CHAPTER I

INTRODUCTION

Minimizing a network's length is one of the oldest optimization problems in mathematics and, consequently, it has been worked on by many of the leading mathematicians in history. In the mid-seventeenth century a simple problem was posed: Find the point P that minimizes the sum of the distances from P to each of three given points in the plane. Solutions to this problem were derived independently by Fermat, Torricelli and Cavalieri. They all deduced that either P is inside the triangle formed by the given points and that the angles at P formed by the lines joining P to the three points are all 120°, or P is one of the three vertices and the angle at P formed by the lines joining P to the other two points is greater than or equal to 120°.

In the nineteenth century a mathematician at the University of Berlin, named Jakob Steiner, studied this problem and generalized it to include an arbitrarily large set of points in the plane. This generalization created a star when P was connected to all the given points in the plane, and is a geometric approach to the 2-dimensional center of mass problem.

In 1934 Kössler and Jarník generalized the network minimization problem even further [1]: Given n points in the plane find the shortest possible connected network containing these points. This generalized problem, however, did not become popular until the book, *What is Mathematics*, by Courant and Robbins [2], appeared in 1941. Courant and Robbins linked the name Steiner with this form of the problem proposed by Kössler and Jarník, and it became known as the Steiner Minimal Tree problem. The general solution to this problem allows multiple points to be added, each of which is called a Steiner Point, creating a tree instead of a star.

Much is known about the exact solution to the Steiner Minimal Tree problem. Those who wish to learn about some of the spin-off problems are invited to read the introductory article by Bern and Graham [3], the excellent survey paper on this problem by Hwang and Richards [4], or the recent volume in The Annals of Discrete Mathematics devoted completely to Steiner Tree problems [5]. Some of the basic pieces of information about the

Steiner Minimal Tree problem that can be gleaned from these articles are: (i) the fact that all of the original n points will be of degree 1, 2, or 3, (ii) the Steiner Points are all of degree 3, (iii) any two edges meet at an angle of at least 120° in the Steiner Minimal Tree, and (iv) at most $n-2$ Steiner Points will be added to the network.

This research concentrates on the Steiner Minimal Tree problem, henceforth referred to as the SMT problem. We present several algorithms for calculating Steiner Minimal Trees, including the first parallel algorithm for doing so. Several implementation issues are discussed, some new results are presented, and several ideas for future work are proposed.

In Chapter II we review the first fundamental algorithm for calculating SMT's and we present a simple, effective heuristic which achieves some quite remarkable results. In Chapter III problem decomposition for SMT's is outlined, and a new parallel algorithm for this decomposition is proposed. In Chapter IV we present and discuss the implementation of a parallel algorithm which serves as the basis for the solution to the entire problem. In Chapter V the problem of extracting the correct answer from a list of potential candidates is discussed. Results are presented in Chapter VI, with conclusions and future work following in Chapter VII.

CHAPTER II

INITIAL RESULTS

The First Solution

A typical problem-solving approach is to begin with the simple cases and expand to a general solution. As we saw in Chapter I, the trivial three point problem had already been solved in the 1600's, so all that remained was the work toward a general solution. As with many interesting problems this is harder than it appears on the surface.

The method proposed by the mathematicians of the mid-seventeenth century for the three point problem is illustrated in Figure 1. This method stated that in order to calculate the Steiner Point given points A, B, and C, you first construct an equilateral triangle (ACX) using the longest edge between two of the points (AC) such that the third (B) lies outside the triangle. A circle is circumscribed around the triangle, and a line is constructed from the third point (B) to the far vertex of the triangle (X). The location of the Steiner Point (P) is the intersection of this line (BX) with the circle.

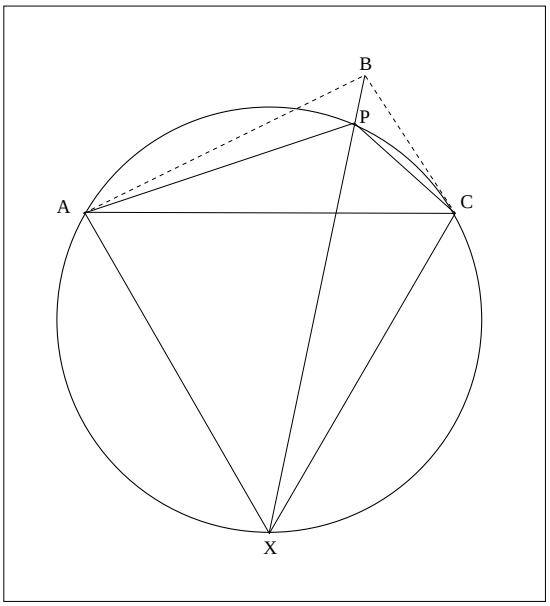

Figure 1. AP + CP = PX.

For the next thirty years after Kössler and Jarník presented the general form of the SMT problem, the only algorithms that existed were heuristics. The heuristics were typically based upon the Minimum-Length Spanning Tree (MST), which is a tree that spanns or connects all vertices whose sum of the edge lengths is as small as possible, and tried in various ways to join three vertices with a Steiner Point. In 1968 Gilbert and Pollak [6] linked the length of the SMT to the length of a MST. It was already known that the length of an MST is an upper bound for the length of an SMT, but their conjecture stated that the length of an SMT would never be any shorter than $\frac{\sqrt{3}}{2}$ times the length of an MST. This conjecture, was recently proved [7], and has led to the MST being the starting point for most of the heuristics that have been proposed in the last 20 years. This combination approach, starting with the MST, will serve as the basis for the heuristic we present later in this chapter.

In 1961 Melzak developed the first algorithm for calculating an SMT [8]. Melzak's Algorithm was geometric in nature and was based upon some simple extensions to Figure 1. The insight that Melzak offered was the fact that you can reduce an n point problem to a set of $n-1$ point problems. This reduction in size is accomplished by taking every pair of points, A and C in our example, calculating where the two possible points, X_1 and X_2, would be that form an equilateral triangle with them, and creating two smaller problems, one where X_1 replaces A and C, and the other where X_2 replaces A and C. Both Melzak and Cockayne pointed out however that some of these sub-problems are invalid. Melzak's algorithm can then be run on the two smaller problems. This recursion, based upon replacing two points with one point, finally terminates when you reduce the problem from three to two vertices. At this termination the length of the tree will be the length of the line segment connecting the final two points. This is due to the fact that $BP + AP + CP = BP + PX$. This is straightforward to prove using the law of cosines, for when P is on the circle, $\angle APX = \angle CPX = 60°$. This allows the calculation of the last Steiner Point (P) and allows you to back up the recursive call stack to calculate where each Steiner Point in that particular tree is located.

This reduction is important in the calculation of an SMT, but the algorithm still has exponential order, since it requires looking at every possible reduction of a pair of points to

a single point. The recurrence relation for an n-point problem is stated quite simply in the following formula:

$$T(n) = 2 * \left(\begin{array}{c} n \\ 2 \end{array} \right) * T(n-1).$$

This yields what is obviously a non-polynomial time algorithm. In fact Garey, Graham, and Johnson [9] have shown that the Steiner Minimal Tree problem is NP-Hard (NP-Complete if the distances are rounded up to discrete values).

In 1967, just a few years after Melzak's paper, Cockayne [10] clarified some of the details from Melzak's proof. This clarified algorithm proved to be the basis for the first computer program to calculate SMTs. The program was developed by Cockayne and Schiller [11] and could compute an SMT for any placement of up to 7 vertices.

A Proposed Heuristic

Background and Motivation

By exploring a structural similarity between *stochastic Petri nets* (see [12] and [13]) and *Hopfield neural nets* (see [14] and [15]), Geist was able to propose and take part in the development of a new computational approach for attacking large, graph-based optimization problems. Successful applications of this mechanism include I/O subsystem performance enhancement through disk cylinder remapping [16, 17], file assignment in a distributed network to reduce disk access conflict [18], and new computer graphics techniques for digital halftoning [19] and color quantization [20]. The mechanism is based on maximum-entropy Gibbs measures, which is described in Reynold's dissertation [21], and provides a natural equivalence between Hopfield nets and the *simulated annealing* paradigm. This similarity allows you to select the method that best matches the problem at hand. For the SMT problem we will implement the Simulated Annealing approach.

Simulated Annealing [22] is a probabilistic algorithm that has been applied to many optimization problems in which the set of feasible solutions is so large that an exhaustive search for an optimum solution is out of the question. Although Simulated Annealing does not necessarily provide an optimum solution, it usually provides a good solution in a user-selected amount of time. Hwang and Richards [4] have shown that the optimal placement

of s Steiner Points to n original vertices yields a feasible solution space of the size

$$2^{-n} \begin{pmatrix} n \\ s+2 \end{pmatrix} \frac{(n-s-2)!}{s!}$$

provided that none of the original points have degree 3 in the SMT. If the degree restriction is removed they showed that the number is even larger. The SMT problem is therefore a good candidate for this approach.

Adding One Junction

Georgakopoulos and Papadimitriou [23] have provided an $\mathcal{O}(n^2)$ solution to the *1-Steiner problem*, wherein exactly one Steiner Point is added to the original set of points. Since at most $n-2$ Steiner Points are needed in an SMT solution, repeated application of the algorithm offers a "greedy" $\mathcal{O}(n^3)$ approach. Using their method, the first Steiner Point is selected by partitioning the plane into Oriented Dirichlet Cells, which they describe in detail. Since these cells do not need to be discarded and recalculated for each addition, subsequent additions can be accomplished in linear time. Deletion of a candidate Steiner Point requires regeneration of the MST, which Shamos showed can be accomplished in $\mathcal{O}(n \; log \; n)$ time if the points are in the plane [24], followed by the cost for a first addition ($\mathcal{O}(n^2)$). This approach can be regarded as a natural starting point for Simulated Annealing by adding and deleting different Steiner Points.

The Heuristic

The Georgakopoulos and Papadimitriou 1-Steiner algorithm and the Shamos MST algorithm are both difficult to implement. As a result, we have chosen to investigate the potential effectiveness of this Annealing Algorithm using a more direct, but slightly more expensive $\mathcal{O}(n^3)$ approach. As previously noted, all Steiner Points have degree 3 with edges meeting in angles of 120°. We consider all $\binom{n}{3}$ triples where the largest angle is less than 120°, compute the Steiner Point for each (a simple geometric construction), select that Steiner Point giving greatest reduction, or least increase in the length of the modified tree (increases are allowed since the Annealing Algorithm may go uphill) and update the MST accordingly. Again, only the first addition requires this (now $\mathcal{O}(n^3)$) step. We use the straightforward, $\mathcal{O}(n^2)$ Prim's algorithm to generate the MST initially and after each deletion of a Steiner Point.

The Annealing Algorithm can be described as a non-deterministic walk on a surface. The points on the surface correspond to the lengths of all feasible solutions, where two solutions are adjacent if they can be reached through the addition or deletion of one Steiner Point. The probability of going uphill on this surface is higher when the temperature is higher but decreases as the temperature cools. The rate of this cooling typically will determine how good your solution will be. The major portion of this algorithm is presented in Figure 2. This non-deterministic walk, starting with the MST has led to some very exciting results.

Results

Before we discuss large problems, a simple introduction into the results from a simple six point problem is in order. The Annealing Algorithm is given the coordinates for six points: (0,0), (0,1), (2,0), (2,1), (4,0), (4,1). The first step is to calculate the MST, which has a length of 7, as shown in Figure 3. The output of the Annealing Algorithm for this simple problem is shown in Figure 4. In this case the Annealing Algorithm calculates the exact SMT solution which has a length of 6.616994.

We propose as a measure of accuracy the percentage of the difference between the length of the MST and the exact SMT solution that the Annealing Algorithm achieves. This is a new measure which has not been discussed (or used) because exact solutions have not been calculated for anything but the most simple layouts of points. For the six point problem discussed above this percentage is 100.0% (the exact solution is obtained).

After communicating with Cockayne, data sets were obtained for exact solutions to randomly generated 100 point problems that were developed for [25]. This allows us to use the measure of accuracy previously described. Results for some of these data sets provided by Cockayne are shown in Table 1.

An interesting aspect of the Annealing Algorithm that cannot be shown in the table is the comparison of execution times with Cockayne's program. Whereas Cockayne mentioned that his results had an execution cut-off of 12 hours, these results were obtained in less than 1 hour. The graphical output for the first line of the table, which reaches over 96% of the optimal value, appears as follows: the data points and the MST are shown in Figure 5,

```
#define EQUILIBRIUM ((accepts>=100 AND rejects>=200) OR
     (accepts+rejects > 500))
#define FROZEN ((temperature < 0.5) OR ((temperature < 1.0)
     AND (accepts==0)))

while(not(FROZEN)){
     accepts = rejects = 0;
     old_energy = energy();
     while(not(EQUILIBRIUM)){
          operation = add_or_delete();
          switch(operation){
               case ADD:
                    ΔE = energy_change_from_adding_a_node();
                    break;
               case DELETE:
                    ΔE = energy_change_from_deleting_a_node();
                    break;
          }
          if(rand(0,1) < e^{min{0.0,-ΔE/temperature}}){
               accepts++;
               old_energy = new_energy;
          }else {
               /* put them back */
               undo_change(operation);
               rejects++;
          }
     }
     temperature = temperature*0.8;
}
```

Figure 2. Simulated Annealing algorithm.

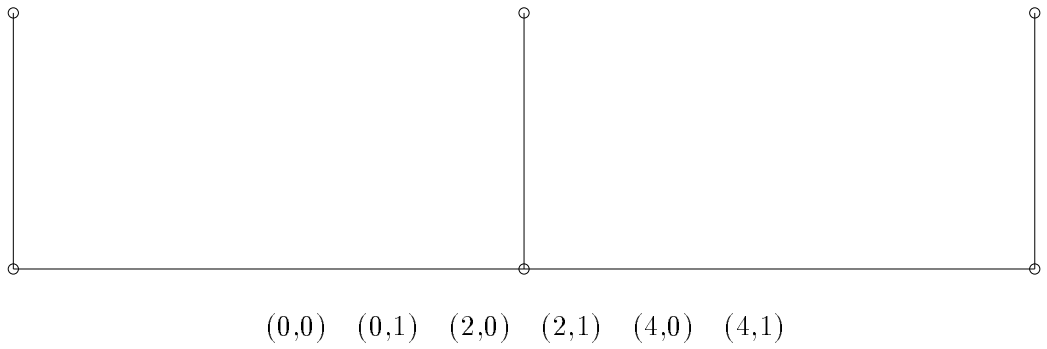

Figure 3. Spanning tree for 6 point problem.

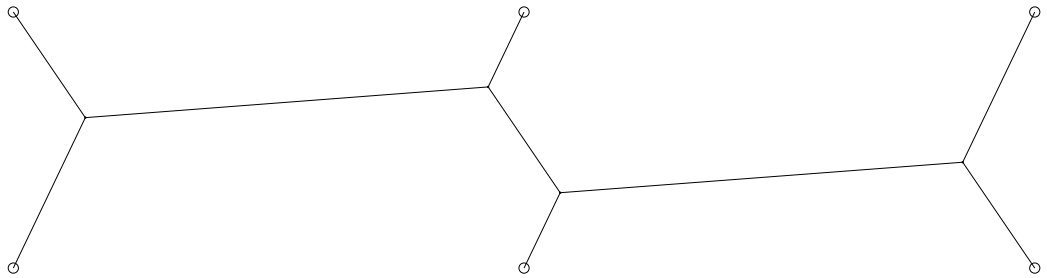

Figure 4. 6 point solution.

