Steiner Minimal Trees: Their Computational Past, Present, and Future

Frederick C. Harris, Jr.
Department of Computer Science
University of Nevada
Reno, Nevada 89557
fredh@cs.unr.edu

Abstract: Given a set of N cities, construct a connected network which has minimum length. The problem is simple enough, but the catch is that you are allowed to add junctions in your network. Therefore the problem becomes how many extra junctions should be added, and where should they be placed so as to minimize the overall network length.

This intriguing optimization problem is known as the Steiner Minimal Tree Problem (SMT), where the junctions that are added to the network are called Steiner Points.

The focus of this paper is twofold. First we look at the computational history of the problem, up through and including a new method to computer SMT's in parallel. Secondly we look at future work in the computation of Steiner Minimal Trees.

keywords: Steiner Minimal Trees

1 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 Jarník and Kössler generalized the network minimization problem even further [27]: 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 [12], appeared in 1941. Courant and Robbins linked the name Steiner with this form of the problem proposed by Jarník and Kössler, 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 [23], or the recent volume in The Annals of Discrete Mathematics devoted completely to Steiner Tree problems [24]. 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 paper 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 Section 2 we review the first fundamental algorithm for calculating SMT's. In Section 3 problem decomposition for SMT's is outlined. In Section 4 we present Winter's sequential algorithm which has been the basis for all computation of SMT's to the present day. In Section 5 we present a parallel algorithm for SMT's. Extraction of the correct answer is discussed in Section 6. Computational Results are presented in Section 7 and Future Work and open problems are presented in Section 8.

2 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 Section 1, 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.

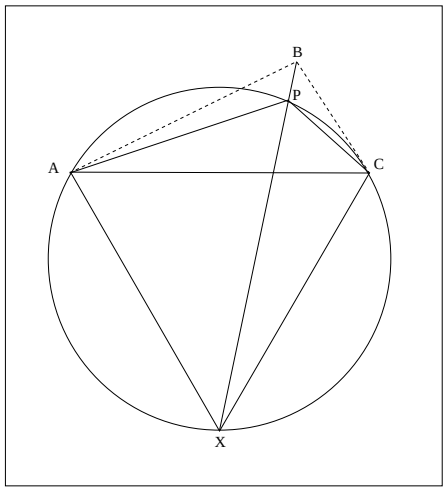

Figure 1: AP + CP = PX.

The next logical extension of the problem, going to four points, is attributed to Gauss. His son, who was a railroad engineer, was apparently designing the layout for tracks between four major cities in Germany and was trying to minimize the length of these tracks. It is interesting to note at this point that a general solution to the SMT problem has recently been uncovered in the archives of a school in Germany [17].

For the next thirty years after Kössler and Jarník presented the general form of the SMT problem, only heuristics were known to exist. The heuristics were typically based upon the Minimum-Length Spanning Tree (MST), which is a tree that spans 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 [16] 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 [13], and has led to the MST being the starting point for most of the heuristics that have been proposed in the last 20 years including a recent one that achieves some very good results [19].

In 1961 Melzak developed the first known algorithm for calculating an SMT [29]. 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 [14] 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 [7] 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.

3 Problem Decomposition

After the early work by Melzak [29], 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 [16] 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 of 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 2.

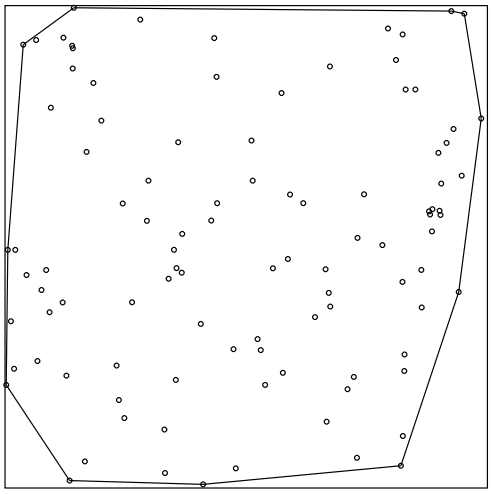

Figure 2: The Convex Hull for a random set of points.

Shamos in his PhD thesis [31] 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 [6]. This approach was refined and made to run in optimal $\mathcal{O}(log\ n)$ time by Aggarwal et.al. [1], and Attalah and Goodrich [2].

