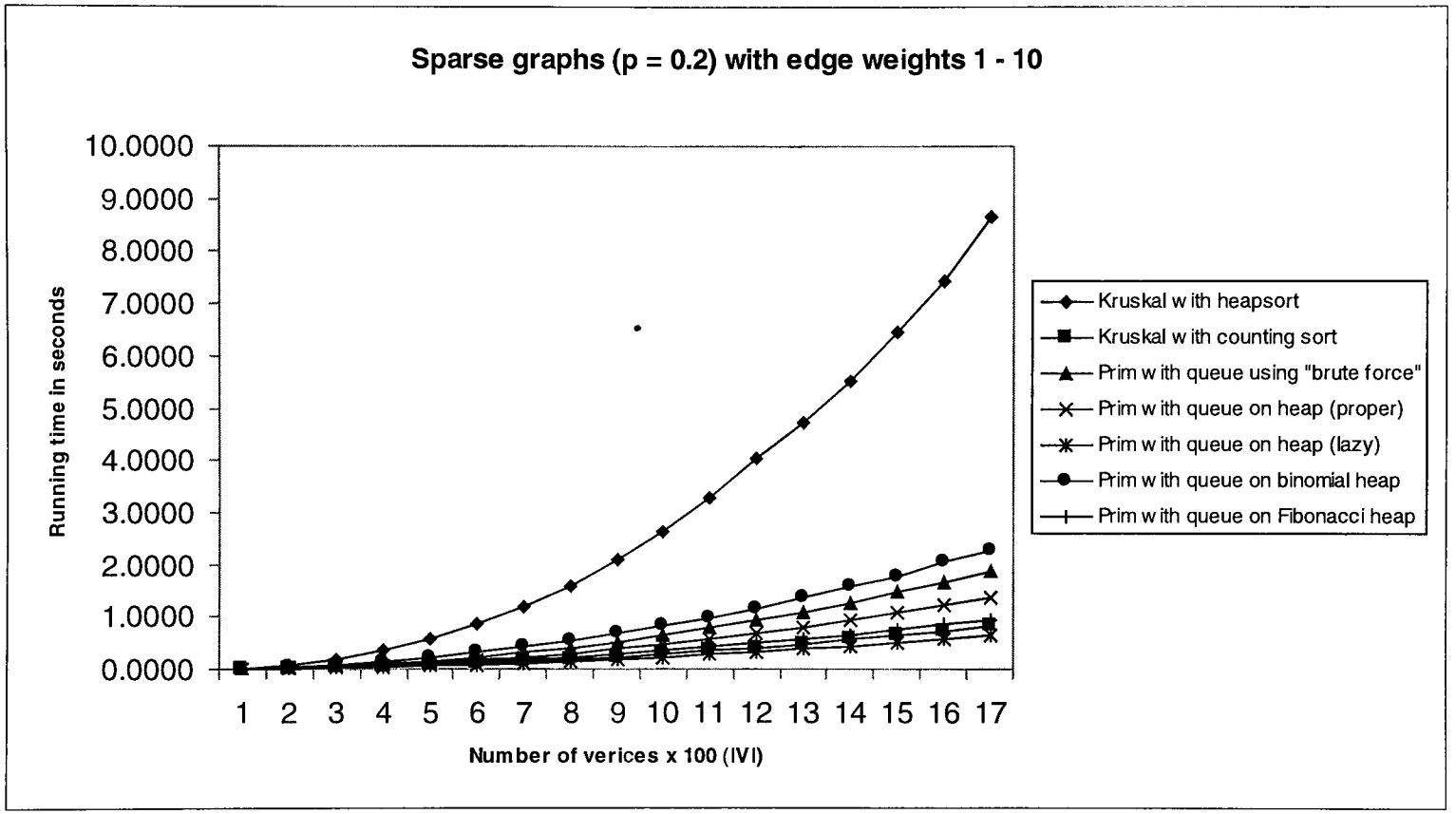

Figure 8 – Performances for sparse graphs with $1 \leq w \leq 10$ (see Table 3)

Graph Density: 0.2 Range of Edge Weights: 1 – 100							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0158	0.0031	0.0065	0.0061	0.0043	0.0140	0.0068
200	0.0771	0.0115	0.0251	0.0218	0.0137	0.0437	0.0203
300	0.2038	0.0254	0.0580	0.0485	0.0283	0.0909	0.0407
400	0.3733	0.0452	0.1006	0.0816	0.0466	0.1516	0.0671
500	0.6168	0.0710	0.1574	0.1262	0.0709	0.2288	0.1003
600	0.9265	0.1021	0.2255	0.1784	0.0977	0.3265	0.1404
700	1.3015	0.1415	0.3099	0.2430	0.1308	0.4323	0.1860
800	1.7522	0.1897	0.4002	0.3111	0.1651	0.5526	0.2385
900	2.2851	0.2443	0.5059	0.3928	0.2064	0.7412	0.2971
1000	2.9086	0.3010	0.6263	0.4823	0.2509	0.8465	0.3699
1100	3.5995	0.3647	0.7598	0.5837	0.2996	1.0157	0.4345
1200	4.5098	0.4345	0.9069	0.6943	0.3529	1.1996	0.5151
1300	5.2311	0.5110	1.0658	0.8095	0.4124	1.4260	0.5999
1400	6.1445	0.5923	1.2423	0.9344	0.4739	1.6142	0.6886
1500	7.1988	0.6805	1.4202	1.0760	0.5400	1.8346	0.7829
1600	8.2506	0.7768	1.6287	1.2313	0.6140	2.0870	0.8917
1700	9.6819	0.8765	1.8304	1.3793	0.6866	2.3454	1.0021

Table 4 – Performances for sparse graphs with $1 \leq w \leq 100$, with times in CPU seconds

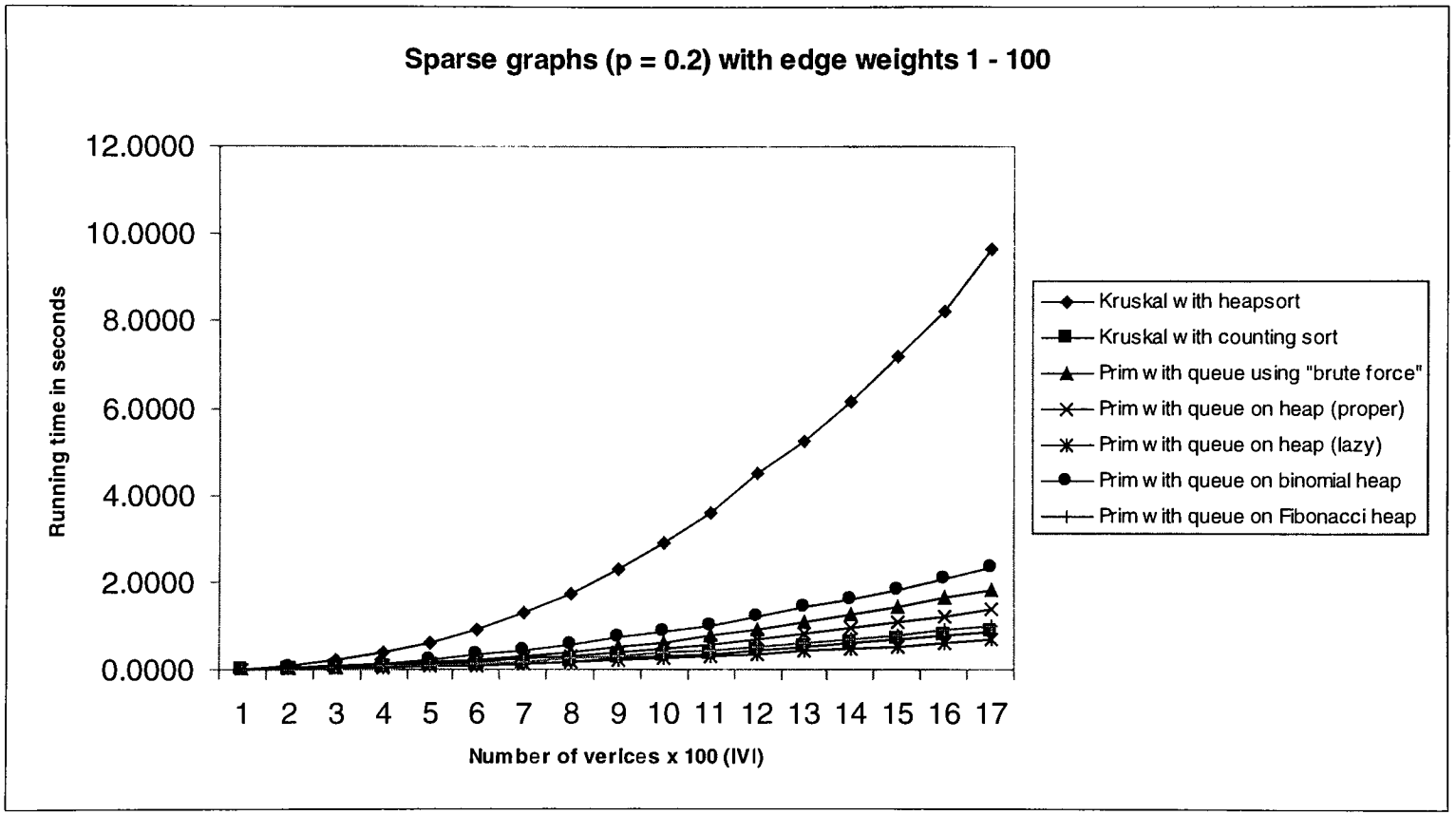

Figure 9– Performances for sparse graphs with $1 \leq w \leq 100$ (see Table 4)

Graph Density: 0.2 Range of Edge Weights: 1020 – 1022							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0115	0.0035	0.0068	0.0053	0.0032	0.0137	0.0063
200	0.0543	0.0119	0.0262	0.0195	0.1020	0.0423	0.0183
300	0.1386	0.0257	0.0603	0.0442	0.0224	0.0875	0.0366
400	0.2578	0.0455	0.1044	0.0767	0.0382	0.1450	0.0620
500	0.4263	0.0711	0.1625	0.1190	0.0588	0.2194	0.0934
600	0.6282	0.1023	0.2334	0.1694	0.0837	0.3164	0.1310
700	0.8994	0.1397	0.3211	0.2312	0.1134	0.4359	0.1737
800	1.1921	0.1824	0.4147	0.2989	0.1461	0.5331	0.2244
900	1.5563	0.2320	0.5249	0.3803	0.1843	0.6742	0.2795
1000	1.9937	0.2866	0.6472	0.4687	0.2269	0.8197	0.3445
1100	2.4258	0.3468	0.7832	0.5652	0.2736	0.9816	0.4136
1200	2.9752	0.4133	0.9339	0.6768	0.3259	1.1618	0.4890
1300	3.4941	0.4851	1.0971	0.7920	0.3823	1.4264	0.5729
1400	4.1360	0.5622	1.2743	0.9168	0.4430	1.5676	0.6563
1500	4.7967	0.6461	1.4588	1.0966	0.5126	1.7764	0.7538
1600	5.4759	0.7384	1.6698	1.2138	0.5809	2.0251	0.8568
1700	6.3664	0.8335	1.8733	1.3595	0.6520	2.3088	0.9662

Table 5 – Performances for sparse graphs with $1020 \leq w \leq 1022$, with times in CPU seconds

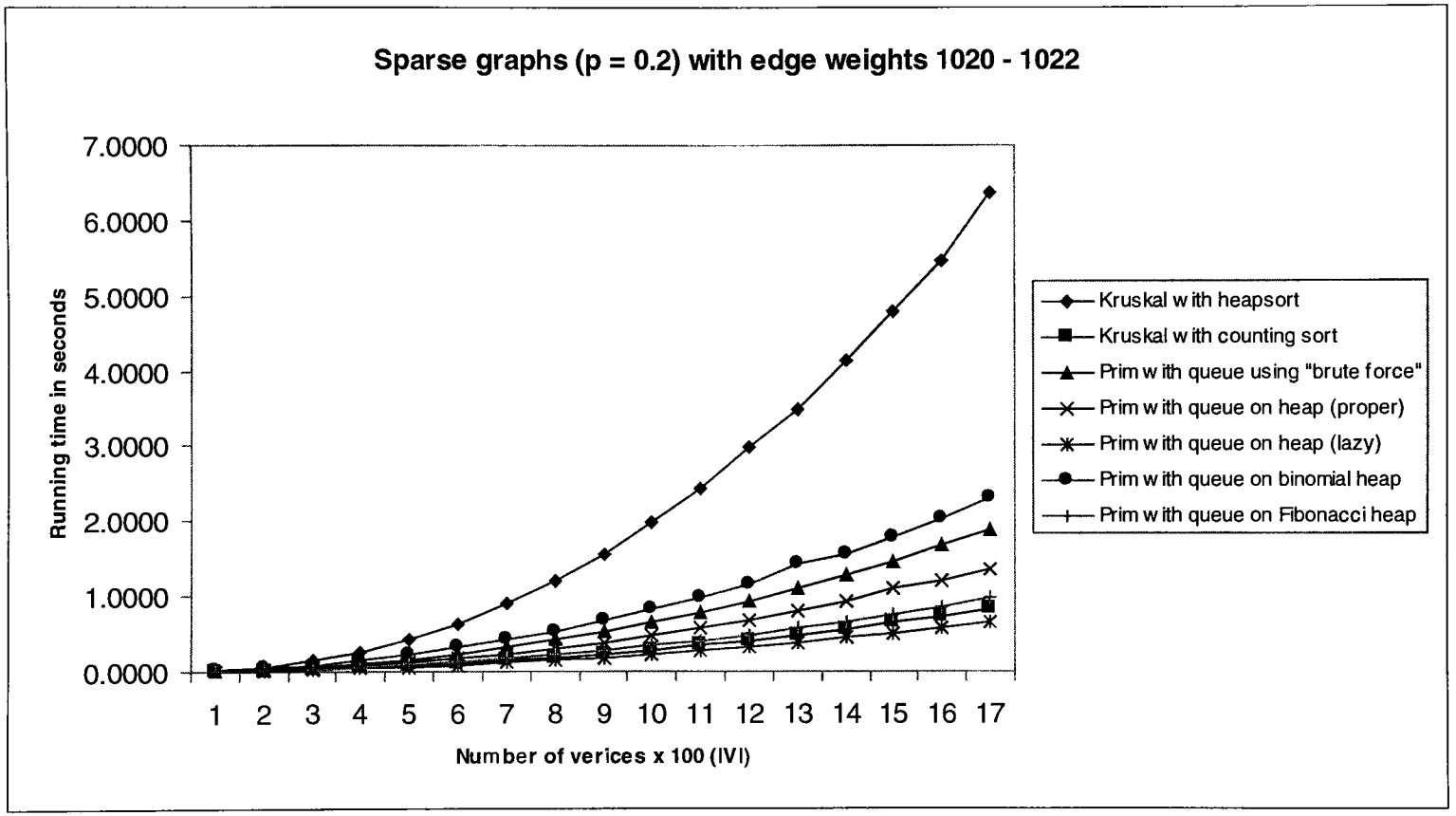

Figure 10 – Performances for sparse graphs with $1020 \leq w \leq 1022$ (see Table 5)

Graph Density: 0.5 Range of Edge Weights: 1 - 10							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0401	0.0070	0.0069	0.0058	0.0038	0.0139	0.0067
200	0.1963	0.0272	0.0270	0.0207	0.0119	0.0431	0.0200
300	0.5096	0.0612	0.0624	0.0468	0.0255	0.0903	0.0406
400	0.6106	0.1087	0.1072	0.0793	0.0431	0.1493	0.0683
500	1.5062	0.1715	0.1676	0.1233	0.0657	0.2262	0.1019
600	2.2659	0.2480	0.2410	0.1776	0.0931	0.3242	0.1438
700	3.1952	0.3379	0.3330	0.2428	0.1263	0.4360	0.1924
800	4.2631	0.4413	0.4277	0.3130	0.1627	0.5527	0.2484
900	5.5471	0.5593	0.5425	0.3969	0.2054	0.7394	0.3114
1000	6.9749	0.6919	0.6685	0.4896	0.2530	0.8501	0.3885
1100	8.5888	0.8370	0.8105	0.5942	0.3059	1.0284	0.4617
1200	10.7420	0.9971	0.9656	0.7102	0.3631	1.2126	0.5472
1300	12.3298	1.1697	1.1351	0.8437	0.4261	1.4154	0.6408
1400	14.5237	1.3661	1.3185	0.9606	0.4942	1.6286	0.7348
1500	16.8494	1.5649	1.5077	1.1116	0.5674	1.8588	0.8431
1600	19.5526	1.7858	1.7280	1.2736	0.6488	2.1169	0.9571
1700	22.8104	2.0134	1.9396	1.4297	0.7288	2.4122	1.0743

Table 6 – Performances for medium density graphs with $1 \leq w \leq 10$, with times in CPU seconds

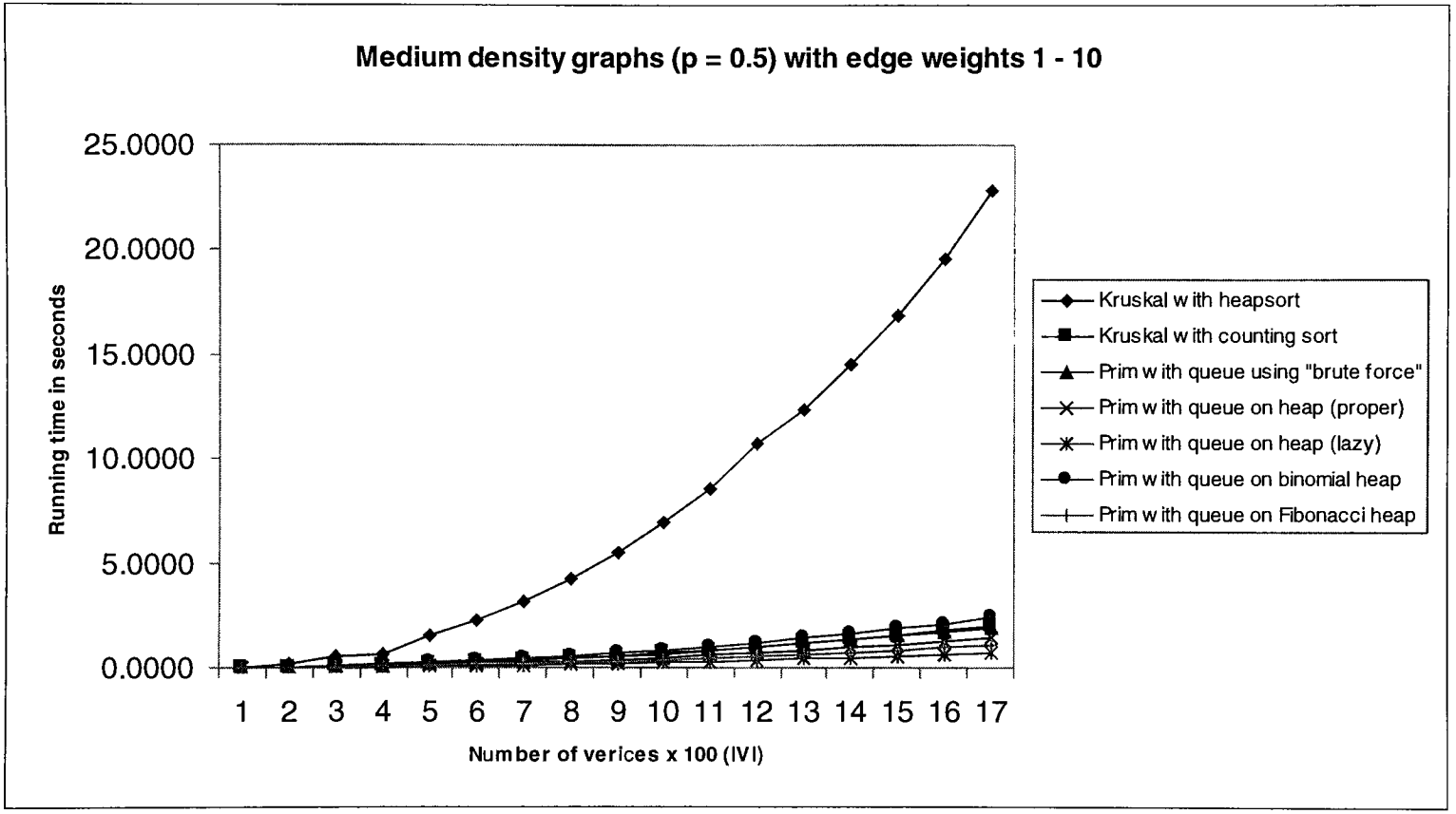

Figure 11 – Performances for medium density graphs with $1 \leq w \leq 10$ (see Table 6)