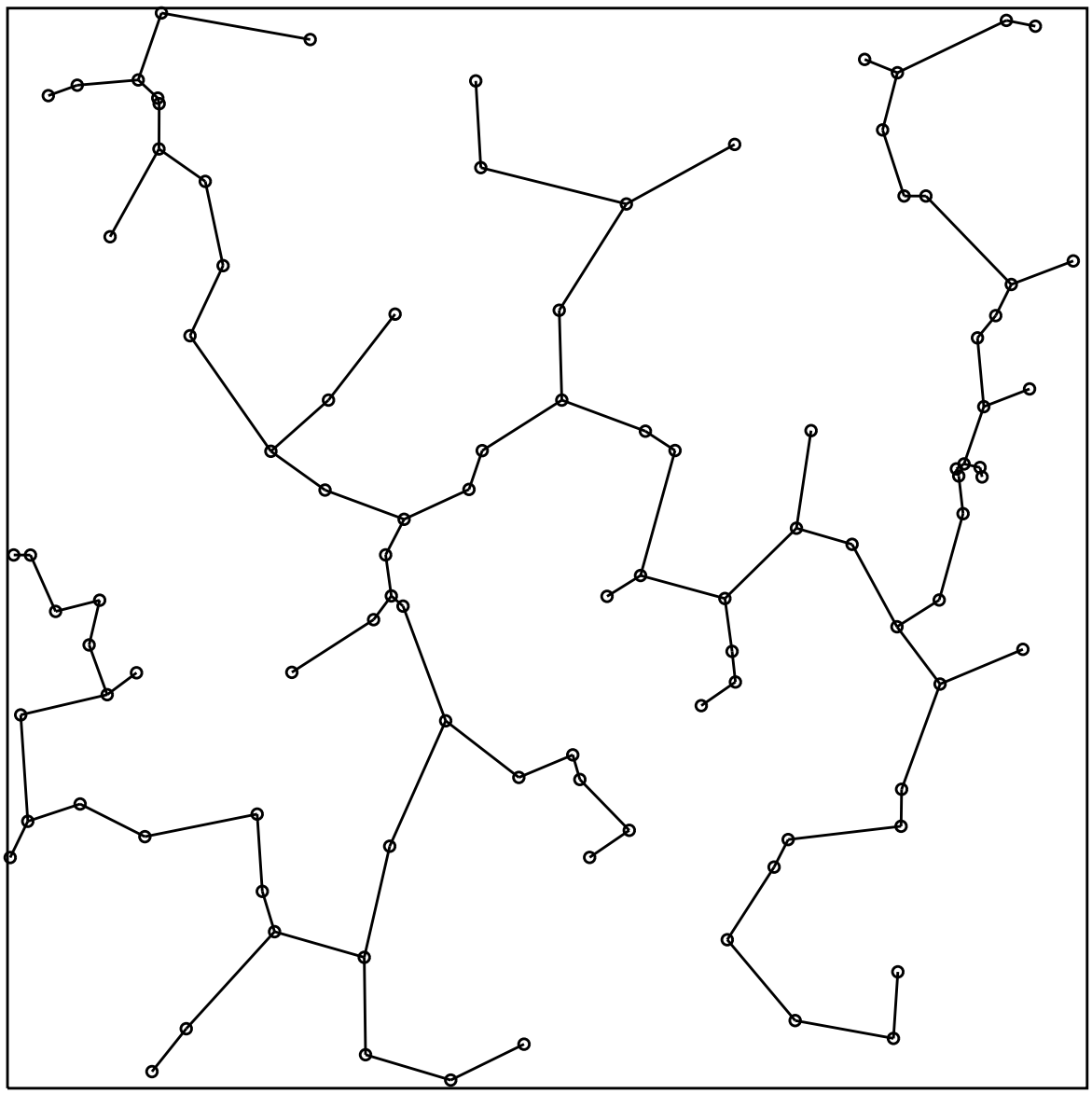

Figure 5. Spanning tree.

the Simulated Annealing Result is in Figure 6, and the Exact SMT Solution is in Figure 7. The solution presented here is obtained in less than $\frac{1}{10}th$ of the time with less than 4% of the possible range not covered. This indicates that we could hope to extend our Annealing Algorithm to much larger problems, perhaps as large as $1,000$ points. If we were to extend this approach to larger problems we would definitely need to implement the Georgakopoulos-Papadimitriou 1-Steiner Algorithm and the Shamos MST Algorithm.

Table 1. Results from 100 point problems.

Exact Solution	Spanning Tree	Simulated Annealing	Percent Covered
6.255463	6.448690	6.261797	96.39%
6.759661	6.935189	6.763495	98.29%
6.667217	6.923836	6.675194	96.89%
6.719102	6.921413	6.721283	99.01%
6.759659	6.935187	6.763493	98.29%
6.285690	6.484320	6.289342	98.48%

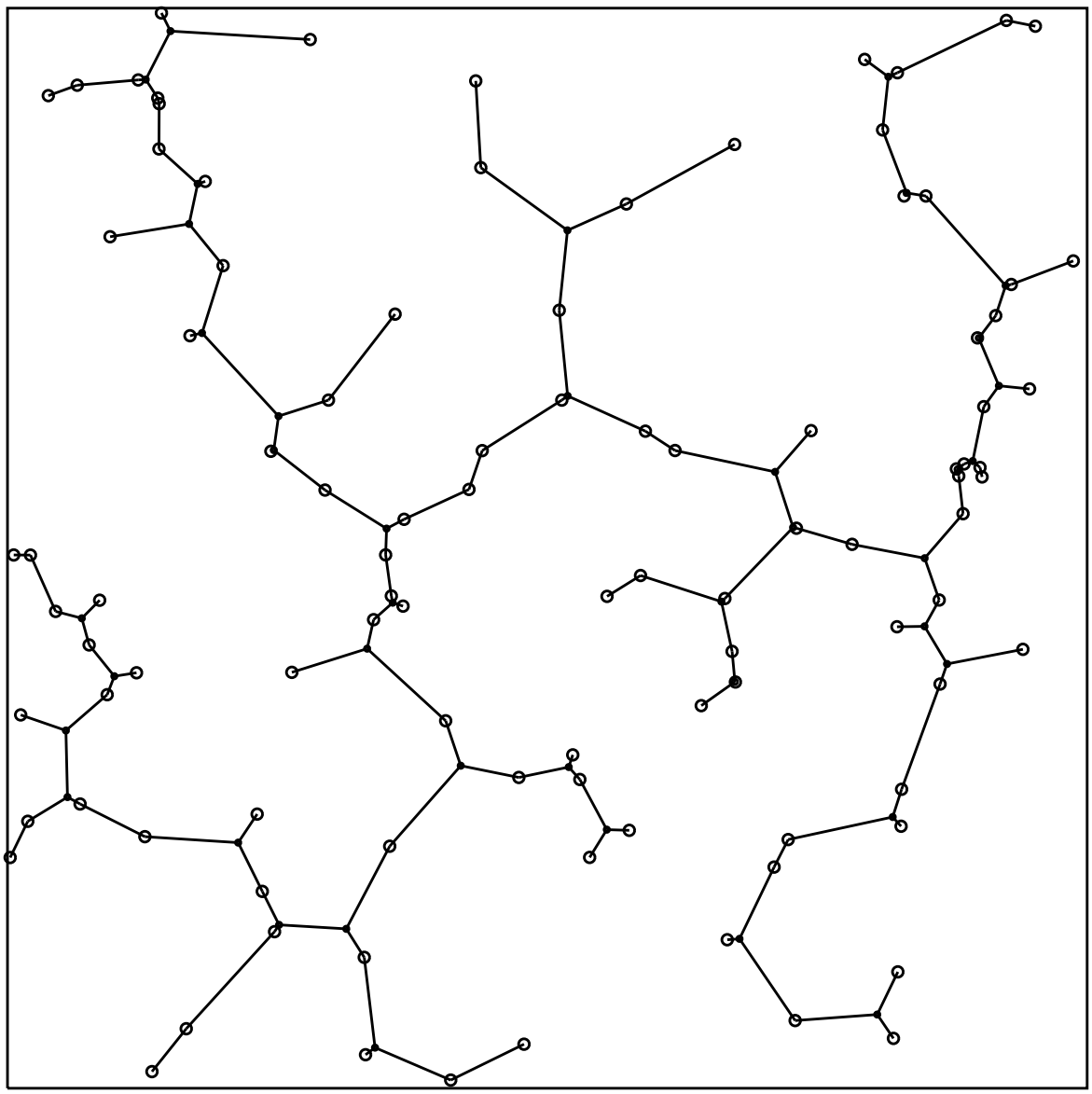

Figure 6. Simulated Annealing solution.

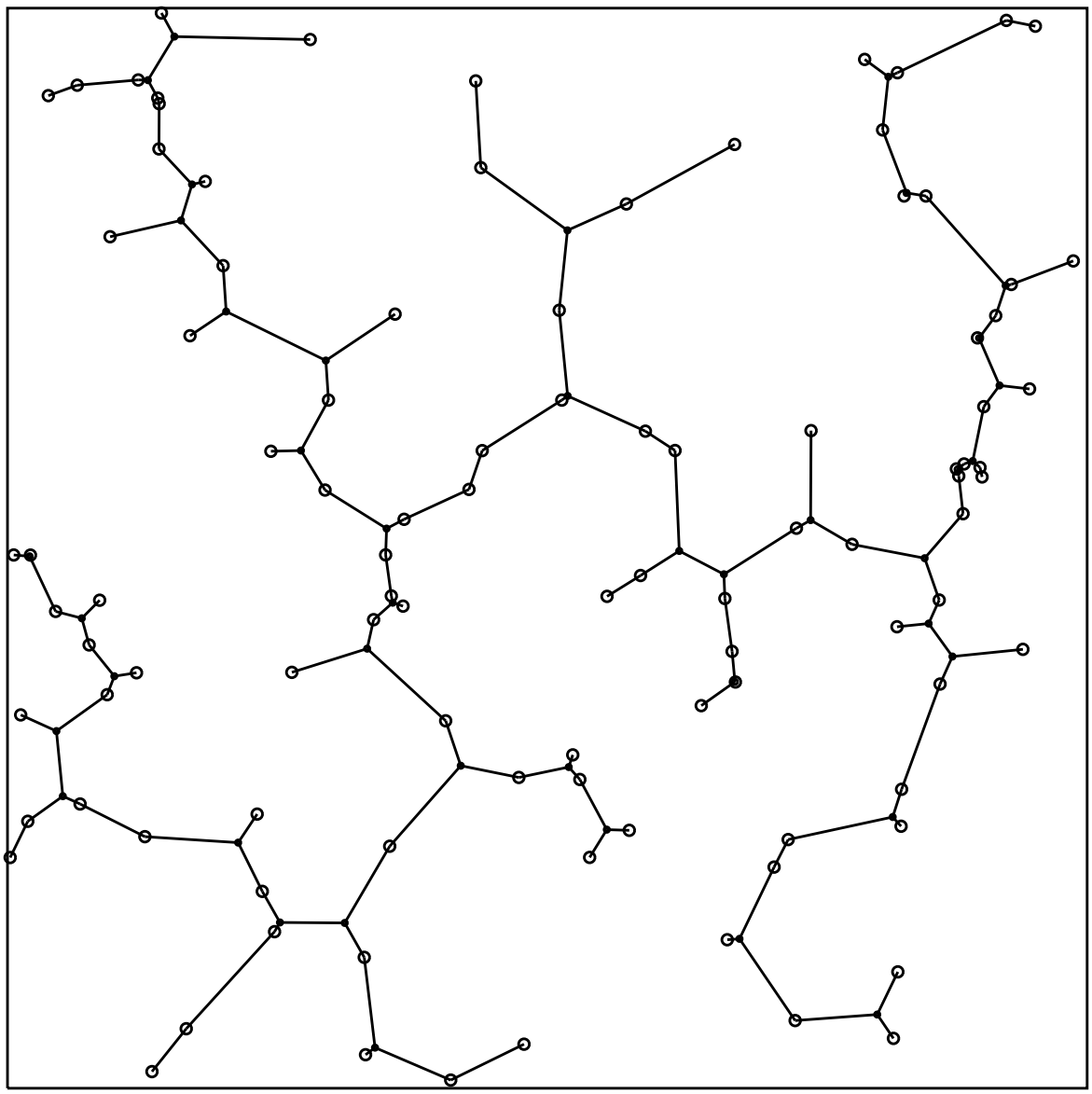

Figure 7. Exact solution.

CHAPTER III

PROBLEM DECOMPOSITION

Motivation and Literature

After the early work by Melzak [8], many people began to work on the Steiner Minimal Tree problem. The first major effort was to find some kind of geometric bound for the problem. In 1968 Gilbert and Pollak [6] showed that the SMT for a set of points, $\mathcal{S}$, must lie within the Convex Hull of $\mathcal{S}$. This bound has since served as the starting point for every bounds enhancement for SMT's.

As a brief review, the Convex Hull is defined as follows: Given a set of points $\mathcal{S}$ in the plane, the Convex Hull is the convex polygon of smallest area containing all the points of $\mathcal{S}$. A polygon is defined to be convex if a line segment connecting any two points inside the polygon lies entirely within the polygon. An example of the Convex Hull for a set of 100 randomly generated points is shown in Figure 8.

Shamos in his PhD thesis [26] proposed a Divide and Conquer Algorithm which has served as the basis for many parallel algorithms calculating the Convex Hull. One of the first such approaches appeared in the PhD thesis by Chow [27]. This approach was refined and made to run in optimal $\mathcal{O}(log\ n)$ time by Aggarwal et.al. [28], and Attalah and Goodrich [29].

The next major work on the SMT problem was in the area of problem decomposition. As with any non-polynomial algorithm, the most important theorems are those that say "If property $\mathcal{P}$ exists, then the problem may be split into the following sub-problems." For the Steiner Minimal Tree Problem property $\mathcal{P}$ will probably be geometric in nature. Unfortunately, decomposition theorems have been few and far between for the SMT problem. In fact, at this writing there have been only three such theorems.

The Double Wedge Theorem

The first decomposition theorem, known as the Double Wedge Theorem, was proposed by Gilbert and Pollak [6]. This is illustrated in Figure 9 and can be summarized quite

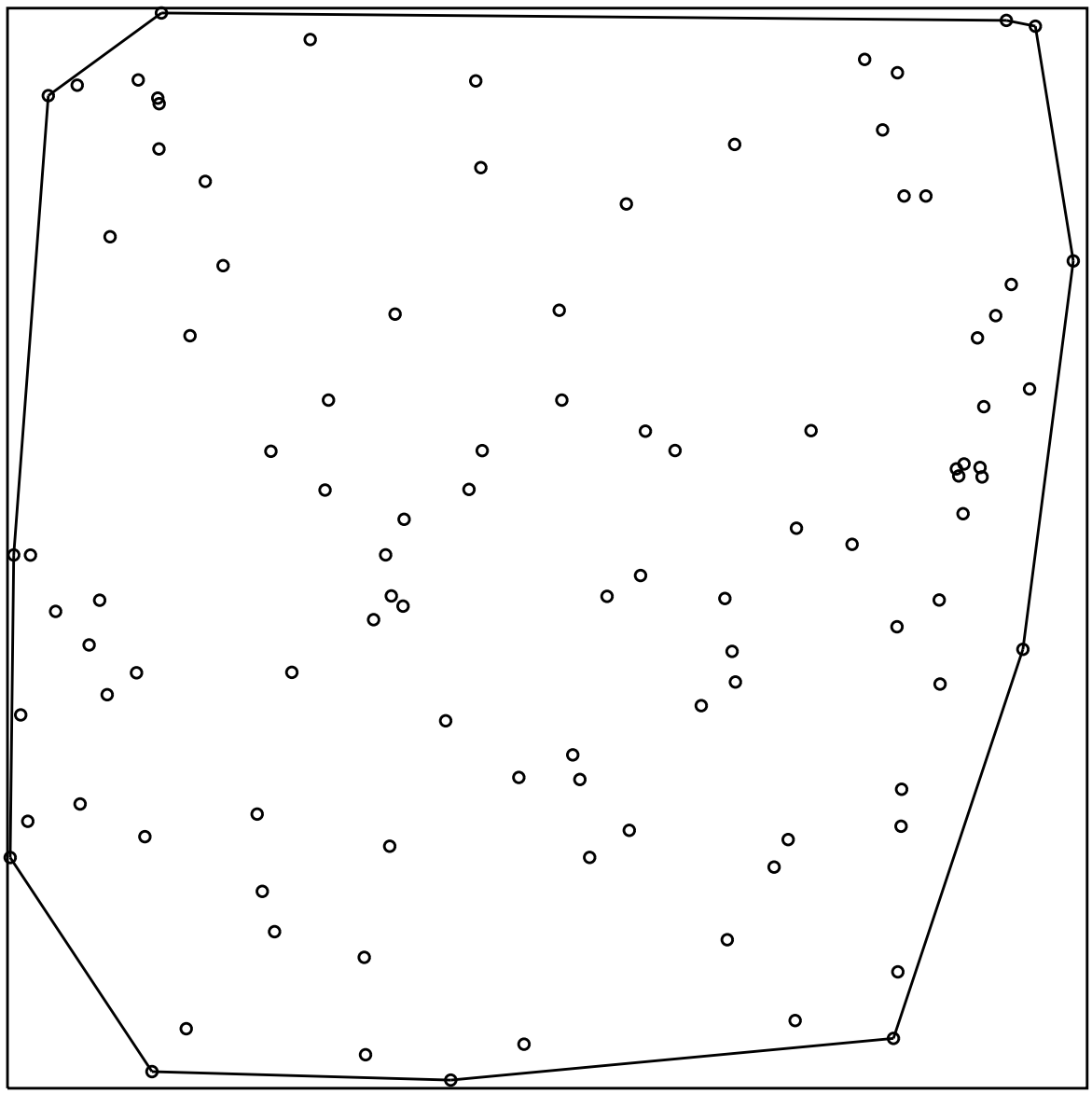

Figure 8. The Convex Hull for a random set of points.

simply as follows: If two lines intersect at point $\mathcal{X}$ and meet at 120°, they split the plane into two 120° wedges and two 60° wedges. If R_1 and R_2 denote the two 60° wedges, and all the points of $\mathcal{S}$ are contained in $R_1 \bigcup R_2$, then the problem can be decomposed. There are two cases to be considered. In case 1 $\mathcal{X}$ is not a point in $\mathcal{S}$; therefore, the Steiner Minimal Tree for $\mathcal{S}$ is the SMT for R_1, the SMT for R_2, and the shortest edge connecting the two trees. In case 2 $\mathcal{X}$ is a point in $\mathcal{S}$; therefore, the Steiner Minimal Tree for $\mathcal{S}$ is the SMT for R_1 and the SMT for R_2. Since $\mathcal{X}$ is in both R_1 and R_2 the two trees are connected.

