An Empirical Examination of Exact Algorithms for the Cardinality Steiner Problem

by

William Brent Johnston

BSc. Simon Fraser University, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© William Brent Johnston 1988
SIMON FRASER UNIVERSITY
August 1988

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Approval

Name: William Brent Johnston

Degree: Master of Science

Title: An Empirical Examination of Exact Algorithms for the Cardinality Steiner Problem

B. Bhattacharya
Chairman

A. L. Liestman
Senior Supervisor

A. H. Dixon

J. G. Peters
External Examiner

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

AN EMPIRICAL EXAMINATION OF EXACT ALGORITHMS FOR THE CARDINALITY STEINER PROBLEM

Author: _____
(signature)

WILLIAM BRENT JOHNSTON
(name)

8/8/88
(date)

Abstract

The *Cardinality Steiner Problem* (*CSP*) is a generalization of the well known and easily solved *shortest path* and *minimum spanning tree* problems. Despite the polynomial solvability of these and other special cases, the *CSP* is NP-complete. This complexity notwithstanding, it is a problem of practical concern, having applications in network design, chip layout, data base queries, and phylogeny. It has been an area of considerable research with no less than seven algorithms advanced for its exact solution. However, computational experience with these exact algorithms is limited and does not allow comparative analysis.

To provide such an examination, four exact algorithms have been implemented and executed over a wide variety of input which was classified according to various graph theoretic properties. Execution times were measured for individual problems and averages determined for the various problem classifications.

The results obtained from this execution alter the previously accepted characterization of these algorithms. In particular, it is shown that the implicit enumeration technique of Shore, Foulds, and Gibbons outperforms the others almost universally. A characterization of those instances for which it performs poorly is given and a means of overcoming this shortcoming suggested. The execution times of the remaining algorithms display little variance for instances of a given graph size. Although it has been stated by several researchers that the dynamic programming technique of Dreyfus and Wagner is more efficient than the similar approach of Levin, it is shown both theoretically and empirically that the opposite is true.

Acknowledgements

Unlike this page, so quickly turned and forgotten, the people mentioned here will live forever in the appreciative memories I hold for my time at Simon Fraser.

To my senior supervisor Dr. Art Liestman, percussive by nature but patient by inclination, mere thanks cannot do justice to the gratitude I feel. Your interest in my academic career, no less than your help with this particular labour, will always be warmly remembered.

To the other member of my committee, Dr. Tony Dixon, many thanks for your interest in and your insights and input into this thesis.

To my External Examiner Dr. Joe Peters, thanks for all your constructive criticism. May this effort in some small manner reward your faith in me.

To Dr. Pavol Hell, thank you for always being an inspiration whatever the work load.

To all the fellow travellers so well met along the way, my very best wishes. Too numerous to list in the particular, let the Hamiltons (Sharon and Howard) receive the thanks you are owed, I know they'll pass it on. Your aid and forbearance will always be remembered.

And to all the denizens that ever frequented Radandt Hall, take off, eh. From Bahram Gustaspi to Mahboob Ashraf it was a great slice of life.

Finally, my sincerest thanks to the support personnel in and around the Computing Science department, in particular to Ethel Inglis. Though I never quite figured out whose staff she was, I never hesitated to lean on her.

Dedication

I would like to dedicate this work to the person who made it both possible and worthwhile, my mother Beatrice. Thank you for all your support, especially during the lean times when it was needed most. And thank you for all your love which was there and appreciated always.

Table of Contents

List of Figures

List of Tables

Chapter 1
INTRODUCTION

With the ongoing growth in computational speeds and storage ability, the mere knowledge that some algorithm requires exponential time or memory in the general case may well be of little practical import. To the unfortunate soul confronted by a problem which is known to be NP-complete, these theoretical generalities seem to ignore the specific nature of his or her dilemma. Of more immediate concern are such questions as how large a problem may be solved in how much time or memory and, given alternatives, which algorithm is best. For any well developed problem, a practical examination of these issues logically extends the theoretical underpinnings in a functionally useful manner.

This thesis proposes to examine one such problem area, a combinatorial optimization problem known as the *Steiner Problem in Graphs*, hereafter abbreviated as the *SPG*. While it includes some well known and easily solved special cases, in general the problem is certainly NP-complete. This complexity notwithstanding, it is a problem of practical concern, having applications in network design, chip layout, data-base queries, and phylogeny. For this reason it has been an area of considerable research with no less than seven exact algorithms appearing in the literature. Along with numerous heuristic approaches, several examinations of special cases, and a number of surveys, the publication list is considerable. Despite this, computational experience for the *SPG* is almost non-existent and what is available serves more to verify the correctness of these algorithms than provide comparative analysis.

To provide such a empirical examination in the area of the *SPG*, four algorithms advanced for its exact solution have been implemented and executed over a wide variety of input. The metric used in this comparison is the amount of CPU time spent to generate a solution for the given problem instance. These input instances were sorted according to various graph theoretic properties and presented in large blocks, in order that measurements be available both for individual problems and averages over many "similar" problems. By organizing the testbed in this manner, it was hoped that some relation might be observed

between a definable property of the input and the executional behaviour of the algorithm. Thus, the comparative analysis undertaken is not merely between the competing solution techniques, but within each one individually.

Beyond this characterization of functionality, the implementation of these algorithms should prove more than mere exercise. Within the theoretical formalisms inherent in the written proposal of each lies the detail of abstraction representation, data structures, and execution sequencing. The effort of codification brings insight, if not hard certainty, into these issues: it may well be that improvements even at the level of efficient bookkeeping are of some significance. In addition, one of these proposed techniques has apparently never been implemented.

These issues are presented in the following manner. Chapter 2 introduces, defines, and outlines the problem. Chapter 3 discusses the detail of the solution algorithms available and the implementations undertaken. Chapter 4 outlines the problem of input selection and the approach adopted for generation and sequencing. Chapter 5 provides the empirical results obtained and an analysis of the regularities or abnormalities revealed. Finally, chapter 6 states the immediate conclusions available and gives the areas most promising for further investigation and future research.

Chapter 2
THE STEINER PROBLEM IN GRAPHS

In a graph $G = (V,E)$ with positive weights or distances provided for each edge, a basic property of interest is the minimal means of connecting or *spanning* a specified group of vertices using elements of E, the edgeset. If this specified group consists of only two vertices (say x and y) then this optimal connection is termed the *shortest path* between x and y, denoted $SP(x,y)$. If, at the other extreme, the entire vertex set is so specified, then a *minimal spanning tree* (*MST*) is called for. In either instance, it is natural to refer to the sum of the weights of the edges providing this connection as the *weight* or *value* of the structure.

For both of these problems elegant, if not straightforward, techniques exist which provide exact solutions in a very efficient manner. For instance, the shortest path between a vertex and every other vertex in V may be found with computational labour of $O(n^2)$ (where $n = |V|$) using the *single source shortest paths* algorithm of Dijkstra [9]. Alternatively, Floyd's [11] *all pairs shortest path* algorithm is an $O(n^3)$ procedure which determines the shortest path between every pair of vertices in G. As regards the MST, Prim [30] and Kruskal [24] have stated algorithms whose labour is $O(n^2)$ and $O(e \log n)$ (where $e = |E|$) respectively.

2.1. Definition

Despite these promising subcases, in the generalization of this problem, where the specified vertex set consists of an arbitrary subset of V whose cardinality lies between these bounds, the solution is somewhat more difficult to identify. Termed the *Steiner Problem in Graphs*, it is typically formulated as follows:

> **Definition 1:** Given an undirected graph $G = (V,E)$ with positive edge weights $W = \{w_e \mid e \in E\}$ and a subset S of V with $|S| = m$ $(2 < m < n)$, determine a subgraph of G which spans S and is of minimal weight.

Since this minimally weighted subgraph must surely be a tree, it is termed a *Steiner minimal tree* (*SMT*) for S in G and denoted T^*. It is natural to refer to non-minimal trees spanning S as merely *Steiner trees* denoted T_S. Once again, by summing the weights of the edges used, the weight of these trees may be determined.

Regarding a Steiner minimal tree, it should be immediately apparent that only vertices of S may be leaves, for a leaf not in S could be deleted from the tree to give a structure which still spans S and is of lower weight. However, the interior points may well include vertices not in S which provide shorter connecting paths than are possible by merely using vertices in the desired set. Thus in any Steiner tree, whether minimal or not, vertices may be of two types: *compulsory points* ($v \in S$), frequently termed *special* vertices in the literature, or *Steiner points* ($v \in V \setminus S$).

An instance of an *SPG* is described by a weighted graph G and a set S, though a standard representation of this is to present G as an $n \times n$ weight matrix $W(G)$ or merely W when G is understood. Since both vertices in G and rows and columns of $W(G)$ may be given integer labels from 1 to n, the graphs represented are termed *labelled* graphs. Vertices may thus be conveniently referenced by labels of the form v_i $(1 \leq i \leq n)$ and edges represented by the labels of vertices they connect: $e(i,j)$ (equivalently e_{ij}) is *incident* on vertices v_i and v_j. The weight of edge e_{ij} will be denoted $w(i,j)$ (or w_{ij}), which is defined to be infinite if $e_{ij} \notin E$ and 0 if $i = j$. A point of note regarding these weight matrices is that, given the undirected nature of the edges, where e_{ij} and e_{ji} refer to the same edge, the matrix will be symmetric and represent no more than $n \times (n-1) / 2$ different edges. With W permuted to place the elements of S in the first m rows and columns, as is the norm in this representation, W and m together fully describe the instance. The *problem size* of an *SPG* instance is defined to be the pair (n,m).

Unfortunately, the dual nature of this metric describing the size of an *SPG* instance upsets the sense of ordering implied by its usage. Certainly problems on graphs having an equal number of vertices may be directly compared. For instance given graphs of 100 vertices, an *SPG* problem of size (100,50) is "larger" than one of size (100,20). Where the instance graphs themselves are of different sizes, the situation regarding ordering is not clear at all. With two problems of size (100,20) and (80,40), the notion of which is "larger" can only be intuitive. Further complicating this issue is the exact relation between size and complexity. While it's easy enough to say that a problem of size (100,50) is "larger" than one of (100,20), it is not necessarily the case that the first is more complex. For these reasons it would seem that the concept of problem size is of limited usefulness, giving some means of describing a single problem but, in general, no means of comparing two.

An additional structure frequently referred to is the *complete shortest distance matrix D(G)* again of size

$n \times n$, where $d(i,j)$ (or d_{ij}) contains the length of the shortest path connecting vertices v_i and v_j. If G contains no path between some pair of vertices v_i and v_j, the graph is disconnected and $d(i,j)$ is set to infinity. This matrix may be easily calculated from $W(G)$ using Floyd's algorithm. The lengths given satisfy the *triangle inequality*, a property inherent in the consideration of a shortest path: for any vertex pair (i,j) there cannot be an intermediate vertex k such that $d_{ij} > d_{ik} + d_{kj}$. Clearly such a vertex, if existing, would lie on a shorter path and the value d_{ij} would be in error. Thus, for all i,j,k, $d_{ij} \leq d_{ik} + d_{kj}$.

This matrix may be used to form an approximate solution of some value. Consider the $m \times m$ submatrix $D(S)$ of shortest path lengths among elements of S. Finding the MST based on these shortest paths results in the so-called *edited MST* of S in G. The weight of this *eMST* (T_e) has a worst case ratio to the optimal tree T^* of $2(1 - 1/l)$ where l is the number of leaves in T^*. No heuristic algorithm proposed to date offers a significantly better ratio than this [37].

2.2. Cardinality Steiner Problem

A special case of this general *SPG* is the *cardinality Steiner problem*, abbreviated below as (*CSP*), where the weights given for edges in the graph instance are restricted to a value of one. It is now possible to reformulate the problem to ask for a tree spanning S which has the least number of edges or which includes the least number of Steiner points, since an SMT for such a graph will meet both these criteria. This by no means eases the difficulty of the problem and, to date, no special case algorithms have been advanced to take advantage of this restriction. The cardinal case remains as intractable as the general case: both are NP-complete [16].

The limitation of edge weights to a value of one does not strictly characterize the *CSP*, since any *SPG* with uniform edge weights (regardless of value) may trivially be viewed as a cardinality problem. More generally, notice that any instance of an *SPG* with integral edge weights may be transformed to a *CSP* by replacing each edge e_{ij} with a path of length w_{ij} by introducing $w_{ij} - 1$ *pseudo*-vertices. A slightly more complex transformation is available in the case of rational weights. In either case, since it is not generally possible to describe these weights in terms of either e or n, this will not be a polynomial transformation, despite its intuitive clarity.

Clearly, the range and/or distribution of edge weights will play a role in the determination of a solution to

any given problem and is an issue of note as regards its impact on any algorithm's action or effectiveness. However, given that none of the algorithms examined specifically exploit unit-valued edge weights and, in all cases, the implementations provided for these algorithms remain equally usable regardless of the weights, it would seem that research into this restricted area would yield not unduly restricted results. This applicability of results in the *CSP* to the more general problem suggests that it is an area that may profitably be examined. Indeed, the paucity of results based on experimental observations into any form of the *SPG* casts the *CSP* as an attractive area for initial investigation. By concentrating on the cardinality version of the problem, an examination of the issue of variation and distribution of edge weights is postponed. Hopefully, this approach will allow such future research to be placed in a more meaningful context.

2.3. History

The *SPG*, effectively dating from 1970, has a short history which may be briefly stated. The first widely published mention of the problem appears to be a paper by Hakimi [19] which states the problem and its relation to various covering problems in graphs and offers an algorithm for its solution. The intractable nature of the *SPG* was indicated by Karp's inclusion of the problem in his seminal paper [22] wherein the NP-completeness of several problems was established. He formulated the *SPG* as a decision problem. Augmenting the input indicated in *Definition 1* with a positive integer k, this variation asks for a *yes/no* response to the question "does G have a subgraph of weight $\leq k$ containing the set S?" He established it as NP-complete by providing a polynomial reduction from the problem *Exact Cover*. Besides showing the difficulty of the *SPG* and its close relation to the other fundamental problems presented, Karp's work clearly indicates the *SPG* as a problem of interest to theoretical computing science.

