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.

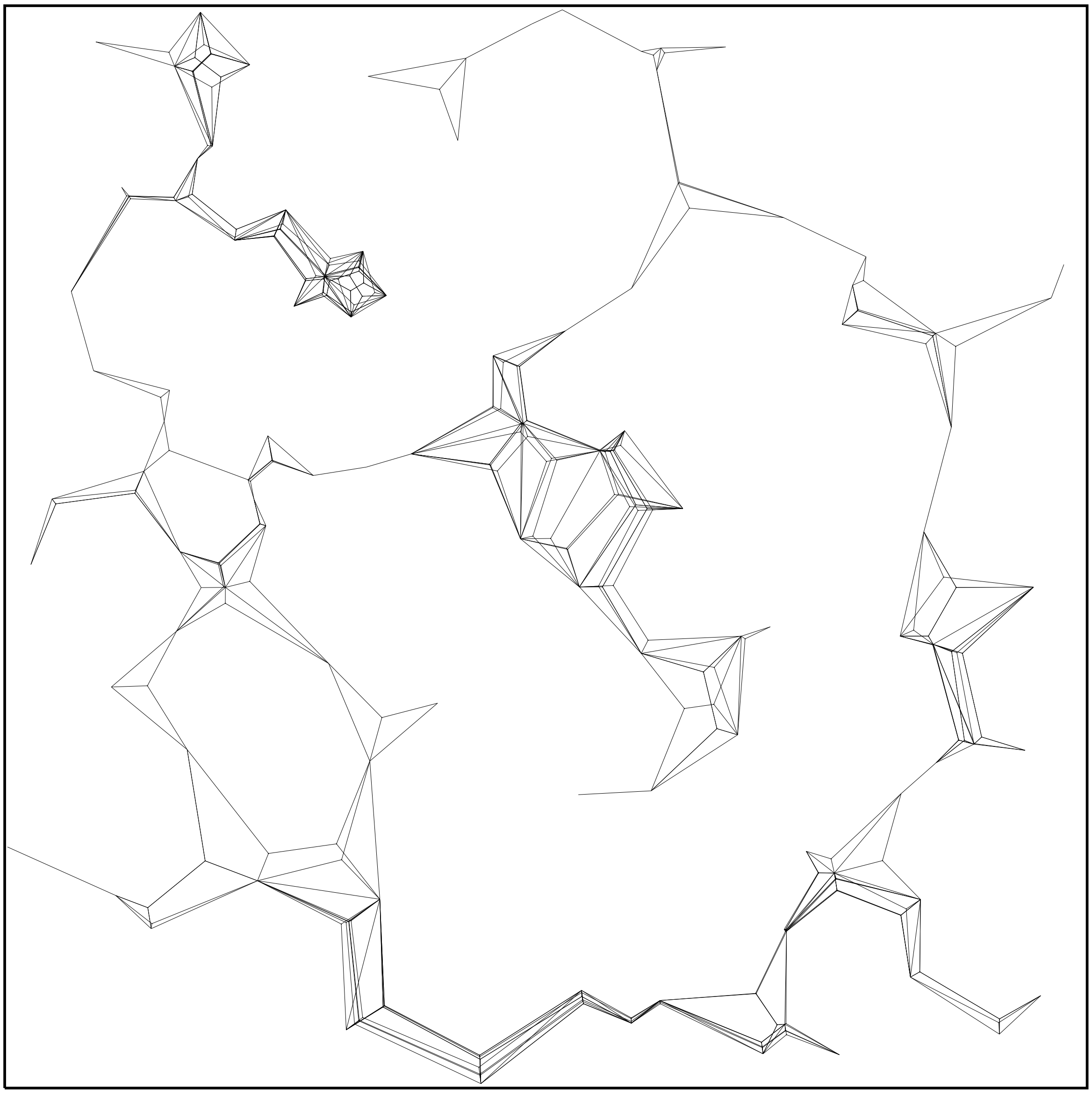

Figure 26. T_list with more than 1 cycle.

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.

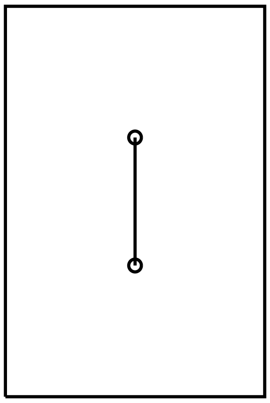

Figure 27. Building block $\mathcal{I}$.