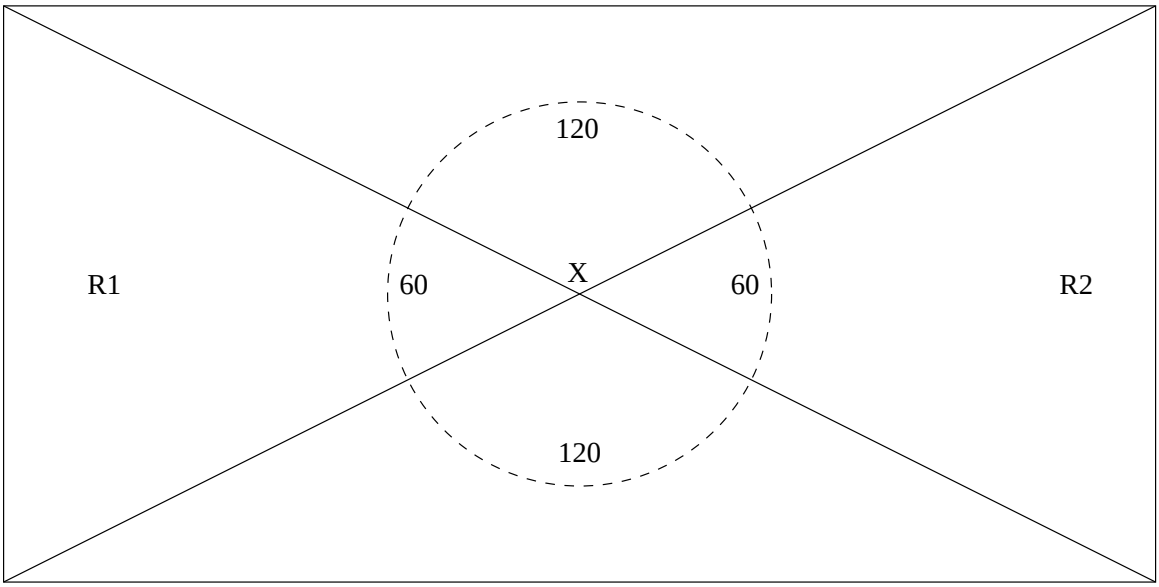

Figure 9. An illustration of the Double Wedge.

The Steiner Hull

The next decomposition theorem is due to Cockayne [30] and is based upon what he termed the *Steiner Hull*. The Steiner Hull is defined as follows: let P_1 be the convex Hull. P_{i+1} is constructed from P_i by finding an edge (p, r) of P_i that has a vertex (q) near it such that $\angle pqr \geq 120°$ and there is not a vertex inside the triangle pqr. The final polygon, P_i, that can be created in such a was is called the Steiner Hull. The Algorithm for this construction is shown in Figure 10. The Steiner Hull for the 100 points shown in Figure 8 is given in Figure 11.

The initial Steiner Polygon, P_1, is the Convex Hull.
Repeat
    Create Next Steiner Polygon P_{i+1} from P_i by
        1) find a set of points $pqr \in \mathcal{S}$ such that:
            p and r are adjacent on P_i
            $\angle pqr \geq 120^\circ$
            $\nexists$ a point from $\mathcal{S}$ in the triangle pqr
        2) remove the edge pr.
        3) add edges pq and qr.
Until($P_i == P_{i+1}$)
Steiner Hull = P_i

Figure 10. The Steiner Hull algorithm.

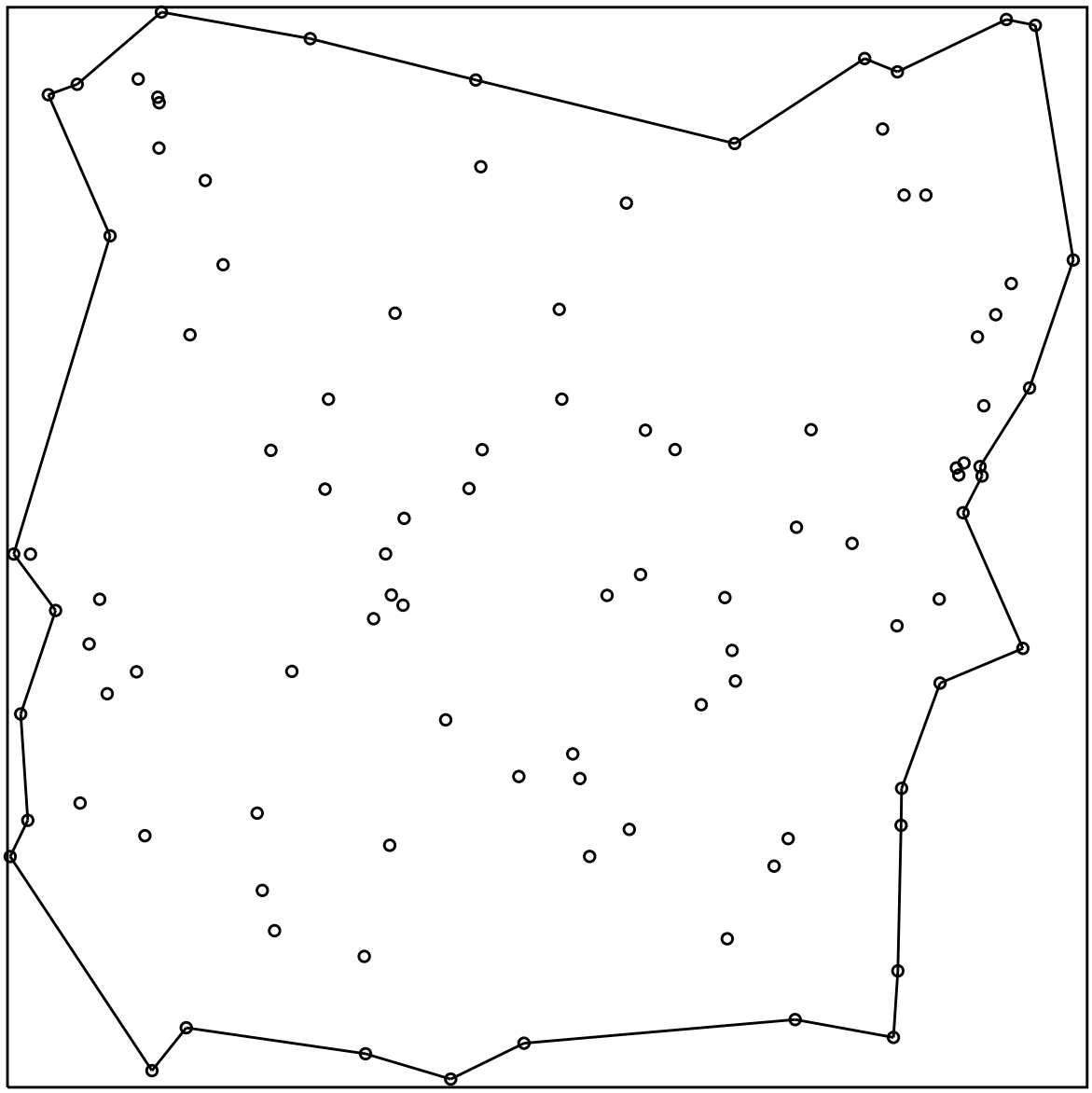

Figure 11. The Steiner Hull for a random set of 100 points.

After defining the Steiner Hull, Cockayne showed that the SMT for $\mathcal{S}$ must lie within the Steiner Hull of $\mathcal{S}$. This presents us with the following decomposition: The Steiner Hull can be thought of as an ordered sequence of points, $\{p_1, p_2, \ldots, p_n\}$, where the hull is defined by the sequence of line segments, $\{p_1p_2, p_2p_3, \ldots, p_np_1\}$. If there exists a point p_i that occurs twice in the Steiner Hull, then the problem can be decomposed at point p_i. If a Steiner Hull contains such a point, then the Steiner Hull is referred to as *degenerate*. This decomposition is accomplished by showing that the Steiner Hull splits $\mathcal{S}$ into two contained subsets, R_1 and R_2, where R_1 is the set of points contained in the Steiner Hull from the first time p_i appears until the last time p_i appears, and R_2 is the set of points contained in the Steiner Hull from the last time p_i appears until the first time p_i appears. With this decomposition it can be shown that $\mathcal{S} = R_1 \bigcup R_2$, and the SMT for S is the union of the SMT for R_1 and the SMT for R_2. This decomposition is illustrated in Figure 12. Cockayne also proved that the Steiner Hull decomposition includes every decomposition possible with the Double Wedge Theorem.

In their work on 100 point problems, Cockayne and Hewgill [25] mention that approximately 15% of the randomly generated 100 point problems have degenerate Steiner Hull's. The Steiner Hull shown in Figure 11 is not degenerate, while that in Figure 12 is.

The Steiner Hull Extension

The final decomposition belongs to Hwang, Song, Ting and Du [31]. They proposed an extension to the Steiner Hull as defined by Cockayne. Their extension is as follows:
If there exist four points a, b, c, d on a Steiner Hull such that:

1. a, b, c, d form a convex quadrilateral,
2. there does not exist a point from $\mathcal{S}$ in the quadrilateral (a, b, c, d),
3. $\angle a \geq 120°$ and $\angle b \geq 120°$,
4. the two diagonals (ac) and (bd) meet at O, and $\angle bOa \geq \angle a + \angle b - 150°$,

then the SMT for $\mathcal{S}$ is the union of the SMT's for R_1 and R_2 and the edge ab where R_1 is the set of points contained in the Steiner Hull from c to b with the edge bc, and R_2 is the set of points contained in the Steiner Polygon from a to d with the edge ad. This decomposition is illustrated in Figure 13. These three decomposition theorems will be combined in the parallel algorithm for decomposition presented next.

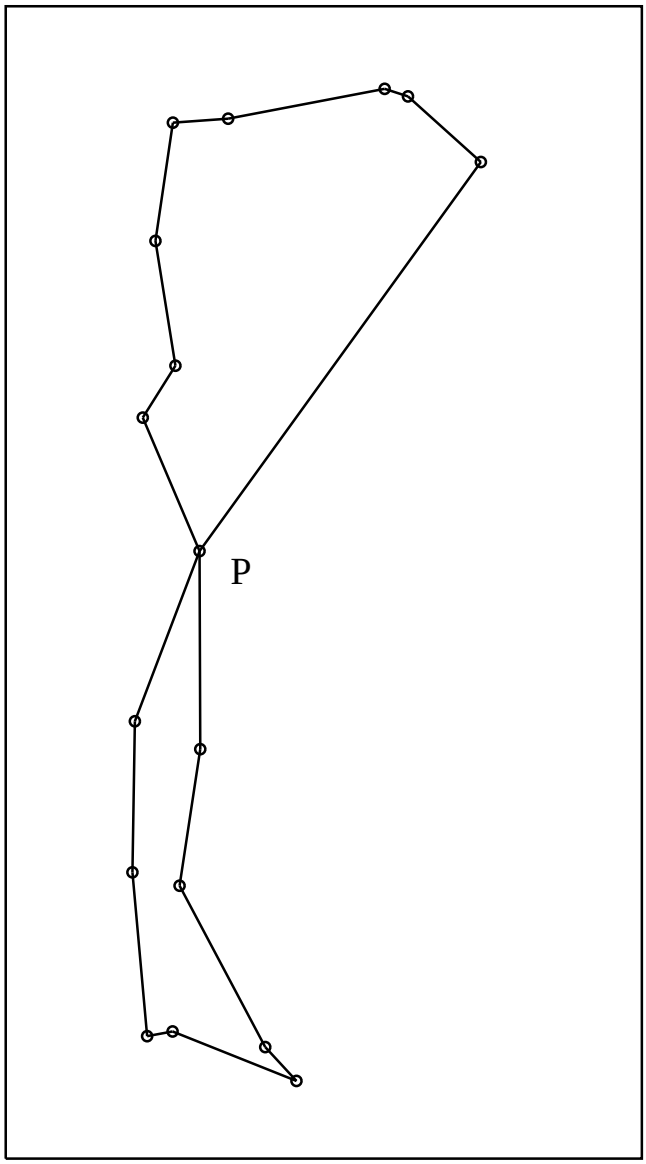

Figure 12. An illustration of the Steiner Hull decomposition.

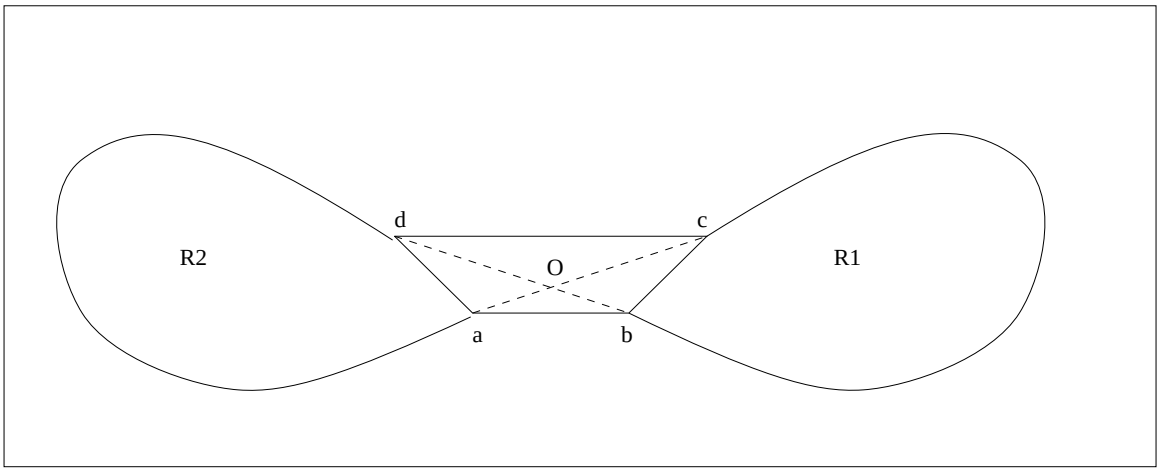

Figure 13. An illustration of the Steiner Hull extension.

A Parallel Algorithm to Calculate the Steiner Hull

A Introduction to Parallelism

Parallel computation is allowing us to look at problems that have previously been impossible to calculate, as well as allowing us to calculate faster than ever before problems we have looked at for a long time. It is with this in mind that we begin to look at a parallel algorithm for the decomposition of the Steiner Minimal Tree problem.

There have been volumes written on parallel computation and parallel algorithms; therefore, the author will not rehash the material that has already been so excellently covered by many others more knowledgeable on the topic, but will refer the interested readers to various books currently available. For a thorough description of parallel algorithms, and the PRAM Model the reader is referred to the book by Joseph JáJá [32], and for a more practical approach to implementation on a parallel machine the reader is referred to the book by Vipin Kumar et.al. [33] or the book by Justin Smith [34].