The first major survey of results on the problem is that of Foulds and Rayward-Smith [12]. They discuss several applications of the *SPG* and appraise both exact and approximate solution methods presented in the literature. The four exact algorithms dealt with are only briefly outlined as to their general approach and theoretical time requirements. Despite some analytical shortcomings, their conclusion that any algorithm to determine an exact solution will require exponential execution time is well-founded given Karp's treatment of the related decision problem. They note that this fact has motivated research in two areas: heuristic approaches, yielding approximate results for the general *SPG*, and special cases of the problem which may prove solvable in polynomial time. While some advancements in these directions have been made, they

conclude that the field is still open and state that to provide pragmatic results "testing must be done at a practical as well as a theoretical level".

Recently Winter [37] has provided a more detailed treatment of the area. In addition to a survey of exact and heuristic algorithms, he includes useful sections on problem reduction techniques and polynomially solvable classes of graphs. To the four exact algorithms discussed by Foulds and Rayward-Smith he adds three new proposals, each of which have the characteristic of reformulating the *SPG* as a *set-covering* or *0-1 integer linear programming* problem. In reporting the computational experience offered by the various authors, Winter notes that "it is very difficult to say anything definite about the performance of the algorithms". He identifies two specific areas where these results may be criticized as to their comparative utility: implementational variations both in programming languages and machines used and shortcomings in the selection of input problem instances. Based on the published results, he concludes that "it is almost impossible to decide which algorithm will perform best for dense/sparse networks with few/many" compulsory vertices.

2.4. Euclidean Steiner Problem

The *SPG* was apparently first advanced by Hakimi as a graph theoretic version of a problem in geometry on the Euclidean plane. Though it may be traced back as far as Fermat [8], this problem has become credited to Jacob Steiner (1797-1863) [7]. Briefly put, Steiner asked for a minimal system of roads to join three villages. The solution to this relatively simple problem depends on the maximum angle present in the triangle formed by the three villages. If it exceeds 120°, merely connect the two villages to that village at which the angle is maximal. If not, there is an "interior" point which, when connected by straight lines to each of the three villages, gives the shortest total distance. This interior point became known as a Steiner point and in the generalization of the problem to larger numbers of villages involving any number of Steiner points the solution was termed a Steiner minimal tree.

Gilbert and Pollack [18] have provided an extensive treatment of this *Euclidean Steiner Problem* (*ESP*) including an algorithm advanced by Melzak [27] for its general solution. Essentially, the procedure is to enumerate all possible relatively minimal trees over all possible subsets of compulsory points. While his proposal has been refined and improved upon through the years, most notably by Cockayne [6] and

Winter [36], it would seem that this general approach is the only one available when an exact solution is required. As the appeal of this algorithm is its finiteness (!), it would be hard to overstate the extreme difficulty of the problem. For instance, Boyce [4] has implemented an improved version of the refinement due Cockayne, but warns that "runs of around 10 or more [points] should be attempted with an eye on one's computer account". The current state of the art would appear to be Winter's algorithm, which effectively handles sets of up to 15 points. He claims that further improvements have the potential to take this maximum size up to 30.

2.5. Rectilinear Steiner Problem

Similar in spirit to the *GSP* is the *Rectilinear Steiner Problem* (*RSP*) where distances in the plane are measured in the rectilinear, or so-called Manhattan, metric. The *RSP* was advanced by Hanan [20] as a way of attacking the problems of providing the optimal means of wire layout for circuit boards or routing electrical or mechanical systems on city street networks or in buildings. Unlike the *GSP*, Hanan's formulation of this problem as occupying a rectangular grid formed by horizontal and vertical lines passing through each compulsory vertex casts the *RSP* as a special case of the *SPG*. Since this grid may be interpreted as a graph with vertices at line intersections and edges as the line segments between them, any algorithm for the general *SPG* may be used to provide a solution for the *RSP*.

Garey and Johnson [15] have proven this problem to be NP-complete whether viewed as a geometric or graph theoretic problem. Despite the regularity of this grid, no special purpose *RSP* algorithm has been advanced which exploits either this structure or special properties of the metric. It would seem that most of the noteworthy research into this problem has centered on the development of useful heuristics of little application to the general *SPG*.

2.6. Special Cases of the SPG

The identification of restricted classes of graphs for which polynomial algorithms may be developed has been an interesting and, at times, productive avenue of research. Though the results in many cases have been either trivial or negative, the existence of tractable classes of input is significant enough to warrant mention.

Garey *et al* [14] have established that the *SPG* remains NP-hard when the input graph G is planar.

However, if G is a tree it is quite easy to construct T^* in linear time. This has motivated research into those classes of graphs which may lie between these boundaries of complexity. Wald and Colburn have been very active in this area, giving linear time algorithms for the *SPG* in outerplanar graphs [33] and series-parallel graphs [34] (2-trees and partial 2-trees). Winter's survey includes a linear time algorithm he has developed for the class of Halin networks. Finally, White, Farber, and Pulleyblank [35] give a polynomial algorithm for the *SPG* in strongly chordal graphs and establish the NP-completeness of the cardinality version for chordal graphs.

Chapter 3
EXACT ALGORITHMS

As indicated, seven algorithms have been advanced which provide an exact solution for the *SPG*. True to the NP-complete nature of the problem, all of these procedures perform, in the worst case, an amount of labour which is exponential in terms of some metric of the instance. Depending on the approach taken, this metric may be the cardinality of the compulsory set or the number of vertices or edges present in the instance graph. As this exponentiality is limited to one of these metrics, with regards to labour every algorithm analyzed will be exponential in some respect and polynomial in another. In terms of memory usage, some of these algorithms are surprisingly frugal with only two exhibiting exponential requirements.

Each of these largely theoretical proposals may be discussed either broadly, as to their overall execution strategy, or narrowly, at the level of implementational detail such as data representation and structures, recognition and modularization of subprocesses, and executional control mechanisms. Beyond these individual characterizations, perhaps the key issue to impinge upon their implementation is the requirement that they certify their execution by producing both the weight and elements of T^*. It is not necessarily the case that these results are determined together and, in any event, the processing undertaken for their determination may well be distinct. Thus the effort of these algorithms *may* have two separable components: *length-generation* to determine the weight of T^* and *tree-generation* to determine its edges.

An additional point of note is that, despite restricting the examination undertaken to the cardinality version of the problem, in no case was the implementation so restricted. Each of these algorithms was proposed for the general problem; each implementation is usable for a general *SPG* instance. Thus, while the discussion of these algorithms may offer improvements possible for the special case of the *CSP*, these were *not* incorporated into the program representing that algorithm. Strictly speaking, this thesis will examine algorithms for the *SPG*, executed in the restricted domain of the *CSP*.

As these algorithms exploit various optimization techniques of wide application, much insight into their

operation is possible merely by outlining the approach chosen. The first work published, Hakimi's *enumeration* algorithm, is not surprisingly the most straightforward in that its action is simply to enumerate the MSTs for all subgraphs of the given instance. Inelegant as it may be, this procedure is guaranteed to yield the correct solution despite its inability to recognize such until all subgraphs have been so considered. At approximately the same time the American research team of Dreyfus and Wagner and a Russian mathematician named Levin independently suggested a *dynamic programming* approach to "build" up the Steiner minimal tree. This involves determining optimal subsolutions for use in later iterations where larger problems may be solved with relative ease given this information. As is the case with the previous approach, this requires the complete enumeration of every subsolution possible for the given instance to ensure the correct final value for the entire solution tree. A subsequent proposal by the New Zealand-based researchers Shore, Foulds, and Gibbons uses a *branch and bound* technique to implicitly evaluate solution trees and curtail enumeration where possible. Besides providing the potential for a faster determination of the optimal tree, this approach is interesting in that it generates a non-optimal solution early in the process which is thereafter improved upon until optimality is detected, allowing for intermediate results of some value.

In terms of being a functional approach to solving an arbitrary instance of an *SPG*, these four algorithms represent the current state of the art. In comparison to the three remaining proposals they share a general applicability and straightforwardness which makes them attractive candidates for empirical examination. This is not to downplay the worth of these others, each of which reformulate the setting of the *SPG* in order to exploit alternative optimization techniques. Rather, it is a reflection of inadequacies in the reformulation undertaken or inherent computational and implementational difficulties with the approach selected which require further development or refinement. In practical and personal terms, the effort required to encode these proposals from the available outlines does not seem warranted given their published results.

3.1. Hakimi's Subset Enumeration Algorithm

Hakimi's presentation of the *SPG* included an algorithm for its solution. It is derived from the observation that in any *SPG* instance, a solution tree is also an MST for the subgraph G^* induced by the combined set of compulsory vertices and Steiner points. Thus, if the correct set of Steiner points is known, T^* may be found quite readily. Hakimi formalized this as a theorem stating that over all subsets X of "candidate" Steiner points (elements of $V \setminus S$), the MST of the subgraph induced by the vertices $X \cup S$ which is of minimal weight is an SMT for S in G.

From this it is possible to construct an algorithm which proceeds by selecting a subset of $V \setminus S$ and determining the MST of the subgraph induced by this subset along with the compulsory vertices. Once all such subsets have been enumerated and the MSTs found, the minimal value observed is the weight of the Steiner minimal tree sought and the associated MST, the solution tree itself.

With the requirement that all 2^{n-m} subsets of candidate Steiner points be considered, this algorithm will be exponential in the cardinality of $V \setminus S$. It will plainly perform very poorly when m is "much" smaller than n but improve as m approaches n. This algorithm has the attraction of Occam's razor: it is a very straightforward approach consisting of two well understood and easily implemented procedures. In addition, the labour per step (determining a subset and calculating the value of its MST) and memory requirement (storing the value and tree edges of the currently minimal MST) are quite reasonable.

Lawler [25] outlines a version of this approach which exploits the fact that, in the case where the weights given for G satisfy the triangle inequality, there exists an SMT in which the number of Steiner points is no more than $m-2$. In such a case, it would only be necessary to enumerate subsets of $V \setminus S$ with cardinality $m-2$ or less in order to identify the optimal tree. This technique will improve the performance of the original algorithm in its area of poorest applicability: where $m < n/2$.

In the special case of the *CSP*, an immediate improvement on the general approach to subset construction is available. Recall that for any instance featuring uniform edge weights an SMT will be a tree spanning S which contains the fewest number of Steiner points. By enumerating subsets of $V \setminus S$ in order of non-decreasing size from 0 up to $n-m$, the first such subset which allows a tree to span S is the optimal set of Steiner points sought. Thus the procedure need only execute until the first spanning tree is discovered. By removing the requirement to completely enumerate the subsets of potential Steiner points, this approach could, on average, be expected to significantly reduce the processing undertaken.

3.1.1. Implementation

The implementation of this algorithm uses none of the optimizations or refinements discussed above, adhering strictly to Hakimi's proposal of complete enumeration and MST determination. The program uses Prim's MST procedure modified slightly to accept as input a vector describing the "live" vertices to be included in the tree constructed. Subset enumeration is handled by a procedure due to Nijenhuis and

Wilf [29] which generates in lexicographic order, the binary strings from 0 to 2^{n-m}. Clearly any such string, when appended to a string of m 1's, may be interpreted as a set containing the compulsory vertices and a subset of Steiner points by the Prim routine. This is an extremely efficient manner of generating these subsets and presenting them to the MST procedure requiring constant time: on average 2 operations per subset amortized over all 2^{n-m} subsets.

3.1.2. Analysis

Using Prim's algorithm for determining the MSTs of these subgraphs, each individual step in the procedure may be performed in labour $O(n^2)$. As outlined in the implementation section, the labour to actually form any one subset is negligible, giving a worst case bound on the entire algorithm of $O(2^{n-m} n^2)$.

3.2. Dreyfus-Wagner's Dynamic Programming Algorithm

The exact algorithm given by Dreyfus and Wagner [12] uses a dynamic programming approach. The theoretical basis for this technique is the fact that any *SPG* may be divided into three smaller *SPG*s by the *optimal decomposition property* (ODP). This property states that for any arbitrary vertex $q \in S$, there exists a partition $[D,E]$ of $S \setminus \{q\}$ which, along with some vertex $p \in V$, divides the Steiner minimal tree for the given instance into three disjoint subsets T_1, T_2, and T_3 where

- T_1 is a shortest path for $\{q,p\}$
- T_2 is a Steiner minimal tree for $\{p\} \cup D$
- T_3 is a Steiner minimal tree for $\{p\} \cup E$

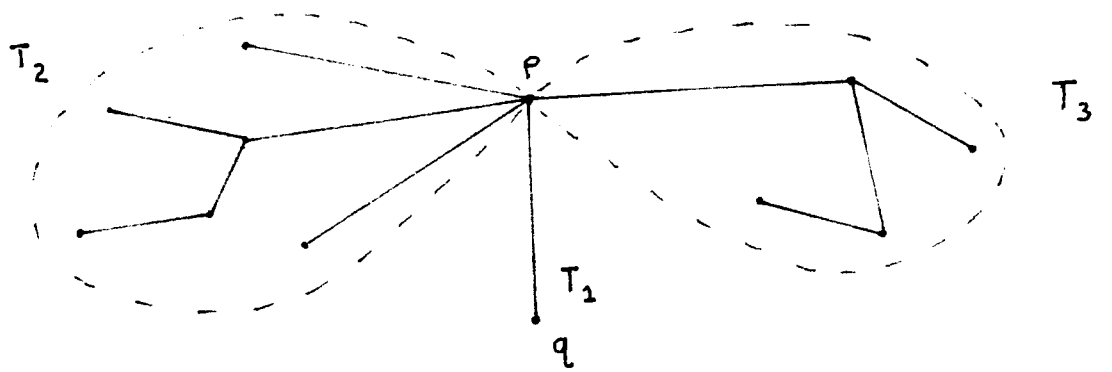

Figure 3-1: The Optimal Decomposition Property

Since T_2 and T_3 are themselves amenable to such decomposition, this property provides the framework for reducing any *SPG* instance to "elemental" subproblems (2 vertex Steiner minimal trees ie. shortest paths) and coalescing optimal subsolutions into larger, still optimal structures. Notice that, given the ODP, any 3 vertex SMT trivially decomposes into a partition $[q,D,E]$ since each of D and E must be non-empty. Thus, to solve a problem of size $(n,3)$ all that is required is to sum over all n possible choices of $p \in V$ the length of the shortest paths from p to the 3 compulsory vertices and record the minimum.