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. Unfortu-

nately, 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 first decomposition theorem, known as the Double Wedge Theorem, was proposed by Gilbert and Pollak [16]. The next decomposition theorem is due to Cockayne [8] and is based upon what he termed the *Steiner Hull*. The final decomposition belongs to Hwang, Song, Ting, and Du [25]. They proposed an extension to the Steiner Hull as defined by Cockayne. These three decomposition theorems were combined into a parallel algorithm for decomposition presented in [18]

4 Winter's Sequential Algorithm

4.1 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 in Section 2, 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 [4]. 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 towards the end of Section 2, 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 [35]. 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 [35]."

4.2 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 3. 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.

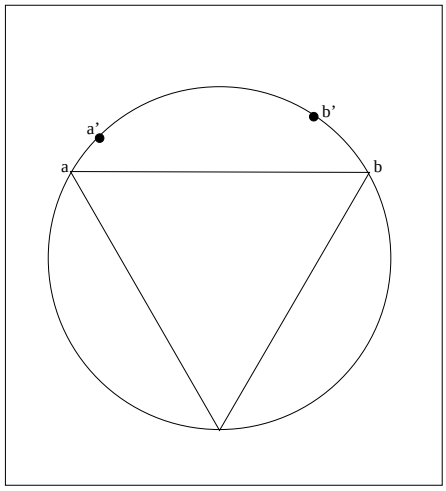

Figure 3: An illustration of Winter's pushing.

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. 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.

This algorithm was a turning point in the calculation of SMT's. It sparked renewed interest into the calculation of SMT's in general. This renewed interest has produced new algorithms such as the Negative Edge Algorithm [34] and the Luminary Algorithm [22]. Winter's algorithm has also served as the foundation for most computerized computation for calculating SMT's and is the foundation for the parallel algorithm we present next.

5 A Parallel Algorithm

5.1 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 Steiner Minimal Tree problem.

There have been volumes written on parallel computation and parallel algorithms; therefore, we 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 [26], and for a more practical approach to implementation on a parallel machine the reader is referred to the book by Vipin Kumar et.al. [28] or the book by Justin Smith [32].

5.2 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 `gotos`. The Initialization (Step 1) does not change, and the translation of steps 2 through 7 appears in Figure 4.