The Parallel Algorithm

The algorithm presented here is designed to operate on a CREW PRAM machine, which is a shared memory machine which allows concurrent reads from memory but allows only one processor to write to a specific location at a time. Other models allow concurrent

writing to the same location, but this requires an arbitration scheme for what is being written.

This algorithm starts with its input being the Convex Hull for $\mathcal{S}$. The vertices on the Convex Hull are placed in an array called *Edges*. This array is made up of structures, where each structure contains an integer (the vertex number) and two pointers that form a doubly linked list through the ordered entries. Initially the linked list goes straight through the edges of the Convex Hull. Two counters are kept for each round. The first counter is named *Previous_count* and is initialized to 0. This counter keeps track of the count of the beginning vertex of the first edge alive on this round. The second counter is named *Current_count* and it is initialized to the number of edges in the Convex Hull. This counter keeps track of where new edges created this round will begin to be placed. Processor i is assigned to the edge defined by vertices $i, i+1$ and the calculation begins. Each processor determines whether the edge it was assigned is to be replaced via the Steiner Hull definition and it's extension. An integer representing the number of edges that will replace this edge (0 – no replacement, 2 – standard Steiner Hull definition, or 3 – Steiner Hull Extension) is placed into *New_Edges*[i]. The prefix sum is then calculated for this array, which results in *New_Edges*[i] = $\sum_{j=0}^{i}$. This value determines the offset into *Edges* where the new edges are to be placed. Processor i places its new edges into the array beginning at $Current_count + New_Edges[i-1]$. Processor i then updates the doubly linked list through *Edges*, starting with the previous link pointing to edge i, updating the list through the new edges, and then updating the next link at i. The processor assigned to edge *Current_count* - 1 then does the end of round updating: $Previous_count = Current_count$, $Current_count = Current_count + New_Edges[i]$. For the next round the processors are assigned the edges from *Previous_count* to *Current_count*, and the calculation begins again. This continues until $Previous_count == Current_count$.

When analyzing this algorithm, whose code is given in Figure 14, There is a case where only one edge will find a new set of edges to add to the current Steiner Polygon. This makes the algorithm end up with $\mathcal{O}(n)$ rounds of computation. In practice however, for the 100 point point problems we solved here, we never saw the algorithm go past 4 rounds of computation.

```
Parallel_Steiner_Hull()
     Input:  The Convex Hull for S
     Output:  The Steiner Hull for S
{
     int i,k;

     Previous_count = 0;
     Current_count = Convex_Hull.size;
     par_for(i = 0; i < Current_count; i++) {
          Edges[i] = Convex_Hull.edges[i];
     }
     do {
          par_for(i = Previous_count; i < Current_count; i++){
               int j;
               j = i - Previous_count;
               New_Edges[j] = Number_of_new_edges(i);
               P_S[j] = Prefix_Sum(j);
               for(k=0; k < New_Edges[j].count; k++){
                    Edges[Current_count + P_S[j-1] + k] =
                         New_Edges[j].edges[k];
               }
               Update_Links(i,Current_count + P_S[j-1], k);
               if(i == Current_count -1){
                    Previous_count = Current_count;
                    Current_count = Current_count + P_S[j];
               }
          }
     } while(Previous_count != Current_count);
}
```

Figure 14. A parallel Steiner Hull algorithm.

CHAPTER IV

GENERATION OF A LIST OF POSSIBLE ANSWERS

Winter's Sequential Algorithm

Overview and Significance

The development of the first working implementation of Melzak's algorithm sparked a move into the computerized arena for the calculation of SMT's. As we saw earlier Cockayne and Schiller [11] had implemented Melzak's Algorithm and could calculate the SMT for all arrangements of 7 points. This was followed almost immediately by Boyce and Seery's program which they called STEINER72 [35]. Their work, done at Bell Labs could calculate the SMT for all 10 point problems. They continued to work on the problem and in personal communication with Cockayne said they could solve 12 point problems with STEINER73. Yet even with quite a few people working on the problem, the number of points that any program could handle was still very small.

As mentioned in Chapter II, Melzak's algorithm yields invalid answers and invalid tree structures for quite a few combinations of points. It was not until 1981 that anyone was able to characterize a few of these invalid tree structures. These characterizations were accomplished by Pawel Winter and were based upon several geometric constructions which enable one to eliminate many of the possible combinations previously generated. He implemented these improvements in a program called GeoSteiner [36]. In his work he was able to calculate in under 30 seconds SMT's for problems having up to 15 vertices and stated that "with further improvements, it is reasonable to assert that point sets up to 30 V-points could be solved in less than an hour [36]."

Winter's Algorithm

Winter's breakthrough was based upon two things: the use of extended binary trees, and what he termed *pushing*. Winter proposed an extended binary tree as a means of constructing trees only once and easily identifying a Full Steiner Tree (FST: trees with n vertices and $n-2$ Steiner Points) on the same set of vertices readily.

Pushing came from the geometric nature of the problem and is illustrated in Figure 15. It was previously known that the Steiner Point for a pair of points, a and b, would lie on the circle that circumscribed that pair and their equilateral third point. Winter set out to limit this region even further. This limitation was accomplished by placing a pair of points, a' and b', on the circle at a and b respectively, and attempting to push them closer and closer together. In his work Winter proposed and proved various geometric properties that would allow you to push a' towards b and b' towards a. If the two points ever crossed then it was impossible for the current branch of the sample space tree to contain a valid answer.

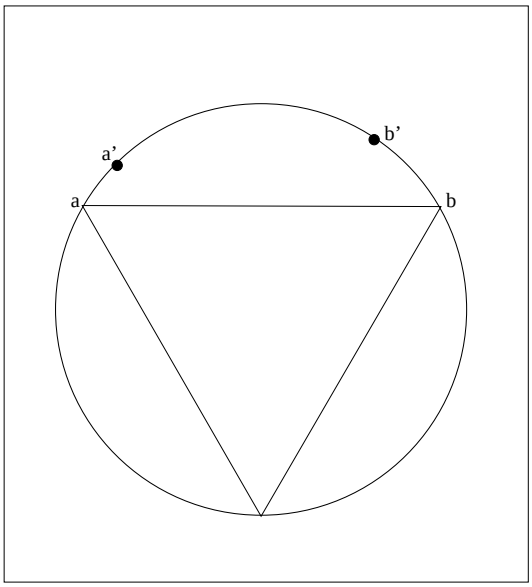

Figure 15. An illustration of Winter's pushing.

Unfortunately, the description of Winter's algorithm is not as clear as one would hope, since the presence of `goto` statements rapidly makes his program difficult to understand, and almost impossible to modify. The initialization, which is given in Figure 16, is basically quite plain in its description. Winter's goal is to build a list of FST's which are candidates for inclusion in the final answer. This list, called T_list, is primed with the edges of the MST, thereby guaranteeing that the length of the SMT does not exceed the length of the MST.

The rest of the algorithm sets about to expand what Winter termed as Q_list, which is a list of partial trees that the algorithm attempts to combine until no combinations are possible. Q_list is primed with the original input points. The legality of a combination is determined in the *Construct* procedure, which uses *pushing* to eliminate cases. While this combination proceeds, the algorithm also attempts to take newly created members of Q_list and create valid FST's out of them. These FST's are then placed onto T_list. Winter's Algorithm appears in Figure 17, and a brief description of the terms used is provided in Figure 18.

This algorithm was a turning point in the calculation of SMT's. It has since served as the basis for every algorithm for calculating SMT's and is the foundation for the parallel algorithm we present next.

```
1 Step 1 /INITIALIZATION/:  Construct SP of V; Construct MST for V;
2    For every V-point q = 1,...,n, let
3         LF(q) := q; H(q) := MIN(q) := 0;
4         E(q) := coordinates of q;
5         LSP(q) := RSP(q) := if q ∈ SP then q else 0;
6         L(q) := R(q) := NIL;
7    For every edge i_t j_t, t = 1, ..., n-1, of MST let
8         SROOT(t) := i_t;ROOT(q) := j_t
9    q := n+1; t := n; p := 0;
```

Figure 16. Step 1 of Winter's algorithm.

A Parallel Algorithm

Overview and Proper Structure

When attempting to construct a parallel algorithm for a problem the sequential code for that problem is often the starting point. In examining sequential code, major levels of parallelism may become self-evident. Therefore for this problem the first thing to do is to look at Winter's algorithm and convert it into structured code without `goto`s. This

```
1 STEP 2 /NEW LEFT SUBTREE/:  p := p + 1;
2    If p = q then go to STEP 7;
3    r := 0; AP := A(p);

4 STEP 3 /NEW RIGHT SUBTREE/:  R := r + 1;
5    If H(p) < H(r) or r = q then go to STEP 2;
6    If H(p) = H(r) and r < p then r := p;
7    If V(r) ⊈ AP then go to STEP 3
8    p* := p; r* := r; LABEL := "+";

9 STEP 4 /E-POINT [p*, r*]/:  AQ := A(q);
10   Construct [p*, r*] (subject to tests described in Sections 4 and 5);
11   In case of failure go to STEP 6;
12   L(q) := p; R(q) := r; LBL(q) := LABEL;
13   LF(q) := LF(p); H(q) := H(p) + 1;
14   E(q) := coordinates of [p*, r*];
15   MIN(p) := max{MIN(p)-1, H(r))};
16   q* := q; q := q+1;
17   LSP(q) := if LSP(p) ≠ 0 then LSP(p) else LSP(r);
18   RSP(q) := if RSP(r) ≠ 0 then RSP(r) else RSP(p);

19 STEP 5 /FSTs WITH q* AS ROOT/:
20   If MIN(q* = 0 and |AQ| > 0 and LF(q*) < LF(R(q1) < ...
21   < LF(R(qv)) < LF(R(q*)) where q1,..., qv are central nodes between
22   LF(q*) and q* (H(q1) < ...< H(qy)) with H(q*) - H(qu) = H(R(qu)),
23   u = 1, ...v,
24   then (subject to tests described in Sections 4 and 5)
25   SROOT(t) := j; ROOT(t) := q*;
26   t := t + 1 for every j ∈ AQ with LF(R(q*)) < j;

27 STEP 6 /CHANGE OF LABEL/:  If LABEL = "-" then go to STEP 3;
28   p* := r; r* := p; LABEL := "-"; Go to STEP 4;

29 STEP 7 /DETERMINATION OF SMT/:  Form unions of FSTs saved in TLIST
30   (subject to tests described in Section 6); The shortest union is
31   the SMT for V; TERMINATE;
```

Figure 17. Steps 2 through 7 of Winter's algorithm.

```
COUNTERS
     p, q, r, p*, q*, r*

SETS (subsets of the original vertices)
     AP, AQ

Q_list STRUCTURE DEFINITIONS
     E(p) --   The coordinates of the point p
     H(p) --   The height of the tree with p as the root
     V(p) --   The vertices in the tree with p as the root
     L(p) --   The left child of the node p
     R(p) --   The right child of the node p
     LBL(p) -- The Label associated with node p,
               used to determine which one of two orientations
               the associated Steiner Point would take.
     LF(p) --  The leftmost leaf in the tree with p as the root
     MIN(p) -- The minimum number of nodes needed above p for it
               to be considered for membership in T_list
     LSP(p) -- Left most leaf on the Steiner Polygon of the tree
               with p as the root.
     RSP(p) -- Right most leaf on the Steiner Polygon of the tree
               with p as the root.
T_list STRUCTURE DEFINITIONS
     SROOT(t) -The Top of the Extended Binary Tree (FST)
     ROOT(t) --The child of the SROOT(t)
```

Figure 18. Definitions for terms in Winter's algorithm.

translation of the Initialization (Step 1 of Winter's Algorithm) appears in Figure 19 and the rest (steps 2 through 7) appears in Figure 20.

```
/* Step 1 */
1 Construct_Steiner_Polygon();
2 Construct_Minimal_Spanning_Tree();
3 for(i=0; i<n; i++){ /* for all fixed points */
4    LF(q) = q;
5    H(q) = MIN(q) = 0;
6    if(Member_of_Steiner_Polygon(q))
7         LSP(q) = RSP(q) = q;
8    else
9         LSP(q) = RSP(q) = 0;
10   L(q) = R(q) = NULL;
11 }
12 for(t = 0; t < n − 1; t++){ /* for all edges in MST */
13   Sroot(t) = edge_i(t);
14   Root(t) = edge_j(t);
15 }
16 q = n;
17 t = n-1;
```

Figure 19. Initialization code.

Notice that the code in Figure 20 lies within a `for` loop. In a first attempt to parallelize anything you typically look at loops that can be split across multiple processors. Unfortunately, upon further inspection, the loop continues while `p<q` and, in the large if statement in the body of the loop, is the statement `q++`. This means that the number of iterations is data dependent and is not fixed at the outset. This loop cannot be easily parallelized.

Mating of Vertices

Since the sequential version of the code does not lend itself to easy parallelization, the next thing to do is back up and develop an understanding for how the algorithm works. The first thing that is obvious from the code is that you select a left subtree and then try to

```
/* Step 2 */
1 for(p=0; p<q; p++){
2       AP = A(p);
3       /* Step 3 */
4       for(r=0; ((H(p) > H(r)) AND (r!=q)); r++){
5               if((H(p) == H(r)) AND (r<p))
6                       r = p;
7               if(Subset(V(r), AP)){
8                       p_star = p;
9                       r_star = r;
10                      for(Label = PLUS; Label <= MINUS; Label++){
11                              /* Step 4 */
12                              AQ = A(q);
13                              if(Construct(p_star,r_star,&(E(q)))){
14                                      L(q) = p;
15                                      R(q) = r;
16                                      LBL(q) = Label;
17                                      LF(q) = LF(p);
18                                      H(q) = H(p) + 1;
19                                      /* next line is different */
20                                      Min(q) = max(Min(p)-1,H(r));
21                                      if(Lsp(p) != 0)
22                                              Lsp(q) = Lsp(p)
23                                      else
24                                              Lsp(q) = Lsp(r)
25                                      if(Rsp(r) != 0)
26                                              Rsp(q) = Rsp(r)
27                                      else
28                                              Rsp(q) = Rsp(p)
29                                      q_star = q;
30                                      q++;
31                                      /* Step 5 */
32                                      if(Proper_to_Add_Tree_to_Tlist(q_star)){
33                                              for_all(j in AP with Lf(R(q_star)) < j){
34                                                      SRoot(t) = j;
35                                                      Root(t) = q_star;
36                                                      t++;
37                                              }
38                                      }
39                              }
40                              /* Step 6 */
41                              p_star = r;
42                              r_star = p;
43                      }
44              }
45      }
46 }
```

Figure 20. The main loop properly structured.

mate it with possible right subtrees. Upon further examination we come to the conclusion that a left tree will mate with all trees that are shorter than it, and all trees of the same height that appear after it on Q_list, but it will never mate with any tree that is taller.

This analysis yields the need for the first function in our parallel algorithm which we will call `a_before_b()`, which is shown in Figure 21. This piece of code will determine if node a would have come before node b in Winter's Q_list. This is necessary since in a parallel implementation the calculations will not happen in the same order as they would have in the sequential version, so things will not be added to Q_list in the same order. With this function we are able to finish the function `mate()` previously described. This boolean function, which determines if node a (left) can mate with node b (right) is given in Figure 22.

```
int a_before_b(
     node a,
     node b){

     node a_parent;
     node b_parent;

     if(a.height < b.height)
          return(TRUE);
     if(a.height > b.height)
          return(FALSE);
     if(a.left == b.left) /* same parent */
          return(a.number < b.number);
     else{
          find_node(a.left,&a_parent);
          find_node(b.left,&b_parent);
          return(a_before_b(a_parent,b_parent));
     }
}
```

Figure 21. Function a_before_b().

```
int mate(
     node p,
     node r){

     if(!r.good_right) /* cannot be a right subtree */
          return(FALSE);
     if(r.height > p.height)
          return(FALSE);
     if(r.height < p.height)
          return(TRUE);
     return(a_before_b(p,r));
}
```

Figure 22. Function mate().

First Approach

Now that the foundational building blocks are completed, design attention can be turned to the main algorithm. The description of this is in a master–slave perspective. This perspective was taken due to the structure of most parallel architectures, as well as the fact that all nodes on the Q_list need a sequencing number assigned to them. The master will therefore be responsible for numbering the nodes and maintaining the main Q_list and T_list.