In the general case $(m > 3)$ exploitation of this property is complicated by the fact that subsolutions must be generated not simply for all partitions $[D,E]$ of $S \setminus \{q\}$. Since the coalescing stage requires the correct choice of $p \in V$, each of the potential partition sets D and E must have a separate subsolution for each possible choice of p. This requires the determination of the weight of the SMT spanning vertex sets of the form $\{s_1, \ldots, s_k, v \mid s_i \in S, v \in V\}$. This weight will be termed the *label* of that vertex set. This necessitates the storing of n labels for each of the $2^{m-1} - 2$ partition sets in order to solve the original problem.

The algorithm of Dreyfus and Wagner implements this property iteratively, by generating these subsolutions for all subsets of $S \setminus \{q\}$ of size 1, then 2, and so on up to $m - 1$. Once all these subsolutions are known, the generation of the label for the vertex set $\{s_1, \ldots, s_{m-1}, s_m\}$ will yield the weight of the SMT for S in G.

The first iteration is quite easily handled since labels for sets of the form $\{s,v \mid s \in S, v \in V\}$ are nothing more than the length of the shortest paths for each pair of vertices (s,v). This task is best performed by establishing the distance matrix $D(G)$, which is required later in any event. Without loss of generality, the k^{th} $(2 \le k \le m-1)$ iteration requires the formation of $\binom{m-1}{k}$ subsets of compulsory vertices (say $C = \{s_1, \ldots, s_k\}$) and for each, as indicated above, the determination of the label of $C \cup \{v\}$ for every $v \in V$.

The Steiner minimal tree for such a set may occur in one of two ways, either as a *split*, where v connects 2 disjoint subsets of C, or as a *join*, where v does not directly participate in T_C (the Steiner minimal tree spanning C) but is connected to some vertex in the tree by a shortest path. To express this in terms of the optimal decomposition property, for an arbitrary vertex $v \in V$ and set C, an SMT may be formed with $v = p$ (a split) or $v = q$ (a join). Thus for each possible choice of C and for each vertex $v \in V$ it is necessary to

perform two separate types of calculations. First, a calculation which determines a value termed *splitcost*(C,v) : over all non-empty partitions $[C_1,C_2]$ of C, the minimum sum of the labels of $C_1 \cup \{v\}$ and $C_2 \cup \{v\}$. Following this, a calculation which determines the sum $d_{v,v'} + splitcost(C,v')$ for each $v' \in V$, termed a *join value* for v. Notice that in the case where $v' = v$, the join value will merely be *splitcost*(C,v). The label for $C \cup \{v\}$ will be the minimum value among the n join values for v.

Dreyfus and Wagner handle this *connect* phase, in which labels are generated for the given set C and each $v \in V$, rather inefficiently. It begins by the calculation of a splitcost value for some vertex v. Since both C_1 and C_2 are smaller sets than C, the labels necessary will have been calculated previously. Thus, the procedure to determine the splitcost is to generate all partitions of C and check their label sums. With *splitcost*(C,v) known, the join values for each $v' \in V$ based on it are assigned as the *tentative* label for $C \cup \{v'\}$. The connect phase then continues with the next $v \in V$, with the new join values based on that vertex' splitcost assigned as tentative labels only when they improve the current value. After all n splitcosts are generated, and for each the n join costs checked, the labels for all $v \in V$ are guaranteed to be correct, though notice that they may have been set and reset a number of times.

An unfortunate aspect of this algorithm is that, even though iteration $m-1$ allows for the determination of the weight of T^*, the actual tree edges are still not known. To obtain these edges requires a tree-generation stage using this known weight along with the set of labels to correctly perform the optimal decompositions. By exactly reversing the coalescing stages to find the paths inherent in the optimal substructures implicitly identified, it is possible to record the edges used. This backtracking is typical of the dynamic programming methodology and by no means impugns the validity of the approach. The issue is not treated thoroughly by the authors who indicate that there are several strategies available to implement this requirement and outline two, which they term "pure" approaches. To quote, "the first method of tree-construction involves less computation while the second uses less computer storage". This would indicate that, in relation to one another, one approach is CPU-time efficient and the other is memory-space efficient.

The time efficient approach is to record, during the connect phase for each set C and vertex $v \in V$, both the vertex (say v') whose splitcost lead to the labelling of $C \cup \{v\}$ and the partition sets C_1 and C_2 which gave that splitcost. With this information stored it is quite straightforward to backtrack, connecting v' and v by shortest path edges and recursively handling C_1 and C_2. This time efficiency is acquired with the trade-off of quadrupling the already vast memory usage of this algorithm.

The space efficient approach is to record only the splitcost values for each $C \cup \{v\}$. Along with the known path lengths and labels, these may be used to "quickly" recompute (for the given v) which particular v' gave rise to the label for $C \cup \{v\}$ and the partitions C_1 and C_2 yielding this splitcost. This information is used as above with v and v' connected and C_1 and C_2 handled recursively. Dreyfus and Wagner assert that this latter method requires no more than $1/n$ the labour of length generation, since the $n-1$ "unnecessary" splitcosts need not be recomputed, and advise that its implementation is the one of choice.

To this must be added the purest of all space efficient approaches: storing nothing and recomputing everything. Notice that the space efficient method given above still requires double the memory needed strictly and unavoidably for length-generation. Clearly an approach which used no extra space would be of some interest given the already exponential memory requirements of storing the needed label values. While this could result in doubling the execution time of the algorithm in the worst case, it is worth noting that the recomputation of splitcosts and partition sets need not be complete. The known label gives a target value which may well be found before the complete regeneration of all possible substructures.

3.2.1. Implementation

Three separate versions of this algorithm are provided, each implementing a different tree-generation scheme: program *TE* using Dreyfus and Wagner's time efficient method, program *SE* using their space efficient method, and program *Hybrid* using the indicated complete regeneration method. Outside of the storing of the information required for the backtracking process, the length generation stage of these programs is exactly identical. Though compiled in separate modules, these programs share as much code as possible in order that the timing information obtained truly reflect the nuances of the backtracking methodologies and not an implementational idiosyncrasy.

Obviously the central data structure involved is the massive label array. Each such array, whether explicitly for labels or, in the case of programs *TE* and *SE*, some vertex or set representation, is declared statically to be of the largest size capable of compilation. Within this array, each subset C of compulsory vertices is assigned a row and each $v \in V$ a column, thus giving a unique address to each $C \cup \{v\}$ "pair". By interpreting the set representation of C to be an $m-1$ bit binary number, with each bit corresponding to a vertex in $S \setminus \{q\}$, it is quite easy to generate an integer row address for each such set. This approach also simplifies partition generation since, for any set C and known subset C_1, the address of the partition pair C_2

is merely the difference between the addresses of C and C_1. Notice that this allows the *TE* implementation to function with only two extra arrays, one for the partition vertex and another for one of the partition sets.

3.2.2. Analysis

The labour performed by this algorithm may be determined by analyzing the effort spent by the connect phase in setting the required $n \cdot 2^{m-1}$ labels. To determine *splitcost*(C,v) with $|C| = k$ involves the examination of $O(2^{k-1})$ partition sets. Each partition may be determined in constant time using an enumeration technique also due to Nijenhuis and Wilf. In addition, this enumeration may be implemented in such a manner as to make the effort of determining and summing the labels of the partition sets equally efficient. Thus, it is sufficient merely to count the number of times this $O(2^{k-1})$ procedure is used. Each iteration $(2 \le k \le m-1)$ requires $\binom{m-1}{k}$ splitcosts for each of the n vertices of G. Putting this together gives an amount of labour over all splitcosts of $\sum_{k=2}^{m-1} \binom{m-1}{k} 2^{k-1} n$. Since $\sum_{k=2}^{m-1} \binom{m-1}{k} 2^{k-1} \approx 3^{m-1}$, this value is $O(3^m n)$.

The number of join costs determined at each connect phase is n^2. Over all connect phases, this gives $\sum_{k=2}^{m-1} \binom{m-1}{k} n^2$ such operations, which is $O(2^m n^2)$. Including the initial determination of $D(G)$ gives a total labour of $O(3^m n + 2^m n^2 + n^3)$ for the length-determination stage of this algorithm. This is exponential in the cardinality of S and polynomial in the cardinality of V.

3.3. Levin's Dynamic Programming Algorithm

At approximately the same time as Dreyfus and Wagner, Levin [26] proposed a very similar dynamic programming algorithm. Without doubt, translational difficulties in the available English version of his paper complicate both its reading and understanding. However, once worked through, it seems clear enough that within the methodology employed Levin's work offers significant efficiencies over that of the others. Unfortunately, this has never been noted in the literature where exactly the opposite has occurred. His effort has been given rather short shrift, it has been misanalyzed and, without exception or proof, castigated as being inferior to the algorithm of Dreyfus and Wagner. In relative terms, Levin's work is an elegant and well developed improvement, its superiority is both provable in theory and observable in application.

To begin, Levin's presentation is certainly complete. His theoretical formulations are quite in order and

each step of the algorithm is given in detail. The ODP, though not named as such, is provided as the "obvious" justification for the approach. His outline calls for the computation and storing of labels (he uses the term *coordinates*) giving the value of the SMT spanning vertex sets of the form $\{s_1, \ldots, s_k, v \mid s \in S, v \in V\}$. He proposes to generate these values iteratively, in $m-1$ steps, at the k^{th} step performing $\binom{m-1}{k}$ substeps which calculate coordinates for a k-set and each vertex $v \in V$. He describes at great length and in complete detail the *connect* phase which assigns the n coordinates within each substep. He discusses the need to perform tree-generation after the computation of all coordinates and correctly describes how this may be accomplished by backtracking using the stored labels. Finally, he provides an adequate analysis of both the memory and labour requirements of the algorithm.

The basis of Levin's efficiency is in the connect phase; indeed, in all other respects it is identical to the approach of Dreyfus and Wagner. Levin attains this efficiency by organizing the labelling procedure in a manner which exploits the limited regularity and inherent connection patterns within the sub-structures being formed. The regularity is due to the fact that, within any connect phase for a subset $C = \{s_1, \ldots, s_k\}$ at least k of the choices of $v \in V$ (namely the k elements of C) already have a known label value which may be assigned with no computation. Given the incremental approach of generating subsets of compulsory vertices and determining SMTs spanning these sets and each $v \in V$, it is clear that during step $k-1$ a label was assigned to k subsets of the form $C \setminus \{s_i\} \cup \{s_i\}$ $(1 \le i \le k)$. For each of these identical representations of the set C, the label is identical at both the $k-1^{\text{st}}$ and k^{th} steps! Thus, where Dreyfus and Wagner's algorithm calculates n splitcost values, Levin's approach calculates only $n-k$ of these.

In considering the underlying connection pattern for the $n-k$ vertices which are not elements of C, there are a number of points to note. First, a vertex v may implicitly participate as a Steiner point in T_C. In this case, the value determined for $splitcost(C,v)$ will equal the known label for the k elements of C and this value will also be the label for $C \cup \{v\}$. Secondly, those vertices v for which the SMT spanning $C \cup \{v\}$ is formed by joining v to some vertex v' in the SMT spanning C, whether v' is explicitly included in C or implicitly in T_C, may have this join value calculated by using the known label of $C \cup \{v'\}$. Rather than blindly considering the sum $d_{vv'} + splitcost(C,v')$ for each (v,v') pair in V, it is only necessary to perform this operation for unlabelled vertices v and labelled vertices v' which are neighbours. Thus, the calculation is of the form $w_{vv'} + splitcost(C,v')$. Finally, since each label is known to be the minimum value of the splitcost or some join cost for that vertex, it is possible to sequentially assign these labels in non-decreasing

order. By keeping track of the optimum join value based on the current nearest labelled vertex, the minimum among all the splitcosts and current join costs of the unlabelled vertices will be the next label to assign and all vertices which have this value as one of their respective costs may be so labelled.

To exploit these properties, Levin offers a connect phase operating in a series of stages during which the currently ascertained labels are assigned and those vertices eliminated from further processing. Using efficient data structures both to list labelled and unlabelled vertices and to sort and store intermediary cost values, this approach is a definite improvement on that of Dreyfus and Wagner. In the first stage the k elements of C are given the label previously assigned the set $C \setminus \{s_i\} \cup \{s_i\}$ as described above. The next stage establishes a splitcost for each of the $n-k$ unlabelled vertices, holding them in a structure termed the *cost-heap*. After these splitcosts have been sorted into non-decreasing order, it is a simple matter to label those vertices whose splitcost equals the label assigned in stage one and delete them from the cost-heap. At the conclusion of this stage top of heap represents the minimum splitcost of an unlabelled vertex. The third stage considers the remaining unlabelled vertices, for each finding the nearest labelled neighbour, all of which have the same label at this point, to determine its current best join value. These are inserted into a separate *join-heap*. Though no vertices are labelled in this stage, the two sorted lists of costs greatly simplify the work remaining.