Graph Density: 0.5 Range of Edge Weights: 1 – 100							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0436	0.0070	0.0068	0.0063	0.0047	0.0165	0.0086
200	0.2147	0.0273	0.0261	0.0225	0.0147	0.0449	0.0217
300	0.5585	0.0616	0.0603	0.0499	0.0301	0.0930	0.0436
400	0.9993	0.1091	0.1041	0.0837	0.0491	0.1556	0.0731
500	1.6661	0.1780	0.1631	0.1306	0.0747	0.2398	0.1202
600	2.4838	0.2621	0.2349	0.1840	0.1022	0.3339	0.1514
700	3.4783	0.3577	0.3239	0.2504	0.1373	0.4437	0.2012
800	4.6982	0.4661	0.4190	0.3220	0.1747	0.5683	0.2580
900	6.1086	0.5903	0.5308	0.4065	0.2174	0.7081	0.3310
1000	7.7067	0.7303	0.6568	0.4995	0.2656	0.8679	0.3971
1100	9.5028	0.8841	0.7977	0.6034	0.3189	1.0441	0.4741
1200	11.9055	1.0768	0.9501	0.7232	0.3776	1.2346	0.5627
1300	13.6841	1.2354	1.1190	0.8427	0.4413	1.4683	0.6559
1400	16.0854	1.4419	1.2987	0.9749	0.5098	1.6574	0.7534
1500	18.8288	1.6544	1.4870	1.1228	0.5841	1.8760	0.8591
1600	21.5607	1.8884	1.7049	1.2850	0.6647	2.1367	0.9801
1700	25.2302	2.1301	1.9153	1.4441	0.7508	2.4498	1.1613

Table 7 – Performances for medium density graphs with $1 \leq w \leq 100$, with times in CPU seconds

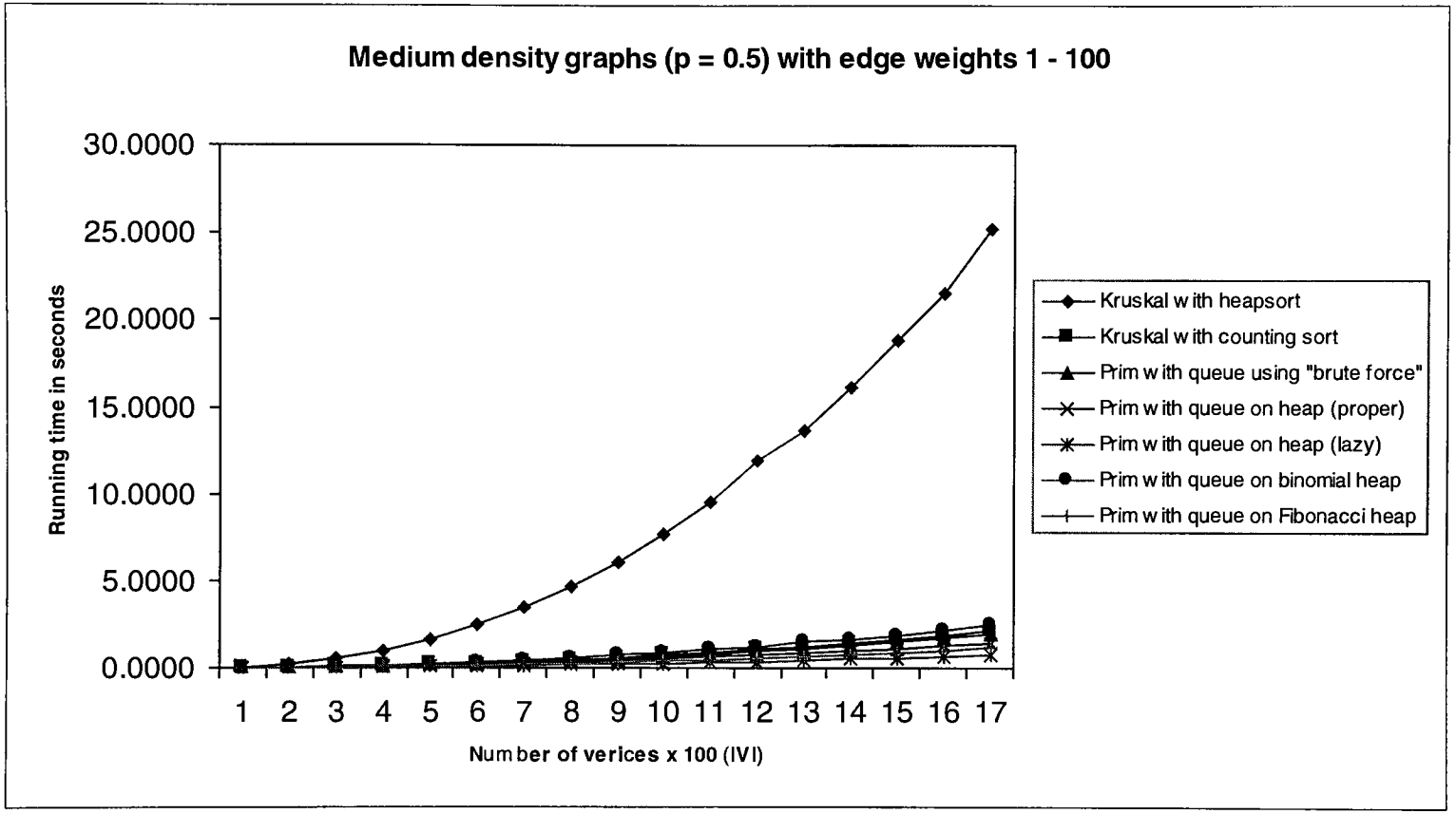

Figure 12 – Performances for medium density graphs with $1 \leq w \leq 100$ (see Table 7)

Graph Density: 0.5 Range of Edge Weights: 1020 – 1022							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0313	0.0073	0.0070	0.0053	0.0032	0.0135	0.0063
200	0.1490	0.0278	0.0272	0.0201	0.0109	0.0426	0.0191
300	0.3857	0.0618	0.0717	0.0461	0.0240	0.0915	0.0390
400	0.6841	0.1009	0.1078	0.0796	0.0411	0.1482	0.0657
500	1.1420	0.1721	0.1680	0.1234	0.0637	0.2259	0.1000
600	1.6992	0.2492	0.2415	0.1764	0.0908	0.3223	0.1404
700	2.3797	0.3390	0.3321	0.2414	0.1237	0.4299	0.1883
800	3.1604	0.4430	0.4325	0.3132	0.1603	0.5653	0.2436
900	4.1703	0.5614	0.5636	0.3978	0.2025	0.6927	0.3058
1000	5.2069	0.6934	0.6694	0.4899	0.2492	0.8463	0.3747
1100	6.3499	0.8401	0.8099	0.5934	0.3013	1.0249	0.4611
1200	7.9240	0.9995	0.9663	0.7106	0.3596	1.2236	0.5635
1300	9.3117	1.1728	1.1354	0.8324	0.4219	1.4489	0.6269
1400	10.7743	1.3621	1.3199	0.9657	0.4904	1.6257	0.7244
1500	12.4001	1.5644	1.5121	1.1116	0.5621	1.8579	0.8392
1600	14.3191	1.7913	1.7267	1.2743	0.6427	2.1243	0.9525
1700	16.7918	2.0175	1.9417	1.4319	0.7251	2.3949	1.0694

Table 8 – Performances for medium density graphs with $1020 \leq w \leq 1022$, with times in CPU seconds

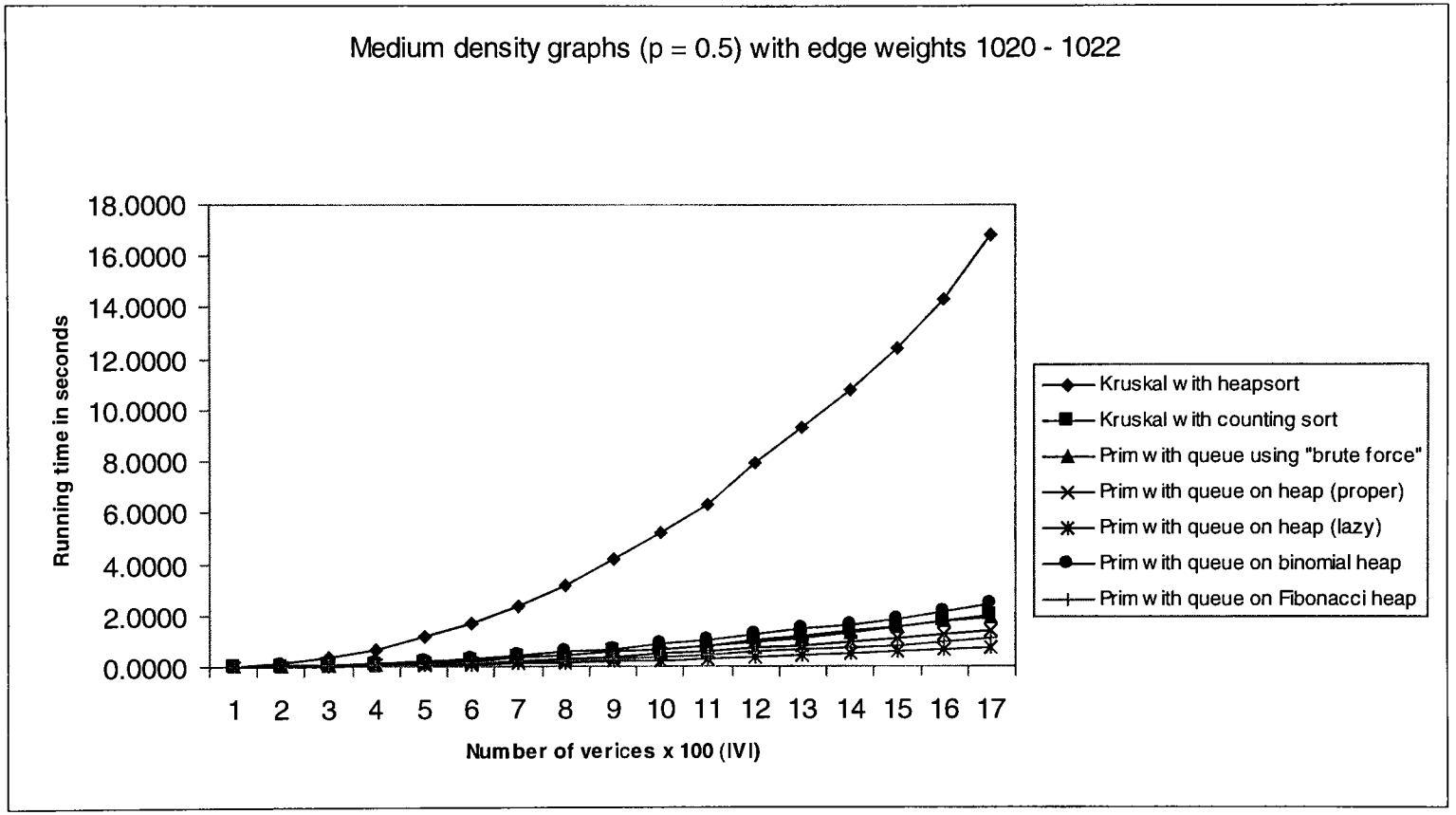

Figure 13 – Performances for medium density graphs with $1020 \leq w \leq 1022$ (see Table 8)

Graph Density: 0.8 Range of Edge Weights: 1 – 10							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0682	0.0108	0.0070	0.0059	0.0038	0.0140	0.0069
200	0.3294	0.0429	0.0277	0.0210	0.0121	0.0438	0.0207
300	0.8385	0.0968	0.0639	0.0477	0.0265	0.0917	0.0421
400	1.5404	0.1746	0.1101	0.0828	0.0450	0.1525	0.0720
500	2.5484	0.2741	0.1716	0.1275	0.0694	0.2340	0.1084
600	3.7630	0.3946	0.2469	0.1825	0.0986	0.3328	0.1547
700	5.2957	0.5371	0.3396	0.2496	0.1338	0.4434	0.2049
800	7.0743	0.7012	0.4389	0.3241	0.1733	0.5696	0.2654
900	9.2075	0.8895	0.5555	0.4104	0.2186	0.7184	0.3337
1000	11.5917	1.0985	0.6846	0.5054	0.2693	0.8783	0.4108
1100	14.2646	1.3391	0.8291	0.6645	0.3282	1.0556	0.4957
1200	17.6639	1.5826	0.9891	0.7329	0.3882	1.2538	0.5808
1300	20.6639	1.8575	1.1623	0.8591	0.4546	1.4923	0.6869
1400	23.9797	2.1588	1.3493	0.9953	0.5275	1.6834	0.7978
1500	27.8348	2.4826	1.5447	1.1479	0.6058	1.9195	0.9337
1600	32.0384	2.8346	1.7676	1.3158	0.6924	2.2692	1.0410
1700	37.4164	3.1890	1.9893	1.4811	0.7829	2.4799	1.1652

Table 9 – Performances for dense graphs with $1 \leq w \leq 10$, with times in CPU seconds

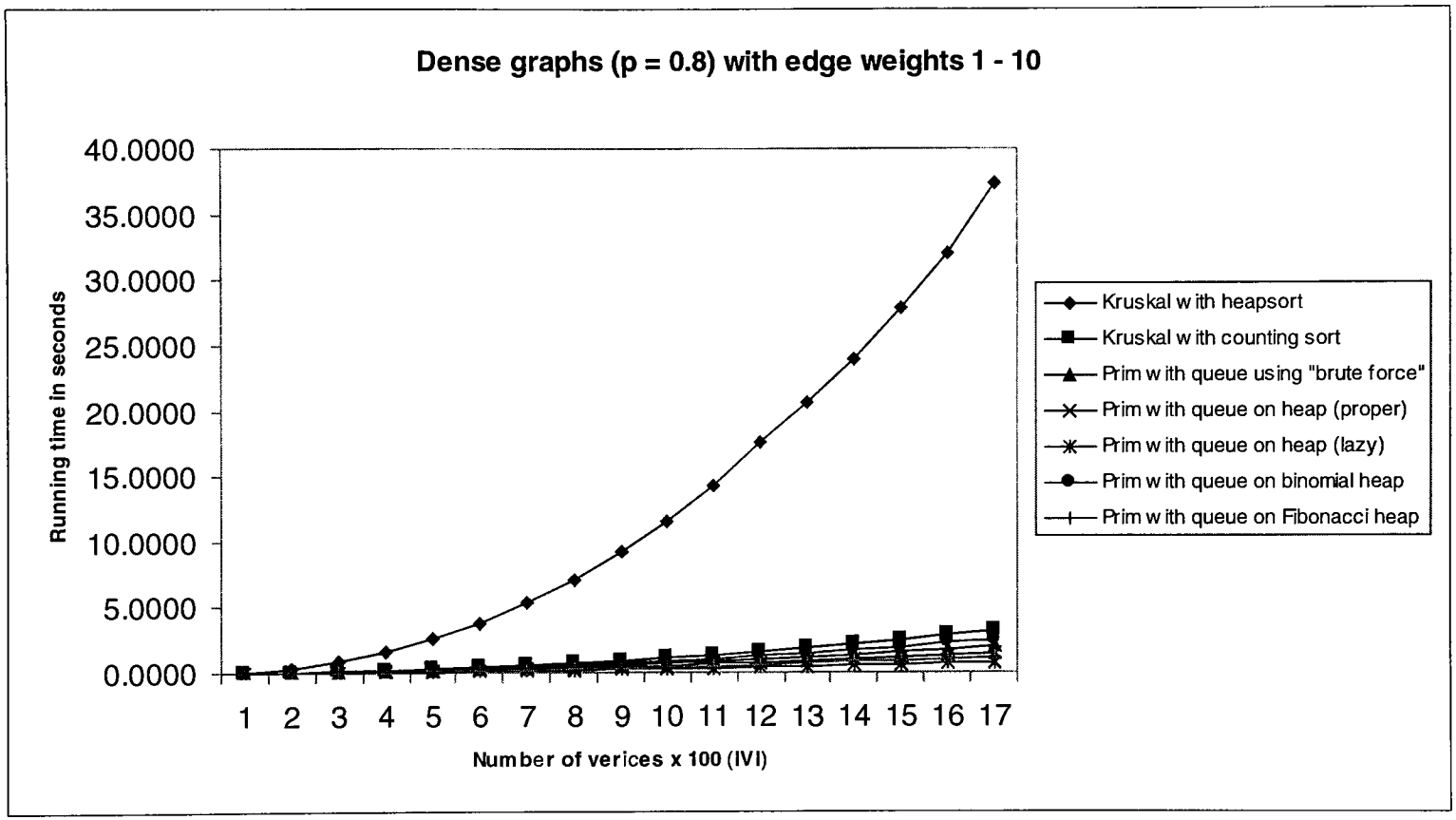

Figure 14 – Performances for dense graphs with $1 \leq w \leq 10$ (see Table 9)

Graph Density: 0.8 Range of Edge Weights: 1 – 100							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0738	0.0108	0.0069	0.0064	0.0048	0.0146	0.0076
200	0.3597	0.0430	0.0266	0.0229	0.0151	0.0465	0.0226
300	0.9298	0.0970	0.0621	0.0507	0.0311	0.0947	0.0451
400	1.6901	0.1811	0.1075	0.0859	0.5080	0.1659	0.0770
500	2.7794	0.2888	0.1684	0.1320	0.0759	0.2394	0.1143
600	4.2202	0.4176	0.2425	0.1871	0.1055	0.3455	0.1603
700	5.8406	0.5668	0.3339	0.2560	0.1428	0.4540	0.2162
800	7.8653	0.7396	0.4324	0.3297	0.1816	0.5950	0.2758
900	10.2333	0.9385	0.5476	0.4168	0.2278	0.7305	0.3441
1000	12.8628	1.1590	0.6759	0.5134	0.2783	0.8865	0.4253
1100	15.7993	1.4029	0.8196	0.6191	0.3356	1.0770	0.5096
1200	19.7110	1.6724	0.9767	0.7412	0.3994	1.2881	0.6162
1300	22.9228	1.9649	1.1531	0.8676	0.4682	1.5077	0.7033
1400	26.6568	2.2788	1.3379	1.0060	0.5401	1.7072	0.8096
1500	30.8877	2.6208	1.5322	1.1597	0.6193	1.9407	0.9321
1600	35.8009	2.9895	1.8616	1.3248	0.7103	2.2323	1.0511
1700	41.5835	3.3666	1.9668	1.4885	0.7938	2.5511	1.2263

Table 10 – Performances for dense graphs with $1 \leq w \leq 100$, with times in CPU seconds