The description from the slave's perspective is quite simple. A process is spawned off for each member of Q_list that is a proper left subtree (Winter's algorithm allows members of Q_list that are not proper left subtrees). Each new process is then given all the current nodes on Q_list. With this information the slave then can determine, using the mate function previously described, with which nodes its left subtree could mate. This mating creates new nodes that are sent back to the master, assigned a number and added to the master's Q_list. The slave also attempts to create an FST out of the new Q_list member, and if it is successful, this FST is sent to the master to be added to the T_list. When a process runs out of Q_list nodes to check it sends a request for more nodes to the master.

The master also has a simple job description. It has to start a process for each initial member of the Q_list, send them all the current members of the Q_list and wait for their messages. Those messages will be in one of three categories: (i) Out_of_Data, (ii) New_Node and (iii) New_Tree.

The Out_of_data message will signal the master that a slave has processed all the members of Q_list it was given and is waiting for additional members. If there are no more nodes to process, this slave can be idled. If all the slaves working on Q_list nodes of the same height are idle then the master will tell them all to terminate. The process terminates if all levels are shut down.

The New_Node message will include the information necessary to add a new node to the Q_list. At this point the master will also have to spawn off a slave and start it up with this new node as its left subtree. If there are any idle slaves that can mate with this node the master will send them this new node to process.

The New_Tree message will be accompanied by a tree which is to be placed on T_list. These are kept and used in the extraction process described in the next chapter.

This structure worked quite well for smaller problems (up to about 15 points), but for larger problems it reached a grinding halt quite rapidly. This was due to various reasons such as the fact that for each slave started the entire Q_list had to be sent. This excessive message passing quickly bogged down the network. Secondly in their work on 100 point problems Cockayne and Hewgill [25] made the comment that T_list has an average length of 220, but made no comment about the size of Q_list, which is the number of slaves that would be started. From our work on 100 point problems this number easily exceeds $1,000$ which means that over $1,000$ processes are starting, each being sent the current Q_list. From these few problems, it is quite easy to see that some major changes needed to be made in order to facilitate the calculation of SMT's for large problems.

Current Approach

The idea for a modification to this approach came from a paper by Quinn and Deo [37], on parallel algorithms for Branch-and-Bound problems. Their idea was to let the master have a list of work that needs to be done. Each slave is assigned to a processor.

Each slave requests work, is given some, and during its processing creates more work to be done. This new work is placed in the master's work list, which is sorted in some fashion. When a slave runs out of work to do, it requests more from the master. They noted that this leaves some processors idle at times (particularly when the problem was starting and stopping), but this approach provides the best utilization if all branches are independent.

This description almost perfectly matches the problem at hand. First, we will probably have a fixed number of processors which can be determined at runtime. Secondly we have a list of work that needs to be done. The hard part is implementing a sorted work list in order to obtain a better utilization. This i implemented in what we term the Proc_list, which is a list of the processes that either are currently running or have not yet started. This list is primed with the information about the initial members of Q_list, and for every new node put on the Q_list, a node which contains information about the Q_list node, is placed on the Proc_list in the a_before_b() sorted order.

In order to complete the implementation a few pointers are also maintained. The first, called next_proc_to_start, is kept at the earliest member of the Proc_list that has not started. The next, called earliest_proc_alive, is placed on the earliest member of the Proc_list that is currently running.

Execution using this version is quite simple and fairly similar to the previous version. The same messages previously described are used in this version.

The first message is Out_of_data. If a node sends a message to the master requesting more data, and there is nothing that it could mate with, it is idled, but if it had been the earliest_proc_alive, its node on the Proc_list is finished, and is removed from the Proc_list. The slave associated with that node would be told to change to the node pointed to by next_proc_to_start which is then updated. That slave is then sent any members of the Q_list that it had not received. If the slave is told to change to a new node, the earliest_proc_alive pointer would need to be updated. This pointer is walked up the Proc_list, and if it meets a node that is idle, that process is terminated and the slave it was associated with would be told to change to the next_proc_to_start node. This walk would continue until the pointer meets a node that is currently running.

The second message is New_Node. This type of message will still include the information necessary to add a new node to the Q_list, but, as noted before, it will also cause a new node to be added to the Proc_list. If there are any idle slaves that could mate with this node the master sends them all the members of Q_list they have not yet received.

The third message is New_Tree. This type of message is handled in the same way it is in the previous version.

The only change made to the slave would be the addition of a Change_Node message, which could come back to it after it requested more data. Implementation of this is quite trivial. The results for this approach are quite exciting, and the timings are discussed in Chapter VI.

CHAPTER V

EXTRACTION OF THE CORRECT ANSWER

Introduction and Overview

Once the T_list discussed in Chapter IV is formed, the next step is to extract the proper answer from it. Winter described this in step 7 of his algorithm (Figure 17). His description stated that unions of FST's saved in T_list were to be formed subject to constraints described in his paper. The shortest union is the SMT for the original points.

The constraints he described were quite obvious considering the definition of an SMT. First, the answer had to cover all the original points. Second, the union of FST's could not contain a cycle. Third, the answer is bounded in length by the length of the MST.

This led Winter to implement a simple exhaustive search algorithm over the FST's in T_list. This approach yields a sample space of size $\mathcal{O}(2^m)$ (where m is the number of trees in T_list) that has to be searched. This exponentiality is born out in his work where he stated that for problems with more than 15 points "the computation time needed to form the union of FST's dominates the computation time needed for the construction of the FST's [36]." An example of the input the last step of Winter's algorithm receives (T_list) is given in Figure 23. The answer it extracts (the SMT) is shown in Figure 24.

Incompatibility Matrix

Although the SMT problem attracted many people through the early 70's the lack of results seemed to dampen most of their spirits. It wasn't until Winter's algorithm was published that the research community received the spark it needed to work on the SMT problem with renewed vigor. With the insight Winter provided into the problem, an attempt to computerize the solution of the SMT problem began anew.

Enhancement of this algorithm was first attempted by Cockayne and Hewgill at the University of Victoria. For this implementation Cockayne and Hewgill spent most of their work on the back end of the problem, or the extraction from T_list, and used Winter's algorithm to generate T_list. This work on the extraction focused on what they termed an

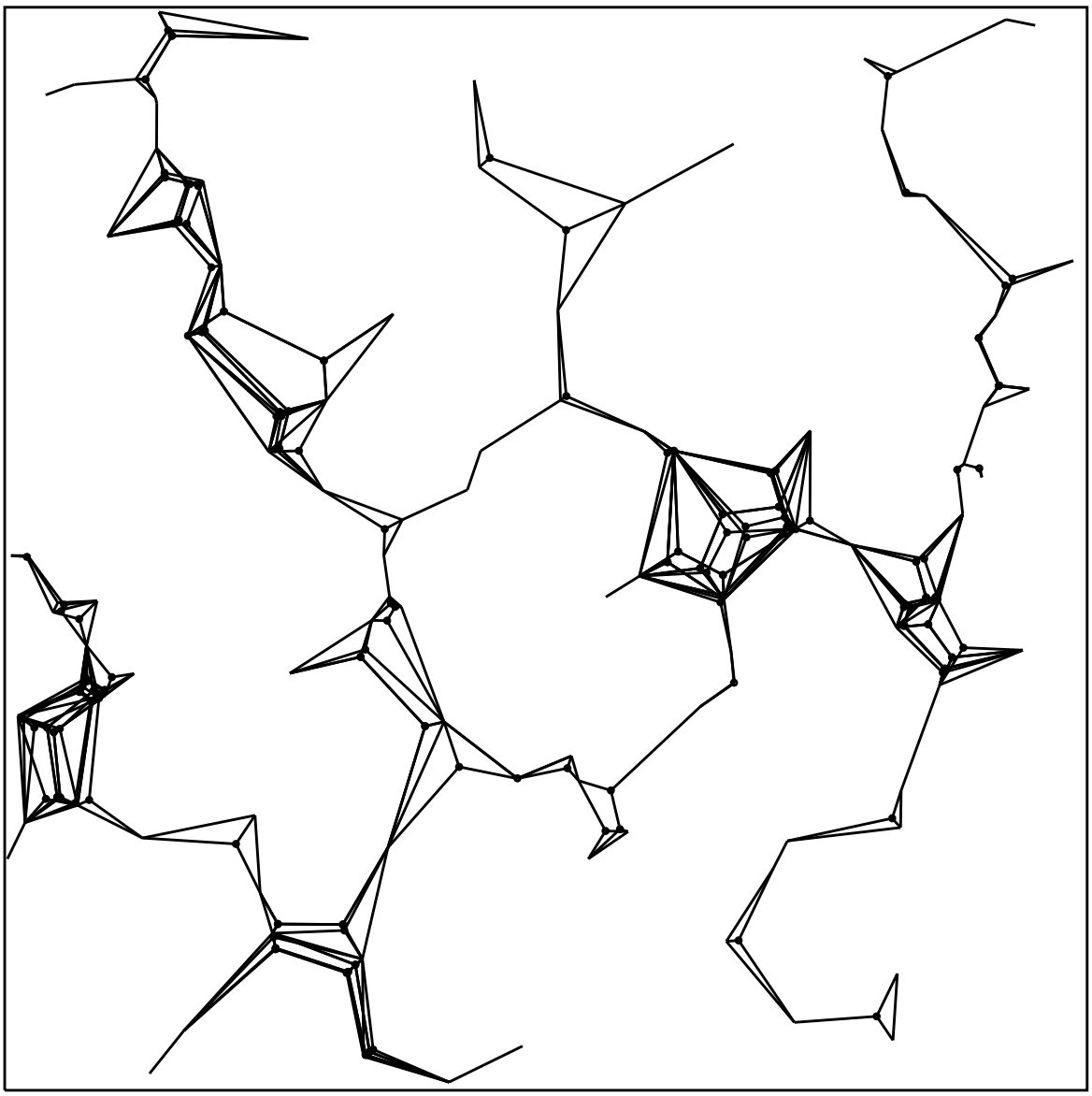

Figure 23. T_list for a random set of points.

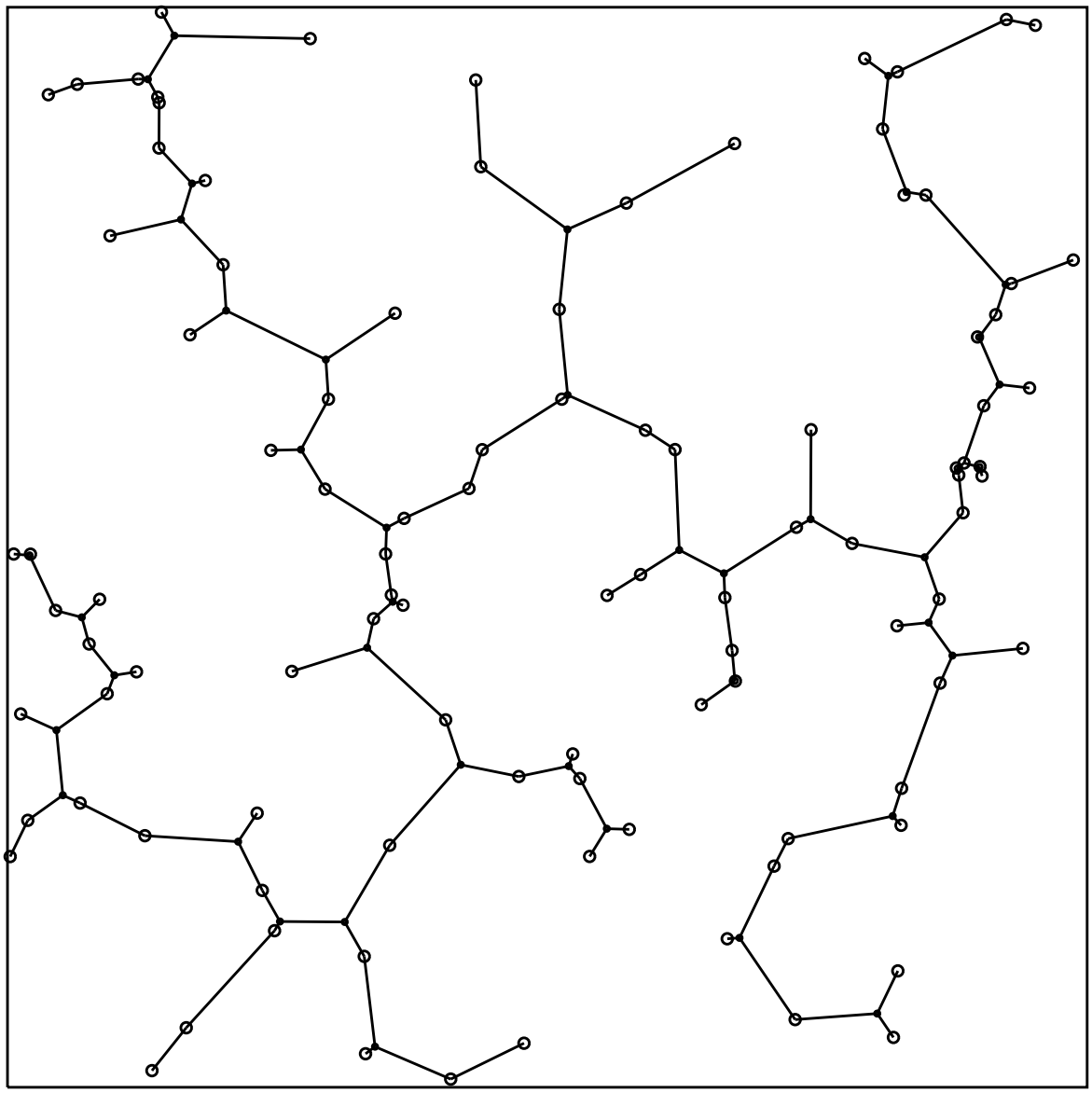

Figure 24. SMT extracted from T_list for a random set of points.

incompatibility matrix. This matrix had one row and one column for each member of T_list. The entries in this matrix were flags corresponding to one of three possibilities: *compatible*, *incompatible*, or *don't know*. The rationale behind the construction of this matrix is the fact that it is faster to look up the value in a matrix than it is to check for the creation of cycles and improper angles during the union of FST's.

The first value calculations for this matrix were straightforward. If two trees do not have any points in common then we *don't know* if they are incompatible or not. If they have two or more points in common then they form a cycle and are *incompatible*. If they have only one point in common and the angle at the intersection point is less than 120° then they are also *incompatible*. In all other cases they are *compatible*.

This simple enhancement to the extraction process enabled Cockayne and Hewgill to solve all randomly generated problems of size up to 17 vertices in a little over three minutes [38].

Decomposition

The next focus of Cockayne and Hewgill's work was in the area of the decomposition of the problem. As was discussed earlier in Chapter III, the best theorems for any problem, especially non-polynomial problems, are those of the form "If property $\mathcal{P}$ exists then the problem can be decomposed." Since the formation of unions of FST's is exponential in nature any theorem of this type is important.

Cockayne and Hewgill's theorem states: "Let A_1 and A_2 be subsets of A satisfying (i) $A_1 \bigcup A_2 = A$ (ii) $|A_1 \bigcap A_2| = 1$ and (iii) the leaf set of each FST in T_list is entirely contained in A_1 or A_2. Then any SMT on A is the union of separate SMT's on A_1 and A_2 [38]." This means that if you break T_list into biconnected components, the SMT will be the union of the SMT's on those components.

Their next decomposition theorem allowed further improvements in the calculation of SMT's. This theorem stated that if you had a component of T_list left from the previous theorem and if the T_list members of that component form a cycle, then it might be possible to break that cycle and apply the previous algorithm again. The cycle could be broken if there existed a vertex v whose removal would change that component from one biconnected component to more than one.

With these two decomposition theorems, Cockayne and Hewgill were able to calculate the SMT for 79 of 100 randomly generated 30 point problems. The remaining 21 would not decompose into blocks of size 17 or smaller, and thus would have taken too much computation time [38]. This calculation was implemented in the program they called EDSTEINER86.

Forest Management

Their next work, which was implemented in a program called EDSTEINER89, focused on improvements to the *incompatibility matrix* previously described. Their goal was to reduce the number of *don't know*'s in the matrix and possibly remove some FST's from T_list altogether.