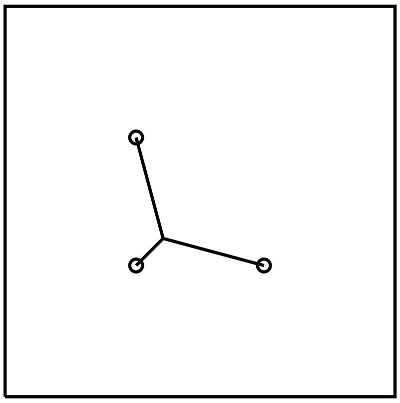

Figure 28. Building block $\mathcal{Y}$.

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

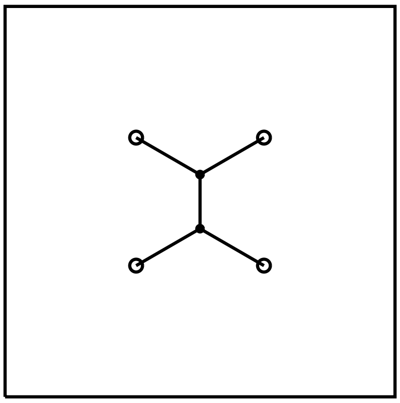

Figure 29. Building block $\mathcal{X}$.

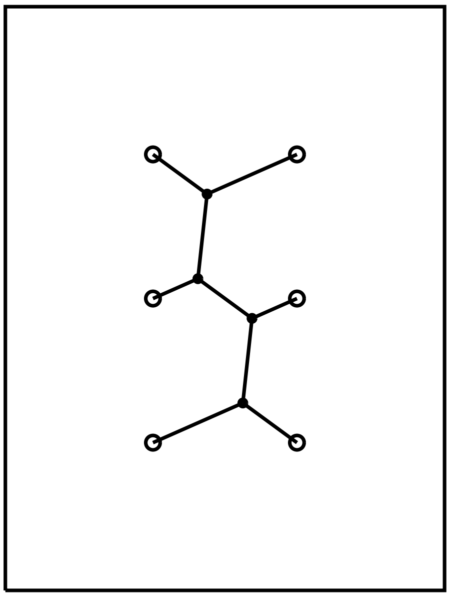

Figure 30. Building block $\mathcal{S}$.

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.

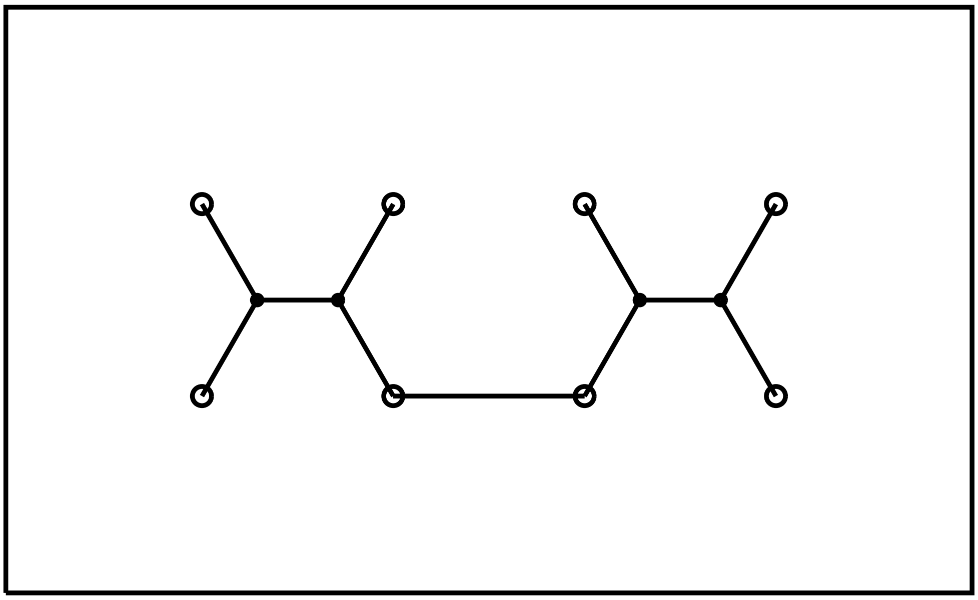

Figure 31. SMT for a 2 x 4 grid.

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.