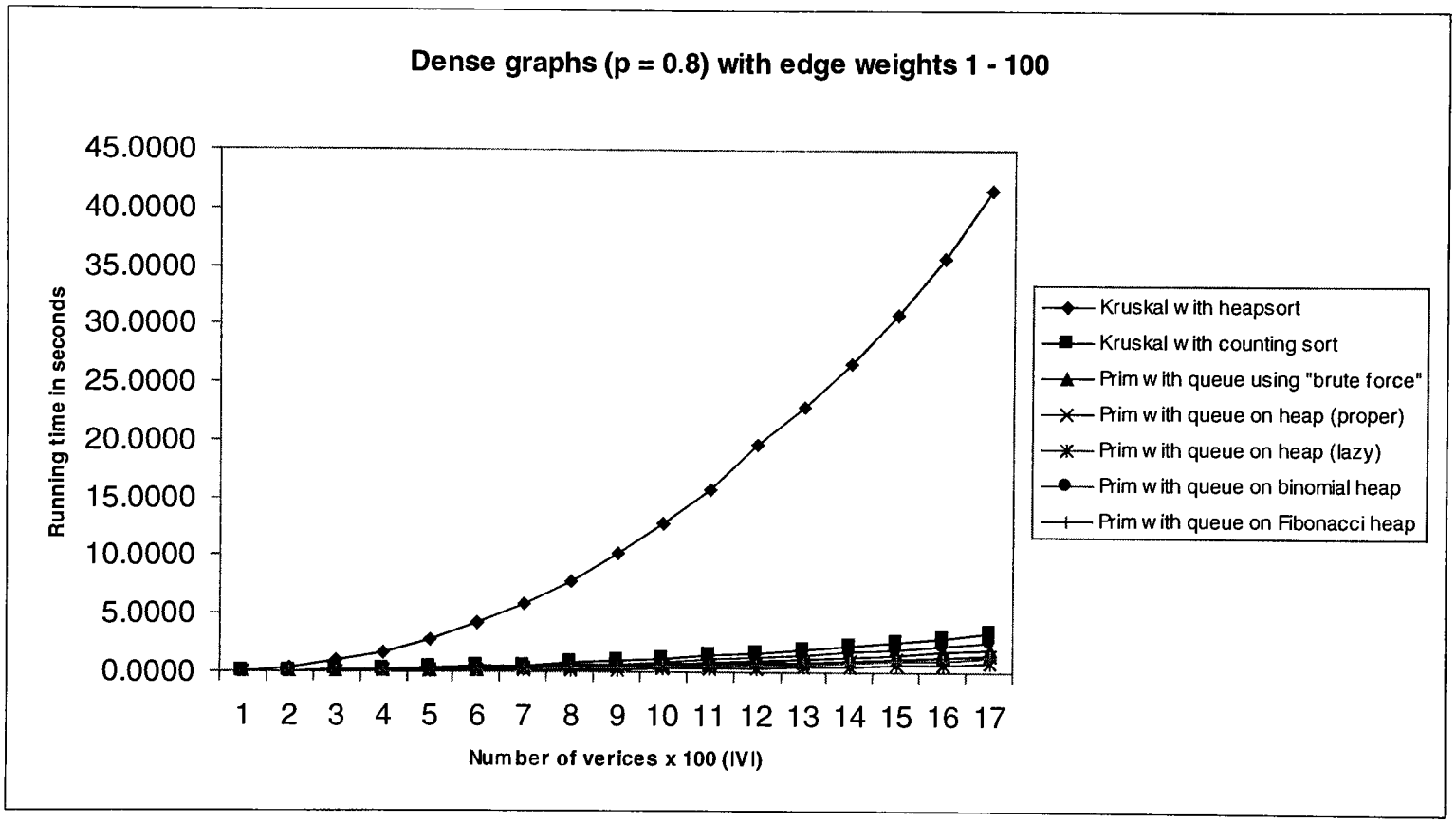

Figure 15 – Performances for dense graphs with $1 \leq w \leq 100$ (see Table 10) , with times in CPU seconds

Graph Density: 0.8 Range of Edge Weights: 1020 – 1022							
Number of Vertices	**khs**	**kcs**	**pbf**	**php_a**	**php**	**pbinh**	**pfibh**
100	0.0525	0.0562	0.0071	0.0054	0.0032	0.0138	0.0065
200	0.2452	0.2170	0.0277	0.0209	0.0113	0.0436	0.0198
300	0.6237	0.4881	0.0639	0.0473	0.0252	0.0914	0.0403
400	1.1482	0.8768	0.1101	0.0815	0.0436	0.1518	0.0694
500	1.9403	1.3727	0.1718	0.1269	0.0674	0.2324	0.1053
600	2.7999	1.9819	0.2475	0.1824	0.0963	0.3381	0.1475
700	3.9653	2.7812	0.3394	0.2494	0.1316	0.4442	0.2016
800	5.2624	3.5189	0.4387	0.3239	0.1702	0.5688	0.2595
900	6.8018	4.4565	0.5551	0.4101	0.2154	0.7154	0.3272
1000	8.6712	5.5038	0.6849	0.5059	0.2651	0.8739	0.4026
1100	10.5708	6.6648	0.8303	0.6134	0.3209	1.0603	0.4818
1200	12.9411	7.9342	0.9886	0.7350	0.3829	1.2528	0.5730
1300	15.0497	9.3148	1.1636	0.8613	0.4504	1.4629	0.6703
1400	17.9377	10.8058	1.3494	0.9968	0.5216	1.7059	0.7770
1500	20.7476	12.4265	1.5450	1.1484	0.6003	1.9234	0.8810
1600	23.6403	14.2014	1.7680	1.3169	0.6853	2.1915	1.0054
1700	27.3645	15.9829	2.0573	1.4789	0.7751	2.4779	1.1341

Table 11 – Performances for dense graphs with $1020 \leq w \leq 1022$, with times in CPU seconds

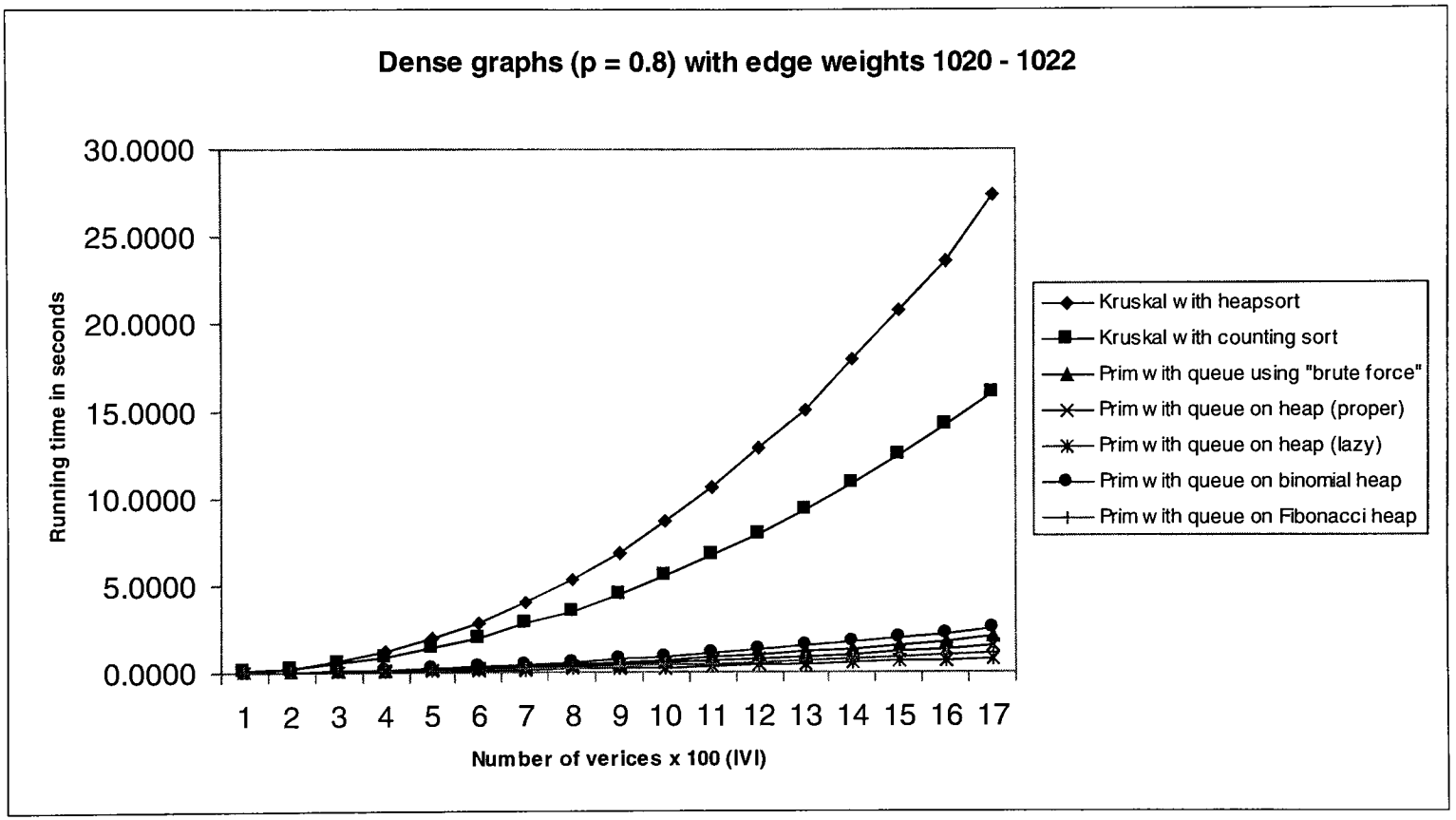

Figure 16 – Performances for dense graphs with $1020 \leq w \leq 1022$ (see Table 11)

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.

Graph Density: 0.2 Range of Edge Weights: 1020 – 1022			
Number of Vertices	**khs** (Heapsort)	**kcs** (Counting Sort)	**sort not timed**
100	0.0525	0.0562	**0.0026**
200	0.2452	0.2170	**0.0094**
300	0.6237	0.4881	**0.0210**
400	1.1482	0.8768	**0.0350**
500	1.9403	1.3727	**0.0570**
600	2.7999	1.9819	**0.0818**
700	3.9653	2.7812	**0.1124**

Table 12 – Performance of Kruskal's Algorithm excluding the sorting time

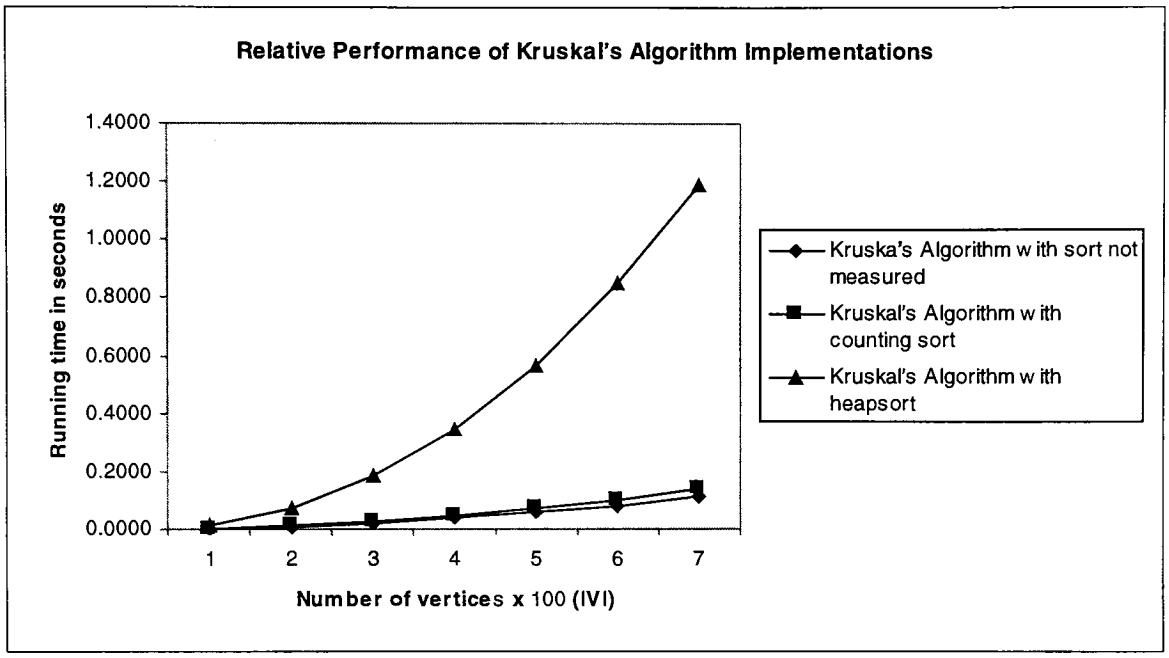

Figure 17 – Performance of Kruskal's Algorithm excluding the sorting time (see Table 12)

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 Kruskl's algorithm performing the disjoint-set operations and the time spent sorting. We arrive at the results by dividing time spent soring by the time spent performing disjoint set operations $R = \frac{T - T_S}{T_S}$.

Graph Density: 0.2 Range of Edge Weights: 1020 – 1022		
Number of Vertices	R_{HS}	R_{CS}
100	19.1923	20.6154
200	25.0851	22.0851
300	28.7000	22.2429
400	31.8057	24.0514
500	33.0404	23.0825
600	33.2286	23.2286
700	34.2785	23.7438

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.

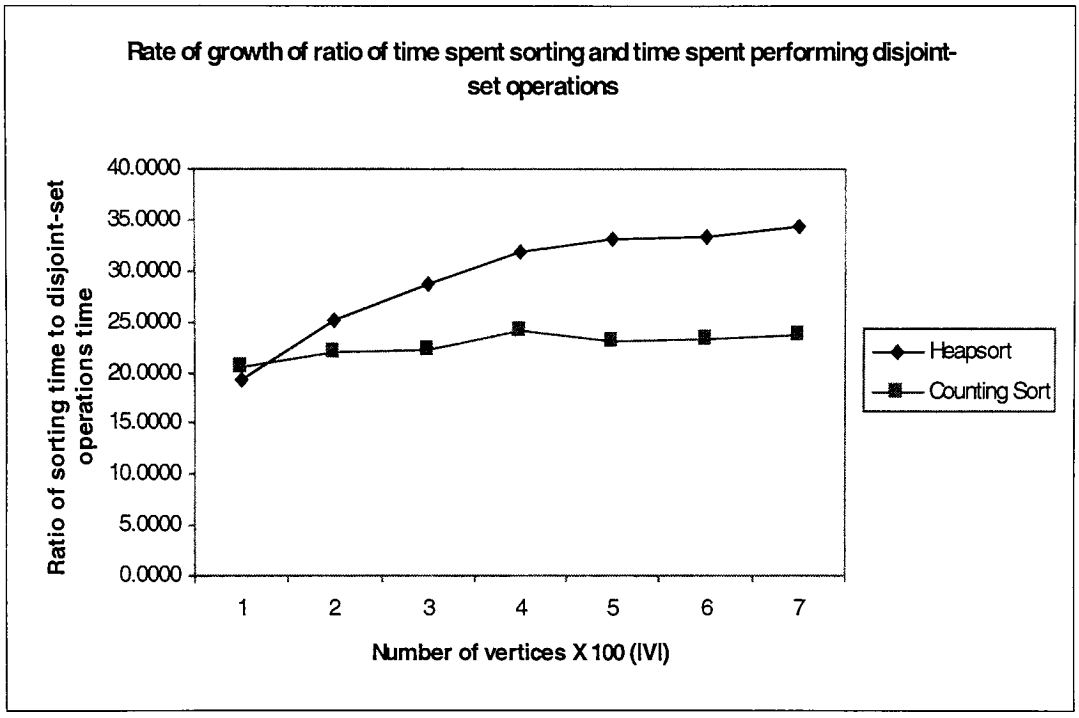

Figure 18 – Growth ratio of two implementations of Kruskal's algorithm

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 $O(E \lg E)$, 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.

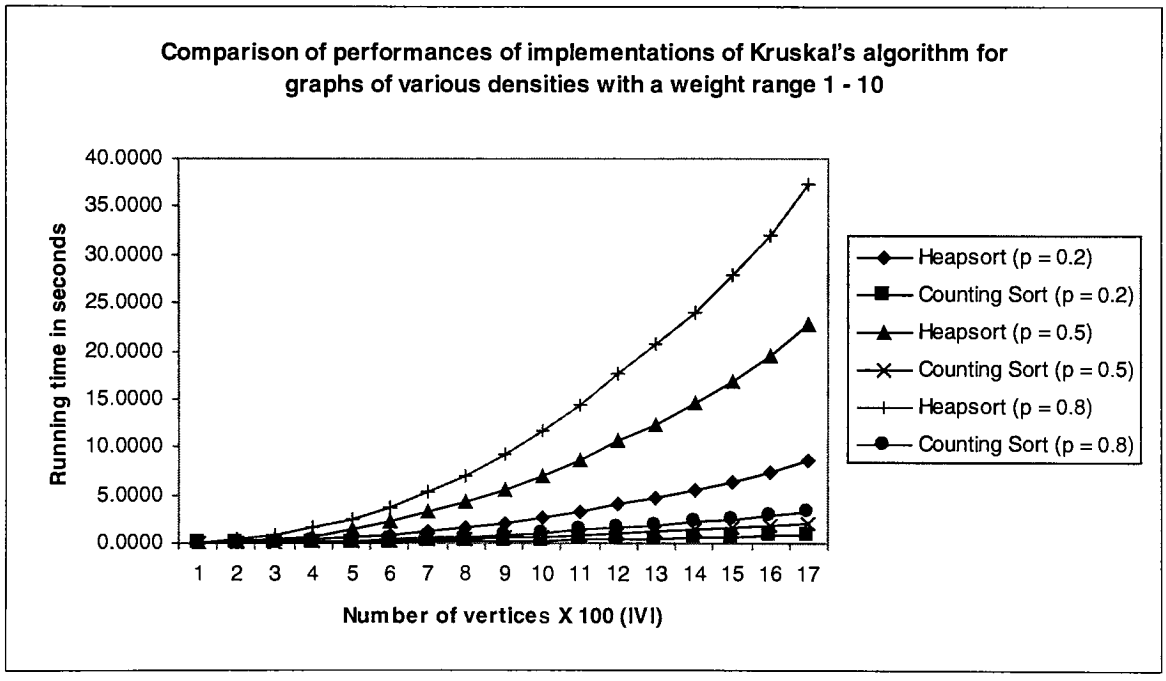

Figure 19 – Effect of density on performance of Kruskal's algorithm for graphs with $1 \leq w \leq 10$.

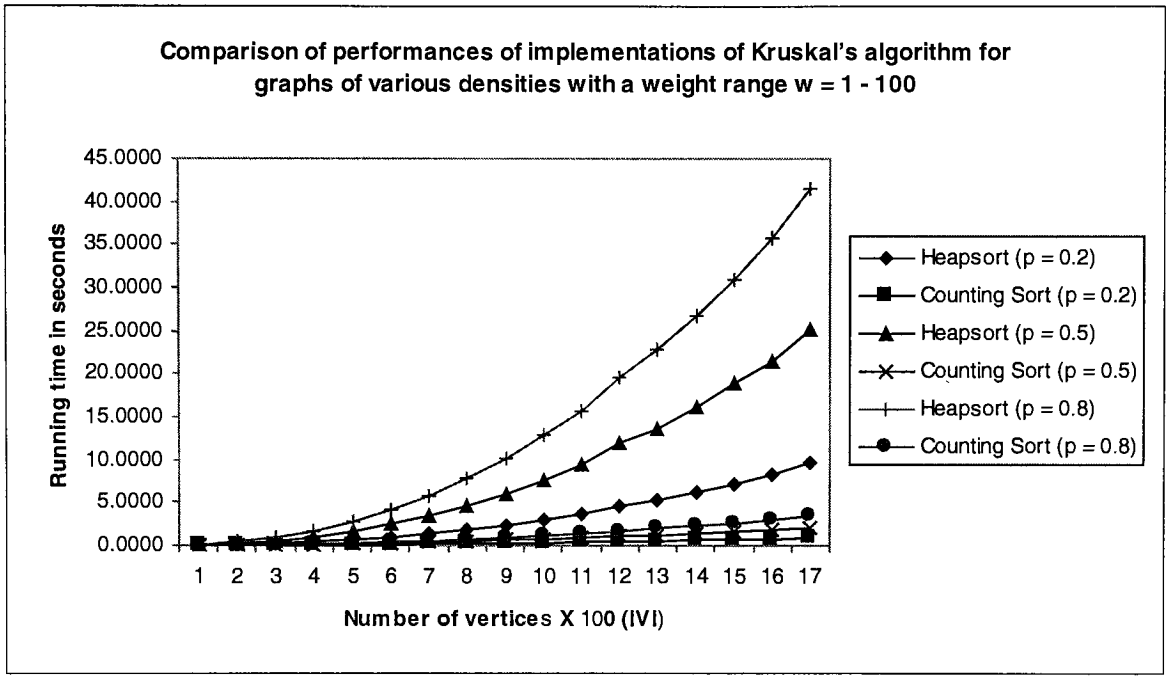

Figure 20 – Effect of density on performance of Kruskal's algorithm for graphs with $1 \leq w \leq 100$.