They proposed two refinements for calculating the entry into the *incompatibility matrix* and one Tree Deletion Theorem. The Tree Deletion Theorem stated that if there exists an FST in T_list that is incompatible with all FST's containing a certain point a then the original FST can be deleted since at least one FST containing a will be in the SMT.

This simple change allowed Cockayne and Hewgill to calculate the SMT for 77 of 100 randomly generated 100 point problems [25]. The other 23 problems could not be calculated in less than 12 hours and were therefore terminated. For those that did complete, the computation time to generate T_list had become the dominate factor in the overall computation time.

So the pendulum had swung back from the extraction of the correct answer from T_list to the generation of T_list dominating the computation time. In Chapter VI we will look at the results of the parallel algorithm presented in Chapter IV to see if the pendulum can be pushed back the other way one more time.

CHAPTER VI

RESULTS

Previous Computation Times

Before presenting the results for the parallel algorithm presented in Chapter IV, it is worthwhile to review the computation times that have preceded this algorithm in the literature. The first algorithm for calculating FST's was discussed in a paper by Cockayne [30] where he mentioned that preliminary results indicated his code could solve any problem up to 30 points that could be decomposed with the Steiner Hull into regions of 6 points or less.

As we saw in Chapter III, the first computational results were presented by Cockayne and Schiller [11]. Their program, called STEINER, was written in FORTRAN on an IBM 360/50 at the University of Victoria. STEINER could calculate the SMT for any 7 point problem in less than 5 minutes of cpu time. When the problem size was increased to 8 it could solve them if 7 of the vertices were on the Steiner Hull. When this condition held it could calculate the SMT in under 10 minutes, but if this condition did not hold it would take an unreasonable amount of time.

Cockayne called STEINER a prototype for calculating SMT's and allowed Boyce and Serry of Bell Labs to obtain a copy of his code to improve the work. They improved the code, renamed it STEINER72, were able to calculate the FST for all 9 point problems and most 10 point problems in a reasonable amount of time [35]. Boyce and Serry continued their work and developed another version of the code that they thought could solve problems of size up to 12 points, but no computation times were given.

The breakthrough we saw in Chapter IV was by Pawel Winter. His program called GEOSTEINER [36] was written in SIMULA 67 on a UNIVAC-1100. GEOSTEINER could calculate SMT's for for all randomly generated sets with 15 points in under 30 seconds. This improvement was put into focus when he mentioned that all previous implementations took more than an hour for non-degenerate problems of size 10 or more. In his work, Winter tried randomly generated 20 point problems but did not give results since some of them did not finish in his cpu time limit of 30 seconds. The only comment he made for problems

bigger than size 15 was that the extraction discussed in Chapter V was dominating the overall computation time.

The next major program, EDSTEINER86, was developed in FORTRAN on an IBM 4381 by Cockayne and Hewgill [38]. This implementation was based upon Winter's results, but had enhancements in the extraction process. EDSTEINER86 was able to calculate the FST for 79 out of 100 randomly generated 32 point problems. For these problems the cpu time for T_list varied from 1 to 5 minutes, while for the 79 problems that finished the extraction time never exceeded 70 seconds.

Cockayne and Hewgill subsequently improved their SMT program and renamed it EDSTEINER89 [25]. This improvement was completely focused on the extraction process. EDSTEINER89 was still written in FORTRAN, but was run on a SUN 3/60 workstation. They randomly generated 200 32-point problems to solve and found that the generation of T_list dominated the computation time for problems of this size. The average time for T_list generation was 438 seconds while the average time for forest management and extraction averaged only 43 seconds. They then focused on 100 point problems and set a cpu limit of 12 hours. The average cpu time to generate T_list was 209 minutes for these problems, but only 77 finished the extraction in the cpu time limit. These programs and their results are summarized in Table 2.

The Implementation

The Significance of the Implementation

The parallel algorithms we have presented in this thesis have been implemented in a program called PARSTEINER94. This implementation is only the second SMT program since Winter's GEOSTEINER in 1981 and is the first parallel code. The major reason that the number of SMT programs is so small is due to the fact that any implementation is necessarily complex.

PARSTEINER94 currently has over 13,000 lines of C code. While there is a bit of code dealing with the parallel implementation, certain sections of Winter's Algorithm have a great deal of code buried beneath the simplest statements. For example line 14 of Figure 20 is the following:

```
if(Construct(p_star,r_star,&(E(q)))){.
```

To implement the function `Construct()` over 4,000 lines of code were necessary, and this does not include the geometry library with functions such as `equilateral_third_point()`, `center_of_equilateral_triangle()`, `line_circle_intersect()` and a host more.

Another important aspect of this implementation is the fact that there can now be comparisons made between the two current SMT programs. This would allow verification checks to be made between EDSTEINER89 and PARSTEINER94. This verification is important since with any complex program it is quite probable that there are a few errors hiding in the code. This implementation would also allow other SMT problems, such as those discussed in Chapter VII, to be explored independently, thereby broadening the knowledge base for SMT's even faster.

Table 2. SMT Programs, authors, and results.

Program	Author(s)	Points
STEINER	Cockayne & Schiller Univ of Victoria	7
STEINER72	Boyce & Serry ATT Bell Labs	10
STEINER73	Boyce & Serry ATT Bell Labs	12
GEOSTEINER	Winter Univ of Copenhagen	15
EDSTEINER86	Cockayne & Hewgill Univ of Victoria	30
EDSTEINER89	Cockayne & Hewgill Univ of Victoria	100
PARSTEINER94	Harris Clemson University	100

The Platform

In the design and implementation of parallel algorithms you are faced with many decisions. One such decision is what will your target architecture be? There are times when this decision is quite easy due to the machines at hand or the size of the problem. In

our case we decided not to target a specific machine, but an architectural platform called PVM.

PVM, which stands for Parallel Virtual Machine, is a software package available from Oak Ridge National Laboratory. This package allows a collection of parallel or serial machines to appear as a large distributed memory computational machine (MIMD model). This is implemented via two major pieces of software, a library of PVM interface routines, and a PVM demon that runs on every machine that you wish to use.

The library interface comes in two languages, C and Fortran. The functions in this library are the same no matter which architectural platform you are running on. This library has functions to spawn off (start) many copies of a particular program on the parallel machine, as well as functions to allow message passing to transfer data from one process to another. Application programs must be linked with this library to use PVM.

The demon process, called *pvmd* in the user's guide, can be considered the back end of PVM. As with any back end, such as the back end of a compiler, when it is ported to a new machine the front end can interface to it without change. The back end of PVM has been ported to a variety of machines, such as a few versions of Crays, various Unix machines such as Sun workstations, HP machines, Data General workstations and DEC Alpha machines. It has also been ported to a variety of true parallel machines such as the iPSC/2, iPSC/860, CM2, CM5, BBN Butterfly and the Intel Paragon.

With this information it is easy to see why PVM was picked as the target platform. Once a piece of code is implemented under PVM it can be re-compiled on the goal machine, linked with the PVM interface library on that machine, and run without modification. In our case we designed PARSTEINER94 on a network of SUN workstations, but, as just discussed, moving to a large parallel machine should be trivial.

Errors Encountered

When attempting to implement any large program from another person's description you often reach a point where you don't understand something. At first you always question yourself, but as you gain an understanding of the problem you learn that there are times when the description you were given is wrong. Such was the case with the SMT problem.

Therefore, to help some of those that may come along and attempt to implement this problem after us we are including a few of the errors we have found so your way may be a little smoother.

The first category of errors we wish to discuss are the notation errors. Whenever a great deal of notation is used it is easy to mix things up and mark things with the wrong subscripts. Fortunately this did not occur too often.

The first notation error is fairly easy to spot. It occurred in Figure 16 at line 8. $Root(q) = j_t$ was given as the second instruction of the line and it should be $Root(t) = j_t$. At this point the T_list is being primed with the edges of the spanning tree.

The second notation error occurred in Figure 17 on line 15. The left side of the equal sign is given as `MIN(p)` and it should be `MIN(q)`. At this point a new node on Q_list has just been created, and values are being loaded into its structure. Node p was created long ago and already has its values, which will not change.

The third error is just down the Figure and it can be solved in one of two ways. Either line 16 needs to be moved after line 18, or lines 17 and 18 need to have all references to q changed to q^*. Node q on Q_list was just created and it needs certain data filled into its structure, but node `q1+` has not been created, and therefore cannot have any data.

The last notational error we wish to mention here did not appear in the algorithm presented in this work, but appeared elsewhere in Winter's paper. In Lemma 3 of Winter's paper the following formula appeared:

$$p^* = \begin{cases} p' & if\ p' \in P \\ m_2 \bigcap O(q)\ \{q\} & if\ p' \notin P, p' \in O(q) \\ O(p) \bigcap O(q)\ \{q\} & if\ p' \notin P, p' \notin O(q). \end{cases}$$

On the right hand side there are two references to p' that are incorrect. The first is $p' \in O(q)$ and the second $p' \notin O(q)$. Both of these references should replace p' with p'_L.

The other category we wish to discuss is not really an error, but can very easily lead to an error. This category is based in the fact that Winter's mother tongue is not English, and as such the structure of sentences, ie verb placement, is different. This is easiest to point out in Figure 17, lines 25 and 26. These two lines appear in a `then` section of an `if` statement, and therefore do not depend upon other words for interpretation. Line 25 has two statements that each have a semicolon after them, signifying an end of statement. Line

26 is made up of one complete statement with various parts to it. Typically this would be interpreted as for every value that meets the condition to the right of the `for` increment `t`. But that is not what is meant. The meaning would come across better if it were re-written as shown in Figure 25. There are a few other places where the mother tongue of the author can confuse you if English is your mother tongue and you are not careful.

```
for every j ∈ AQ with LF(R(q*)) < j;
     SROOT(t) := j;
     ROOT(t) := q*;
     t := t + 1;
```

Figure 25. Clearer presentation of a section of code.

Random Problems

100 Point Random Problems

From the literature it is obvious that the current standard for calculating SMT's has been established by Cockayne and Hewgill. Their work on SMT's has pushed the boundary of computation out from the 15 point problems of Winter to being able to calculate SMT's for a large percentage of 100 point problems.

Cockayne and Hewgill, in their investigation of the effectiveness of EDSTEINER89, randomly generated 100 problems with 100 points inside the unit square. They set up a CPU limit of 12 hours, and 77 of 100 problems finished within that limit. They described the average execution times as follows: T_list construction averaged 209 minutes, Forest Management averaged 27 minutes, and Extraction averaged 10.8 minutes.

While preparing the code for this thesis, Cockayne and Hewgill were kind enough to supply us with 40 of the problems generated for [25] along with their execution times. These data sets were given as input to PARSTEINER94, and the calculation timed. The Wall Clock time necessary to generate T_list for the two programs appear in Table 3. For

Table 3. Comparison of T_list times.

Test Case	PARSTEINER94	EDSTEINER89
1	650	8597
2	1031	13466
3	1047	15872
4	1687	17061
5	874	13258
6	1033	15226
7	1164	12976
8	1109	16697
9	975	15354
10	554	8650
11	660	9894
12	946	13057
13	858	13687
14	978	17132
15	819	11333
16	752	12766
17	896	13815
18	788	10508
19	618	10550
20	724	11193
21	983	11357
22	889	12999
23	1449	15028
24	890	14417
25	912	17562
26	1125	12395
27	943	15721
28	583	10014
29	1527	18656
30	681	10033
31	873	16401
32	791	10217
33	1132	18635
34	1097	18305
35	1198	19657
36	803	11174
37	923	15256
38	824	12920
39	826	12538
40	972	15570
Avg.	939	13748

all 40 cases, the average time to generate T_list was less than 20 minutes. This is exciting because we have been able to generate T_list properly, while cutting an order of magnitude off the time.

It is worth noting in all fairness that not all of this speedup obtained can be attributed to the parallel implementation. The machines that the two programs were run on are quite different in performance. The SUN technical manuals show that the machines we used to run PARSTEINER94 on are any where from 4 to 7 times faster than the SUN 3/60 that EDSTEINER89 was run on. This range is given since the network of workstations that PVM was running on was not a homogeneous collection. After the hardware change is taken into consideration a speedup factor in the range of 2 can be attributed to the parallel approach invoked.

These results are quite promising for various reasons. First, the parallel implementation presented in this work is quite scalable, and therefore could be run with many more processors, thereby enhancing the speedup provided. Secondly, with the PVM platform used, we can in the future port this work to a real parallel MIMD machine, which will have much less communication overhead, or to a shared memory machine, where the communication could all but be eliminated, and expect the speedup to improve much more.

It is also worth noting that proper implementation of the Cycle Breaking which Cockayne and Hewgill presented in [38], is important if extraction of the proper answer is to be accomplished. In their work, Cockayne and Hewgill mentioned that 58% of the problems they generated were solvable without the Cycle Breaking being implemented, which is approximately what we have found with the data sets they provided. An example of such a T_list that would need cycles broken (possibly multiple times) is provided in Figure 26.

Larger Random Problems

Once the 100 point problems supplied by Cockayne and Hewgill had been successfully completed, the next step was to try a few larger problems. This was done with the hope of gaining an insight into the changes that would be brought about from the addition of more data points.

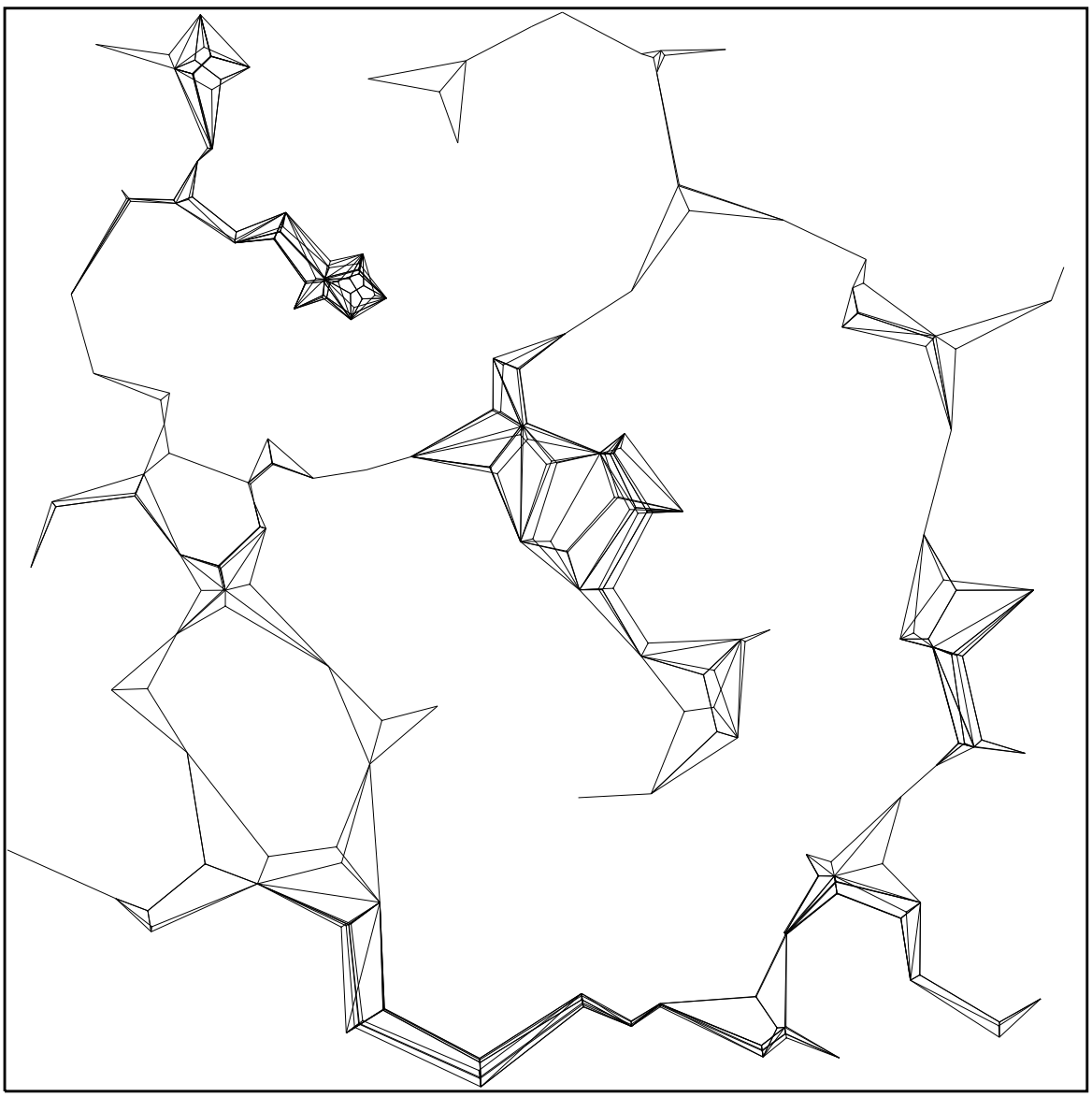

Figure 26. T_list with more than 1 cycle.