Each subsequent stage labels at least one of the remaining vertices, specifically the vertex (or vertices) corresponding to the minimal value in the two heaps. Notice that this label value may be found merely by comparing the topmost heap elements. At the conclusion of the label assignment for the stage it is necessary to update those join-heap elements for which a newly-labelled vertex creates an improved join value by being a nearer neighbour. Note that the labour performed in these stages will reflect the density of the graph, as the number of updates on the join-heap is bounded by the number of edges incident at these $\leq n-k$ vertices.

The backtracking methodology Levin proposes exactly reverses the action of the connect phase. In using no extra storage besides the label array, this regeneration scheme is very much like the hybrid approach outlined for Dreyfus and Wagner's algorithm. One feature of Levin's approach is that it considers only edges in W, unlike the others' use of paths in G represented by the distance matrix D. This removes the requirement of recovering path elements for a pair of endpoints.

3.3.1. Implementation

Given their organizational similarities, the implementation of this algorithm will be very close to that provided for Dreyfus and Wagner's. In fact, aside from the connect procedure, every routine required here was available almost directly from one of the programs *SE* or *Hybrid*. It was also possible to use exactly the same data structure and addressing mechanism to store the labels. Thus the only implementational details of note are concerning the far more complicated connect phase required here.

Fortunately, this complication is more comparative than real and despite its length the process is fairly straightforward. The major design consideration was the implementation of the two data structures used to hold the split and join costs. The term *heap* used to describe them is only half true. Once the splitcosts have been determined for the $n - k$ required vertices these values will never change and, aside from the minimum amongst them, are of no interest. It is thus possible to implement this merely as a sorted array with a pointer to the minimal element, performing deletions implicitly by skipping over elements which are known to be labelled. The join heap does not share this characterization as knowledge of all its values is required to perform updates which may alter the heap order. This structure was implemented in a manner outlined by Tarjan [32].

It is certainly noteworthy that this segment of the algorithm may be amenable to further optimization. Fredman and Tarjan [13] have proposed a data structure for implementing heaps allowing an asymptotic improvement for several network algorithms. These *Fibonacci heaps* support most operations (in particular *insert* and *decrease-key*) in O (1) amortized time. Despite this attractive property, the *delete-min* operation required frequently in this case is still an O (log n) task. In addition, for such small heap sizes it is unclear as to whether the asymptotic improvement will be noticeable. A better use of *F-heaps* may be in the organization of the edges incident at the unlabelled vertices. This issue was not addressed in the implementation provided here and is certainly an area open to further research.

3.3.2. Analysis

As is the case with the algorithm of Dreyfus and Wagner, the labour here is best measured in terms of setting the required $n \cdot 2^{m-1}$ labels. This is still performed in $m-1$ iterations, each of $\binom{m-1}{k}$ steps. At the k^{th} $(2 \le k \le m-1)$ such iteration however, each step requires the calculation of only $n-k$ splitcosts. This fundamental operation is identical for both algorithms, $O(2^{k-1})$. Over all splitcosts then, the labour for Levin's approach is:

$$\sum_{k=2}^{m-1}\binom{m-1}{k} 2^{k-1}(n-k) = \sum_{k=2}^{m-1}\binom{m-1}{k} 2^{k-1} n - \sum_{k=2}^{m-1}\binom{m-1}{k} 2^{k-1} k$$

$$= n/2 \sum_{k=2}^{m-1}\binom{m-1}{k} 2^{k} - (m-1)\sum_{k=2}^{m-1}\binom{m-2}{k-1} 2^{k-1}$$

$$= n/2\, 3^{m-1} - (m-1)\, 3^{m-2} \qquad (1)$$

$$= 3^{m-2}\left(\frac{3n}{2} - m + 1\right)$$

Once again this is $O(3^m n)$. To give some measure of the comparative efficiency of Levin's algorithm, consider (1). The first term expresses the total number of splitcosts determined by Dreyfus and Wagner's approach, while the second term expresses the saving attained by Levin. Dividing (1) by the first term gives the ratio of splitcost labour between the two algorithms:

$$1 - \frac{(m-1)\, 3^{m-2}}{n/2\, 3^{m-1}} \approx 1 - \frac{2m}{3n}$$

For a reasonably sized compulsory set (say $m < n/2$) this algorithm will offer a reduction in the number of splitcost calculations of up to 1/3. Notice also, that this saving will increase with m.

Levin's labelling procedure also offers savings in the number of join operations performed at any substage. Though the worst case analysis is the same as the others', its method of converging on the correct label value should be expected to perform significantly better on average. This segment of the connect phase

commences with the establishment of the join and cost heaps: $O(n \log n)$. Following this, the effort of updating the join heap is subsumed by the effort of checking edges. For the entire segment this will be no more than n^2 or m, depending on the implementation. Therefore, the total "join" labour is $\sum_{k=2}^{m-1} \binom{m-1}{k} (n \log n + n^2)$ which is $O(2^m n^2)$. The first iteration of this algorithm is handled by determining the shortest paths from each compulsory vertex to each $v \in V$, $O(m n^2)$. This gives a total labour of $O(3^m n + 2^m n^2)$ for the length-determination stage of this algorithm.

3.4. Shore, Foulds, and Gibbons' Branch and Bound Algorithm

An undeniably different approach to the *SPG* is a branch and bound algorithm given by Shore, Foulds, and Gibbons [31]. Their procedure systematically constructs sets of edges spanning S in G in a sequence of steps which consider a single edge for inclusion in or exclusion from the tree being formed. This process generates a series of partial solutions for the given problem, each of which may be evaluated as to its ability to participate in T^*. By abandoning those subsolutions whose participation is no longer feasible, this technique of implicit enumeration offers the potential to avoid much of the unproductive computation characteristic of a more explicit approach. While in its worst case this could require the examination of all 2^e possible subsets of edges, it would seem intuitive that, on average, better performance could be observed. In any event, merely to discuss an algorithm's behaviour in terms such as best or average cases is of some significance, given the unwavering exponentiality of those algorithms previously presented.

Their method of constructing these Steiner trees uses a heuristic much akin to Kruskal's MST algorithm. Recall that this algorithm builds a spanning tree in $n - 1$ "greedy" steps. Starting with each vertex in its own component, at each step select the edge which connects the two nearest components, that is the two components at which the minimal weight edge is incident, and merge the two components into one. The $n - 1$ minimal edges selected constitute an MST for G.

To adapt this approach to generate Steiner trees, Shore, Foulds, and Gibbons' algorithm also starts with n components and at each step merges two of them, one of which must be an *essential* component (intuitively, one which contains a compulsory vertex). This continues until all $s \in S$ are contained in a single component. The heuristic rule used to select the connecting edge is to determine the largest difference in weights between the two cheapest edges incident at each essential component. Once the component with the largest such

"penalty" is found, the cheapest edge incident is the edge chosen. Roughly put, this heuristic identifies the edge which, if not selected for inclusion, would give the largest increase in the weight of the tree being formed. To generate all Steiner trees spanning S using this approach, every edge so identified by the penalty process and included in some T_S must also be excluded from consideration in order to identify the next-best edge to include in the current structure. In this manner, the procedure is able to determine all subsets of edges which give rise to Steiner trees for S.

The decision tree corresponding to this enumeration consists of nodes which represent partial Steiner trees. Each such node may be evaluated by considering the combined weight of the edges selected to that point along with some measure of the minimal means of connecting the remaining components. This gives a lower bound on the tree being formed which is compared to the currently best upper bound known for the value of an SMT to determine the feasibility of the current structure. If it is feasible, the edge selection process described above is continued. If not, a node is abandoned as every Steiner tree which includes the set of edges to that point is certain to be non-optimal.

The edge selection process creates two branches for each node. One of these offspring is an *Included Edge Node* (*IEN*) where the identified edge is added to the set of edges in the partial solution and the set of components reduced by one. The other is an *Excluded Edge Node* (*EEN*) where the partial solution and set of components remain the same but the identified edge is removed from subsequent consideration. Information regarding the sets of included and excluded edges at any node is very conveniently held in a modified weight matrix whose rows and columns represent components, instead of vertices, in G. Without exception, each node will have its own unique version of the original matrix $W(G)$ formed by an operation (*merging* or *exclusion*) on the weight matrix of its parent. With edge e_{ij} $(i < j)$ selected, at the *IEN*, merging is performed by setting elements of row and column i to the minimum of the corresponding entries in rows i and j, specifically set $w_{ik} = w_{ki} = \min(w_{ik}, w_{jk})$ $(1 \le k \le n)$. At the *EEX*, exclusion merely requires the setting of $w_{ij} = w_{ji} = 0$.

The authors note the storage problems which would arise if each node retained in memory its own matrix. An additional complication arising from such an approach would be the loss of the labels of vertices in G. Consider the effect of merging some component into another. Any subsequent node selecting an edge incident at this new super-component would have no means of determining the actual vertex labels of this

edge other than by "unravelling" the previous merging steps to find the original labels. For these reasons it is impractical to form a "new" matrix at each node. They claim to handle this problem by means of a component vector C $(1 \times n)$ whose entries c_i record the label of the component which contains the vertex labelled i in G. Unfortunately, the actual use made of this vector is never stated in their explication. Presumably, though, given its own version of this vector, each node may "view" $W(G)$ in its own way by considering all elements of C which contain the same value as a component. In this light, merging is very easily accomplished. To coalesce components labelled i and j, merely set those elements of C which are currently labelled j to i, performing no operation whatsoever on $W(G)$. Consider, however, the ramifications of this approach, specifically the difficulty this introduces into the task of determining the minimally weighted edge between a pair of components (say X and Y). In addition to an examination of the entire component vector to find the vertex labels for elements of X and Y, each of the $|X| \times |Y|$ candidate edges in $W(G)$ must be checked. Recall that the penalty heuristic requires that this operation be performed to determine the 2 minimal edges for each component at every node. Clearly, at any step the labour of "unravelling" vertex labels has been avoided at the expense of "remerging" the current components. Fortunately, this issue of unravelling is rather deceiving; as discussed below the implementation provided avoids it altogether.

In processing the offspring of any node, the strategy is to *fathom* (evaluate) the *IEN* before the *EEN*, giving a depth-first flavour to the search undertaken. This allows the value of a feasible solution to be established quickly and used to aid the bounding mechanism in the search for optimality. As is typical with this methodology, the determination of meaningful bounds is crucial to the early elimination of non-optimal solutions. Complicating this truism is the trade-off almost always present between labour and precision: the more time spent on its generation, the tighter the bound. In this regard, Shore, Foulds, and Gibbons clearly favour speed above all. This philosophy seems eminently reasonable given the extreme difficulty of generating even near-optimal *SPG* solutions as evidenced in the worst case ratio available for any heuristic algorithm discussed previously.

They outline this method of bounding by means of a theorem which states that the weight of any SMT can be no less than the minimum of two easily calculated values. In considering an SMT for some set of vertices S (and more generally for a set of essential components respective to S), note that it must trivially satisfy one of two cases: it includes Steiner points or it does not. If it does not, then T^* is known to be an MST for S,

and will contain $m-1$ edges. Consider a set of m edges formed by taking the minimal edge (to another vertex in S) incident at each $v \in S$. Notice that the minimally weighted edge among these m edges has been included twice: once at each vertex where it is incident. By "deleting" this extra inclusion and summing over the $m-1$ edges chosen, a lower bound on any MST for S is established, since no MST could do better than include these $m-1$ edges. On the other hand, if T^* does include at least one Steiner point, note that it will span at least $m+1$ vertices and thus contain at least m edges. Now, consider a set of m edges formed by taking the minimal edge (to *any* $v \in V$) incident at each vertex of S. The sum of these m edges similarly bound the lowest value of such a T^*. Clearly, the minimum of these two values represents a lower bound on any SMT for S. While this bound may be arbitrarily bad, it is found quite easily, since both values may be established by examining n rows in $W(G)$ giving a labour requirement of $O(m\,n)$.

This technique may be employed at any node in the decision tree, where it will give a lower bound on the weight of the edges required to connect the essential components remaining in any partial solution being processed. By adding this bound value to the sum of the weights of edges already included, a lower bound on any T_S based on that partial solution is established. The authors mention that they determine no upper bound on a partial solution at a node, and thus make no attempt to implicitly recognize an easily completed sub-structure. Presumably, the issue here is, again, one of utility versus labour. Since an upper bound is only useful as a means of fathoming in cases where it exactly equals the lower bound, the task of determining it, even through some quick heuristic approximation, would, intuitively at least, be wasted labour at the majority of nodes.

Upon the fathoming of a node, whether by bounding or by the generation of a new solution, backtracking commences. Since the search has concentrated on the *IEN* offspring at any node, backtracking is accomplished by moving "up" the decision tree, generating the unvisited *EEN*. Once the exclusion operation has established the weight matrix for the *EEN*, the bound is calculated and, if feasible, edge selection recommenced. If the node is infeasible, backtracking continues. The entire process terminates when backtracking from the initial node is completed.

3.4.1. Implementation

One of the more appealing aspects of the branch and bound methodology is the availability of a well established format within which any specific algorithm of its type may execute. As given by Garfinkel and Nemhauser [17], this format is highly modularized and requires very few design decisions of a crucial nature. This agreeable characterization of the general approach is particularly evident in the case of the algorithm of Shore, Foulds and Gibbons. Both their method of branching by edge selection and bounding are trivially implemented routines to search the weight matrix current at any node. Indeed, the only implementation issue of note is the exact means of handling the label disambiguation apparently required to determine the set of edges included in any partial solution and the impact of this technique on the matrix structure employed.

As discussed above, the process of merging inherently creates many differently sized component structures which make the timely and efficient identification of vertex or edge labels based upon these "compressed" matrices problematic. To explicitly identify, in terms of G, each edge included in a partial solution involves the labour of either unravelling the merging steps at previous nodes or maintaining the original labels from one node to another. It is preferable to avoid a node by node commitment to explicit edge identification. Such an approach is readily available within the outline given by the authors merely by taking a slightly different view of the exact structure required from an application of this algorithm.