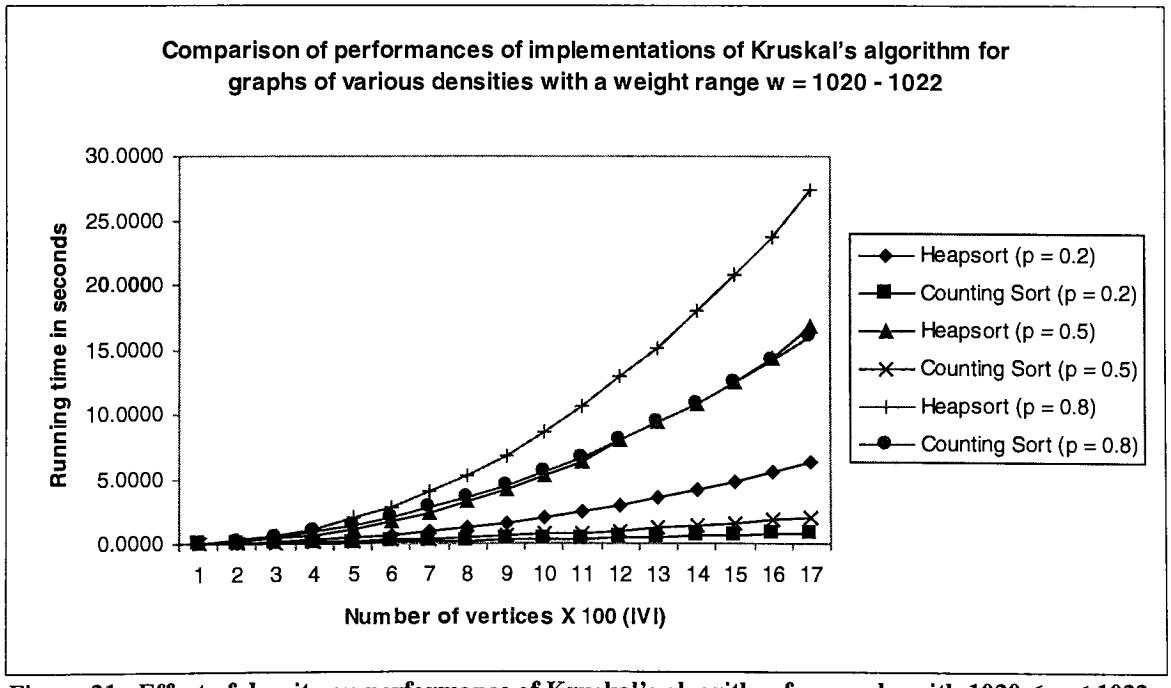

Figure 21– Effect of density on performance of Kruskal's algorithm for graphs with $1020 \leq w \leq 1022$.

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 $O(EV) = O(V^3)$ 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 ε.

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:

V	$p = 0.2$				$p = 0.5$				$p = 0.8$			
	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}
100	1.1379	0.6552	2.4310	1.1379	1.1897	0.6552	2.3966	1.1552	1.1864	0.7500	2.2813	1.1875
200	1.2585	0.5707	2.1073	1.0341	1.3043	0.5749	2.0821	0.9662	1.3190	0.6594	2.0306	0.9869
300	1.3106	0.5308	1.9648	0.8546	1.3333	0.5449	1.9295	0.8675	1.3396	0.6134	1.8679	0.8895
400	1.2365	0.5498	1.7815	0.7875	1.3518	0.5435	1.8827	0.8613	1.3297	0.8625	2.8166	0.8964
500	1.3501	0.5117	1.8543	0.8065	1.3593	0.5328	1.8345	0.8264	1.3459	0.5750	1.8136	0.8659
600	1.3555	0.5061	1.8506	0.7811	1.3570	0.5242	1.8255	0.8097	1.3529	0.5639	1.8466	0.8568
700	1.3673	0.4989	1.7998	0.7595	1.3715	0.5202	1.7957	0.7924	1.3606	0.5578	1.7734	0.8445
800	1.3668	0.4955	1.7784	0.7572	1.3665	0.5198	1.7658	0.7936	1.3542	0.5508	1.8047	0.8365
900	1.3704	0.4930	1.7611	0.7490	1.3668	0.5175	1.8629	0.7846	1.3536	0.5465	1.7526	0.8256
1000	1.3723	0.4919	1.7962	0.7496	1.3654	0.5167	1.7363	0.7935	1.3546	0.5421	1.7267	0.8284
1100	1.3738	0.4898	1.7364	0.7377	1.3640	0.5148	1.7307	0.7770	1.2477	0.5421	1.7396	0.8231
1200	1.3669	0.4857	1.7141	0.7304	1.3596	0.5113	1.7074	0.7705	1.3496	0.5389	1.7379	0.8314
1300	1.3768	0.4882	1.7110	0.7301	1.3454	0.5050	1.6776	0.7595	1.3529	0.5396	1.7378	0.8106
1400	1.3767	0.4855	1.7057	0.7209	1.3726	0.5145	1.6954	0.7649	1.3557	0.5369	1.6970	0.8048
1500	1.3740	0.4791	1.6617	0.7072	1.3563	0.5104	1.6722	0.7585	1.3457	0.5340	1.6735	0.8037
1600	1.3594	0.4787	1.6786	0.7070	1.3568	0.5094	1.6621	0.7515	1.3434	0.5362	1.6850	0.7934
1700	1.3708	0.4818	1.6744	0.6866	1.3566	0.5098	1.6872	0.7514	1.3431	0.5333	1.7139	0.8238

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 $1 \le w \le 10$

V	$p = 0.2$				$p = 0.5$				$p = 0.8$			
	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}
100	1.0858	0.7049	2.2951	1.1148	1.0794	0.7460	2.8190	1.3851	1.0781	0.7500	2.2813	1.1675
200	1.1514	0.8284	2.0048	0.9312	1.1800	0.8533	1.9958	0.9844	1.1818	0.8594	2.0306	0.9869
300	1.1959	0.5835	1.8742	0.8392	1.2084	0.8032	1.8837	0.8737	1.2249	0.8134	1.8879	0.8895
400	1.2328	0.5711	1.8578	0.8223	1.2437	0.5868	1.8590	0.8734	1.2515	0.5914	1.9313	0.8984
500	1.2472	0.5818	1.8130	0.7948	1.2489	0.5720	1.8381	0.9204	1.2758	0.5750	1.8138	0.8659
600	1.2640	0.5478	1.8302	0.7870	1.2788	0.5554	1.8147	0.8228	1.2981	0.5639	1.8488	0.8568
700	1.2753	0.5383	1.7790	0.7854	1.2935	0.5483	1.7720	0.8035	1.3043	0.5578	1.7734	0.8445
800	1.2884	0.5307	1.7783	0.7888	1.3012	0.5425	1.7849	0.8012	1.3115	0.5508	1.8047	0.8385
900	1.2879	0.5255	1.8870	0.7584	1.3058	0.5348	1.7419	0.8143	1.3138	0.5485	1.7528	0.8258
1000	1.2988	0.5202	1.7551	0.7870	1.3149	0.5317	1.7375	0.7950	1.3185	0.5421	1.7287	0.8264
1100	1.3017	0.5133	1.7401	0.7444	1.3220	0.5265	1.7304	0.7857	1.3239	0.5421	1.7398	0.8231
1200	1.3082	0.5083	1.7278	0.7419	1.3137	0.5221	1.7071	0.7781	1.3177	0.5389	1.7379	0.8314
1300	1.3188	0.5095	1.7818	0.7411	1.3279	0.5237	1.7424	0.7783	1.3291	0.5398	1.7378	0.8108
1400	1.3295	0.5072	1.7275	0.7389	1.3321	0.5229	1.7001	0.7728	1.3299	0.5389	1.6970	0.8048
1500	1.3199	0.5019	1.7050	0.7278	1.3244	0.5202	1.6708	0.7851	1.3212	0.5340	1.6735	0.8037
1600	1.3227	0.4987	1.6950	0.7242	1.3288	0.5173	1.6828	0.7827	1.4052	0.5382	1.6850	0.7934
1700	1.3270	0.4978	1.7004	0.7285	1.3283	0.5199	1.6964	0.8042	1.3213	0.5333	1.7139	0.8238

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 $1 \leq w \leq 100$

V	$p = 0.2$				$p = 0.5$				$p = 0.8$			
	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}	R_{pbf}	R_{php}	R_{pbinh}	R_{pfibh}
100	1.2630	0.8036	2.5649	1.1867	1.3208	0.8036	2.5472	1.1867	1.3148	0.5928	2.5556	1.2037
200	1.3438	0.5231	2.1892	0.9365	1.3532	0.5423	2.1194	0.9502	1.3254	0.5407	2.0881	0.9474
300	1.3643	0.5088	1.9798	0.8281	1.5553	0.5208	1.9848	0.8480	1.3510	0.5328	1.9323	0.8520
400	1.3611	0.4980	1.8892	0.8083	1.3543	0.5183	1.8818	0.8254	1.3509	0.5350	1.8828	0.8515
500	1.3713	0.4982	1.8515	0.7882	1.3814	0.5182	1.8308	0.8104	1.3538	0.5311	1.8314	0.8298
600	1.3778	0.4941	1.8878	0.7733	1.3890	0.5147	1.8271	0.7959	1.3589	0.5280	1.8538	0.8087
700	1.3888	0.4905	1.8854	0.7513	1.3757	0.5124	1.7809	0.7800	1.3809	0.5277	1.7811	0.8083
800	1.3874	0.4888	1.7835	0.7508	1.3809	0.5118	1.8049	0.7778	1.3544	0.5255	1.7581	0.8012
900	1.3802	0.4848	1.7728	0.7349	1.4188	0.5090	1.7413	0.7887	1.3538	0.5252	1.7445	0.7979
1000	1.3808	0.4841	1.7489	0.7350	1.3864	0.5087	1.7275	0.7848	1.3538	0.5240	1.7274	0.7958
1100	1.3857	0.4841	1.7387	0.7318	1.3848	0.5078	1.7272	0.7770	1.3538	0.5231	1.7288	0.7855
1200	1.3799	0.4815	1.7188	0.7225	1.3598	0.5081	1.7219	0.7930	1.3450	0.5210	1.7045	0.7798
1300	1.3852	0.4827	1.8010	0.7234	1.3840	0.5088	1.7408	0.7531	1.3510	0.5229	1.8985	0.7782
1400	1.3899	0.4832	1.7099	0.7159	1.3888	0.5078	1.8834	0.7501	1.3537	0.5233	1.7114	0.7795
1500	1.3303	0.4874	1.8199	0.8874	1.3803	0.5057	1.8714	0.7549	1.3454	0.5227	1.8749	0.7872
1600	1.3757	0.4788	1.8884	0.7059	1.3550	0.5044	1.8870	0.7475	1.3425	0.5204	1.8841	0.7835
1700	1.3779	0.4798	1.8983	0.7107	1.3580	0.5084	1.8725	0.7488	1.3911	0.5241	1.8755	0.7889

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 $1020 \leq w \leq 1022$

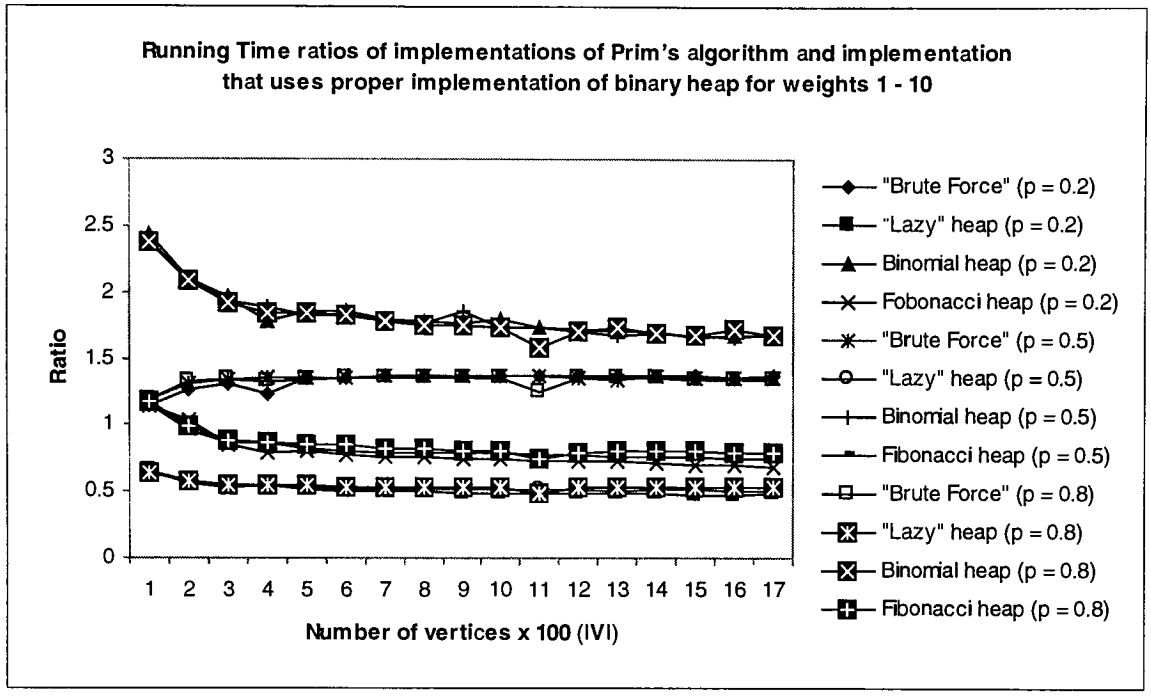

Figure 22 – Ratios obtained for all implementations of Prim's algorithm and the implementation that uses a proper implementation of binary heap (php_a) for $1 \leq w \leq 10$

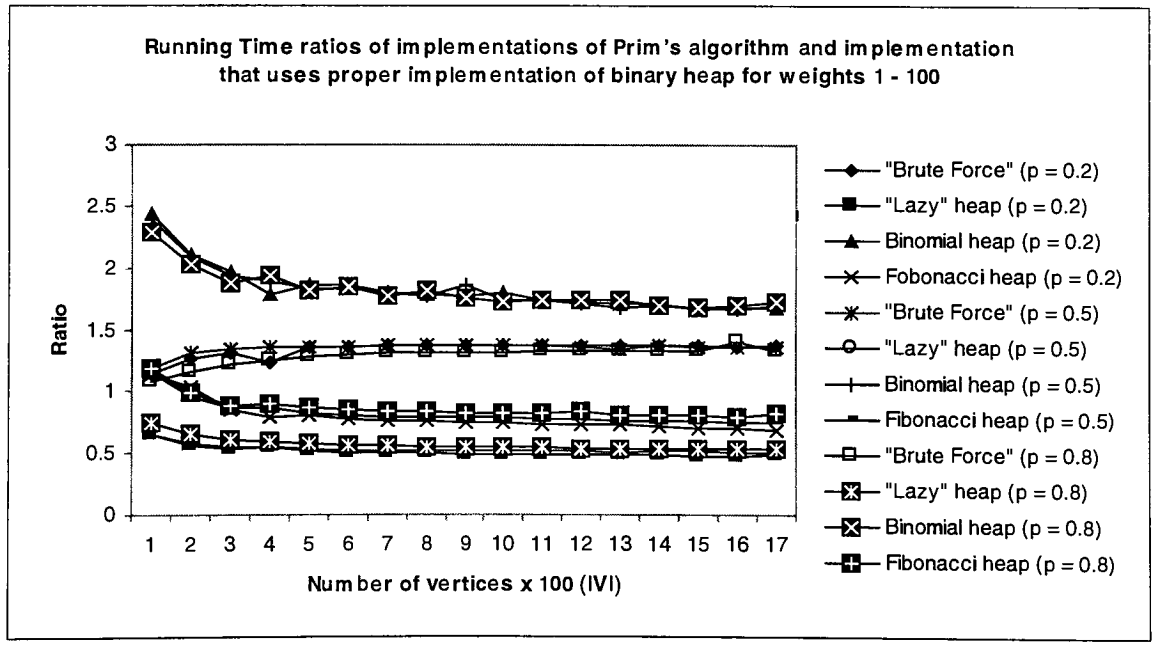

Figure 23 – Ratios obtained for all implementations of Prim's algorithm and the implementation that uses a proper implementation of binary heap (php_a) for $1 \leq w \leq 100$

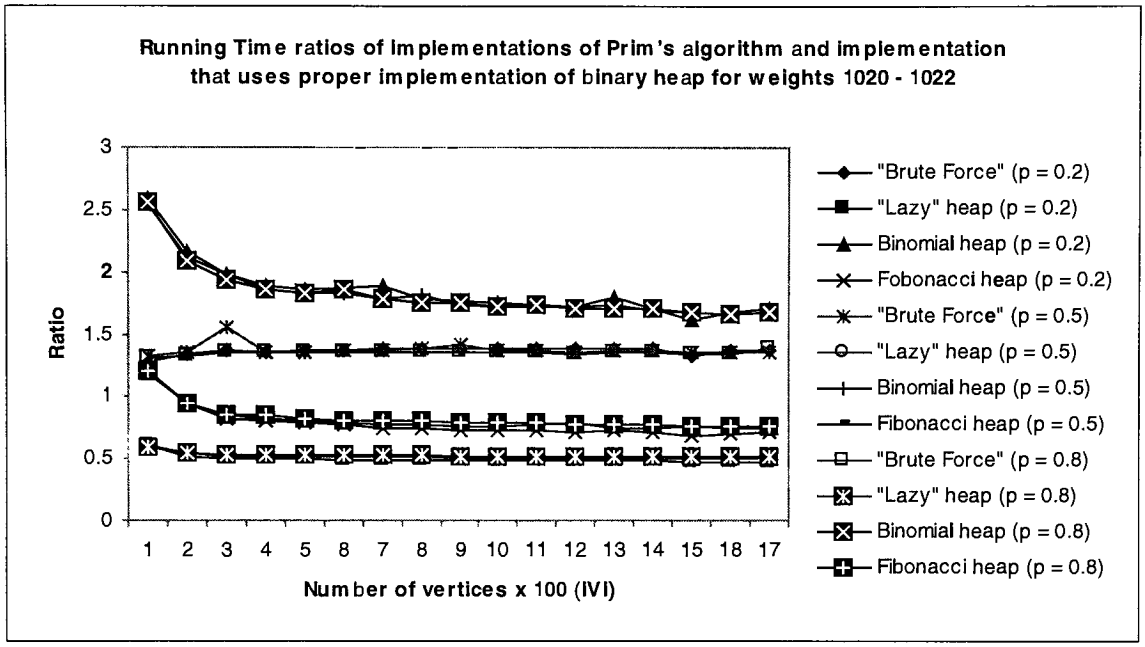

Figure 24 – Ratios obtained for all implementations of Prim's algorithm and the implementation that uses a proper implementation of binary heap (php_a) for $1020 \leq w \leq 1022$

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 ε. 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.