For this attempt we generated several random sets of 110 points each. The length of T_list increased by approximately 38%, from an average of 210 trees to an average of 292 trees. The time to compute T_list also increased, but the growth more than doubled, going from an average of 15 minutes to an average of more than 40 minutes.

The interesting thing that jumped out the most was the increase in the number of large bi-connected components. Since the extraction process must do a complete search of all possibilities, the larger the component the longer it will take. This is a classic example of an exponential problem, where when the problem size increases by 1 the time doubles. With this increased component size, none of the random problems generated finished inside a 12 hour cut off time.

This rapid growth puts into perspective the importance of the work previously done by Cockayne and Hewgill. Continuation of their work with incompatibility matrices as well as decomposition of T_list components appears at this point to be very important for the future of SMT calculations.

Grids

Introduction

The problem of determining SMT's for grids was mentioned to the author by Ron Graham. In this context we are thinking of a grid as a regular lattice of unit squares. The literature has very few pieces of information regarding SMT's on grids, and most of the information that is given is conjectured and not proven. The first proof for anything besides a 2 x 2 grid came in a paper by Chung and Graham [39] in which they proved the optimality of their characterization of SMT's for 2 x m grids. The only other major work was presented in a paper by Chung, Gardner, and Graham [40]. They argued the optimality of the SMT on a 2 x 2, 3 x 3, and a 4 x 4 lattice and gave conjectures and constructions for those conjectures for SMT's on all other square lattices.

In their work Chung, Gardner, and Graham specified three building blocks from which all SMT's on square (n x n) lattices were constructed. The first, labeled $\mathcal{I}$, is just a K_2, or a path on two vertices. This building block is given in Figure 27. The second, labeled $\mathcal{Y}$, is a FST on 3 vertices of the unit square. This building block is given in Figure 28. The

third, labeled $\mathcal{X}$, is a FST on all 4 vertices of the unit square. This building block is given in Figure 29. For the generalizations we are going to make here, we need to introduce one more building block, which we will label $\mathcal{S}$. This building block is a FST on a 3 x 2 grid and appears in Figure 30.

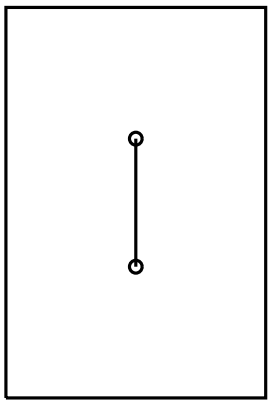

Figure 27. Building block $\mathcal{I}$.

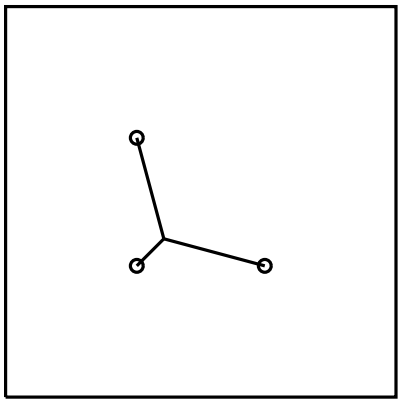

Figure 28. Building block $\mathcal{Y}$.

2 x m Grids

Grids of size 2 x m have two basic types of SMT's. The first is a FST on all the vertices in the 2 x m grid. An example of this for a 2 x 3 grid is given in Figure 30. The other SMT is constructed from the building blocks previously described. We hope that

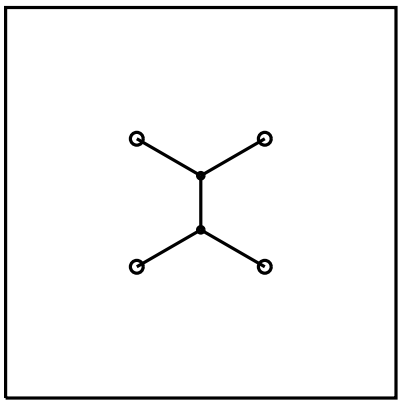

Figure 29. Building block $\mathcal{X}$.

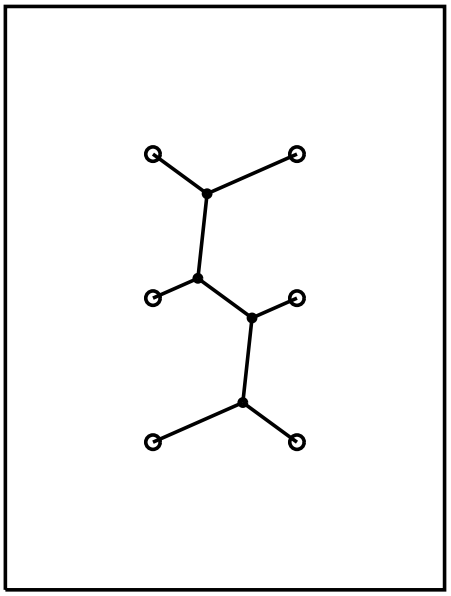

Figure 30. Building block $\mathcal{S}$.

these building blocks, when put in conjunction with the generalizations for 3 x m and 4 x m, will provide the foundation for a generalization of m x n grids in the future.

In their work on ladders (2 x m grids) Chung and Graham established and proved the optimality of their characterization for 2 x m grids. Before giving their characterization, a brief review of the first few 2 x m SMT's is in order. The SMT for a 2 x 2 grid is shown in Figure 29, the SMT for a 2 x 3 grid is shown in Figure 30, and the SMT for a 2 x 4 grid is given in Figure 31.

Chung and Graham [39] proved that SMT's for ladders fell into one of two categories. If the length of the ladder was odd, then the SMT was the FST on the vertices of the ladder. The SMT for the 2 x 3 grid in Figure 30 is an example of this. If the length of the ladder was even, the SMT was made up of a series of $\frac{m}{2} - 1$ $\mathcal{XI}$'s followed by one last $\mathcal{X}$. The SMT for the 2 x 4 grid in Figure 31 is an example of this.

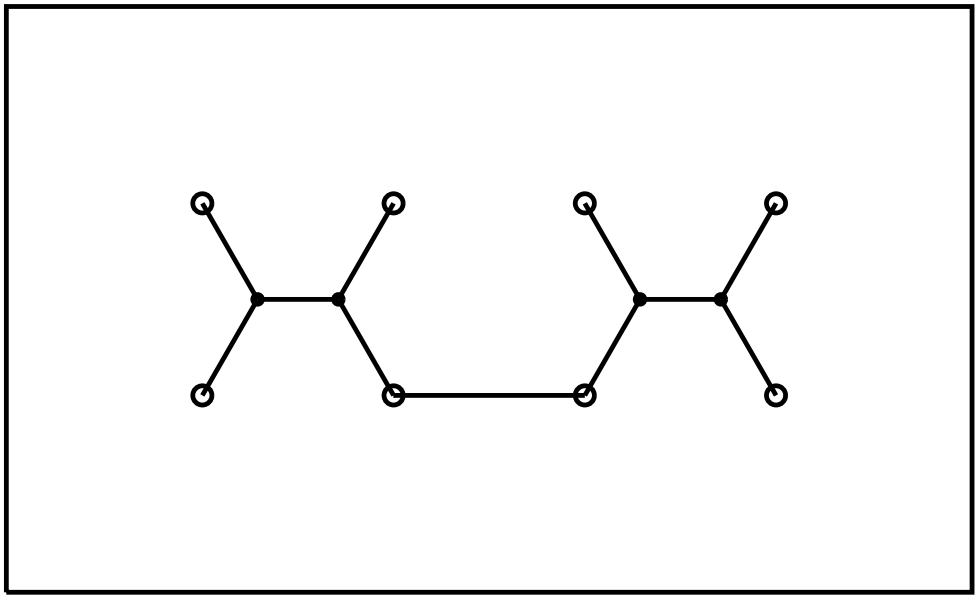

Figure 31. SMT for a 2 x 4 grid.

3 x m Grids

The SMT for 3 x m grids has a very easy characterization which can be seen once the initial cases have been presented. The SMT for the 3 x 2 grid is presented in Figure 30. The SMT for the 3 x 3 grid is presented in Figure 32.

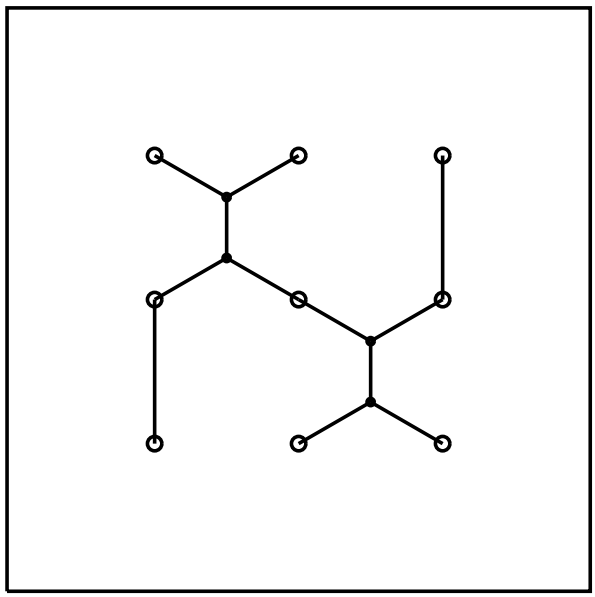

Figure 32. SMT for a 3 x 3 grid.

From here we can characterize all 3 x m grids. Except for in the 3 x 2 grid, which is an S building block, there will be only two basic building blocks present, $\mathcal{X}$'s and $\mathcal{I}$'s. There will be exactly two $\mathcal{I}$'s and m-1 $\mathcal{X}$'s. The two $\mathcal{I}$'s will appear on each end of the grid. The $\mathcal{X}$'s will appear in a staggered checkerboard pattern, one on each column of the grid the same way that the two $\mathcal{X}$'s are staggered in the 3 x 3 grid. The 3 x 5 grid is a good example of this and is shown in Figure 33.

4 x m Grids

The foundation for the 4 x m grids has already been laid. In their most recent work, Cockayne and Hewgill presented some results on Square Lattice Problems [25]. They looked at 4 x m grids for m=2 to m=6. They also looked at the SMT's for these problems when various lattice points in that grid were missing. What they did not do, however, was characterize the structure of the SMT's for all 4 x m grids.

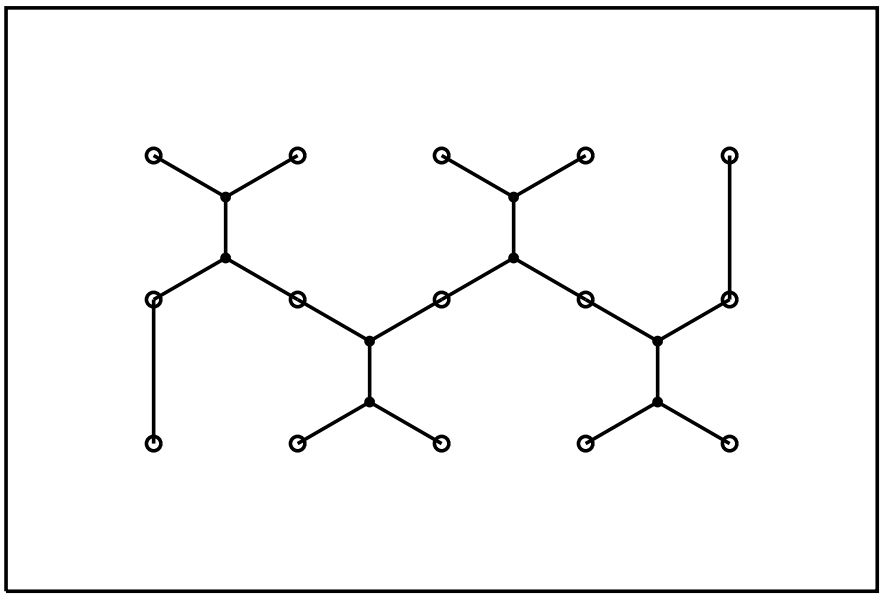

Figure 33. SMT for a 3 x 5 grid.

The 4 x 2 grid is given in Figure 31. From the work of Chung, Gardner, and Graham [40], we know that the SMT for a 4 x 4 grid is a checkerboard pattern of 5 $\mathcal{X}$'s. This layout gives us the first two patterns we will need to describe the 4 x m generalization. The first pattern, which we will call pattern A, is the same as the 3 x 4 grid without the two $\mathcal{I}$'s on the ends. This pattern is given in Figure 34. The second pattern, denoted as pattern B, is the 2 x 4 grid in Figure 31 without the connecting $\mathcal{I}$. This is shown in Figure 35.

Before the final characterization can be made, two more patterns are needed. The first one, called pattern C, is a 4 x 3 grid where the pattern is made up of two non-connected 2 x 3 SMT's, shown in Figure 36. The final pattern, denoted pattern D, is the same as pattern A except the upper left $\mathcal{X}$ of the 2 x 3 grid is replaced with a $\mathcal{Y}$. This is shown in Figure 37.

Now we can begin the characterization. The easiest way to present the characterization is to with some simple string re-writing rules. Since the 4 x 2, 4 x 3, and 4 x 4 patterns have already been given, the rules will begin with a 4 x 5 grid. This grid has the string AC. The first rule is that whenever there is a C replace it with DB. Therefore a 4 x 6 grid is ADB. The next rule is that whenever there is a B replace it with a C. Therefore for $m \geq 4$

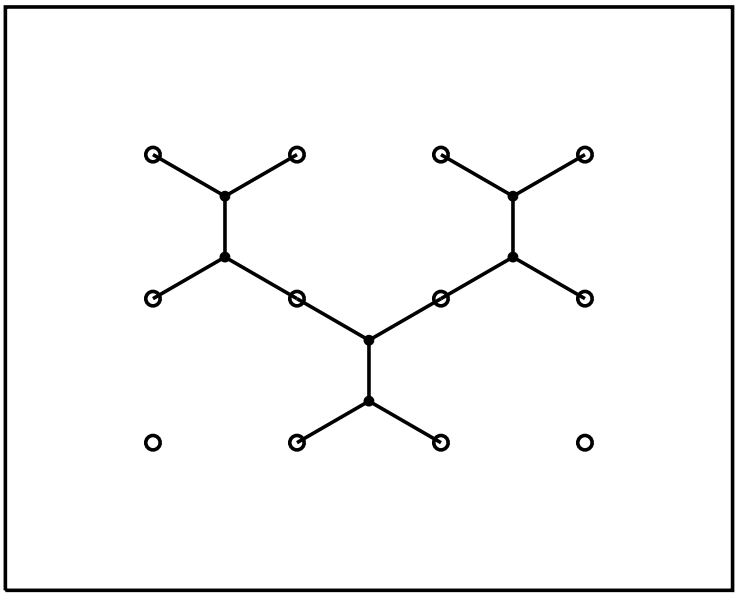

Figure 34. 4 x m pattern A.