To begin, notice that not only is edge identification costly, it is information that is required for only one of the many nodes established in the enumeration: over all nodes where the component structure consists of a single essential component (a candidate solution) the one where the sum of the weights of the included edges is minimal. Furthermore, this edge information is not even required at the time this node is established, since all candidate solutions must be considered before this minimum is known and T^* is identified. Clearly, any attempt to disambiguate some partial structure is nothing more than an unnecessary attempt to perform tree generation in tandem with length generation.

Consider the effect of maintaining at each node, in place of a list or set of explicitly selected edges, the vector C which holds the current component label for each vertex in G. While matrices and component vectors proliferate (polynomially), the merging and exclusion processes function quite effectively with the local labels available for a given matrix regardless of the compression it may embody. The bounding mechanism is equally satisfied with a "running total" of the weights of included edges. Finally, not only are

candidate solutions easily recognized by scanning C, but the contents of this vector at the optimal solution node are themselves sufficient to describe the tree implicit in this structure! Hakimi's theorem has established that if the optimal component of G which includes S is known, T^* may be recovered as the MST of this induced subgraph.

In this context, the effect of alterations in elements of the weight matrix and the incumbent loss of labelling information is completely eliminated. Rather than leading to a proliferation of these matrices, all the necessary processing is done within the original matrix provided for the instance. Representing each node in the decision tree by a data structure which contains the relatively few bookkeeping values needed, this procedure may be implemented quite efficiently. Merging is accomplished by reserving, at the parent node, the current C vector, the component labels of the edge selected (say $e_{ij}\,(i<j)$) and the contents of the current matrix row i. The coalescing operation, which creates the matrix for the new *IEN* may now overwrite the contents of row i quite safely, leaving the contents of row j untouched. This component is eliminated at the new node by creating a new version of C where all elements of the previous vector which were j are set to i. Upon backtracking, the *EEN* is easily established by reversing this process, restoring the parent C vector and row i in the matrix then deleting edge e_{ij}. Additional values held in this node structure include its lower bound and a pair of vectors which hold the nearest component to each live component: one to the nearest essential component, the other to any component. These vectors significantly simplify both the bounding and edge selection process.

The depth first nature of the execution allowed the actual structure holding these nodes to be implemented as a last-in first-out stack. Once both successor branches are fathomed a node need never be revisited, so the stack of active nodes could never exceed the number of edges in the initial graph.

3.4.2. Analysis

The effort required at any node, whether an *IEN* or an *EEN*, is to spawn and evaluate a pair of offspring nodes. Though these may be created at separate times in the process, it is quite straightforward to analyze the labour necessary to establish the pair. The steps in the process are:

1. determine the edge to include or exclude
2. create adjusted weight matrix for the *IEN* and *EEN*,
3. calculate new lower bound for each

Step 1 requires the examination of the edges incident at each current essential component. As there can never be more than the initial m compulsory vertices and, for each, no more than n edges, the penalty operation is O ($m\,n$). Step 2 is performed twice, once for each offspring, and corresponds to the operations of merging and exclusion. For the *IEN*, this involves finding the minimum value in each row for a pair of columns, O (n). At the *EEN* the operation is far simpler: only two matrix elements need to be reset. To perform step 3 in its entirety would involve the examination of no more than m columns in the weight matrix, thus for each node, the labour would be O ($m\,n$). In actual fact, given the nature of the construction process and the similarity of weight matrices between parent and offspring, this calculation is rarely performed from scratch and may be implemented in O (n) time.

The number of nodes established will be no greater than the number of subsets of edges possible in the instance graph. Including the final MST determination, the total labour will be O ($2^e\,m\,n + n^2$).

The memory requirement at each node is also minimal. There is no need to store the entire weight matrix associated with that node. As the implementation comments outline, it is only necessary to store the list of active components and the previous values in the row altered in addition to a few bookkeeping values. As the number of these nodes will be no more than the number of edges in the graph, the entire memory usage will be O ($e\,n$).

3.5. Reformulation Algorithms

The first of these reformulation techniques to appear in the literature is Aneja's *Set Covering* algorithm [1]. Roughly put, the approach is to enumerate a series of cutsets for some partition of V and iterate using Linear Programming (with Relaxations) to converge on the minimally weighted selection of edges which satisfies the constraints inherent in forming a cover of these cutsets. At each iteration, the lower bound on the weight of the cover represented is established by solving the linear program associated with the set covering problem. Non-optimal covers have the effect of generating more cut constraints for the original formulation. Thus, the procedure seeks to solve a problem in the form of a matrix which has a fixed number of columns (e) but where the number of rows (O (e) to start) increases during convergence to the point where, even if possible initially, they may no longer be dealt with explicitly. This unacceptable growth is handled implicitly by a method termed the *row generation scheme*. While this technique has been successfully employed for other problems formulated in this manner, Aneja expresses dissatisfaction with its applicability to the *SPG*.

He offers a short summary of the computational experience gleaned from an implementation of the Set Covering algorithm. In attempting 50 problems of size $\geq$ (30,10) he obtained 29 solutions within the 60 CPU second time limit allowed each instance. Though never stated, it would appear that this limit was established to prevent problems with execution of the Linear Program. In 20 of the 21 unsolved problems, the first iteration did not terminate. This would appear to be the major shortcoming of the approach, as his final comment is that "some method other than linear programming which is able to provide good bounds at each iteration of the algorithm could make this approach much more efficient". As he selected that technique to perform this task because it is "extremely useful from a computational point of view", it is problematic as to how the algorithm may in fact be improved. Aneja offers no analysis of the Set Covering algorithm, rather a pointless exercise given that an Linear Program solution is required at each step of the algorithm.

One final point may be made regarding the test bed for his empirical observations. In no case does his input instance contain more than 11% of the possible edges for the given vertex set. Consider the effect of this extreme sparseness on the matrix used for the Linear Program. The initial matrix formed is known to have e columns and may contain up to $e/2$ rows. By constraining e to $\approx$ 10% of its potential, his testing strategy limits this matrix to $\approx$ 1% of its potential number of elements! Even so, he reports that the Set Covering algorithm is unable to make any progress whatsoever in more than 40% of these instances.

In the same Linear Programming vein as Aneja, Wong [38] has offered a *Dual Ascent* algorithm for the *SPG*. It is apparently motivated by Aneja's own observation that the bottleneck of his Set Covering algorithm is the requirement that the Linear Program at any iteration be solved exactly to give a lower bound for the iteration. Wong attempts to improve upon this by considering the dual version of the Linear Program which, rather than being solved, is approximated by an ascent heuristic. While this procedure leads to a faster determination of the desired lower bound, it cannot be said to immediately yield an improved solution technique for the *SPG*. Wong's explicitly described algorithm is, in fact, a heuristic process where his dual ascent technique is the first phase, used to "help find a feasible solution". This solution is then processed in an identical manner to the edited spanning tree heuristic previously outlined. If the weight of the structure resulting from this operation equals the lower bound determined by the heuristic, then T^* has been identified. If not, Wong suggests "that a branch and bound procedure should have little difficulty in determining an optimal solution". Clearly, this outline falls somewhat short of being algorithmic. This appraisal is supported by Winter, who considers Wong's proposal merely as a heuristic to provide a quicker

lower bound for the Linear Program formulation and implies that an algorithm, such as Aneja's, which exploits this efficiency has yet to be developed.

The final approach is Beasley's *Lagrangean Relaxation* algorithm for a 0-1 Linear Program formulation of the problem [3]. This procedure operates in a setting much like Shore *et al*'s branch and bound proposal, the main difference being that Lagrangean Relaxation algorithm devotes considerably more effort to generate the lower bound on any partial solution. Beasley's work is apparently the most developed of this group as it includes two separate methods of generating the bounds sought. In addition, Winter's survey includes an outline of yet another technique, giving three versions of the Lagrangean Relaxation algorithm. Whether this trio of bounding methods represent distinct approaches or merely evolutionary advances is difficult to determine, though certainly the pair presented in his original paper are discussed in competitive terms. The first algorithm generates tighter bounds while the second executes faster. Winter avers that the third version is an improvement over the others in both areas, "even though [*sic*] LRA3 was implemented on a CRAY-1S computer".

An analysis of the labour performed by these methods would be most informative, however Beasley provides none whatsoever. Consider though, that at any node in the decision tree searched the procedure performs 7 reduction techniques to eliminate edges and/or vertices from consideration, recalculates the distance matrix for the (reduced) graph, and performs at least 5 iterations of subgradient optimization before determining the Lagrangean multipliers. However "simple" and "easy" this process may be, the empirical results offered by Beasley suggest that this algorithm has a very narrow range of applicability.

He presents timing information for his algorithm executed on 18 problems ranging in size from (50,9) to (100,50) for graphs whose density never exceeds 8% of the possible edges. Even for these unrealistically sparse instances, the first and second versions of this algorithm were unable to terminate successfully in 3 cases. In this regard, his inclusion of problem reduction as an explicit step in the algorithm must be viewed with some apprehension. His results clearly indicate that for most of the problems attempted the input graphs were considerably reduced in size by the application of this technique at the first node in the decision tree. For one problem of size (100,50) on a graph of 200 edges, the initial reduction process identified 56 edges of T^* and eliminated 58 others from consideration. On the other hand, for the three unsolved problems, the effect of reduction was relatively minor: a problem of size (100,17) had only 15 SMT edges identified and another 30 edges rejected, leaving 155 edges for consideration.

Whether this relationship between reducability and solvability is widespread is impossible to determine from only 18 samples, however Winter's summary of the results offered for the third version of the Lagrangean Relaxation algorithm does little to improve this situation. While the problem sizes of up to (500,250) are impressive, the three density values (0.5%, 0.8%, and 2%) are extremely low. Of the 12 problems attempted, 11 were solved within 20 CPU minutes, with Winter noting an improvement in the effectiveness of the algorithm as the compulsory set becomes larger. It is worth mentioning that the effectiveness of the reduction process is most noticeable in sparse graphs with relatively many compulsory points, seemingly the exact characterization of the Lagrangean Relaxation algorithm. In the absence of further evidence of its general utility, this approach may be viewed as an effective technique only in the case of sparse graphs which are amenable to the reduction heuristic.

3.6. Comparative Results

Given that the presentations of these algorithms concentrate more on theoretical *bona fides* than computational experience, such results as are stated serve primarily to display their utility rather than justify any deeper analysis. The only comparative study of these proposals available in the literature is included in the work of Shore, Foulds and Gibbons. Using implementations of the algorithms of Hakimi and Dreyfus-Wagner along with their own, their test strategy was to provide each of the three with the same problem instance and record the amount of CPU time required to generate the solution tree. The experimental testbed consisted of five random instances for each of nine different problem sizes: three values of m, roughly "small" ($m \leq 5$), "medium" ($m = n/2$), and "large" ($m \geq n - 5$), for each of three complete graphs (K_{10}, K_{20}, K_{30}). The edge weights were chosen at random, initially in the range 1 to 50.

Clearly this organization will restrict the results obtained to displaying the effects of problem size and edge weights on the algorithms' execution. Though the conclusions offered are so limited, the authors mention in passing the opinion that, given that all the input consisted of complete graphs, they are examining "worst case results, the algorithms would probably perform better with sparser graphs". Despite its intuitive appeal, in the absence of supporting data this statement merely indicates edge density as an issue worthy of investigation.

The majority of the conclusions advanced are in accordance with the theoretical analysis of the algorithms:

- when m is "close" to n (ie. $2^{n-m} < 2^m$), Hakimi's algorithm outperforms the others
- when m is "small" in relation to n (ie. $2^m < 2^{n-m}$), Dreyfus-Wagner's is superior
- branch and bound had greater variance in times for different problems of the same size
- both the Hakimi and Dreyfus-Wagner algorithms provided relatively consistent times within the same problem sizes.

In addition, they note that their branch and bound technique "appears to fill a gap left by the other two" algorithms as it displays the best performance when m is equal to $n/2$.

In an effort to judge the impact of the specific problem instances on the observed results, a second test run was made with the range of random edge weights narrowed to a value of 1 or 2. Within the five similar instances this produced a wider fluctuation of execution times for their branch and bound technique, but only a slight change to the others. They suggested that it was perhaps the distribution (apparently in relation to the compulsory vertices) of the edge weights rather than the actual variance which affected performance.

They end by noting the difficulty of advancing helpful conclusions from the results of their small testbed and state that "a fruitful line of future research could be to attempt to characterize the class of graphs for which the algorithm[s] perform well".

Chapter 4
TEST STRATEGY

Such then is the state of research into the general *SPG*. While the literature contains a number of algorithms for its exact solution, there are only scattered and rudimentary experimental results available which serve merely to display the potential of these algorithms to provide useful answers. If the general utility of these solution methodologies is accepted, the next stage of development in this field is a broad and rigorous examination of the specific and functional aspects of these competing approaches. As has been noted by most of the researchers surveyed, the need for such a practical underpinning to this heretofore largely theoretical area is clear.

In this regard, the work of Shore *et al* may be seen as an initial step. Though their testbed was limited both in size and scope, restricting their conclusions to those of a general nature, the experimental approach was quite sound. This thesis shall build on that approach, following their indication that an "attempt to characterize the class of graphs" for which these algorithms exhibit satisfactory performance would be of value. Thus, the explicit goal here is to examine the behavior characteristics of the four major algorithms over a well defined testbed of input, both to provide general empirical information and to determine if some specific property of the input has a discernible impact on that execution. This type of investigation clearly requires a carefully constructed test strategy in order to examine an algorithm's behaviour over a large enough range of specifically characterized input as to give results amenable to a similarly specific characterization.

4.1. Input Classification

Certainly it is difficult to state with assurance exactly which graphical properties may impinge on an *SPG* algorithm's execution. Indeed, there are so many properties from which to choose that even to compile a list of candidates would be no small task. Perhaps the most general set of well defined properties are those collectively known as *graph invariants*. An invariant may be defined as a function whose domain is the set of simple, undirected graphs with no isolated nodes and whose value for a given graph is fixed regardless of the labelling which is applied or may be applied to that graph. Harary [21] defines a *complete set* of invariants as those which "determine a graph up to isomorphism". As no such set is known, the development of new invariants remains an area of active research. A comprehensive listing of many of these invariants and the relations between them has been offered by Brighton and Dutton [5].