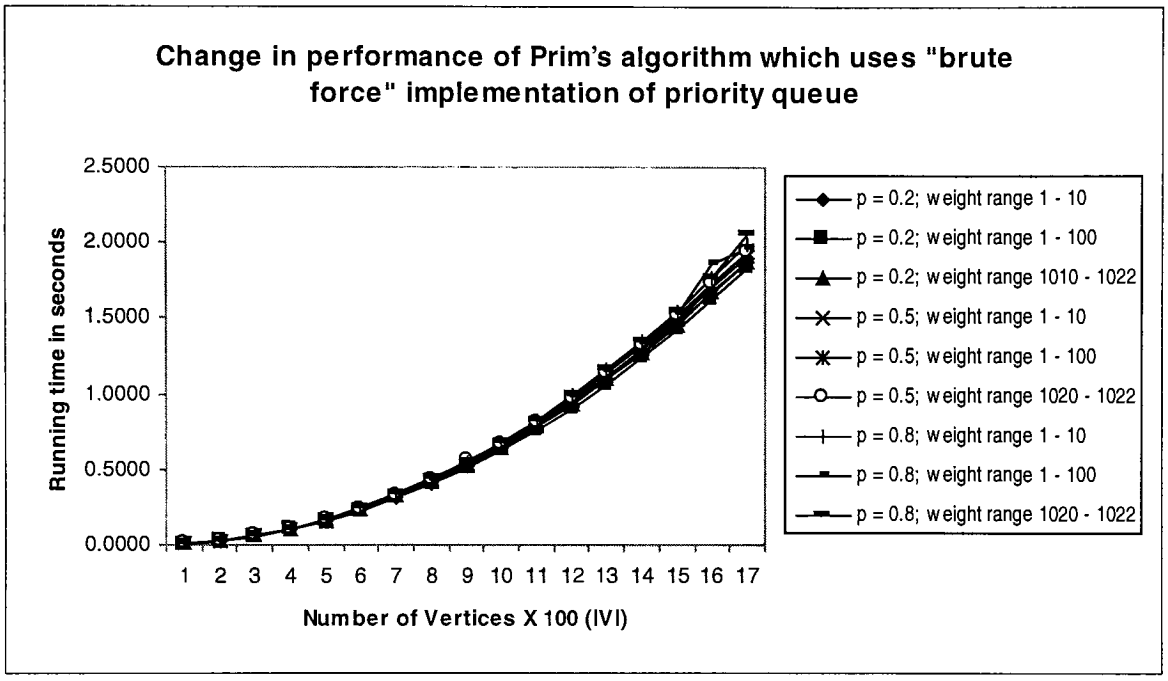

Figure 25 – Change in performance of "brute force" implementation of Prim's algorithm for different data sets.

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".

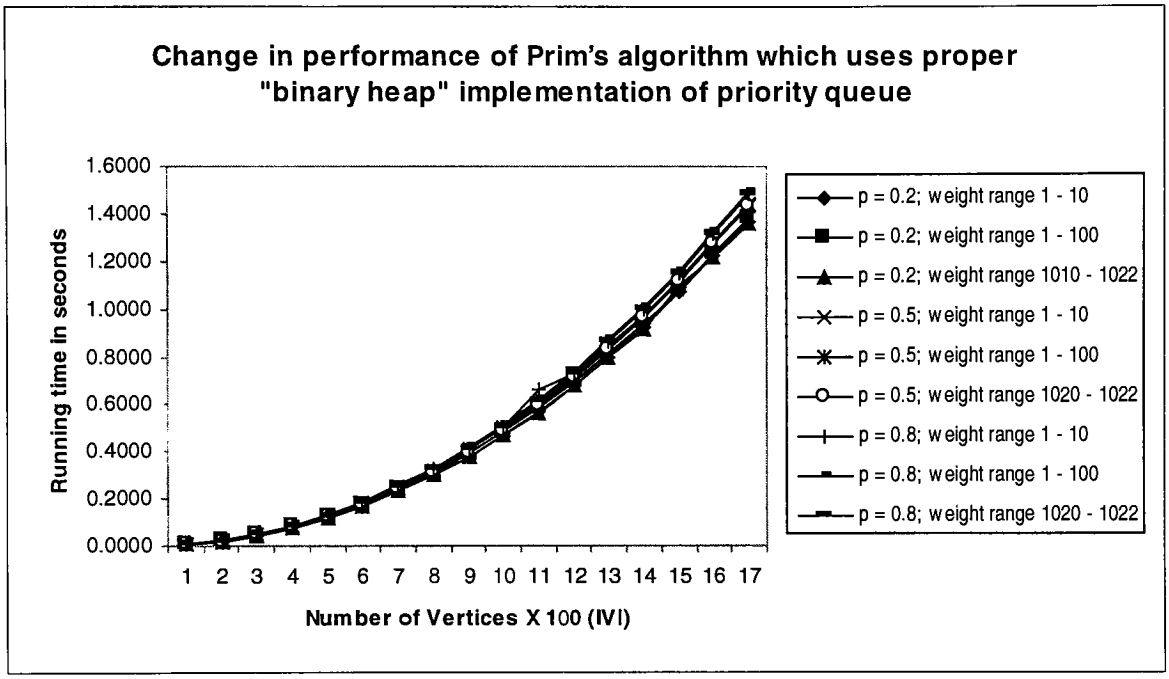

Figure 26 – Change in performance of "binary heap" implementation of Prim's algorithm for different data sets.

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 he 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 $0 < p \leq 0.2$.

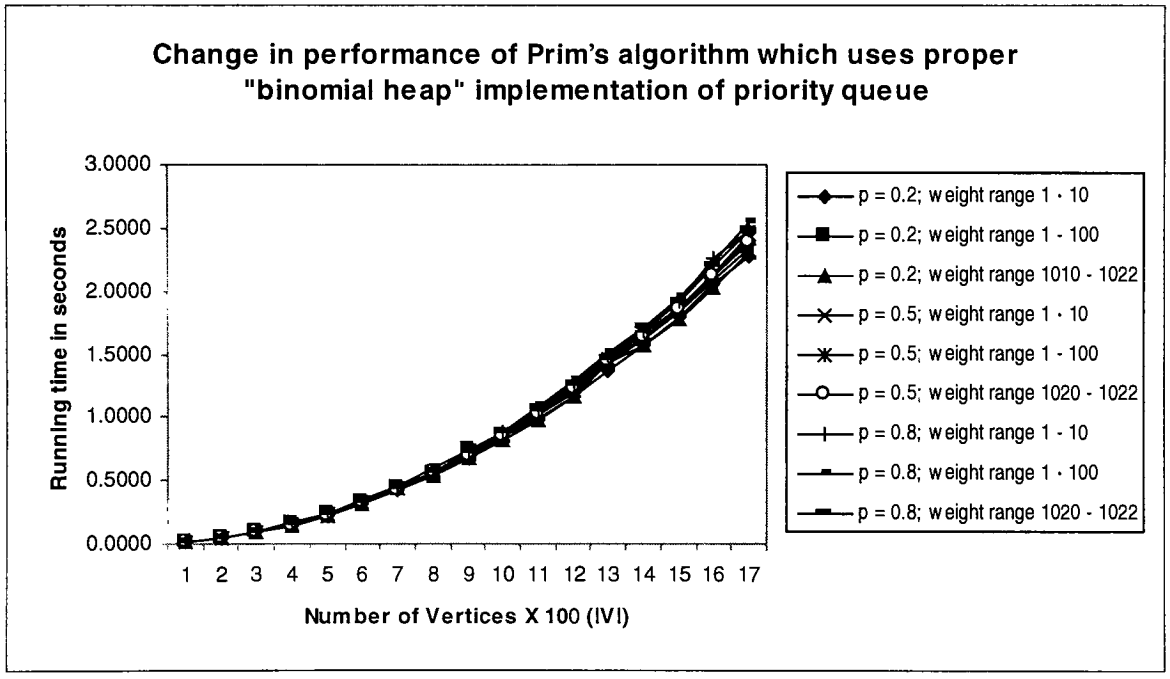

Figure 27 – Change in performance of "binomial heap" implementation of Prim's algorithm for different data sets.

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, ε becomes a significant factor in algorithm's performance. Hence, for low values of ε, 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. For very dense graphs, the performance of this implementation becomes almost quadratic, with respect to the number of edges. Unlike other implementations, the implementation of Prim's algorithm that uses a Fibonacci heap produces visible results for a relatively small number of vertices. 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.

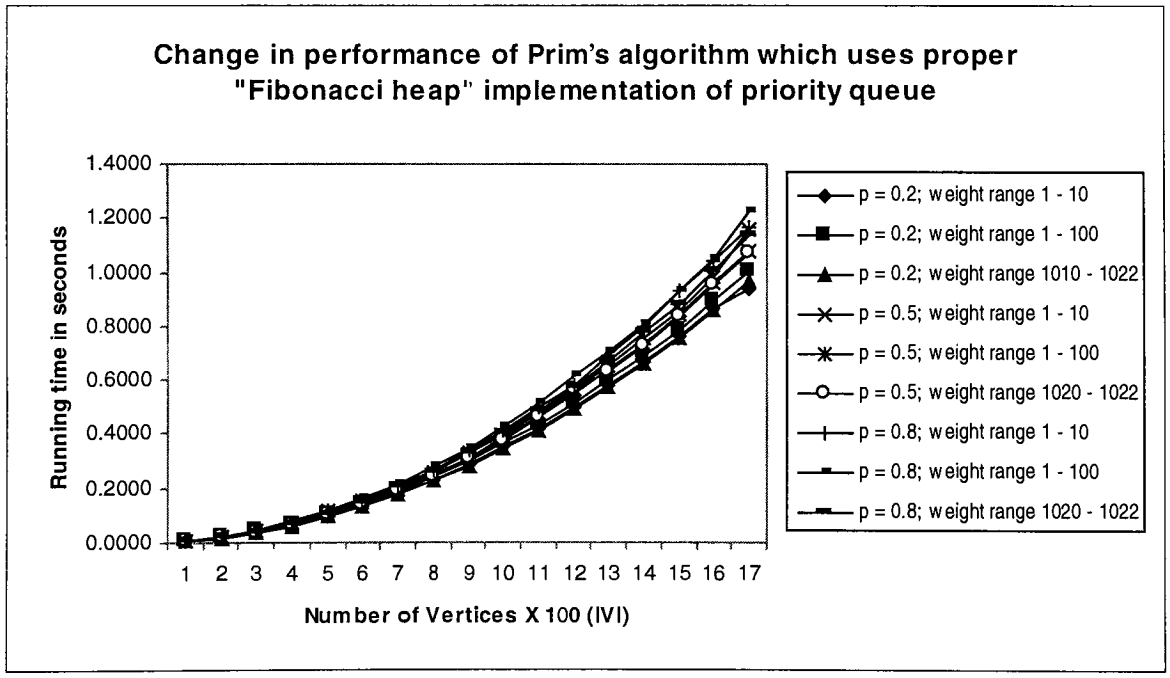

Figure 28 – Change in performance of "Fibonacci heap" implementation of Prim's algorithm for different data sets.

"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.

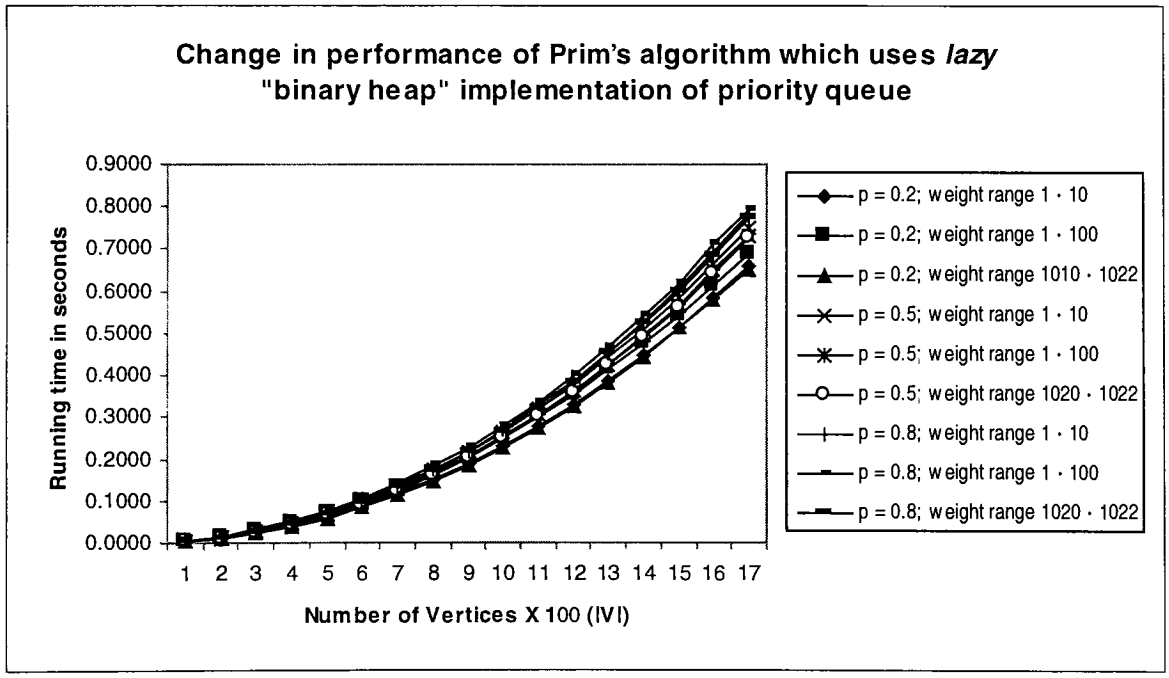

Figure 29 – Change in performance of "lazy heap" implementation of Prim's algorithm for different data sets.

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.

	$0 < p \leq 0.2$	$p \cong 0.5$	$0.8 \leq p < 1$
Narrow weight range	1. Prim with "lazy" heap 2. Kruskal with counting sort	1. Prim with "lazy" heap 2. Prim with Fibonacci heap	1. Prim with "lazy" heap 2. Prim with Fibonacci heap
Medium weight range	1. Prim with "lazy" heap 2. Kruskal with counting sort	1. Prim with "lazy" heap 2. Prim with Fibonacci heap	1. Prim with "lazy" heap 2. Prim with Fibonacci heap
High weight range	1. Prim with "lazy" heap 2. Kruskal with counting sort	1. Prim with "lazy" heap 2. Prim with Fibonacci heap	1. Prim with "lazy" heap 2. Prim with Fibonacci heap

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 lager 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

[1] R.L. Graham and Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43-57, 1985

[2] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48 – 50, 1956

[3] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1398 – 1401, 1957.

[4] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Progamming*. Addison-Wesley, 1968. Third Edition

[5] Gilles Brassard, Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[6] Seymour Lipshutz, Marc Lipson. *Theory and Problems Discrete Mathematics*. Shaum's Outlines. McGraw Hill, 1997.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990

[8] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké.Spdeknosti* 3: 37 – 58, 1926.

[9] D. Cheriton, R. E. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*. 5:724-742, 2976

[10] V.K. Balakrishnan. *Graph Theory*. Shaum's Outlines. MgGraw Hill. 1997. McGraw Hill, 1990

[11] Robert E. Tarjan, Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*. 31:245 – 281, 1984

[12] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Progamming*. Addison-Wesley, 1998. Second Edition

[13] Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306 – 318, 1985.