Notice that the code in Figure 4 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

```
/* 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 4: The main loop properly structured.

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.

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 of how the algorithm works. The first thing that is obvious from the code is that you select a left subtree and then try to 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.

5.3 First Approach

The description of this parallel algorithm 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 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.

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 [10] 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.

5.4 Current Approach

The idea for a modification to this approach came from a paper by Quinn and Deo [30], 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 was 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 a sorted order.

The results for this approach are quite exciting, and the timings are discussed in Section 7.

6 Extraction of the Correct Answer

6.1 Introduction and Overview

Once the T_list discussed in Section 4 is formed, the next step is to extract the proper answer from it. Winter described this in step 7 of his algorithm. 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 [35]." An example of the input the last step of Winter's algorithm receives (T_list) is given in Figure 5. The answer it extracts (the SMT) is shown in Figure 6.

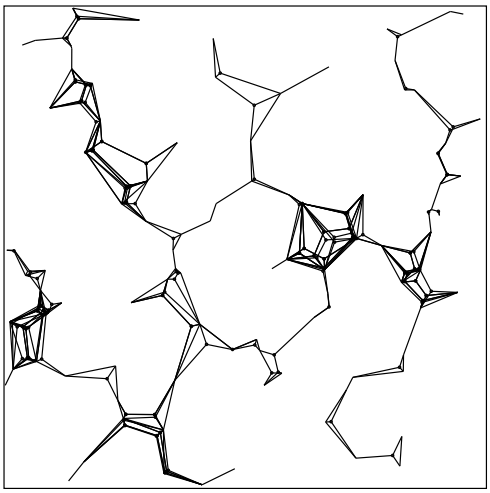

Figure 5: T_list for a random set of points.

6.2 Incompatibility Matrix

Once Cockayne published the clarification of Melzak's proof in 1967 [7] and Gilbert and Pollak published their paper giving an upper bound the the SMT length [16] many people were attracted to this problem. From this time until Winter's work was published in 1985 [35] quite a few papers were published dealing with various aspects of the SMT Problem, but the attempt to computerize the solution of the SMT problem bogged down around 12 vertices. It wasn't until Winter's algorithm was published that the research community received the spark it needed to work on this aspect of 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

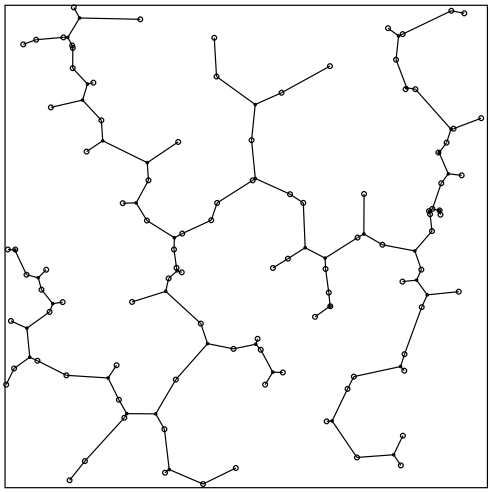

Figure 6: SMT extracted from T_list for a random set of points.

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 *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 [9].

6.3 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 Section 3, 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 [9]." 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 [9]. This calculation was implemented in the program they called EDSTEINER86.

6.4 Forest Management

Their next work focused on improvements to the *incompatibility matrix* previously described and was implemented in a program called EDSTEINER89, 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 [10]. 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 Section 7 we will look at the results of the parallel algorithm presented in Section 8 to see if the pendulum can be pushed back the other way one more time.

7 Computational Results

7.1 Previous Computation Times

Before presenting the results for the parallel algorithm presented in Section 5, 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 [8] 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 Section 2, the next 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 [4]. 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 Section 4 was by Pawel Winter. His program called GEOSTEINER [35] 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 Section 6 was dominating the overall computation time.

The next major program, EDSTEINER86, was developed in FORTRAN on an IBM 4381 by Cockayne and Hewgill [9]. 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 [10]. 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 1.

Table 1: 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 Univ of Nevada	100

7.2 The Implementation

7.2.1 The Significance of the Implementation

The parallel algorithm we presented has been implemented in a program called PARSTEINER94 [18, 20]. 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 13 of Figure 4 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 we will discuss in Section 8, to be explored independently, thereby broadening the knowledge base for SMT's even faster.

7.2.2 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 [15].

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.

7.2.3 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 recommend that you look at the list of errors we found while implementing Winter's Algorithm [18].

7.3 Random Problems

7.3.1 100 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 project, Cockayne and Hewgill were kind enough to supply us with 40 of the problems generated for [10] along with their execution times. These data sets were given as input to the parallel code PARSTEINER94, and the calculation timed. The Wall Clock time necessary to generate T_list for the two programs appear in Table 2. For all 40 cases, the average time to generate T_list was less than 20 minutes.

Table 2: 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

This is exciting because we have been able to generate T_list properly, while cutting an order of magnitude off the time.

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 [9], 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 7.

Figure 7: T_list with more than 1 cycle.

7.3.2 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.

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.

8 Future Work

8.1 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 Section 6.

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 [30] that served as the basis for the parallel algorithm in Section 5.