The two most obvious properties of any graph are the number of vertices and edges: respectively its *order* and *size*. As the amount of labour performed by any of the algorithms to be examined is expressed in terms of these values, they are clearly of central importance to input classification. In particular, the influence of a graph's order on an algorithm's execution is so overriding and well stated by theoretical analysis as to preclude a useful examination of the effect of any other property. That is to say that by varying the order of the input the results observed would be directly attributable solely to that difference in order. Certainly there are issues worthy of investigation as regards the impact of the size of the vertex set, for example how large is too large or what executional relation may be observed between graph order and size of the compulsory set. However, as is the case with edge weights, it is an issue which may be conveniently isolated from consideration here and investigated profitably in its own right. Thus, the approach will be to fix the order of the input at some value and test over a range of other properties. Notice that fixing the order of the instance graph in this manner does not fix the size of the problem itself. By varying the size of the compulsory set, a useful examination of a variety of problem sizes may be performed. Indeed, this will be the major means of segmenting the problem space. This focuses the testing undertaken on the subtle variations which may be observed within comparably sized problems.

The only criteria used to set the graph order come from implementational considerations. Clearly, given the potentially vast time requirements of the algorithms, the input should not be too large to be reasonably handled. On the other hand, it should allow a problem size which is in some sense non-trivial, so that a

reasonable amount of effort is expended for its solution. Beyond this intuitive trade-off between too much and too little labour, the number of vertices must be sufficiently large as to create a suitable sample space of candidate graphs from which instances may be drawn. For these reasons a vertex set of order 30 was selected, giving a maximum edge count of 435 and a total sample space of 2^{435} labelled graphs.

In considering the second of these most fundamental invariants: graph size, it should be apparent that an analogous situation does not exist. Despite the opinion offered by Shore *et al* on this matter, that complete graphs represent worst cases, such a characterization is not immediately observable based on theoretical analysis of the algorithms. Note that this is true even for their own $O(2^e)$ procedure which, though exponential in the number of edges *in the worst case*, is apparently not amenable to a description of the number of edges constituting this worst case. More importantly, the effect of edge count or density has never been considered in any implementational testing reported in the literature. Thus, an examination of a wide range of graph sizes will be of significant value. With the order fixed, this variation on the size of the instance graphs will be the primary means of partitioning the input, an approach giving two very agreeable benefits.

Firstly, it allows for a further classification of the input based on other invariants in a manner which may enable a more refined analysis of the observed results. A subsequent division within a partition based on order and size may yield results attributable to the effect of the invariant which gave rise to that particular subdivision. Conversely, merely adding this additional categorization does not complicate consideration of the original size-based partition. Secondly, there exists a well established theory governing the generation of random elements of these fixed order and size partitions which greatly eases implementation of such a scheme.

As regards these further invariants, the intuitive footing seems less sure. In so clear a field, perhaps even a well intentioned guess would suffice as an initial foray, however it is possible to identify two characteristics upon which their selection may be based. Given the desire to examine these algorithms over a large amount of input, practical considerations require that the value of this property be readily established. This eliminates a number of invariants whose determination is itself NP-complete such as clique number, vertex covering or independence number, chromatic number, and circumference. Within those invariants remaining, it would seem natural to examine those which apparently influence or constrain the tree structure

possible in a given graph. These include distribution and size of vertex degree, graph diameter and radius, and vertex and edge connectivity values. With the exception of the last pair, all of these invariants may be easily determined in one pass through the matrix given for the graph. While still a polynomial time procedure, setting the connectivity values is a considerably more complicated matter requiring the solution of a large number of network flow problems. To speculate, these properties may well represent the boundary of easily computed invariants.

To summarize then, the testbed shall be organized in the following manner. Random graphs of order 30 will be grouped by edge count into nine partitions ranging from 50 to 250 edges. These partitions will be composed of three to five *runsets*, each of 75 graphs which have an identical value for diameter, radius, and edge connectivity. In total the testbed consists of 35 runsets representing 2625 graphs. Depending on the algorithm, up to seven distinct problem sizes were actually tested, giving a total of no less than 8,000 problem instances for each of the six implementations examined.

4.2. Input Generation

In order for the results of these executions to be experimentally meaningful, it is required that the instances selected as input be randomly representative of the partitions described. A number of methodologies have been utilized for the generation or construction of such a testbed of random graphs. Depending upon the nature of the structure desired, these have a stylistic range from *ad hoc* to algorithmic. Fortunately, the area of random graphs has been actively researched in the past few years. Palmer's [29] excellent compilation of theoretical results on the topic provides precisely the means of defining the sample space to be explored and generating structures from this space with uniform probability.

Methods used in forming test sequences for the *SPG* may be quickly reviewed. Shore *et al* generated their random instances in a very simple manner. Given that they tested complete graphs, the only randomness required was in the assignment of edge weights. Aneja used a scheme, later adopted by Beasley, geared to construct connected graphs of arbitrary sparseness. This procedure commences with the generation of a random spanning tree on the desired number of vertices, ensuring connectedness. Following this, additional edges are generated in some random fashion and added to the tree until the desired density is reached. Once this process is complete random edge weights are assigned. This last procedure comes closest to constructing the range of graphs desired here, however a superior approach is available.

Palmer states two *probability models* for viewing the sample space of undirected graphs of a given order, appropriately enough Models A and B. For theoreticians in the area of random graphs, these models allow for a probabilistic examination of the asymptotic behaviour of the number of graphs with some property of interest. To those who merely require a few (or many) random graphs, the value of this formalism is more concrete. By fully describing the set of objects desired, randomness in the construction of their elements may be assured merely by undertaking that construction in such a manner that each of the elements are equally likely to occur. Both Models A and B offer as elegant accessories exactly this means of construction.

The sample space of Model A consists of all labelled graphs of a fixed order n where edges have an equal and fixed probability p $(0 < p < 1)$ of being present. With p set at 1/2, each of the $2^{\binom{n}{2}}$ possible labelled graphs of order n have an equal probability of being chosen. Model B offers the more restrictive sample space of all labelled graphs with both order and size (e) fixed, the number of which is $\binom{\binom{n}{2}}{e}$. Notice that this model exactly defines the elements of the partitions desired for the testbed here.

To generate, with equal probability, one of these elements, it is sufficient to choose a suitably random set of edges. This may be accomplished very easily by means of an algorithm of Nijenhuis and Wilf which generates uniformly at random a k-subset of an n-set. By establishing a labelling of the 435 edges available in a graph of order 30, it is possible to interpret such a random k-subset of 435 as a collection of e edges.

Regardless of this algorithmic rigour, it is a fact that to attain true randomness in any implementation is problematic. Any program of this type will function by using a routine programmed to generate a sequence of random numbers of some sort (real or integer) which are then processed into the random structure required. The difficulty lies with the inescapable reliance on the *random number generator* used. In a computer environment, the problem of ensuring that such a sequence is "uniformly at random" is well known [23] and clearly beyond the scope of this thesis. Instead, the best that can be hoped for is an acceptable level of *pseudo-randomness*. In this regard the generation routine for these instances is at the mercy of the *Sun processor* upon which this program was developed. Within its standard *C-library* is contained a *Random* routine which generates integers in the range $(0..2^{31}-1)$ with a cyclic period $> 2^{69}$. Statistics gathered over a large amount of output from this system function [2] indicate that it *may* in fact provide these integers uniformly at random. Hopefully, the pseudo-randomness of the structures generated here may be viewed with the same degree of confidence.

The actual construction of the testbed was performed in the following manner. For a fixed number of edges, 2000 random graphs were generated by the means outlined above. Each was processed to determine the required properties and the set of 2000 sorted on the basis of diameter, radius, and edge connectivity. Parenthetically, it should be noted that in most cases edge and vertex connectivity values were identical: fewer than 50 of the 18,000 random graphs generated displayed different values. For this reason, no distinction between edge and vertex connectivity will be made and the generic term *connectivity* will refer to both.

In general, the sorting process yielded no more than 10 subsets, several of which were very small (less than 20 elements) in relation to the 3 or 4 largest. Runsets were then established by the arbitrary selection of a set of 75 graphs from these largest groups. In no case was a subset having at least 75 graphs not included as a runset. This characterizes the input selected as graphs having combinations of diameter, radius, and connectivity which occur in at least 3.75% of the graphs of a fixed order and size which were randomly generated. Whether this method ensures a representative collection of graphs of a fixed size is open to question, certainly no claim of absolute representativeness is intended.

Chapter 5
EMPIRICAL RESULTS

To reiterate, the hypothesis under which these experiments were conducted was that a correlation would be observed between some graph theoretic property and the execution time of the algorithms tested. To this end, the input was segmented as previously described: by edge count and by diameter, radius, and connectivity. For each such unique input characterization tests were made for several sizes of compulsory sets in a range deemed suitable for the algorithm in question. These ranges were chosen in accordance with the theoretical analysis of the algorithms, the conclusions advanced by Shore, Foulds, and Gibbons as regards their comparative merits, and implementational considerations. Given that n, the number of vertices in the input graph, was fixed at 30, this variation in values of m was the sole difference in the problem sizes tested.

The results reported here are in the form of the mean execution time of a runset element. This was determined by summing the total CPU seconds required to find the weight and elements of the Steiner minimal tree for each instance in the runset and dividing by the size of that runset. Execution times were obtained through a UNIX system function which monitors various resources utilized by active processes. The measurement was made for the amount of time spent in so-called "user mode", that is actual time of program execution in the CPU and includes no system overhead. As there was no input as such and the result of each individual problem was printed after the time of execution obtained, there are no *I/O* operations affecting these results in any manner. Finally, the system clock upon which these measurements are based functions by "ticking" every 0.02 seconds. It is to this accuracy that results are reported.

5.1. Impact of Diameter, Radius, and Connectivity

It is possible to quickly dispense with one of the dimensions segmenting the testbed. The results reveal no observable correlation whatsoever between execution time and some differentiation in the input's diameter, radius and connectivity values. Certainly there were variations in the timings of individual runsets, but there emerged no overall pattern upon which any supportable conclusion could be drawn. This was true both for the complete enumeration algorithms and the more volatile branch and bound routine.

This situation is not unexpected in the case of the explicit enumeration techniques, where only Levin's approach may have the capacity to display a large sensitivity to subtle input variations. As the connect phase of the algorithm features a segment where paths of increasing length are formed, it could be the case that the less the graph's diameter, the quicker this process would terminate. If so, the cumulative effect should be more noticeable at a large problem size.

7	229.82				
6	229.66	228.98			
5	229.20	228.46	228.12	227.64	
4		228.48	227.92	227.30	226.68
3					226.40
	50	60	70	80	100

Table 5-1: Levin results by diameter and edge count

Table 5-1 summarizes the results of a range of diameter values for 5 edge counts at a compulsory set of 12. While there is a downward trend in times for decreasing diameters, it is very slight indeed. Notice that this decrease *within* each edge count is always exceeded by the decrease in times *between* edge counts at a fixed diameter. It would seem that in the case of Levin's algorithm any effect of diameter variation is secondary to that of edge count.

The three Dreyfus-Wagner implementations did not display even the slight consistency of Levin's. The times could be either marginally more or less across decreasing diameter values, in no overall pattern. Ignoring diameter values, the impact of radius and/or connectivity was similarly inconsistent for all algorithms of this class.

Turning to the algorithm of Shore *et al*, the effect of these properties is equally difficult to judge. While it

displayed far greater differences in times between individual runsets, there was absolutely no correlation between these differences and the input characterization. As an example, the following table, showing times for several sizes of compulsory sets at $e = 60$ is typical of the fluctuations observed between different diameter values.

4	8.76	16.08	34.10	426.36	221.30
5	10.98	7.40	50.16	70.10	1073.84
6	6.70	24.44	60.44	360.68	1888.20
	10	12	15	18	20

Table 5-2: Shore, Foulds, and Gibbons results by diameter and compulsory set size

Depending on the size of the compulsory set, each diameter value may give the lowest, middle, or highest execution time. Even more confusing was the fact that the 75 graphs comprising these runsets had an equally wide fluctuation of times. The organization used in table 5-1, fixing the size of the compulsory set and varying edge count, would show a lack of pattern similar to table 5-2.

Certainly the experimental treatment given these three invariants did not allow for a broad examination of their impact. Criticism could be made regarding the limited range of values used and the fact that they were studied as a group, rather than individually. These results then, should be viewed not as a conclusive statement, but as an indication that these properties have no impact on the algorithms in question. Besides correcting these stated shortcomings, one means of providing a deeper consideration of these invariants could be to characterize not just the input instance, but to extend these notions to the set of compulsory vertices within the underlying graph. This intuitively generalizes these invariants in much the same way a Steiner tree generalizes the idea of a spanning tree. It may be of some interest to examine a testbed organized by these *Steiner-diameter* and *Steiner-radius* values.

In summation, it would seem that this secondary means of input characterization provides little insight into the algorithms' execution patterns and may be ignored in reporting the general results. As indicated previously, this by no means prejudices the utility of the testbed. By coalescing the runsets into larger blocks, it may be viewed as 9 sets of at least 225 graphs each, representative of the most common values of diameter, radius, and connectivity. It is in this form that the remainder of the results will be reported.

5.2. Dreyfus-Wagner Results

The results obtained for the three Dreyfus-Wagner implementations support the appraisal given for the general algorithm in all respects. The tables below state these results by program for the three values of *m* tested (7, 10, and 12) and the nine edge counts.