MST.mak


```
#
# This makefile created by makemake on Tue Mar 19 16:46:19 1996
#

#
# Definitions
#

.SUFFIXES:
.SUFFIXES:      .o .C .C~ $(SUFFIXES:.o %=%)

.C:
        $(LINK.cc) -o $@ $< $(LDLIBS)
.C.o:
        $(COMPILE.cc) $(OUTPUT_OPTION) $<
.C.a:
        $(COMPILE.cc) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
        $(RM) $%

CCFLAGS =       +w -g
CC =    cc
CFLAGS =        -g
LIBFLAGS =      -g -lm -lsunmath
#LIBFLAGS =     -g -lrwtool

CPFILES =
CFILES =        MST.c gengraph.c heap.c mtx.c primbf.c primhp.c krusk_un.c \
                krusk_bf.c BinHeap.c primbh.c FibHeap.c primfh.c krusk_cs.c
HFILES =        MST.h BinHeap.h FibHeap.h
SOURCEFILES =   $(HFILES) $(CPFILES) $(CFILES)
.precious:      $(SOURCEFILES)
OBJFILES =      heap.o mtx.o primbf.o primhp.o krusk_un.o krusk_bf.o BinHeap.o\
                primbh.o FibHeap.o primfh.o krusk_cs.c

#
# Main targets
#

all:    MST gengraph

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

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

#heaptest:      heaptest.o $(OBJFILES)
#       $(CC) $(CFLAGS) -o heaptest heaptest.o $(OBJFILES) $(LIBFLAGS)

#
# Dependencies
#

MST.o:          MST.h
gengraph.o:     MST.h
heap.o:         MST.h
#heaptest.o:    MST.h
mtx.o:          MST.h
primbf.o:       MST.h
primhp.o:       MST.h
krusk_un.o:     MST.h
krusk_cs.o:     MST.h
krusk_bf.o:     MST.h
primbh.o:       MST.h BinHeap.h
BinHeap.o:      MST.h BinHeap.h
primfh.o:       MST.h FibHeap.h
FibHeap.o:      MST.h FibHeap.h

#
# Housekeeping
#

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

clean:
        /bin/rm -rf $(OBJFILES) MST.o gengraph.o ptrepository Templates.DB .sb core
```

```
/*#define __DEBUG*/

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

/****
 **** Definitions for method specification
 ****/
#define KRUSKAL_INSTSRT    "kis"
#define KRUSKAL_HEAPSRT    "khs"
#define KRUSKAL_COUNTSORT  "kcs"
#define PRIM_BF            "pbf"
#define PRIM_HEAP          "php"
#define PRIM_HEAP_1        "php_a"
#define PRIM_BINHEAP       "pbinh"
#define PRIM_FIBHEAP       "pfibh"

/****
 **** Generic Definitions
 ****/
#define INFINITY          0x7fff
#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                  ***
 *****************************************************************************/

/****
 **** Priority Queue Itself
 ****/
typedef struct      _tagPriQueue_
        {
           int*     vertices;                    /* Array of vertieces */
           int      head;                        /* Head of the queue  */
           int      tail;                        /* Tail of the queue  */
           int      size;                        /* Total size of queue*/
           int      length;                      /* Total length of que*/
        } PBFQueue;

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

/****
 **** MACRO TO DETERMINE IF THE ABOVE QUEUE IS EMPTY
 ****/
#define IS_QUE_EMPTY(Q)    (((Q).length) <= 0)


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

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

typedef PBFQueue HeapStruct;                     /* Heap structure      */

/***
 *** Vertex list implementation to be used with the Binomial Heap
 ***/
typedef struct __tagVertList__
        {
           int       u;                          /* u values            */
           BOOL      bRemoved;
        } VertList, *PVertList;



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



/*****************************************************************************
 *** DEFINITONS FOR KRUSKAL'S ALGORITHM                                     ***
 *****************************************************************************/
typedef struct _tagEdges
        {
           int u;
           int v;
           int weight;
        } Edge, *EdgeList;


/*****************************************************************************
 *** CPU TIMINGS                                                           ***
 *****************************************************************************/
#include <sys/time.h>
#include <sys/resource.h>
extern int getrusage();
#define CPUDEFS struct rusage ruse;
#define CPUTIME (getrusage(RUSAGE_SELF,&ruse),\
  ruse.ru_utime.tv_sec + ruse.ru_stime.tv_sec + \
  1e-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.                           ***
 ***                                                                       ***
 *****************************************************************************
 *****************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sunmath.h>
#include "MST.h"


/*****************************************************************************/
/*** 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 __tagBinMode_
        {
            /***
             *** 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;
/*****************************************************************************/



typedef struct __tagCollection_
        {
           int                 nCapacity;
           int                 nTrees;
           int                 nCurrentSize;
           BinTree*            aTrees;
           int                 nMin;
           BOOL*               aInQueue;
           PBinNode*           apVertexNodes;
        } BinQueue, *PBinQueue;

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









/*****************************************************************************/
/*** Structure to provide the memory store of all nodes -- to avoid the    ***/
/*** dynamic allocation, which may be costly.                              ***/
/*****************************************************************************/
typedef struct __tagBinHeapMemMgt_
        {
           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_create(int size, NodeStore* pStore);
void 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 DEFINITINS                                                 ***/
/*****************************************************************************/
#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

#define __BINHEAP_H__
#endif
```

```
#ifndef __FIBHEAP_H__

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

#include <stdio.h>
#include <math.h>
#include <sunmath.h>
#include "MST.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 tree that contains a minimum node, and an array of   ***/
/*** 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 */
                                            /*  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                   Dn;        /* D(n)                         */

        } FibQueue, *PFibQueue;
/*****************************************************************************/




/*****************************************************************************/
/*** Structure to provide the memory store of all nodes -- to avoid the    ***/
/*** dynamic allocation, which may be costly.                              ***/
/*****************************************************************************/
typedef struct __tagFibHeapMemMgt_
        {
           FibNode*  aNodes;                /* Array of nodes              */
           PFibNode* apNodes;               /* Array of pointers           */
           unsigned  numNodes;              /* Total nodes                 */
           unsigned  top;                   /* top of the stack            */
        } FibNodeStore;
/*****************************************************************************/




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

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

PFibNode   FibHeap_concatLists(PFibNode pL1, PFibNode pL2);
PFibQueue  FibHeap_insert(PFibQueue h, PFibNode pNode);
PFibQueue  FibHeap_union(PFibQueue h1, PFibQueue h2);
PFibNode   FibHeap_extractMin(PFibQueue h);
PFibQueue  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);




#define __FIBHEAP_H__
#endif
```

```
/*****************************************************************************
 ***                                                                       ***
 *** PROGRAM:          gengraph                                            ***
 ***                                                                       ***
 *** WRITTEN BY:       Alec Berenbaum                                      ***
 ***                                                                       ***
 *** COURSE:           icsg800 - Theory of Algorithms                      ***
 ***                                                                       ***
 *** DATE:             Winter Quarter of 1995-1996                         ***
 ***                                                                       ***
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MST.h"

extern char* optarg;
extern int   optind;
double drabd48();
void   srand48();


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:         PrintUsageMessage                                   ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:  FILE* pOutFile - output file or stream to which the ***
 ***                                    message is printed                 ***
 ***                                                                       ***
 ***                  char* sProgName - name of the program               ***
 ***                                                                       ***
 ***                  char* sErrStr - error message string.  If this arg  ***
 ***                                  is NULL, error message is not prited***
 ***                                                                       ***
 *** RETURNS:          NOTHING                                             ***
 ***                                                                       ***
 *** DESCRIPTION:      This function is used by teh main program to print  ***
 ***                   the usage message to a specified stream when one or ***
 ***                   more of the command line arguments is not used      ***
 ***                   correctly.  The main purpose for this function is   ***
 ***                   to maintain the code readibility.                   ***
 ***                                                                       ***
 *****************************************************************************/
void PrintUsageMessage(FILE* pFile, char* sProgName, char* sErrStr)
{
   if (sErrStr != NULL)
      fprintf(pFile, "%s:  %s\n", sProgName, sErrStr);
   fprintf(pFile, "USAGE: %s %s %s %s %s %s\n", sProgName,
                  "-n <int>",
                  "-p <float>",
                  "-w <int>",
                  "-W <int>",
                  "[-f filename]\n");
}
/*****************************************************************************/



/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:         GenGraph                                            ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:  GraphRow* aGraph - graph buffer to be filled up by  ***
 ***                                      this function with connections   ***
 ***                                                                       ***
 ***                  int nVertices - number of vertices the graph is to  ***
 ***                                  have.                                ***
 ***                                                                       ***
 ***                  double dProb - probability of an edge being genera- ***
 ***                                 ted.                                  ***
 ***                                                                       ***
 ***                  int nMinWeight - minimum edge weight                 ***
 ***                                                                       ***
 ***                  int nMaxWeight - maximum edge weight                 ***
 ***                                                                       ***
 *** RETURNS:          NOTHING                                             ***
 ***                                                                       ***
 *** DESCRIPTION:      This function fills the graph with connetions based ***
 ***                   on the probability and the weight range passed via  ***
 ***                   the input arguments.                                ***
 ***                                                                       ***
 *****************************************************************************/
void GenGraph(GraphRow* pGraph, int nVertices, double dProb, int nMinWeight,
              int nMaxWeight)
{
   int i, j;                        /* Indeces into the graph matrix */
   double dNumGenerated;           /* Random number generated */
   int    nWeight;                 /* Weight generated        */
   int    nRandVal;

   /****
    **** Start Generationg
    ****/
   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:         main                                                ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:  int argc - number of command line arguments to the  ***
 ***                              program                                  ***
 ***                                                                       ***
 ***                  char** argv - array of pointers to command line      ***
 ***                                arguments                              ***
```

```
 ***                                                                    ***
 *** RETURNS:          NOTHING                                         ***
 ***                                                                    ***
 *** DESCRIPTION:      Main function of the program.  This function checks ***
 ***                   the command line arguments to make sure that the ***
 ***                   parameters specified for the graph generation are ***
 ***                   valid.  Then this function initiates the graph   ***
 ***                   generation sequence.                             ***
 ***                                                                    ***
 ************************************************************************/
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     dfProbability = -1.0;/* Probability of edge being generated     */
   int        nMinWeight = -1;      /* Minimum edge weight                     */
   int        nMaxWeight = -1;      /* Maximum edge weight                     */
   GraphRow*  aGraph = 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 (dfProbability > -1)
                {
                    PrintUsageMessage(stderr, argv[0],
                          "Illegal use of command line arguments");
                    if (sFileName != NULL) free(sFileName);
                    exit(1);
                }
                else
                    dfProbability = 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");
                    free(sFileName);
                    exit(1);
                ]
                else
                {
                    sFileName = (char *)malloc(strlen(optarg));
                    strcpy(sFileName, optarg);
                }
                break;
      default:  PrintUsageMessage(stderr, argv[0], "Illegal Argument used");
                if (sFileName != NULL) free(sFileName);
                exit(0);
   }
   /****
    **** Make sure that all required command line argumetns have been entered
    ****/
   if ((nVertices < 0) || (dfProbability < 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, &aGraph)) == 0)
   {
      fprintf(stderr, "%s:  Cannot allocate memory needed to create graph.\n",
                      argv[0]);
      if (sFileName != NULL) free(sFileName);
      exit(0);
   }
   else if (rc == -1)
   {
      fprintf(stderr, "%s:  Illegal number of vertices specified. %s.\n",
                     argv[0], "Please use a value n, such that [n >= 2]");
      if (sFileName != NULL) free(sFileName);
      exit(1);
   }

   /****
    **** We can now fill the graph with edges
    ****/
   GenGraph(aGraph, nVertices, dfProbability, nMinWeight, nMaxWeight);

   /****
    **** Save the graph to the file
    ****/
   if ((rc = SaveGraph(aGraph, nVertices, sFileName)) == 0)
   {
```

```
    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);
}
```

```
#include <stdio.h>
#include "MST.h"

/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:               CreateGraph                                  ***
 ***                                                                       ***
 *** WRITTEN BY:             Alec Berenbaum                               ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:        int nVertices - number of vertices in the    ***
 ***                                        graph                          ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:       int** pGraph - pointer to the array where    ***
 ***                                       the graph is stored             ***
 ***                                                                       ***
 *** RETURNS:                int - 1 = graph created succesfully          ***
 ***                               0 = couldn't allocate memory for graph***
 ***                              -1 = illegal value for nVertices         ***
 ***                                                                       ***
 *** DESCRIPTION:            This function creates an empty graph of the  ***
 ***                         number of vertices specified by nVertices     ***
 ***                         argument.                                     ***
 ***                                                                       ***
 *****************************************************************************/
int CreateGraph(int nVertices, GraphRow** pGraph)
{
   int i, j;                          /* Indeces into the raph 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)
       {
         for (j = 0; j < i; j ++)
            if ((*pGraph)[j] != NULL) free (*pGraph);
         free(*pGraph);
         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;
}
/*****************************************************************************/


/*****************************************************************************
```

```
 ***                                                                       ***
 *** FUNCTION:               DestroyGraph                                 ***
 ***                                                                       ***
 *** WRITTEN BY:             Alec Berenbaum                               ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:        int** pGraph - pointer to the array where    ***
 ***                                       the graph is stored             ***
 ***                                                                       ***
 ***                         int nVertices - number of vertices in the    ***
 ***                                        graph                          ***
 ***                                                                       ***
 *** RETURNS:                NOTHING                                      ***
 ***                                                                       ***
 *** DESCRIPTION:            This function destroys the graph by          ***
 ***                         deallocaating resources.                      ***
 ***                                                                       ***
 *****************************************************************************/
void DestroyGraph(GraphRow **pGraph, int nVertices)
{
   int i;                          /* Array Index */

   for (i = 0; i < nVertices; i++)
      free((*pGraph)[i]);
   free(*pGraph);
}
/*****************************************************************************/



/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:               SaveGraph                                    ***
 ***                                                                       ***
 *** WRITTEN BY:             Alec Berenbaum                               ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:        int nVertices - number of vertices in the    ***
 ***                                        graph                          ***
 ***                                                                       ***
 ***                         int** pGraph - pointer to the array where    ***
 ***                                       the graph is stored             ***
 ***                                                                       ***
 ***                         char* sFileName - name of the file to which  ***
 ***                                           the graph is to be written***
 ***                                   IF THIS VALUE IS NULL, stdout is    ***
 ***                                   USED.                               ***
 ***                                                                       ***
 *** RETURNS:                int - 1 = graph saved successfully           ***
 ***                               0 = couldn't open file                  ***
 ***                                                                       ***
 *** DESCRIPTION:            This function creates an empty graph of the  ***
 ***                         number of vertices specified by nVertices     ***
 ***                         argument.                                     ***
 ***                                                                       ***
 *****************************************************************************/
int SaveGraph(GraphRow* pGraph, int nVertices, char* sFileName)
{
   FILE*     pOutFile;             /* Output file handle */
   int       i, j;                 /* Row Indexes        */

   /****
    **** See if the stdout is to be used
    ****/
   if (sFileName == NULL)
      pOutFile = stdout;
```

```
   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 FALUE 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 succesfully                   ***
 ***                                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, GraphRow** pGraph, int* pnBig)
{
   int   i = 0, j;                  /* Graph Indeces */
   FILE* pFile;                     /* Pointer to the file to be read */
   int   stat;                      /* Status of file read            */
   int   rc;                        /* Return code   */

   /****
    **** See if the stdin is to be used for intput
    ****/
   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)[i][j] > *pnBig)
            *pnBig = (*pGraph)[i][j];
         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;
}
/*****************************************************************************/
```

```
/*****************************************************************************
 ***                                                                       ***
 *** PROGRAM:          mst                                                 ***
 ***                                                                       ***
 *** WRITTEN BY:       Alec Berenbaum                                      ***
 ***                                                                       ***
 *** COURSE:           icsg800 - Theory of Algorithms                      ***
 ***                                                                       ***
 ***                                                                       ***
 *****************************************************************************/

#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 <grpah file> -m <method> [-o <output file>]",
                    progName);
}

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

   int        nLargestWeight = 0;

   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);
                }
                sscanf(optarg, "%d", &nTimesToIterate);
                bGotI = 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);
                }
                sGraphFileName =(char*)malloc(strlen(optarg)*sizeof(char) + 1);
                strcpy(sGraphFileName, optarg);
                bGotG = 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);
                }
                sMethod = (char *)malloc(strlen(optarg) * sizeof(char) + 1);
                strcpy(sMethod, optarg);
                bGotM = 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(1);
                }
                sOutputFileName =(char*)malloc(strlen(optarg)*sizeof(char) + 1);
                strcpy(sOutputFileName, optarg);
                bGotO = 1;
                break;
      default:  PrintUsageMessage(stderr, argv[0]);
                if (sGraphFileName != NULL) free(sGraphFileName);
                if (sMethod != NULL) free(sMethod);
                if (sOutputFileName != NULL) free(sOutputFileName);
                exit(0);
   }
   /****
    **** Make sure that all required command line argumetns 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_HEAPSRT)    != 0) &&
       (strcmp(sMethod, KRUSKAL_COUNTSORT)  != 0) &&
       (strcmp(sMethod, PRIM_BF)            != 0) &&
```

```
      (strcmp(sMethod, PRIM_HEAP)          != 0) &&
      (strcmp(sMethod, PRIM_HEAP_1)        != 0) &&
      (strcmp(sMethod, PRIM_BINHEAP)       != 0) &&
      (strcmp(sMethod, PRIM_FIBHEAP)       != 0))
   {
      fprintf(stderr, "%s:  %s.\n         %s: [%s, %s, %s, %s, %s or %s]\n\n",
              argv[0], "Illegal MST method specified",   "Valid methods are"
              KRUSKAL_INSTSRT, KRUSKAL_HEAPSRT, PRIM_BF, PRIM_HEAP,
              PRIM_BINHEAP, PRIM_FIBHEAP) ;
      if (sGraphFileName != NULL) free(sGraphFileName);
      if (sMethod != NULL) free(sMethod);
      if (sOutputFileName != NULL) free(sOutputFileName);
      exit(1);
   }

   /****
    **** Create the graph and load it
    ****/
   if ((rc = ReadGraph(sGraphFileName, &nVertices, &aGraph, &nLargestWeight))
           == -2)

   {
      fprintf(stderr, "%s:  Couldn't allocate memory for graph.\n");
      if (sGraphFileName != NULL) free(sGraphFileName);
      if (sMethod != NULL) free(sMethod);
      if (sOutputFileName != NULL) free(sOutputFileName);
      exit(1);
   }
   else if (rc == -1)
   {
      fprintf(stderr, "%s:  Bad data or number of vertices.\n");
      if (sGraphFileName != NULL) free(sGraphFileName);
      if (sMethod != NULL) free(sMethod);
      if (sOutputFileName != NULL) free(sOutputFileName);
      exit(1);
   }
   else if (rc == 0)
   {
      fprintf(stderr, "%s:  Coudln't open file \"%s\" \n", sGraphFileName);
      if (sGraphFileName != NULL) free(sGraphFileName);
      if (sMethod != NULL) free(sMethod);
      if (sOutputFileName != NULL) free(sOutputFileName);
      exit(1);
   }

   /****
    **** We now have the graph -- Do MST
    ****/
   for (i = 0; i < nTimesToIterate; i++)
   {
/*
      printf("Iteration #%3d:    , i + 1);
*/
      CreateGraph(nVertices, &aOutGraph);
      if (strcmp(sMethod, PRIM_BF) == 0)
         nTotCost = pbf_mst(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, PRIM_HEAP) == 0)
         nTotCost = php_mst(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, KRUSKAL_HEAPSRT) == 0)
         nTotCost = kpc_mst(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, KRUSKAL_COUNTSORT) == 0)
         nTotCost = kcs_mst(aGraph, nVertices, aOutGraph, nLargestWeight,
                            &dfTime);
      else if (strcmp(sMethod, KRUSKAL_INSTSRT) == 0)
         nTotCost = kis_mst(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, PRIM_HEAP_1) == 0)
         nTotCost = php_mst_1(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, PRIM_BINHEAP) == 0)
         nTotCost = pbh_mst(aGraph, nVertices, aOutGraph, &dfTime);
      else if (strcmp(sMethod, PRIM_FIBHEAP) == 0)
         nTotCost = pfh_mst(aGraph, nVertices, aOutGraph, &dfTime);

      /***
       *** Accumulate time
       ***/
      dfAccTime += dfTime;

      if (i < nTimesToIterate - 1)
         DestroyGraph(&aOutGraph, nVertices);

      /***
       *** Print Results
       ***/
/*
      printf("Cost = %5d;  Time = %f seconds\n", nTotCost, dfTime);
*/
   }

   SaveGraph(aOutGraph, nVertices, sOutputFileName);


   /****
    **** Cleanup
    ****/
   DestroyGraph(&aGraph, nVertices);
   DestroyGraph(&aOutGraph, nVertices);
   if (sGraphFileName != NULL) free(sGraphFileName);
   if (sMethod != NULL) free(sMethod);
   if (sOutputFileName != NULL) free(sOutputFileName);
/*
   printf("%s %d %s %f %s\n", "The accumulated time to find the MST for",
                              nTimesToIterate,
                              (nTimesToIterate == 1) ? "iteration is"
                                                     "iterations is",
                              dfAccTime, "seconds.");
   printf("The average time is: %f\n", dfAccTime / (double)nTimesToIterate);
*/
   printf("%4d   %8.4f : %8.4f\n", nVertices, dfAccTime,
                                   dfAccTime / (double)nTimesToIterate);
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include "MST.h"

/*****************************************************************************
 ***  INSERTION SORT                                                       ***
 *****************************************************************************/

/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:              kis_insSort                                    ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:       EdgeList E - edges of the graph                ***
 ***                                                                       ***
 ***                        int nEdgeCount - number of edges in the list   ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:      Same as the input argument EdgeList E          ***
 ***                                                                       ***
 *** RETURNS:               NOTHING                                        ***
 ***                                                                       ***
 *** DESCRIPTION:           Sorts the edges in non-decreasing order by weigh***
 ***                        using the insertion sort.                      ***
 ***                                                                       ***
 *****************************************************************************/
void kis_insSort(EdgeList E, int nEdgeCount)
{
   int i, j;                    /* Array indeces                          */
   Edge   key;                  /* Entry against which comarisons made    */

   /****
    **** 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;
   }
}
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:              kis_findCycle                                  ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:       int nStartVert - vertex frow which to start DFS ***
 ***                                                                       ***
 ***                        int nPrevVert - Pi of the vertex that is used in***
 ***                                determining the cycle.                 ***
 ***                                                                       ***
 ***                        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 determin if the edge, of which it is ***
 ***                        part, forms a cycle if added to the graph.     ***
 ***                                                                       ***
 *****************************************************************************/
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 1;

   /****
    **** 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;
}
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:          kis_mst                                           ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:   GraphRow* pGraph - Adjacency matrix which represents ***
 ***                                      the input graph                  ***
 ***                                                                       ***
 ***                    int nVertices - number of vertices in the graph    ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS: GraphRow* pOutGraph - Adjacency matrix in which the ***
 ***                                        output graph is to be stored   ***
 ***                                                                       ***
 ***                    double* pdfTime - the trime it took to get mst     ***
 ***                                                                       ***
 *** RETURNS:           int - Total cost of the MST                       ***
 ***                                                                       ***
 *** DESCRIPTION:       This function finds the MST using Kruskal's algorithm***
 ***                    with insertion sort and without find-union algorithms***
```

```
 ***                                                                        ***
 ******************************************************************************/
int kis_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
            double* pdfTime)
{
   int  nCost = 0;                      /* Cost of the tree                 */
   int             u, v, k;             /* Vertices examinined at one time  */

   EdgeList        E;                   /* Edges of the graph               */
   int             nEdgeCount = 0;      /* Number of edges in the graph     */

   unsigned short  unStartTime;         /* Start time miliseconds part      */
   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                   */

   /*************************************************************************
    *** 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);
   tStartTime = tp.time;            /* Get seconds                      */
   unStartTime = tp.millitm;        /* Get milliseconds                 */

   /****
    **** Sort the edges of E by non-decreasing weight w
    ****/
   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++;
   }

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


   /****
    **** We now completed the MST - release the allocated aresources
    ****/
   free(E);

   return nCost;
}
/*****************************************************************************/
```

```
#include <stdio.h>
#include <sys/types.h>
/*#include <sys/time.h>*/
#include <sys/timeb.h>
#include "MST.h"


/*****************************************************************************
 ***  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: NONE                                               ***
 ***                                                                       ***
 *** RETURNS:           int - pointing value at the X                      ***
 ***                                                                       ***
 *** 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 eack 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]++;
   }
}
/*****************************************************************************/



/*****************************************************************************
 ***                                                                       ***
 *** 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 eack vertex         ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS: NONE                                               ***
 ***                                                                       ***
 *** RETURNS:           NOTHING                                            ***
 ***                                                                       ***
 *** DESCRIPTION:       Joins two vertice 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);
}



/*****************************************************************************
 ***  HEAPSORT IMPLEMENTATION                                              ***
 *****************************************************************************/

#define KR_LEFT(I)    (((I) * 2) + 1)
#define KR_RIGHT(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;
}



/*****************************************************************************
 ***                                                                       ***
 *** FUMCTION:          kr_heapify                                        ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:   EdgeList E - list of edges to be sorted using the ***
 ***                                heapsort                               ***
 ***                                                                       ***
 ***                    int i - index around which the heapify is to take  ***
 ***                            place.                                     ***
 ***                                                                       ***
 ***                    int nNumEdges - number of edges in the list        ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS: NONE                                               ***
 ***                                                                       ***
 *** RETURNS:          NOTHING                                           ***
 ***                                                                       ***
 *** DESCRIPTION:      Maintains the heap property of the list of edges.  ***
 ***                                                                       ***
 *****************************************************************************/
void 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*/

   /****
    **** Get the left and the right values
    ****/
   l = KR_LEFT(i);
   r = KR_RIGHT(i);

   /****
    **** Get the largest value
    ****/
   largest = ((l < nNumEdges) && (E[l].weight > E[i].weight)) ? l . i;
   if ((r < nNumEdges) && (E[r].weight > E[largest].weight))
      largest = r;

   /****
    **** See if the values need to be exchanged
    ****/
   if (largest != i)
   {
      edge_swap(&(E[largest]), &(E[i]));

      kr_heapify(E, largest, nNumEdges);
   }
}
/*****************************************************************************/


void  kr_build_heap(EdgeList E, int nEdgeCount)
{
   int i;

   for (i = (nEdgeCount / 2) - 1; i >= 0; i--)
      kr_heapify(E, i, nEdgeCount);
}




void kr_heapsort(EdgeList E, int nEdgeCount)
{
   int i;
   int size;

   kr_build_heap(E, nEdgeCount);
   size = nEdgeCount;
   for (i = (nEdgeCount - 1); i >= 1; i--)
   {
      edge_swap(&(E[0]), &(E[i]));
      kr_heapify(E, 0, --size);
   }
}


/*****************************************************************************
 ***  MST                                                                  ***
 *****************************************************************************/

/*****************************************************************************
 ***                                                                       ***
 *** FUMCTION:          kpc_mst                                           ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:   GraphRow* pGraph - Adjacency matrix which represents ***
 ***                                       the input graph                 ***
 ***                                                                       ***
 ***                    int nVertices - number of vertices in the graph    ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS: GraphRow* pOutGraph - Adjacency matrix in which the ***
 ***                                        output graph is to be stored   ***
 ***                                                                       ***
 ***                    double* pdfTime - the trime it took to get mst     ***
 ***                                                                       ***
 *** RETURNS:          int - Total cost of the MST                       ***
 ***                                                                       ***
 *** DESCRIPTION:      This function finds the MST using Kruskal's algorithm***
 ***                    with heapsort and path compression.               ***
 ***                                                                       ***
 *****************************************************************************/
int 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 examinined at one time   */

   EdgeList        E;               /* Edges of the graph                */
   int             nEdgeCount = 0;  /* Number of edges in the graph      */
```

```
   unsigned short  unStartTime;        /* Start time miliseconds part       */
   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          dfTmpTime;          /* Temporary time for intermediate   */
                                       /*   measurements                    */
   struct rusage   ruse;
   double          dfStartCPUTime;
   double          dfEndCPUTime;


   /****
    **** 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) 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);
   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
    ****/
   kr_heapsort(E, nEdgeCount);

   /****
    **** 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)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *pdfTime = dfEndCPUTime - dfStartCPUTime;

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

   return nCost;
}
/*****************************************************************************/
```

```
#include <stdio.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] += 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 elemetns less than or equal to i
    ***/
   for (j = nLength - 1; j >= 0; j--)
   {
      edge_copy(&B[C[A[j].weight - 1] - 1], &A[j]);
      C[A[j].weight - 1]--;
   }
}


/*****************************************************************************
 ***  MST                                                                  ***
 *****************************************************************************/

/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:          kcs_mst                                           ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:   GraphRow* pGraph - Adjacency matrix which represents ***
 ***                                      the input graph                  ***
 ***                                                                       ***
 ***                    int nVertices - number of vertices in the graph    ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:  GraphRow* pOutGraph - Adjacency matrix in which the ***
 ***                                         output graph is to be stored  ***
 ***                                                                       ***
 ***                    double* pdfTime - the trime it took to get mst     ***
 ***                                                                       ***
 *** RETURNS:           int - Total cost of the MST                       ***
 ***                                                                       ***
 *** DESCRIPTION:       This function finds the MST using Kruskal's algorithm***
 ***                    with heapsort and path compression.               ***
 ***                                                                       ***
 *****************************************************************************/
int kcs_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph, int nLargest,
            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 examinined at one time   */

   EdgeList         E;                  /* Edges of the graph                */
   EdgeList         Etmp;               /* Edges of the graph                */
   int              nEdgeCount = 0;     /* Number of edges in the graph      */

   unsigned short   unStartTime;        /* Start time miliseconds part       */
   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                    */

   struct rusage    ruse;
   double           dfStartCPUTime;
   double           dfEndCPUTime;


   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) nEdgeCount ++;
   E = (EdgeList)malloc(nEdgeCount * sizeof(Edge));
   Etmp = (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)
         {
            Etmp[k].u = u;
            Etmp[k].v = v;
            Etmp[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);
   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(Etmp, 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)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *pdfTime = dfEndCPUTime - dfStartCPUTime;

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

   return nCost;
}
/***************************************************************************/
```

```
#include "MST.h"


/*****************************************************************************
 ***  HEAP FUNCTIONS                                                       ***
 *****************************************************************************/


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:           heapify                                          ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:    HeapStruct* pHeap - pointer to the heap structure ***
 ***                                                                       ***
 ***                     int i - index for which the heap is heapified     ***
 ***                                                                       ***
 ***                     KeyArray key - array of keys for every vertex     ***
 ***                                    on the heap                        ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:   NONE                                             ***
 ***                                                                       ***
 *** RETURNS:            NOTHING                                          ***
 ***                                                                       ***
 *** DESCRIPTION:        This function ensures the heap property to be    ***
 ***                     maintained on the heap.                          ***
 ***                                                                       ***
 *****************************************************************************/
void heapify(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;
      /****
       **** Heapify around the smallest to propogate
       ****/
      heapify(pHeap, smallest, key);
   }
}
/*****************************************************************************/
```

```
/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:           heap_build                                       ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:    HeapStruct* pHeap - pointer to the heap structure ***
 ***                                                                       ***
 ***                     KeyArray key - array of keys for every vertex     ***
 ***                                    on the heap                        ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:   NONE                                             ***
 ***                                                                       ***
 *** RETURNS:            NOTHING                                          ***
 ***                                                                       ***
 *** DESCRIPTION:        This function accepts a HeapStruct and arranges  ***
 ***                     internal values such that the stuctue contains a ***
 ***                     heap.                                            ***
 ***                                                                       ***
 *****************************************************************************/
void heap_build(HeapStruct* pHeap, KeyArray key)
{
   int i;                       /* index into the heap */

   pHeap->size = pHeap->length;
   for (i = (pHeap->length / 2) - 1; i >= 0; i--)
      heapify(pHeap, i, 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                                             ***
 ***                                                                       ***
 *** RETURNS:            NOTHING                                          ***
 ***                                                                       ***
 *** 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 vallue array */
   int tmp;                     /* temporary variable used in swapping values */

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

   for (i = (pHeap->length - 1); i >= 1; i--)
   {
      tmp = pHeap->vertices[0];
      pHeap->vertices[0] = pHeap->vertices[i];
      pHeap->vertices[i] = tmp;
      (pHeap->size)--;
      heapify(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;
}
/*****************************************************************************/
```