8.2 Additional Problems

8.2.1 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 8.

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 [21] to allow effective application of an Annealing Algorithm that could walk through graphs within the 2-connected class.

8.2.2 Augmenting Existing Plane Networks

In practical applications, it frequently happens that new points must be joined to an existing Steiner Minimal Tree. Although a new and larger SMT can, in principle, be constructed which connects both the new and the existing points, this is typically impractical. eg. in cases where a fiber optic network has already been constructed. Thus the only acceptable approach is to add the new points to the network as cheaply as possible. Cockayne has presented this problem which we can state as follows:

Augmented Steiner Network: Given a connected plane graph $G = (V, E)$ (i.e. an embedding of a connected planar graph in E^2) and a set V' of points in the plane which are not on edges of G, construct a connected plane supergraph $G" = (V", E")$, such that $V"$ contains $V \bigcup V'$, $E"$ contains E, and the sum of the Euclidean lengths of the set of edges in $E" - E$ is a minimum. In constructing the plane graph $G"$ it is permitted to add an edge connecting a point in V' to an interior point of an edge in G. It is

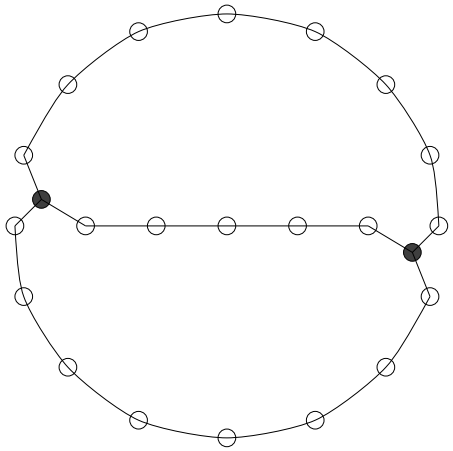

Figure 8: Theta graph.

also permitted to add Steiner points. Thus, strictly speaking, G" need not be a supergraph of G.

The Augmented Steiner Network Problem clearly has applications in such diverse areas as canal systems, rail systems, housing subdivisions, irrigation networks and computer networks. For example, given a (plane) fiber optic computer network $G = (V, E)$ and a new set V' of nodes to be added to the network, the problem is to construct a set F' of fiber optic links with minimum total length that connects V' to G. The set F' of new links is easily seen to form a forest in the plane, because the minimum total length requirement ensures that there cannot be cycles in F'.

As an example, consider the situation in Figure 9 where G consists of a single, long edge and $V' = v_1, \ldots, v_8$. The optimal forest F' consists of three trees joining G at f_1, f_2 and f_3. It is necessary that extra Steiner points s_1, s_2 and s_3 be added so that F has minimum length.

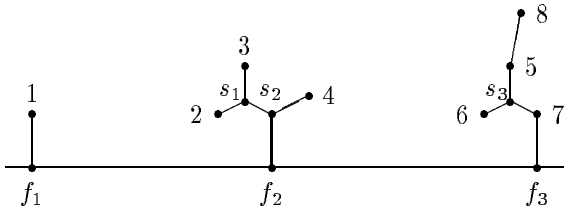

Figure 9: An Optimal Forest.

While we are aware of several algorithms for solving special cases of the Augmented Existing Plane Network Problem, such as those by Chen [5] and Trietsch [33] or the special case where the graph G consists of a single vertex, in which case the problem is equivalent to the classical Steiner Minimal Tree Problem, we are not aware of any algorithms or computer programs available for exact solutions to the general form of this problem. Here, "exact" means provably optimal except for round-off error and machine representation of real numbers. Non-exact (i.e. heuristic) solutions are sub-optimal although they may often be found considerably faster.

References

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

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

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

[4] 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.

[5] G. X. Chen. The shortest path between two points with a (linear) constraint [in Chinese]. *Knowledge and Appl. of Math.*, **4**:1–8, 1980.

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

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

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

[9] 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.

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

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

[13] 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.

[14] 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.

[15] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A User's guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, 1994.

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

[17] R.L. Graham. Private Communication.

[18] F.C. Harris, Jr. *Parallel Computation of Steiner Minimal Trees*. PhD thesis, Clemson, University, Clemson, SC 29634, May 1994.

[19] F.C. Harris, Jr. A stochastic optimization algorithm for steiner minimal trees. *Congr. Numer.*, **105**:54–64, 1994.

[20] F.C. Harris, Jr. Parallel computation of steiner minimal trees. In David H. Bailey, Petter E. Bjorstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virgia J. Torczan, and Layne T. Watson, editors, *Proc. of the* 7^{th} *SIAM Conf. on Parallel Process. for Sci. Comput.*, pages 267–272, San Francisco, California, February 1995. SIAM.

[21] 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.

[22] F. K. Hwang and J. F. Weng. The shortest network under a given topology. *J. Algorithms*, **13**(3):468–488, Sept. 1992.

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

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

[25] 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.

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

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

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

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

[30] 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.

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

[32] Justin R. Smith. *The Design and Analysis of Parallel Algorithms.* Oxford University Press, Inc., New York, NY, 1993.

[33] D. Trietsch. Augmenting Euclidean networks – the Steiner case. *SIAM J. Appl. Math.*, **45**:855–860, 1985.

[34] D. Trietsch and F. K. Hwang. An improved algorithm for Steiner trees. *SIAM J. Appl. Math.*, **50**:244–263, 1990.

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