250	1.78	1.76	1.78
200	1.78	1.76	1.78
150	1.78	1.76	1.78
120	1.78	1.78	1.78
100	1.78	1.78	1.78
80	1.78	1.78	1.78
70	1.78	1.78	1.78
60	1.80	1.78	1.78
50	1.78	1.78	1.80
	SE	Hybrid	TE

Table 5-3: Dreyfus-Wagner results for m = 7

250	32.56	32.42	32.24
200	32.56	32.44	32.30
150	32.58	32.44	32.28
120	32.56	32.46	32.30
100	32.58	32.46	32.32
80	32.62	32.48	32.32
70	32.58	32.48	32.32
60	32.58	32.52	32.34
50	32.60	32.50	32.34
	SE	Hybrid	TE

Table 5-4: Dreyfus-Wagner results for m = 10

250	302.58	302.10	299.38
200	302.68	302.10	299.50
150	302.76	302.26	299.66
120	302.80	302.26	299.50
100	302.84	302.30	299.56
80	302.76	302.40	299.64
70	302.78	302.54	299.66
60	302.80	302.48	299.70
50	302.94	302.56	299.70
	SE	Hybrid	TE

Table 5-5: Dreyfus-Wagner results for m = 12

In comparative terms, the average times will reflect the savings attained by the various backtracking approaches. Table 5-3 indicates that for a small size of compulsory set the backtracking methodology has no impact. The larger sizes reveal that the execution times of the programs appear to obey an ordering of *TE* < *Hybrid* < *SE*. However, the decrease between programs is so slight as to be statistically insignificant, as program *TE* offers only a 1.1% improvement over *SE* and 0.9% over *Hybrid* for a compulsory set of size 12. For $m = 10$, these improvements are 0.9% and 0.6% respectively.

Given the increase in memory usage required to attain these savings, it would be difficult to represent the time efficient approach as the method of choice for the backtracking stage of this algorithm. Between programs *Hybrid* and *TE*, a 200% increase in memory usage gives a speed up of less than 1%. It would seem that by using no extra storage, the *Hybrid* approach is the most reasonable technique of the three.

It is perhaps noteworthy that Dreyfus and Wagner's stated method of choice, the space efficient implementation, displays the poorest performance of the three. Compared to the *Hybrid* approach this may be an unexpected result. Consider though, the action of the two programs. The *Hybrid* method is to regenerate all splitcosts and join values until the partition sets which yield the target label are identified. Program *SE*, by retaining the split vertex, performs this action only for the correct set of splitcosts. Thus, compared to the worst case of the Hybrid program, the *SE* approach could perform $1/n$ of these reverse splitcosts, potentially a large saving. However, notice that this saving is gained at the expense of storing $n \cdot 2^{m-1}$ vertex labels. Clearly, the trade off between this fixed number of assignments and the possibly large number of regenerations favours the latter. That is, the wasted labour of performing reverse splitcosts for the wrong vertices is always less than the wasted labour of storing unneeded vertex labels.

The only other observation that may be made for these results is a slight tendency for the times to decrease as edge count increases. Once again, this decrease is statistically insignificant: no more than 0.1% between $e = 50$ and $e = 250$. Though noticeable in all three programs for the larger compulsory sets, it is more pronounced in the *SE* and *Hybrid* programs at $m = 12$. Two factors may be identified as influences in this regard.

First, sparse instances tend to require more Steiner points and thus have more edges. It would require more effort by the backtracking stage to recover these edges. Then too, this trend may be due to denser graphs

having an increased probability that a label value is assigned through a join operation, rather than a split. During the backtracking stage for programs *SE* and *Hybrid* this "join edge" would be found without performing a reverse splitcost calculation, a saving of some note. In the case of program *TE*, which retains this information at the time of label assignment and thus performs no such calculations, this effect would manifest itself as a reduction in the amount of recursion required, giving a slight saving of an implementation dependent nature.

5.3. Levin Results

Table 5-6 gives the results of Levin's algorithm for the same values of m used with the Dreyfus-Wagner routines. For comparative purposes the *Hybrid* program was selected to represent the others, as its backtracking mechanism is identical to that of Levin's. For the two largest compulsory sets the superiority of Levin's algorithm is clearly evident. At $m = 12$ the improvement in execution time averages 33%, while at $m = 10$ the average is 15%. Interestingly enough, at $m = 7$ this algorithm is actually about 5% slower than the *Hybrid* program. These figures support the contention that the efficiency of this approach improves with an increase in the size of the compulsory set. However, this appraisal must be tempered with the knowledge that there is a point below which this theoretical superiority will not be realized.

250	1.78	26.62	224.66
200	1.80	26.68	224.86
150	1.88	26.92	225.28
120	1.86	27.10	225.86
100	1.88	27.34	226.56
80	1.92	27.56	227.38
70	1.94	27.74	228.02
60	1.96	27.94	228.60
50	2.00	28.12	229.58
	7	10	12

Table 5-6: Levin results by edge count and compulsory set size

Within these results, notice that the same tendency identified in the Dreyfus-Wagner routines is present. In this case, however, the decrease in execution time as edge count increases is much larger than the others. Between the largest and smallest graph sizes, the improvement for $m = 12$ is almost 5 seconds and represents a 2.2% reduction. As was the case with the previous algorithm, this may be attributed to the fact that the denser the graph, the more likely that a join value will label some vertex. For this algorithm and in the case

of the *CSP*, the effect is much more beneficial. This is due to the manner in which these join values are generated and handled.

In considering the activity of the join heap, it should be intuitive that with few edges present, unlabelled vertices are less likely to have a labelled neighbour. In this case they will enter the join heap with an infinite value. It will take a number of stages for all of these vertices to gain a labelled neighbour, though at each stage a few will and thus percolate upward to the top (or near top) of the heap. Compare this to the situation in a dense graph. With uniform edge weights and many edges, most vertices could have a labelled neighbour and enter the heap at or near the top, to be labelled at the beginning stage of this join segment. Not only is this labelling done faster, but those vertices which are not labelled will have a smaller heap to percolate through. Thus for dense graphs, fewer such stages will be required and the labour at each will be less.

Notice that this characterization of the algorithm is unique to the *CSP*. Given uniform edge weights, the join heap will have only two classes of elements: those about to be labelled and those with an infinite current join value. Any vertex which gains a labelled neighbour at some stage will definitely be labelled itself during the next. To put this another way, a vertex can only percolate upwards once during its time in the heap. This would cast the *CSP* as a best case for Levin's algorithm.

Returning to the situation at $m = 7$, it is noticeable that the execution time at $e = 250$ is virtually the same as the *Hybrid* routine. It may be the case that the overhead of the heap establishment and maintenance, as described above, for less dense graphs overcomes the saving in splitcost calculations. Also, the average heap size would tend to be larger for a smaller compulsory set, further limiting the efficiency of the method. In an effort to more closely characterize this algorithm in comparison to the *Hybrid* routine, tests were conducted for compulsory sets of size 8 and 9 at edge counts of 50, 80, and 120.

120	4.14	4.04	11.16	10.02
80	4.14	4.16	11.16	10.26
50	4.14	4.32	11.18	10.56
	Hybrid m = 8	Levin m = 8	Hybrid m = 9	Levin m = 9

Table 5-7: Comparison of Levin and Hybrid results for m = 8, 9

This table shows that the routine of Levin displays its efficiency at $m = 9$.

5.4. Hakimi Results

As indicated in the outline of this algorithm, it is most effective in cases where relatively few of the vertices are non-compulsory. For this reason, the range of sizes of the compulsory set was limited to between $n/2$ and n. Table 5-8 gives the results obtained for values of m of 18, 20, and 23. This fits the intuitive view of the algorithm as being in a "complementary" relation to the dynamic programming routines. Certainly, being an $O(2^{n-m})$ routine where the others are $O(3^m)$ it should offer better performance at these symmetric sizes. This is observable in the results presented in table 5-8.

250	1.54	11.52	43.88
200	1.54	11.52	43.88
150	1.54	11.54	43.90
120	1.54	11.54	43.90
100	1.54	11.52	43.88
80	1.54	11.48	43.52
70	1.54	11.42	43.28
60	1.54	11.28	42.60
50	1.50	10.90	40.54
	23	20	18

Table 5-8: Hakimi results by edge count and compulsory set size

This algorithm displays only slight variations in times for different edge counts, indeed it would seem that only for $e = 50$ is there a notable change. This reduction may certainly be attributed to the lack of edges, allowing the MST program to execute somewhat faster. At an edge count of 100 or more, the times are virtually identical. Given the strictly enumerative approach of the algorithm, this similarity of execution times for similar problem sizes was an expected result.

As these times for $m = 18$ were still quite low, a few problems were executed on a 70 edge graph at sizes of 15 and 12. These times were 322 and 2370 seconds respectively. There seems little to be drawn in the way of conclusions from this, though it would appear that for suitably small instances, less than 40 vertices perhaps, the technique should be fairly reliable within its known range of applicability.

5.5. Shore, Foulds, and Gibbons Results

As indicated by the comments of the authors regarding their testing of this algorithm, it could be expected to show a wider fluctuation of execution times than the algorithms discussed previously. This characterization is also forthcoming based on the theoretical appraisal and is clearly in evidence in the data obtained. As the authors suggested that it is the method of choice when m is in the middle of its range (close to $n/2$), the sizes of compulsory sets tested were 10, 12, 15, 18, and 20. These results are listed in table 5-9.

250	0.08	0.12	0.16	0.22	0.26
200	0.10	0.12	0.18	0.24	0.28
150	0.12	0.12	0.18	0.24	0.28
120	0.88	0.34	0.18	0.24	0.28
100	0.94	3.52	99.34	∞	56.26
80	3.80	9.42	58.36	849.38	∞
70	5.22	10.28	99.26	260.70	1414.58
60	12.04	23.10	145.90	912.84	1930.40
50	11.64	17.72	20.00	39.34	49.44
	10	12	15	18	20

Table 5-9: Shore, Foulds, and Gibbons results by edge count and compulsory set size

It is noteworthy that the anticipated fluctuation in times manifested itself only in the sparser instances of 100 or fewer edges. Furthermore, the larger values in table 5-9 were due, not to consistently large times, but to a few bad cases with which the algorithm had great difficulty. For instance, the times labelled ∞ were each due to a single instance which failed to terminate after executing for over 300,000 CPU seconds.

With this caveat in mind then, several immediate observations may be made for these averages. Notice that, while the execution times at $e = 250$ are always the lowest for each size of compulsory set, the results at $e = 50$ are never the highest. In all cases, the times peak at some edge count between these extremes, in fact between 60 and 100 edges. There is also a distinct drop in times between edge counts of 100 and 120. This suggests that, in terms of edge count, the algorithm has a narrow band where it performs poorly, but that above and below that band it functions comparatively well.

In considering the effects of problem size, it is almost always the case that times increase with an increase in m. For the instances of less than 100 or more than 120 edges this characterization is strictly true. Indeed, the times at $e = 80$ show an almost exponential growth. This is nearly the case at $e = 100$, but there is a

sharp drop for a compulsory set of size 20. Only at $e = 120$ do the times decrease then increase as m becomes larger, with the minimum at $m = 15$.

Recall the observations advanced by Shore *et al* regarding this algorithm's executional characteristics. While they mention that the algorithm performs well when m is near $n/2$, they offered no deeper analysis of its behaviour for various sized problems. They also remarked that complete graphs probably represent the worst case for this algorithm. This would suggest that the expected results would show a downward trend in times as edge count decreased and as m neared the middle of its range. Given the results in table 5-9, it is possible to make a more specific appraisal of the algorithm's characteristics in the restricted case of the *CSP*.

Regarding the size of the compulsory set, the results in table 5-9 for $m = 15$ show a better average than would have been obtained from any other algorithm. In that comparative respect then, this algorithm does "fill the gap" left by the others when m is near $n/2$. However, it does not appear to support an interpretation of that statement as suggesting that the best case for the algorithm is when m is in that range.

In terms of edge count, though the comment of Shore *et al* was only an opinion, it is not possible to dismiss it out of hand merely on the results here. The only conclusion that may be drawn with certainty from table 5-9 is that *in the case of the cardinality version of this problem*, given a sufficient number of edges the algorithm functions very well. Furthermore, it *may* well be that this sufficient number of edges is relatively low in terms of the total number of edges possible in some instance, for these results indicate that this good performance may be expected when $\approx$ 25% of the possible edges are present. The cause of this characterization is the functionality of the edge selection technique employed by the branching process in the presence of uniform edge weights.

In a graph where the edge weights are identical this technique rarely ever "selects" an edge. Recall that the heuristic operates by identifying the minimal edge incident at the essential component where the difference between the two minimally weighted edges is largest. In the case where the weights are uniform, the only time an edge could be selected is when it is the only edge incident at such a component. Otherwise, the edge chosen is the "first" edge incident at the component labelled 1: ties are broken lexicographically! As long as this edge connects essential components, this choice is optimal. For instance, in a complete graph this approach is guaranteed to select $m - 1$ edges which span the compulsory set with no possibility of an

incorrect selection. The algorithm constructs T^* in $m-1$ steps in a manner identical to Prim's MST algorithm, as components are merged in order into the first component in the labelling. In less dense graphs, so long as a tree spanning the compulsory set using no Steiner points is possible, the same characterization holds: no incorrect edges are chosen.

However, when Steiner points are required, as is more likely the case in sparser graphs, this lexicographic preoccupation may provide very little efficiency. There seems no possibility of choosing an edge on its relative merits, presumably the goal of this heuristic. Thus, the approach has the potential to degenerate to an explicit enumeration of structures formed by considering edges to Steiner points in the order they appear in the instance labelling. Depending on where the selection of the correct edge occurs in this enumeration, a considerable amount of unnecessary labour may be performed. Contributing to this problem is the inability of the bounding mechanism to identify anything more than a disconnected graph structure, too large a set of edges, or a candidate solution.

Fortunately, this characterization of a possible increase in labour with a decrease in edges only holds up to a point. Specifically, there comes a time where the number of edges is so few that even if inefficiently handled, they may all be considered fairly rapidly.