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


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  pbf_alloc_data_structs                     ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           int nVerteces - 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 Prims         ***
 ***                           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)
{
   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:                  pbf_free_data_structs                      ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           NONE                                       ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:          PBFQueue** pQ - structure for priority     ***
 ***                                           queue                       ***
 ***                                                                       ***
 ***                           KeyArray* pKey - array of keys              ***
 ***                                                                       ***
 ***                           PIArray* pPi - array of PI values           ***
 ***                                                                       ***
 *** RETURNS:                  NOTHING                                    ***
 ***                                                                       ***
 *** DESCRIPTION:              This function releases resources used by   ***
 ***                           the datastructures in the Prim's            ***
 ***                           algorithm.                                  ***
 ***                                                                       ***
 *****************************************************************************/
void pbf_free_data_structs(PBFQueue** pQ, KeyArray* pKey, PIArray* pPi)
{
   /****
    **** Release queue resources
    ****/
   free((*pQ)->vertices);
   free(*pQ);
   *pQ = NULL;

   /****
    **** Release PI and Key resources
    ****/
   free(*pKey);
   free(*pPi);
   *pKey = NULL;
   *pPi = NULL;
}
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** 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 que***
 ***                                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)
{
   int i = 0;                      /* Index into the queue */
```

```
   /****
    **** Search exaustively until the vertex is either found or not
    ****/
   if (Q->size <= 0)
      return 0;
   while ((i < Q->length) && (vertex != Q->vertices[i])) i++;

   /****
    **** See what stopped the loop and return accordingly
    ****/
   if (i >= Q->length)
      return 0;
   return (Q->vertices[i] == vertex);
}
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                    pbf_extract_min                         ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:             PBFQueue* Q - queue from which minimum is ***
 ***                                           extracted                   ***
 ***                                                                       ***
 ***                             KeyArray key - array which stores the key ***
 ***                                            values for every vetex     ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:            NONE                                    ***
 ***                                                                       ***
 *** RETURNS:                     int - The actual vertex extracted       ***
 ***                                   NIL if the queue is empty           ***
 ***                                                                       ***
 *** DESCRIPTION:                 This function extracts the vertex with the***
 ***                              minimal key value from the queue using the***
 ***                              BURTE FORCE.  The queue is searched in its***
 ***                              entirety, and the smalles value is      ***
 ***                              returned.                               ***
 ***                                                                       ***
 *****************************************************************************/
int pbf_extract_min(PBFQueue* Q, KeyArray key)
{
   int i, j = NIL;              /* Index into the queue */
   int min = INFINITY;          /* Minimal value        */

   /****
    **** Make sure that the queue is not empty.  If it is, return a -1
    ****/
   if (Q->size <= 0)
      return NIL;

   /****
    **** Now that we know that the queue is not empty, start search
    ****/
   for (i = 0; i < Q->length; i++)
      if ((Q->vertices[i] != NIL) && (key[Q->vertices[i]] <= min))
      {
         min = key[Q->vertices[i]];
         j = i;
      }

   /****
    **** We have now found the minimal vertex.  Extract it
    ****/
   if (j != NIL)
   {
      min = Q->vertices[j];
      Q->vertices[j] = NIL;
      Q->size--;
      return min;
   }
   return NIL;
}
/*****************************************************************************/



/*****************************************************************************
 *** MST                                                                   ***
 *****************************************************************************/


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                    pbf_mst                                 ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:             GraphRow* pGraph - input graph          ***
 ***                                                                       ***
 ***                             int nVertices - number of vertices      ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:            GraphRow* pOutGraph - output graph      ***
 ***                                                                       ***
 ***                             double* pdfTime - time it took to get mst ***
 ***                                                                       ***
 *** RETURNS:                     int - Total cost of the MST             ***
 ***                                                                       ***
 *** DESCRIPTION:                 This function finds a MST in the graph  ***
 ***                              supplied via the pGraph argument.  The  ***
 ***                              Prim's alogorithm is used using the BRUTE ***
 ***                              force to extract the minimum vertex from  ***
 ***                              the priority queue.                     ***
 ***                                                                       ***
 *****************************************************************************/
int pbf_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
            double* pdfTime)
{
   PBFQueue*       Q;                  /* Priority Queue                   */
   KeyArray        key;                /* Keys                             */
   PIArray         Pi;                 /* PI's                             */
   int             u, v, k;            /* Vertices examinined at one time  */
   int             nCost = 0;          /* Cost of the tree                 */
int i, j;
   unsigned short  unStartTime;        /* Start time miliseconds part      */
   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                   */

   struct rusage   ruse;
   double          dfStartCPUTime;
   double          dfEndCPUTime;

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

```
   /****
    **** Take start time reading
    ****/
/*
   ftime(&tp);
   tStartTime = tp.time;
   unStartTime = tp.millitm;
*/
   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 ov 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
    ****                                                    to NIL
    ****/
   u = Q->vertices[0];
   key[u] = 0;
   Pi[u] = NIL;


   /****
    **** Begin iteration over the queue looking for edges
    ****/
   while (Q->size > 0)
   {
      /****
       **** If there is a retained edge, add it to the graph
       ****/
      u = pbf_extract_min(Q, key);

      /****
       **** Iterate over all vertices adjacent to u
       ****/
      for (k = 0; k < Q->length; k++)
      {
         if (Q->vertices[k] != NIL)
         {
            v = Q->vertices[k];
            if ((u != v) && (pGraph[u][v] != 0) && (pGraph[u][v] < key[v]))
            {
               Pi[v] = u;
               key(v] = pGraph[u][v];
            }
         }
      }
   }

   /****
    **** We should now have the MST!!!!
    **** Take the end time reading
    ****/
/*
   ftime(&tp);
   tEndTime = tp.time;
   unEndTime = tp.millitm;
   *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *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
    ****/
   pbf_free_data_structs(&Q, &key, &Pi);

   return nCost;
}
/*****************************************************************************/
```

```
#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 nVerteces - 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 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_free_data_structs                      ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           NONE                                       ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:          PBFQueue** pQ - structure for priority     ***
 ***                                           queue                       ***
 ***                                                                       ***
 ***                           KeyArray* pKey - array of keys              ***
 ***                                                                       ***
 ***                           PIArray* pPi - array of PI values           ***
 ***                                                                       ***
 *** RETURNS:                  NOTHING                                    ***
 ***                                                                       ***
 *** DESCRIPTION:              This function releases resources used by ***
 ***                           the datastructures in the Prim's           ***
 ***                           algorithm.                                 ***
 ***                                                                       ***
 *****************************************************************************/
void php_free_data_structs(PBFQueue** pQ, KeyArray* pKey, PIArray* pPi)
{
   /****
    **** Release queue resources
    ****/
   free((*pQ)->vertices);
   free(*pQ);
   *pQ = NULL;

   /****
    **** Release PI and Key resources
    ****/
   free(*pKey);
   free(*pPi);
   *pKey = NULL;
   *pPi = NULL;
}
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  php_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 que***
 ***                                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 exaustively 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);
}
/*****************************************************************************/


/*****************************************************************************
 *** MST                                                                   ***
 *****************************************************************************/


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                    php_mst                                 ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:             GraphRow* pGraph - input graph          ***
 ***                                                                       ***
 ***                             int nVertices - number of vertices      ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:            GraphRow* pOutGraph - output graph      ***
 ***                                                                       ***
 ***                             double* pdfTime - time it took to get mst ***
 ***                                                                       ***
 *** RETURNS:                     int - Total cost of the MST             ***
 ***                                                                       ***
 *** DESCRIPTION:                 This function finds a MST in the graph  ***
 ***                             supplied via the pGraph argument.  The  ***
 ***                             Prim's alogorithm 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)
{
   PBFQueue*       Q;                  /* Priority Queue                   */
   KeyArray        key;                /* Keys                             */
   PIArray         Pi;                 /* PI's                             */
   int             u, v, k;            /* Vertices examinined 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 miliseconds part      */
   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                   */

   struct rusage   ruse;
   double          dfStartCPUTime;
   double          dfEndCPUTime;


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

   /****
    **** Take start time reading
    ****/
/*
   ftime(&tp);
   tStartTime = tp.time;
   unStartTime = tp.millitm;
*/
   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)
   {
      php_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
    ****                                          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[curNode]] <
                            key[Q->vertices[PARENT(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); */
   }

   /****
    **** We should now have the MST!!!!
    **** Take the end time reading
    ****/
/*
   ftime(&tp);
   tEndTime = tp.time;
   unEndTime = tp.millitm;
   *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *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, &Pi);

   return nCost;
}
/*****************************************************************************/
```

```
/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  php_mst_1                                 ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           GraphRow* pGraph - input graph            ***
 ***                                                                       ***
 ***                           int nVertices - number of vertices        ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:          GraphRow* pOutGraph - output graph        ***
 ***                                                                       ***
 ***                           double* pdfTime - time it took to get mst ***
 ***                                                                       ***
 *** RETURNS:                  int - Total cost of the MST               ***
 ***                                                                       ***
 *** DESCRIPTION:              This function finds a MST in the graph    ***
 ***                           supplied via the pGraph argument.  The    ***
 ***                           Prim's alogorithm is used using the BRUTE ***
 ***                           force to extract the minimum vertex from  ***
 ***                           the priority queue.                       ***
 ***                                                                       ***
 *****************************************************************************/
int php_mst_1(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
              double* pdfTime)
{
   PBFQueue*       Q;                  /* Priority Queue                  */
   KeyArray        key;                /* Keys                            */
   PIArray         Pi;                 /* PI's                            */
   int             u, v, k;            /* Vertices examinined at one time */
   int             nCost = 0;          /* Cost of the tree                */

   int i;

   unsigned short  unStartTime;        /* Start time miliseconds part     */
```

```
   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
    ****/
   dfStartCPUTime = CPUTIME;
/*
   ftime(&tp);
   tStartTime = tp.time;
   unStartTime = tp.millitm;
*/
   /****
    **** Put all vertices onto the queue - Use only connected ones.  The size
    **** member of the queue is used to account for the number ov 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;

   /****
    **** 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
    ****/
/*
   ftime(&tp);
   tEndTime = tp.time;
   unEndTime = tp.millitm;
   *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *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, &Pi);

  return nCost;
}
/*****************************************************************************/
```

```
#include <stdio.h>
#include <sys/types.h>
/*#include <sys/time.h>*/
#include <sys/timeb.h>
#include "BinHeap.h"

/*****************************************************************************
 *** DATA STRUCTURE MANAGEMENT FUNCTIONS                                   ***
 *****************************************************************************/


/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  pbh_alloc_data_structs                     ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           int nVerteces - 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 Heahp, an array of flags which      ***
 ***                           indicates whether the vertex is in the       ***
 ***                           queue, and array of barent 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 = (PBinNode *)malloc(nVertices * sizeof(PBinNode));

   /***
    *** Create the nodes for the binomial heap usage
    ***/
   *pStore = (NodeStore *)malloc(sizeof(NodeStore));
   NodeStore_create(nVertices, *pStore);

   /***
    *** Allocate space for the PI array
    ***/
   *pPi = (int *)malloc(nVertices * sizeof(int));
   for (i = 0; i < nVertices; i++)
   {
      (*aInQueFlags)[i] = FALSE;
      (*pPi)[i] = NIL;
      (*pKey)[i] = NULL;
   }
}
/*****************************************************************************/



/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  pbh_free_data_structs                      ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           NONE                                       ***
 ***                                                                       ***
 *** 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 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;
}
```

```
/*****************************************************************************/



/*****************************************************************************
 *** MST                                                                   ***
 *****************************************************************************/



/*
void pbh_get_vert_list(PBinNode pNode, int arr[], int maxLen, int* count)
{
   if ((*count < maxLen) && (pNode != NULL))
   {
      arr[*count] = pNode->vert;
      (*count)++;
      if (pNode->c != NULL)
         pbh_get_vert_list(pNode->c, arr, maxLen, count);
      if (pNode->sib != NULL)
         pbh_get_vert_list(pNode->sib, arr, maxLen, count);
   }
}
*/
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                  pbh_mst                                   ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:           GraphRow* pGraph - input graph            ***
 ***                                                                       ***
 ***                           int nVertices - number of vertices        ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:          GraphRow* pOutGraph - output graph        ***
 ***                                                                       ***
 ***                           double* pdfTime - time it took to get mst ***
 ***                                                                       ***
 *** RETURNS:                   int - Total cost of the MST               ***
 ***                                                                       ***
 *** DESCRIPTION:               This function finds a MST in the graph    ***
 ***                           supplied via the pGraph argument.  The    ***
 ***                           Prim's alogorithm is used using the BRUTE ***
 ***                           force to extract the minimum vertex from  ***
 ***                           the priority queue.                       ***
 ***                                                                       ***
 *****************************************************************************/
int pbh_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
            double* pdfTime)
{
   PIArray         Pi;                  /* PI's                            */

   int             u, v, k;             /* Vertices examinined 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                     */
   NodeStore       nodeStore;           /* Store of pointers               */
   PBinQueue       pQueue;              /* Pointer to binomial queue       */

   unsigned short  unStartTime;         /* Start time miliseconds part     */
   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                  */

   int             nErrCode;            /* User for debugging              */

   struct rusage   ruse;
   double          dfStartCPUTime;
   double          dfEndCPUTime;


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

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