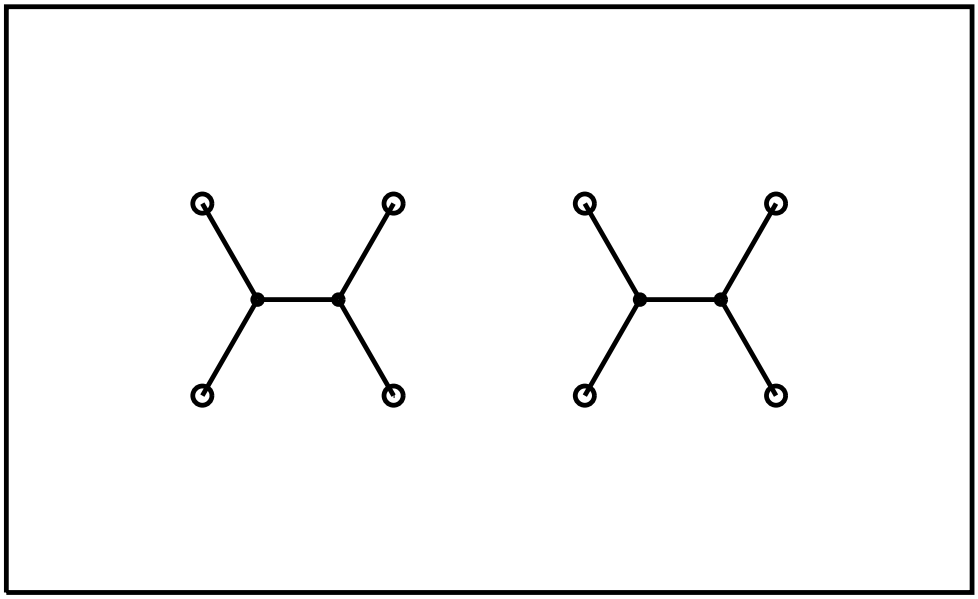

Figure 35. 4 x m pattern B.

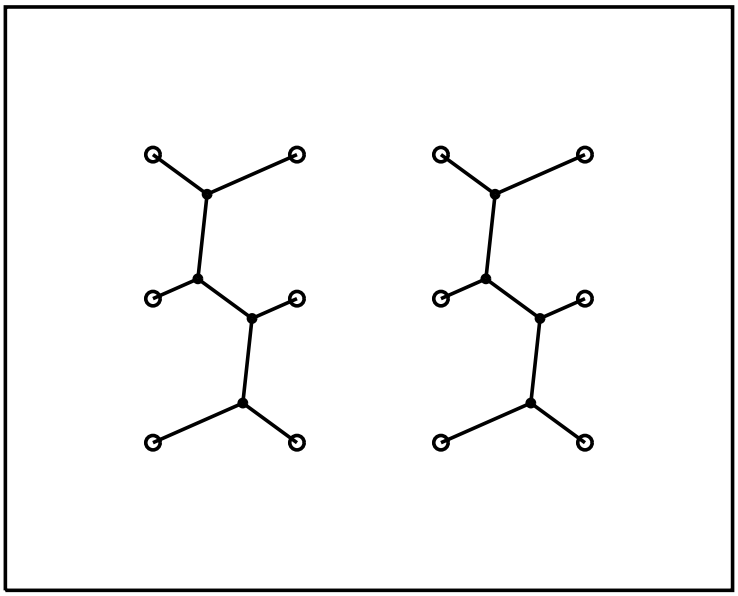

Figure 36. 4 x m pattern C.

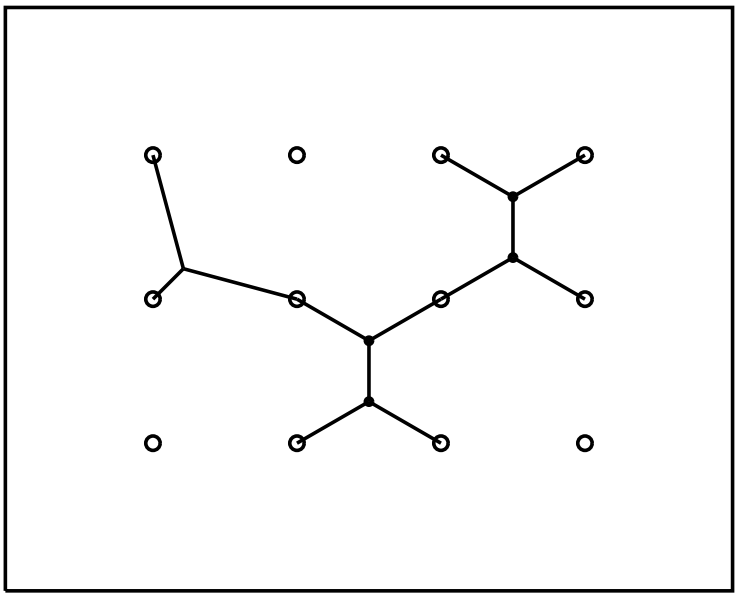

Figure 37. 4 x m pattern D.

and m even, the string is an A at the beginning, a B at the end and there are $(m/2) - 2$ D's in between. If m is odd and $m \geq 5$, the string is an A at the beginning, a C at the end, and $((m-1)/2) - 2$ D's in between. A listing of the strings for m from 5 to 11 is given in Table 4.

Table 4. String representations for 4 x m grids.

m =	5	6	7	8	9	10	11
String	$\mathcal{AC}$	$\mathcal{ADB}$	$\mathcal{ADC}$	$\mathcal{ADDB}$	$\mathcal{ADDC}$	$\mathcal{ADDDB}$	$\mathcal{ADDDC}$

Computation Times

In their work on grids, Cockayne and

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

Conclusions

This work is a synthesis of several new results. In Chapter II we presented a new heuristic for the SMT problem as well as a new measure of accuracy for heuristics. The heuristic was based upon Simulated Annealing and was an outgrowth of the 1-Steiner problem. The measure of accuracy expresses how much of the distance from the MST to the SMT a heuristic covers on a particular problem. This measure showed that our heuristic has achieved a score of over 95% on every randomly generated 100 point problem we tested, while using less than $\frac{1}{10}$th the cpu time currently used to solve the problem exactly.

In Chapter III we presented a new parallel algorithm for calculating the Steiner Hull and thereby decomposing the problem. This algorithm while not having a great theoretical speedup, in practice, computes the Steiner Hull for randomly generated 100 point problems rapidly.

The foundation of this thesis, presented in Chapters IV and VI, is a parallel algorithm for the generation of T_list and its implementation. When Winter developed his algorithm, the time for extraction dominated the overall computation time. After Cockayne and Hewgill's work, the time to generate T_list dominated the overall computation time. The parallel algorithms we presented in Chapters III and IV were implemented in a program called PARSTEINER94, and the results presented in Chapter IV show that the time to generate T_list has now been cut so much that the extraction time once again dominates the overall computation time.

In Chapter IV we also presented a characterization of SMT's for certain sizes of grids. The characterization of the SMT for a 2 x m grid has been studied in depth, and is well defined. The only other characterization for grids previously known were for square grids. In this work we present the characterizations of SMT's for 2 x m grids, as well as conjectured characterizations of SMT's for 3 x m, and 4 x m grids.

Further Parallelization

There remains a great deal of work that can be done on the Steiner Minimal Tree problem in the parallel arena. The first thing to consider is whether there are other ways to approach the parallel generation of T_list that would be more efficient. Improvement in this area would push the computation pendulum even further away from T_list generation and towards SMT extraction.

The next thing to consider is the entire extraction process. The initial generation of the *incompatibility matrix* has the appearance of easy parallelization. The forest management technique introduced by Cockayne and Hewgill could also be put into a parallel framework, thereby speeding up the preparation for extraction quite a bit.

With this initialization out of the way, decomposition could then be considered. The best possible enhancement here might be the addition of thresholds. As with most parallel algorithms, for any problem smaller than a particular size it is usually faster to solve it sequentially. These thresholds could come into play in determining whether to call a further decomposition, such as the cycle decomposition introduced by Cockayne and Hewgill that was discussed in Chapter V.

The final option for parallelization is one that may yield the best results, and that is in the extraction itself. Extraction is basically a branch and bound process, using the *incompatibility matrix*. This branch and bound is primed with the length of the MST as the initial bound, and continues until all possible combinations have been considered. The easiest implementation here would probably be the idea presented in the paper by Quinn and Deo [37] that served as the basis for the parallel algorithm in Chapter IV.

Additional Problems

1-Reliable Steiner Tree Problem

If we would like to be able to sustain a single failure of any vertex, without interrupting communication among remaining vertices, the minimum length network problem takes on a decidedly different structure. For example, in any FST all of the original vertices are of degree 1, and hence any one can be disconnected from the network by a single failure of the adjacent Steiner Point.

We would clearly like a minimum length 2-connected network. The answer can be the minimum length Hamiltonian cycle (consider the vertices of the unit square), but it need not be, as shown in the Θ graph given in Figure 38.

Here we can add Steiner points near the vertices of degree 3, and reduce the network length without sacrificing 2-connectivity. This is not just a single graph, but is a member of a family of graphs that look like ladders, where the Θ graph has only one internal rung. We hope to extend earlier work providing constructions on 2-connected graphs [41] to allow effective application of an Annealing Algorithm that could walk through graphs within the 2-connected class.

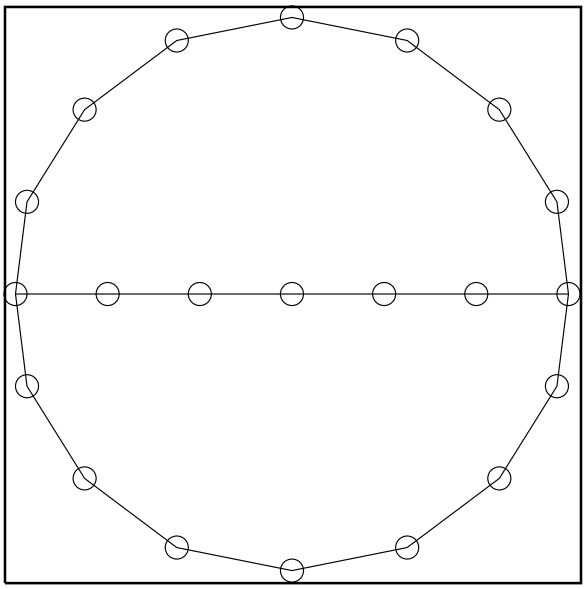

Figure 38. Theta graph.

Addition to Existing Networks

Cockayne has presented the following problem: Given a connected graph G and a set of vertices, V, which are not on edges of G, find a connected graph H which has G as a subgraph and contains the vertices in V.

This problem is very interesting when you consider it in the context of a growing road network. In this case you have a small town with an existing road network and a few

new houses and businesses that have been built outside of the town. Your job is to construct a set of roads with minimum length that connects these new houses and businesses to the town's road network.

The Steiner Minimal Tree problem we have discussed here can be seen as a special case of this new problem. For the SMT, the existing graph is empty, and all new vertices must be connected with minimal length. Therefore it is quite simple to deduce that this is also an NP-Complete problem.

LITERATURE CITED

1. V. Jarník and O. Kössler. O minimálnich gratech obsahujicich n daných bodu [in Czech]. *Casopis Pesk. Mat. Fyr.*, 63:223–235, 1934.

2. R. Courant and H. Robbins. *What is Mathematics? an elementary approach to ideas and methods*. Oxford University Press, London, 1941.

3. M.W. Bern and R.L. Graham. The shortest-network problem. *Sci. Am.*, 260(1):84–89, January 1989.

4. F.K. Hwang and D.S. Richards. Steiner tree problems. *Networks*, 22(1):55–89, January 1992.

5. F.K. Hwang, D.S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Ann. Discrete Math.* North-Holland, Amsterdam, 1992.

6. E.N. Gilbert and H.O. Pollak. Steiner minimal trees. *SIAM J. Appl. Math.*, 16(1):1–29, January 1968.

7. D.Z. Du and F.H. Hwang. A proof of the Gilbert-Pollak conjecture on the Steiner ratio. *Algorithmica*, 7(2/3):121–135, 1992.

8. Z.A. Melzak. On the problem of Steiner. *Canad. Math. Bull.*, 4(2):143–150, 1961.

9. M.R. Garey, R.L. Graham, and D.S Johnson. The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.*, 32(4):835–859, June 1977.

10. E.J. Cockayne. On the Steiner problem. *Canad. Math. Bull.*, 10(3):431–450, 1967.

11. E.J. Cockayne and D.G. Schiller. Computation of Steiner minimal trees. In D.J.A. Welsh and D.R. Woodall, editors, *Combinatorics*, pages 52–71, Maitland House, Warrior Square, Southend-on-Sea, Essex SS1 2J4, 1972. Mathematical Institute, Oxford, Inst. Math. Appl.

12. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

13. Michael K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Trans. Comput.*, C-31(9):913–917, September 1982.

14. S. Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1:17–61, 1988.

15. J.J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Nat. Acad. Sci.*, 81:3088–3092, 1984.

16. R. Geist, D. Suggs, R. Reynolds, S. Divatia, F. Harris, E. Foster, and P. Kolte. Disk performance enhancement through Markov-based cylinder remapping. In Cherri M. Pancake and Douglas S. Reeves, editors, *Proc. of the ACM Southeastern Regional Conf.*, pages 23–28, Raleigh, North Carolina, April 1992. The Association for Computing Machinery, Inc.

17. R. Geist, D. Suggs, and R. Reynolds. Minimizing mean seek distance in mirrored disk systems by cylinder remapping. In *Proc.* 16^{th} *IFIP Int. Symp. on Computer Performance Modeling, Measurement, and Evaluation (PERFORMANCE '93)*, pages 91–108, Rome, Italy, September 1993.

18. R. Geist and D. Suggs. Neural networks for the design of distributed, fault-tolerant, computing environments. In *Proc.* 11^{th} *IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 189–195, Houston, Texas, October 1992.

19. R. Geist, R. Reynolds, and D. Suggs. A markovian framework for digital halftoning. *ACM Trans. Graphics*, 12(2):136–159, April 1993.

20. R. Geist, R. Reynolds, and C. Dove. Context sensitive color quantization. Technical Report 91–120, Dept. of Comp. Sci., Clemson Univ., Clemson, SC 29634, July 1991.

21. W.R. Reynolds. *A Markov Random Field Approach to Large Combinatorial Optimization Problems*. PhD thesis, Clemson, University, Clemson, SC 29634, August 1993.

22. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(13):671–680, May 1983.

23. G. Georgakopoulos and C. Papadimitriou. A 1-steiner tree problem. *J. Algorithms*, 8(1):122–130, Mar 1987.

24. F.P. Preparata and M.I. Shamos. *Computational Geometry: an introduction*. Springer-Verlag, New York, NY, 1988.

25. E.J. Cockayne and D.E. Hewgill. Improved computation of plane Steiner minimal trees. *Algorithmica*, 7(2/3):219–229, 1992.

26. M.I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.

27. A. Chow. *Parallel Algorithms for Geometric Problems*. PhD thesis, University of Illinois, Urbana-Champaign, IL, 1980.

28. A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

29. M.J. Atallah and M.T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, 1988.

30. E.J. Cockayne. On the efficiency of the algorithm for Steiner minimal trees. *SIAM J. Appl. Math.*, 18(1):150–159, January 1970.

31. F.K. Hwang, G.D. Song, G.Y. Ting, and D.Z. Du. A decomposition theorem on Euclidian Steiner minimal trees. *Disc. Comput. Geom.*, 3(4):367–382, 1988.

32. J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1992.

33. V. Kumar, A. Grama, A. Gupta, and Karypis G. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

34. J.R. Smith. *The Design and Analysis of Parallel Algorithms*. Oxford University Press, Inc., New York, NY, 1993.

35. W.M. Boyce and J.R. Seery. STEINER 72 – an improved version of Cockayne and Schiller's program STEINER for the minimal network problem. Technical Report 35, Bell Labs., Dept. of Computer Science, 1975.

36. P. Winter. An algorithm for the Steiner problem in the Euclidian plane. *Networks*, 15(3):323–345, Fall 1985.

37. M.J. Quinn and N. Deo. An upper bound for the speedup of parallel best-bound branch-and-bound algorithms. *BIT*, 26(1):35–43, 1986.

38. E.J. Cockayne and D.E. Hewgill. Exact computation of Steiner minimal trees in the plane. *Info. Proccess. Lett.*, 22(3):151–156, March 1986.

39. F.R.K. Chung and R.L. Graham. Steiner trees for ladders. In B. Alspach, P. Hell, and D.J. Miller, editors, *Annals of Discrete Mathematics:2*, pages 173–200. North–Holland Publishing Company, 1978.

40. F.R.K. Chung, M. Gardner, and R.L. Graham. Steiner trees on a checkerboard. *Math. Mag.*, 62:83–96, 1989.

41. S. Hedetniemi. Characterizations and constructions of minimally 2-connected graphs and minimally strong digraphs. In *Proc.* 2^{nd} *Louisiana Conf. on Combinatorics, Graph Theory, and Computing*, pages 257–282, Louisiana State Univ., Baton Rouge, Louisiana, March 1971.