The results obtained reflect this appraisal. In no case did a problem whose solution value was $m-1$ require more than 0.3 seconds. In addition, every single bad case for the algorithm required a Steiner point. Since merely requiring a Steiner point did not always give a large increase in time, the problem is not inherent in the algorithm, but in some way brought out by a property of the input.

In order to gauge the effect of this decidedly unheuristic tie-breaking methodology, several problems with large execution times were selected and the rows and columns of their instance graph randomly permuted before resubmission to the program. This has the effect of altering the labels applied to the components and thus the order in which they are selected. To ensure that the actual problem not change, only the candidate Steiner points were so permuted. The outcome of this permutation was not encouraging in that the times changed only marginally. However, when this relabelling procedure was performed on the compulsory set, the results were very noticeable.

initial	161.78	198.04	209.18
best 3	17.94	0.46	0.60
	18.28	0.52	0.60
	18.48	0.54	110.74
worst 3	141.14	232.30	342.86
	163.22	270.00	349.62
	163.24	270.70	514.84
	e = 70	e = 60	e = 70
	m = 12	m = 15	m = 18

Table 5-10: Results of compulsory set permutation

Table 5-10 presents a summary of 25 permutations applied to three individual problems with a compulsory set of size 12, 15, and 18 at edge counts of 60 and 70. The results list the times of the unpermuted instance followed by the lowest three and the highest three observed execution times.

Clearly the labels applied to vertices in the instance graph have a major impact on the execution of this algorithm. From these and other, similar results it would seem that it is quite often possible to permute the rows and columns and find a labelling amenable to the algorithm's lexicographic approach. This is a result of some significance to the utility of this algorithm as regards the *CSP*. Given that it is a very effective technique at higher densities, the potential to attain reasonable execution times in its area of poorest performance would make it attractive for all classes of input.

The number of these instances with pathological labellings was relatively few. In order to judge their impact on the averages in table 5-9, the results were sorted and the 5% which gave the highest execution times identified. This 5% figure was selected arbitrarily and totalled 15 graphs for the 50, 60, and 70 edge sets and 18 for 80 and 100. These instances were removed from the testbed and the averages for the remaining 95% recalculated. Table 5-11 gives the results for these best cases.

100	0.32	0.22	0.18	0.24	0.28
80	0.94	2.40	3.08	0.26	0.30
70	2.36	2.80	11.36	15.54	0.30
60	4.72	7.30	18.58	37.40	52.60
50	5.80	7.86	8.00	6.90	6.10
	10	12	15	18	20

Table 5-11: Shore, Foulds, and Gibbons results with "worst" 5% removed

The figures clearly indicate that in regards to the overall averages, the relatively enormous times of the few bad cases impact noticeably on the results. For the testbed employed, most instances are handled very easily by the algorithm. Certainly it is not possible to characterize the routine as being effective for 95% of all graphs or that only 5% of the cases could be expected to be bad. The exact nature of the distribution of these instances cannot be stated with certainty merely from these results. However, it is possible to suggest a characterization of the range of edge counts for which this algorithm exhibits poor performance. From table 5-11 it would seem that the low end of this range, $e = 60$, contains relatively more bad cases but that their execution time is moderately large. For the larger edge counts, the number of these bad cases is less, but they require much more time to execute. For instance, notice that the 2 entries in 5-9 which were given an infinite "average" ($m = 18$, $e = 100$ and $m = 20$, $e = 80$) have shrunk to 0.24 and 0.30 seconds respectively.

As regards the variation in times within edge counts for different sizes of compulsory sets, table 5-11 does not reveal the same patterns as 5-9. Notice the sharp drop at a compulsory set of size 20 for edge counts of 70 or more in these best cases. Given the algorithm's efficiency in handling compulsory vertices, as opposed to candidate Steiner points, it may be the case that in the absence of pathological labellings, an equal number of each could bring out the worst behaviour in the procedure. In terms of the *CSP*, the combination of low edge count and a medium sized compulsory set seems a more likely characterization of the worse case for the algorithm than that provided by the authors.

In an effort to determine the applicability of these results to larger instance graph orders, a small testbed of 75 graphs was constructed for 40 vertices. Tested at three different edge counts (130, 200, and 400) the results proved remarkably consistent with the analysis offered above.

400	0.12	0.20	0.34	0.50	0.68
200	0.50	4.72	0.34	0.50	0.70
130	5.50	120.50	5189.28	∞	∞
	10	15	20	25	30

Table 5-12: Shore, Foulds, and Gibbons results for n = 40

Once again the highest execution times occured at a low edge count and were attributable to a few bad cases. The results at a compulsory set of size 20 for 130 edges were due to five bad instances, with all others completing in under a second. Both of the larger compulsory sets at 130 edges had a single instance which

failed to terminate. At the next largest edge count, which was very near to 25% of the possible edges, the results improved significantly, exactly the characterization of the original testbed.

In summation, this algorithm of Shore, Foulds, and Gibbons is in many respects the best available for the *CSP*. It would appear that if the problem instance is suitably dense, it outperforms any other algorithm in all problem sizes for which it was tested. It is unfortunate that there seems to be no means of characterizing its worse cases other that by broad classification of the edge counts where they are likely to occur. Despite the existence of these cases for which it performs poorly, they may well be sufficiently rare to warrant this algorithm's usage even in its known range of potentially degraded performance. An additional point concerning its general utility is its unique ability to function very well for the input it can handle. That is, if the algorithm is capable of solving a given problem, it will quite often do so immediately. This suggests that it may be amenable to some form of probabilistic improvement, where it could be allowed to execute for a fixed time and if unsuccessful, abandon that execution and restart after a permutation of the input. Regardless of its lack of merit from a theoretical point of view, the idea seems intriguingly appealing in practical terms.

Chapter 6
CONCLUSIONS AND FUTURE DIRECTIONS

Based on the empirical results gathered from the execution of these algorithms, it must be concluded that the accepted characterization of their utility in the area of the *SPG* does not extend to the case of the *CSP*. In addition it is possible to further refine one aspect of their comparative worth even in the general problem area.

In terms of the cardinality version of the problem, the branch and bound algorithm of Shore, Foulds, and Gibbons significantly outperforms all the others across a wide variety of compulsory set sizes. It will occasionally exhibit poor performance in a class of input which may be identified by edge count. Even within its relatively narrow range of potential difficulty, it will frequently function with no degraded performance. No other algorithm tested can match either its domain of application in terms of problem size or its potential to provide a very fast solution.

Of the other algorithms studied, Hakimi's is an effective procedure for problems where more than half the vertices are compulsory. Its performance is directly linked to the number of candidate Steiner points in the problem and execution times for problems of the same size will show little fluctuation. Depending upon the operating speed of the machine used and the time available for execution, this algorithm will be impractical for problems where the number of candidate Steiner points is large.

The algorithm of Levin is an effective process for problems where less than one half the vertices are compulsory and, as is the case with Hakimi's, has a physical limit beyond which its usage will be unattractive. This limit will be based upon the number of compulsory vertices in the problem and, once again, is machine dependent. An added requirement is the need for sufficient memory to store the necessary intermediate subsolutions. This algorithm has been seriously misjudged in the literature and it may be stated conclusively that it will outperform the algorithm of Dreyfus and Wagner significantly at larger sizes of compulsory sets. This characterization will extend to the case of the *SPG*.

The algorithm of Dreyfus and Wagner shares the same executional characteristics as that of Levin. Specifically, it will show little fluctuation in times for problems of the same size, it requires a large amount of memory, and its usage becomes unattractive as the size of the compulsory set grows. However, it must be judged the poorest of the four as it displays the largest execution times and requires the most memory.

In terms of future areas of research revealed by this study, it has already been stated that a valuable contribution would be to conduct an investigation of edge count in the area of the *SPG*. A deeper study of the combined effect of variable edge weights and densities would be of special interest in the case of the algorithm of Shore *et al.* This could also be performed for Levin's algorithm which has never been reported as having been tested for the *SPG*.

In the area of the *CSP* there are a few issues of interest which may be identified. First, a version of Hakimi's algorithm specialized for the *CSP* as indicated previously and provided with an optimized MST subroutine may well prove surprisingly effective. Then too, the permutation heuristic advanced for the branch and bound routine would be an interesting exercise, though it would undoubtedly require some theoretical groundwork. Finally, for those seeking a real challenge, the development of a truly useful edge selection heuristic would immediately improve the branch and bound routine.

In the way of new algorithms, clearly an area that has yet to be investigated is the design of a distributed or parallel algorithm for either problem. All three of the fundamental methodologies of algorithms studied here seem amenable to such techniques.

References

[1] Aneja, Y. P.
An Integer Linear Programming Approach to the Steiner Problem in Graphs.
Networks 10:167-178, 1980.

[2] Ashraf, M.
Personal Communication.
1987.

[3] Beasley, J. E.
An Algorithm for the Steiner Problem in Graphs.
Networks 14:147-159, 1984.

[4] Boyce, W. M.
An Improved Program for the Full Steiner Tree Problem.
ACM Transactions on Mathematical Software 3:359-385, 1977.

[5] Brigham, R. C. and Dutton, R. D.
A Compilation of Relations Between Graph Invariants.
Networks 15:73-107, 1985.

[6] Cockayne, E. J.
On the Efficiency of the Algorithm for Steiner Minimal Trees.
SIAM Journal on Applied Mathematics 18:150-159, 1970.

[7] Courant, H. and Robbins, H.
What is Mathematics?
Oxford Press, New York, 1941.

[8] Coxeter, H. M.
Introduction to Geometry.
Wiley and Sons, New York, 1961.

[9] Dijkstra, E. W.
A Note on Two Problems in Connexion with Graphs.
Numerische Mathematik 1:269-271, 1959.

[10] Dreyfus, S. E. and Wagner, R. A.
The Steiner Problem in Graphs.
Networks 1:195-207, 1971.

[11] Floyd, R. W.
Algorithm 97, Shortest Path.
Communications of the ACM 5:345, 1962.

[12] Foulds, L. R. and Rayward-Smith, V. J.
Steiner Problem in Graphs: Algorithms and Applications.
Engineering Optimization 7:7-16, 1983.

[13] Fredman, M. L. and Tarjan, R. E.
Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms.
In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 338-346. IEEE, 1984.

[14] Garey, M. R., Graham, R. L., and Johnson, D. S. .
The Complexity of Computing Steiner Minimal Trees.
SIAM Journal on Applied Mathematics 32:826-834, 1977.

[15] Garey, M. R. and Johnson, D. S. .
The Rectilinear Steiner Problem is NP-complete.
SIAM Journal on Applied Mathematics 32:826-834, 1977.

[16] Garey, M.R. and Johnson, D. S.
Computers and Intractability a Guide to the Theory of NP-Completeness.
Freeman, San Francisco, 1979.

[17] Garfinkel, R. S. and Nemhauser, G. L.
Integer Programming.
Wiley and Sons, New York, 1972.

[18] Gilbert, E. N. and Pollack, H. O.
Steiner Minimal Trees.
SIAM Journal on Applied Mathematics 16:1-29, 1968.

[19] Hakimi, S. L.
Steiner's Problem in Graphs and its Implications.
Networks 1:113-133, 1971.

[20] Hanan, M.
On Steiner's Problem with Rectilinear Distance.
SIAM Journal on Applied Mathematics 14:255-265, 1966.

[21] Harary, F.
Graph Theory.
Addison-Wesley, Reading, Mass., 1969.

[22] Karp, R. M.
Reducability Among Combinatorial Problems.
In Miller, R. E. and Thatcher, J. W (editors), *Complexity of Computer Computations*, pages 85-193. Plenum Press, New York, 1972.

[23] Knuth, D. E.
The Art of Computer Programming. Volume 2: *Semi-Numerical Algorithms.*
Addison-Wesley, Reading, Mass., 1973.

[24] Kruskal, J. B.
On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem.
Proceedings of the American Mathematical Society 4:48-50, 1956.

[25] Lawler, E. L.
Combinatorial Optimization: Networks and Matroids.
Holt, Rinehart, and Winston, New York, 1976.

[26] Levin, A. J.
Algorithm for the Shortest Connection of a Group of Graph Vertices.
Soviet Mathematics Dolkademi 12:1477-1481, 1971.

[27] Melzak, Z. A.
On the Problem of Steiner.
Canadian Mathematics Bulletin 4:143-148, 1961.

[28] Nijenhuis, A. and Wilf, H. S.
Combinatorial Algorithms for Computers and Calculators.
Academic Press, New York, 1978.

[29] Palmer, E. M.
Graphical Evolution .
Wiley, New York, 1985.

[30] Prim, R. C.
Shortest Connection Networks and Some Generalizations.
Bell Systems Technical Journal 36:1389-1401, 1957.

[31] Shore, M. L., Foulds, L. R., and Gibbons, P. B.
An Algorithm for the Steiner Problem in Graphs.
Networks 10:323-333, 1982.

[32] Tarjan, R. E.
Data Structures and Network Algorithms.
SIAM Journal on Applied Mathematics, Philadelphia, 1983.

[33] Wald, J. A. and Colburn, C. J. .
Steiner Trees in Outerplanar Graphs.
Congressus Numerantium 36:15-22, 1982.

[34] Wald, J. A. and Colburn, C. J. .
Steiner Trees, Partial 2-Trees and Minimum IFI Networks.
Networks 13:159-167, 1983.

[35] White, K., Farber, M., and Pulleyblank, W.
Steiner Trees, Connected Domination, and Strongly Chordal Graphs.
Networks 15:109-124, 1985.

[36] Winter, P.
An Algorithm for the Steiner Problem in the Euclidean Plane.
Networks 15:323-345, 1985.

[37] Winter, P.
Steiner Problem in Networks: A Survey .
Networks 17:129-167, 1987.

[38] Wong, R. T.
A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph.
Mathematical Programming 28:271-287, 1984.