```
      }
   }
   /***
    *** Make sure that the number of connected vertices is at least 2.
    ***/
   if (nVertCount < 2)
   {
      NodeStore_destroy(&nodeStore);
      BinQueue_destroy(&pQueue);
      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;
   Pi[u] = 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->aInQueue[v]) && (pGraph[u][v] != 0) &&
             (pGraph[u][v] < pQueue->apVertexNodes[v]->nKey))
         {
            Pi[v] = u;
            BinQueue_decreaseKey(pQueue, v, pGraph[u][v]);
         }
      }
   }

   /***
    *** We should now have the MST!!!!
    *** Take the end time reading
    ***/
/*
   ftime(&tp);
   tEndTime = tp.time;
   unEndTime = tp.millitm;
   *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *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
    ***/
   NodeStore_destroy(&nodeStore);
   BinQueue_destroy(&pQueue);
   free(Pi);

   return nCost;
}
/*****************************************************************************/
```

```
#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 succesful, FALSE if failed        ***
 ***                                                                       ***
 *** DESCRIPTION:         This function allocates memory necessasry 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->apNodes = (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->apNodes[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->apNodes);
}
/*****************************************************************************/
```

```
/*****************************************************************************
 ***                                                                       ***
 *** 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->apNodes[(pStore->top)++];
}
/*****************************************************************************/
```

```
/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:            NodeStore_free                                   ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:     PBinNode* node - pointer to the pointer that     ***
 ***                                      points to node to be freed       ***
 ***                                                                       ***
 ***                      NodeStore* pStore - node store to which the node ***
 ***                                          is to be returned.           ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:    NONE                                             ***
 ***                                                                       ***
 *** RETURNS:             NOTHING                                          ***
 ***                                                                       ***
 *** DESCRIPTION:         This function returns the allocated node back to ***
 ***                      the store.                                       ***
 ***                                                                       ***
 *****************************************************************************/
void NodeStore_free(PBinNode* node, NodeStore* pStore)
{
   pStore->top--;
   pStore->apNodes[pStore->top] = *node;
   *node = NULL;
}
/*****************************************************************************/



PBinQueue BinQueue_make(int nVertices)
{
   int       i;
   PBinQueue pQueue;

   pQueue = (PBinQueue)malloc(sizeof(BinQueue));

   pQueue->nTrees = floor(log2((double)nVertices)) + 1;
   pQueue->aTrees = (BinTree*)malloc(pQueue->nTrees * sizeof(BinTree));
   for (i = 0; i < pQueue->nTrees; i++)
      pQueue->aTrees[i] = NULL;

   pQueue->apVertexNodes = (PBinNode *)malloc(nVertices * sizeof(PBinNode));
   pQueue->aInQueue = (BOOL *)malloc(nVertices * sizeof(BOOL));
   for (i = 0; i < nVertices; i++)
   {
      pQueue->aInQueue[i] = FALSE;
      pQueue->apVertexNodes[i] = NULL;
   }

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

   return pQueue;
}



void BinQueue_destroy(PBinQueue* ppQueue)
{
   free((*ppQueue)->aTrees);
   free((*ppQueue)->aInQueue);
   free((*ppQueue)->apVertexNodes);
   free(*ppQueue);
}



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                  */
         case 1:  break;                         /* Only h1                   */

         case 2:  h1->aTrees[i] = t2;            /* Only h2                   */
                  h2->aTrees[i] = NULL;
/*
                  if ((h1->nMin == -1) ||
                      (h1->aTrees[h1->nMin]->nKey > t2->nKey))
                     h1->nMin = i;
*/
                  break;

         case 3:                                 /* h1 and h2                 */
                  carry = BinQueue_combineTrees(t1, t2);
                  h1->aTrees[i] = NULL;
                  h2->aTrees[i] = NULL;
                  break;

         case 4:  h1->aTrees[i] = carry;         /* Only carry                */
/*
                  if ((h1->nMin == -1) ||
                      (h1->aTrees[h1->nMin]->nKey > carry->nKey))
                     h1->nMin = i;
```

```
*/
                carry = NULL;
                break;

        case 5:                                 /* h1 and carry              */
                carry = BinQueue_combineTrees(t1, carry);
                h1->aTrees[i] = NULL;
                break;

        case 6:                                 /* h2 and carry              */
                carry = BinQueue_combineTrees(t2, carry);
                h2->aTrees[i] = NULL;
                break;

        case 7:  h1->aTrees[i] = carry;         /* All trees                 */
/*
                if ((h1->nMin == -1) ||
                    (h1->aTrees[h1->nMin]->nKey > carry->nKey))
                   h1->nMin = i;
*/
                carry = BinQueue_combineTrees(t1, t2);
                h2->aTrees[i] = NULL;
                break;
     }
   }
   return h1;
}



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

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

   h->aTrees[nMinInd] = NULL;
   h->nCurrentSize -= pDeletedQueue->nCurrentSize + 1;

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

   BinQueue_destroy(&pDeletedQueue);

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

   return pOldRoot;
}



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 = BINQUEUE_VERTEX_DUPLICATION;
   else
   {
      pTmpQueue = BinQueue_make(h->nCapacity);
      pTmpQueue->nCurrentSize = 1;
      pTmpQueue->aTrees[0] = n;
      pTmpQueue->apVertexNodes[n->unVert] = n;
      pTmpQueue->aInQueue[n->unVert] = TRUE;
      pTmpQueue->nMin = 0;

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



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

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

```
    }
    return nMinInd;
}



int BinQueue_decreaseKey(PBinQueue h, int unVert, int nKey)
{
    PBinMode          pCurNode;          /* Node being decreased               */
    PBinNode          pParentNode;       /* Parent node of node bing decreased */
    PBinNode          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  */

    /***
     *** "Buble Up"
     ***/
    while ((pParentNode != NULL) && (pCurNode->nKey < pParentNode->nKey))
    {
       /***
        *** Exchange the key and all satelite 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
        ***/
       h->apVertexNodes[pCurNode->unVert] = pCurNode;
       h->apVertexNodes[pParentNode->unVert] = pParentNode;

       /***
        *** Bubble up
        ***/
       pCurNode = pCurNode->pParent;
       pParentNode = pParentNode->pParent;
    }
    return BINQUEUE_OK;
)
```

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



/*****************************************************************************
 *** MST                                                                   ***
 *****************************************************************************/



/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:                    pfh_mst                                  ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:             GraphRow* pGraph - input graph           ***
 ***                                                                       ***
 ***                             int nVertices - number of vertices       ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:            GraphRow* pOutGraph - output graph       ***
 ***                                                                       ***
 ***                             double* pdfTime - time it took to get mst ***
 ***                                                                       ***
 *** RETURNS:                    int - Total cost of the MST              ***
 ***                                                                       ***
 *** DESCRIPTION:                This function finds a MST in the graph   ***
 ***                             supplied via the pGraph argument.  The   ***
 ***                             Prim's alogorithm is used using the BRUTE ***
 ***                             force to extract the minimum vertex from ***
 ***                             the priority queue.                      ***
 ***                                                                       ***
 *****************************************************************************/
int pfh_mst(GraphRow* pGraph, int nVertices, GraphRow* pOutGraph,
            double* pdfTime)
{
   PIArray         Pi;                  /* PI's                          */

   int             u, v, k;             /* Vertices examinined 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    */
   PFibNode        pNode;               /* Node that was extracted       */
   PFibNode        pRootVertex;         /* Root vertex                   */

   FibNodeStore    nodeStore;           /* Store of pointers             */
   PFibQueue       pQueue;              /* Pointer to binomial queue     */

   unsigned short  unStartTime;         /* Start time miliseconds part      */
   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                */

   int             nErrCode;            /* User for debugging            */

   struct rusage   ruse;
   double          dfStartCPUTime;
   double          dfEndCPUTime;

   /***
    *** Allocate data structures
    ***/
   FibNodeStore_create(nVertices, &nodeStore);
   pQueue = FibHeap_make(nVertices);
   Pi = (PIArray)malloc(nVertices * sizeof(int));

   /***
    *** Take start time reading
    ***/
/*
   ftime(&tp);
   tStartTime = tp.time;
   unStartTime = tp.millitm;
*/
   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
    ***/
   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->unVert = u;
         if (nVertCount == 0)
         (
            pNode->nKey = 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(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;
  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->apVertexNodes[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.millitm;
   *pdfTime = ((double)tEndTime - (double)tStartTime) +
              ((double)unEndTime - (double)unStartTime) / 1000.0;
*/
   dfEndCPUTime = CPUTIME;
   *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
    ***/
   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 succesful, FALSE if failed        ***
 ***                                                                       ***
 *** DESCRIPTION:         This function allocates memory necessasry 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 = (PFibNode)malloc(size * sizeof(FibNode))) == NULL)
      return FALSE;

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

   /***
    *** Everything is allocated.  Now we set pointers and initialize values
    ***/
   for (i = 0; i < size; i++)
   {
      pStore->apNodes[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].unVert = NIL;
      pStore->aNodes[i].nDegree = 0;
      pStore->aNodes[i].bMark = 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:             PFibNode - 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                          ***
 ***                                                                       ***
 *****************************************************************************/
PFibNode 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)++];
}
```

```
/*****************************************************************************/




/*****************************************************************************
 ***                                                                       ***
 *** FUNCTION:            FibNodeStore_free                                ***
 ***                                                                       ***
 *** INPUT ARGUMENTS:     PFibNode* node - pointer to the pointer that     ***
 ***                                      points to node to be freed       ***
 ***                                                                       ***
 ***                      FibNodeStore* pStore - node store to which the   ***
 ***                                             node is to be returned.   ***
 ***                                                                       ***
 *** OUTPUT ARGUMENTS:    NONE                                             ***
 ***                                                                       ***
 *** RETURNS:             NOTHING                                          ***
 ***                                                                       ***
 *** 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;                  /* Return value                      */
   int       i;                        /* Index into the arrays             */


   /***
    *** Allocate the queue itself and set its rootlist to NULL
    ***/
   pRetVal = (PFibQueue)malloc(sizeof(FibQueue));
   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));
/*
   pRetVal->A = (PFibNode *)malloc(nMaxSize * sizeof(PFibNode));
*/
   for (i = 0; i < nMaxSize; i++)
   {
      pRetVal->apVertexNodes[i] = NULL;
      pRetVal->aInQueue[i] = FALSE;
   }
   pRetVal->Dn = 1 + 8 * sizeof(long);
   pRetVal->A = (PFibNode *)malloc(pRetVal->Dn * sizeof(PFibNode));

   /***
    *** Return queue
    ***/
   return pRetVal;
}




void FibHeap_destroy(PFibQueue* pQue)
{
   free((*pQue)->aInQueue);
   free((*pQue)->apVertexNodes);
   free((*pQue)->A);
   free(*pQue);
   *pQue = NULL;
}




PFibNode FibHeap_concatLists(PFibNode pL1, PFibNode pL2)
{
   PFibNode pTmp;                      /* Temporary pointer to hold place   */

   /***
    *** If both lists are empty, do nothing and return an empty list
    ***/
   if ((pL1 == NULL) && (pL2 == NULL))
      return NULL;

   /***
    *** If either of the lists is empty, simply return the other
    ***/
   if (pL2 == NULL)
   {
      pL1->pRight = pL1;
      pL1->pLeft = pL1;
      return pL1;
   }
   else if (pL1 == NULL)
   {
      pL2->pRight = pL2;
      pL2->pLeft = pL2;
      return pL2;
   }

   /***
    *** At this point, it is clear that neither of the lists is empty.
    *** Perform the concatenation.
    ***/
```

```
   pTmp = pL2->pLeft;
   pL2->pLeft->pRight = pL1;
   pL1->pLeft->pRight = pL2;

   pL2->pLeft = pL1->pLeft;
   pL1->pLeft = pTmp;

   return pL1;
}



PFibQueue FibHeap_insert(PFibQueue h, PFibNode pNode)
{
   /***
    *** Set up the node to be one-node list
    ***/
   pNode->nDegree = 0;                    /* Node initially has no childredn   */
   pNode->pParent = NULL;                 /* It is also the top node           */
   pNode->pChild = NULL;                  /* No child pointer                  */
   pNode->pLeft = pNode;                  /* It is the only node in the list   */
   pNode->pRight = pNode;
   pNode->bMark = FALSE;                  /* It is also unmarked               */

   /***
    *** Concatenate the node pNode with the root list  of h
    ***/
   h->pRootList = FibHeap_concatLists(h->pRootList, pNode);

   /***
    *** Update the minimum
    ***/
   if ((h->pMin == NULL) || (pNode->nKey < h->pMin->nKey))
      h->pMin = pNode;

   /***
    *** Update the counter
    ***/
   h->nCurrentSize++;

   /***
    *** Adjust flags and pointers
    ***/
   h->aInQueue[pNode->unVert] = TRUE;
   h->apVertexNodes[pNode->unVert] = pNode;

   return h;
}



PFibQueue FibHeap_union(PFibQueue h1, PFibQueue h2)
{
   PFibQueue       h;                     /* Pointer to the resulting heap */

   /***
    *** Create and initialize the resulting heap
    ***/
   h = FibHeap_make(h1->nMaxSize);        /* Create the resulting heap     */
   h->pMin = h1->pMin;                    /* Initialize the pointer to min */

   /***
    *** Concatenate the root list of H2 with the root list of H
    ***/
   h->pRootList = FibHeap_concatLists(h1->pRootList, h2->pRootList);

   /***
    *** Adjust the minimum pointer to reflect the min
    ***/
   if ((h1->pMin == NULL) ||
       ((h2->pMin != NULL) && (h2->pMin->nKey < h1->pMin->nKey)))
      h->pMin = h2->pMin;

   h->nCurrentSize = h1->nCurrentSize + h2->nCurrentSize;

   return h;
}



PFibNode FibHeap_extractMin(PFibQueue h)
{
   PFibNode  pMinNode = NULL;             /* Pointer to node with min key */
   PFibNode  pCurChild;                   /* Iteration over child nodes   */
   BOOL      bMadeFullCycle = FALSE;      /* Flags full cycle over list   */

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

         /***
          *** Now for the node itself
          ***/
         pCurChild = pMinNode->pChild;
         pCurChild->pParent = NULL;
         pMinNode->pChild = NULL;

         /***
          *** Concatenate the child list with the heap's root list and set new
          *** minimum
```

```
         ***/
        h->pRootList = FibHeap_concatLists(h->pRootList, pCurChild);
      }

      /***
       *** Remove the node from the min node from the list
       ***/
      if (h->nCurrentSize > 1)
      {
         pMinNode->pRight->pLeft = 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;
      }

/*
      pMinNode->pRight = NULL;
      pMinNode->pLeft  = NULL;
*/

      h->nCurrentSize--;
      h->apVertexNodes[pMinNode->unVert] = NULL;
      h->aInQueue[pMinNode->unVert] = FALSE;
   }


   return pMinNode;
}




PFibQueue FibHeap_consolidate(PFibQueue h)
{
   int            i;                      /* Index into the array           */
   int            D;                      /* Size of the auxiliarry array   */

   PFibNode       pW;                     /* Pointer to the current node    */
   PFibNode       pX, pY;                 /* Nodes X and Y                  */
   PFibNode       pTmp;                   /* Swap pointer                   */

   int            d;                      /* Degree of the current node     */

   BOOL           bFullCycle = FALSE;     /* Made a full cycle over the list*/

   /***
    *** Allocate and initialize the auxilliarry 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->Dn; i++)
      h->A[i] = NULL;

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

   /***
    *** Break the circle to stop
    ***/
   h->pRootList->pLeft->pRight = NULL;
   h->pRootList->pLeft = NULL;
   pW = h->pRootList;

   do
   {
      pX = pW;
      d = pX->nDegree;
      pW = pW->pRight;

      /***
       *** We need another loop because the consolidated result may collide
       *** with another large tree on the root list.
       ***/
      while(h->A[d] != NULL)
      {
         pY = h->A[d];
         if (pY->nKey < pX->nKey)
         {
            pTmp = pX;
            pX = pY;
            pY = pTmp;
         }
         if (pW == pY)
            pW = pY->pRight;
         FibHeap_link(h, pY, pX);
         h->A[d] = NULL;
         d++;
      }
      h->A[d] = pX;
   } while (pW != NULL);

   /***
    *** Now rebuild the root list, find the new minimum, set all root list
    *** nodes' parent pointers to NULL and count the number of subtrees.
    ***/
   h->pRootList = NULL;
   h->pMin = NULL;
   for (i = 0; i < h->Dn; i++)
   {
      if (h->A[i] != NULL)
      {
         /***
          *** Make one-node-list and add it to the root list
          ***/
         h->A[i]->pLeft  = h->A[i];
         h->A[i]->pRight = h->A[i];
         h->pRootList = FibHeap_concatLists(h->pRootList, h->A[i]);
         /***
          *** Set new minimum
          ***/
         if ((h->pMin == NULL) || (h->A[i]->nKey < h->pMin->nKey))
            h->pMin = h->A[i];
      }
   }
   return h;
}
```

```
PFibQueue FibHeap_link(PFibQueue h, PFibNode pY, PFibNode pX)
{
   /***
    *** Remove node Y from the root list
    ***/
   if (pY->pRight != NULL)
      pY->pRight->pLeft = pY->pLeft;
   if (pY->pLeft != NULL)
      pY->pLeft->pRight = pY->pRight;

   /***
    *** Make the node Y a one-node circular list with a parent X
    ***/
   pY->pLeft  = pY;
   pY->pRight = pY;
   pY->pParent = pX;

   /***
    *** If node X has no children, then simply make Y its child
    ***/
   if (pX->pChild == NULL)
      pX->pChild = pY;
   else
   {
      /***
       *** Add node Y to X's child list
       ***/
      pY->pLeft = pX->pChild;
      pY->pRight = pX->pChild->pRight;
      pX->pChild->pRight = pY;
      pY->pRight->pLeft = pY;
   }

   /***
    *** Increase the degree of X
    ***/
   pX->nDegree++;

   /***
    *** Node Y was just made a child, so we can clear its mark
    ***/
   pY->bMark = FALSE;

   return h;
}
```

```
PFibQueue FibHeap_decreaseKey(PFibQueue h, unsigned int unVert, int nKey)
{
   PFibNode pY;
   PFibNode pX;


   /***
    *** Check if the key can be decreased.  If not, exit without doing anything
    ***/
   if ((! h->aInQueue[unVert]) ||
       ((pX = h->apVertexNodes[unVert]) == NULL) ||
       (nKey >= h->apVertexNodes[unVert]->nKey))
      return h;

   /***
    *** Set the new key to the vertex
    ***/
   pX->nKey = nKey;
   pY = pX->pParent;

   if ((pY != NULL) && (pX->nKey < pY->nKey))
   {
      FibHeap_cut(h, pX, pY);
      FibHeap_cascadingCut(h, pY);
   }

   if (pX->nKey < h->pMin->nKey)
      h->pMin = pX;

   return h;
}
```

```
PFibQueue FibHeap_cut(PFibQueue h, PFibNode pX, PFibNode pY)
{

   /***
    *** Remove x from the child list of y, decrementing the degree of y
    ***/
   if (pY->pChild == NULL)
      return h;                          /* Nothing to remove              */

   if (pX->pRight == pX)
      pY->pChild = NULL;                 /* Only one node                  */
   else
   {
      pX->pRight->pLeft = pX->pLeft;     /* More than one child            */
      pX->pLeft->pRight = pX->pRight;
      if (pY->pChild == pX)
         pY->pChild = pX->pRight;
   }
   pY->nDegree--;                        /* Decrement degree of y          */
   pX->pLeft = pX;                       /* Set pointers to self           */
   pX->pRight = pX;
   pX->pParent = NULL;
   pX->bMark = FALSE;                    /* Unmark X                       */

   /***
    *** Add the child to the root list of h
    ***/
   h->pRootList = FibHeap_concatLists(h->pRootList, pX);

   return h;
}
```

```
PFibQueue FibHeap_cascadingCut(PFibQueue h, PFibNode pY)
(
   PFibNode 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;
}
```