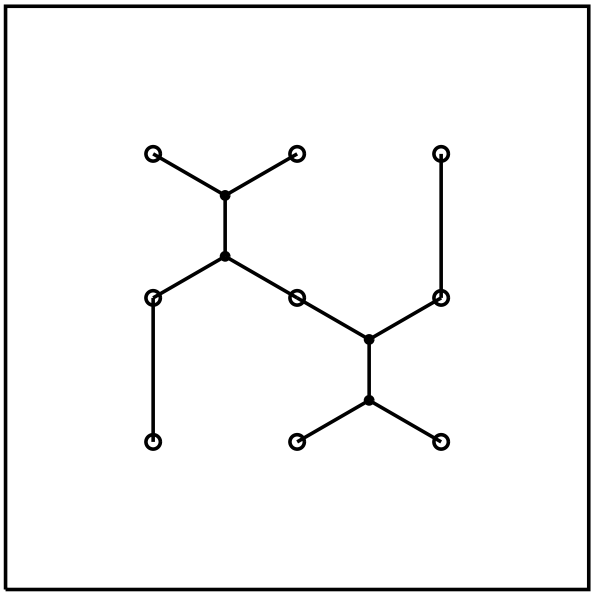

Figure 32. SMT for a 3 x 3 grid.

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.

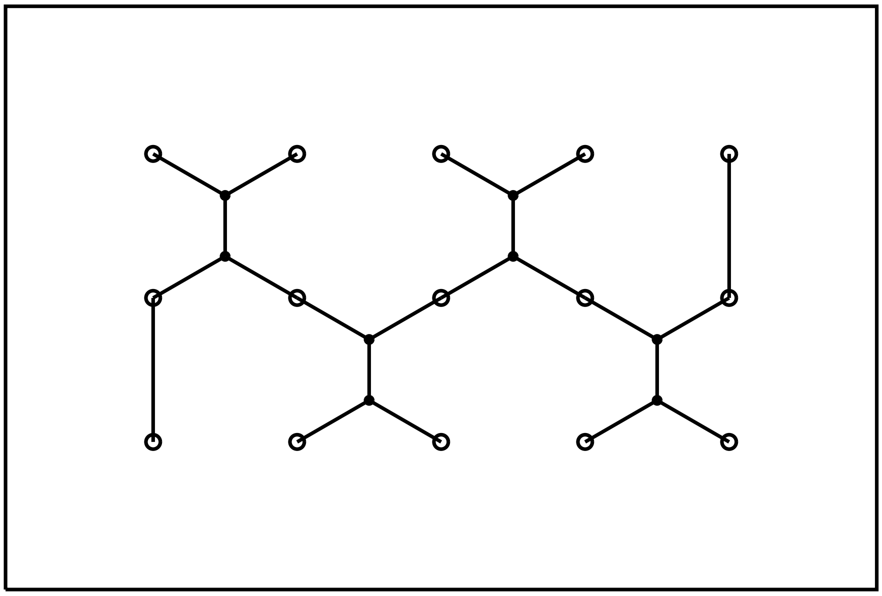

Figure 33. SMT for a 3 x 5 grid.

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 $m \geq 4$

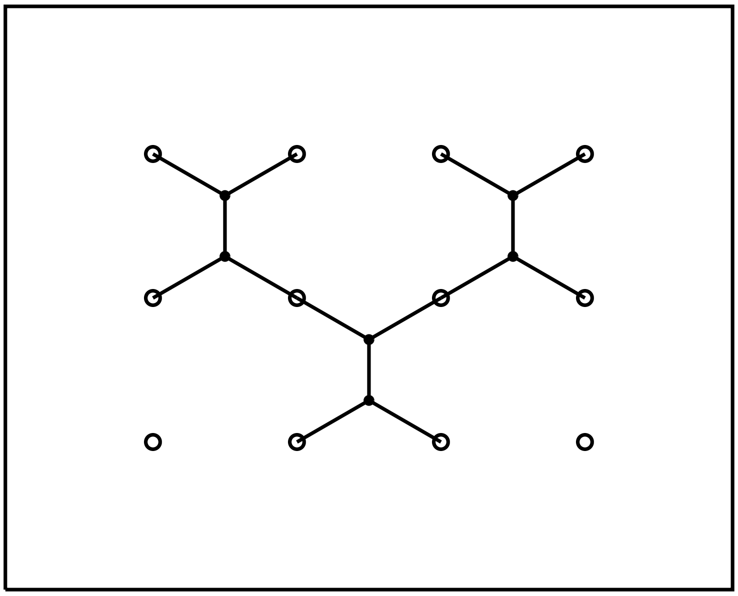

Figure 34. 4 x m pattern A.

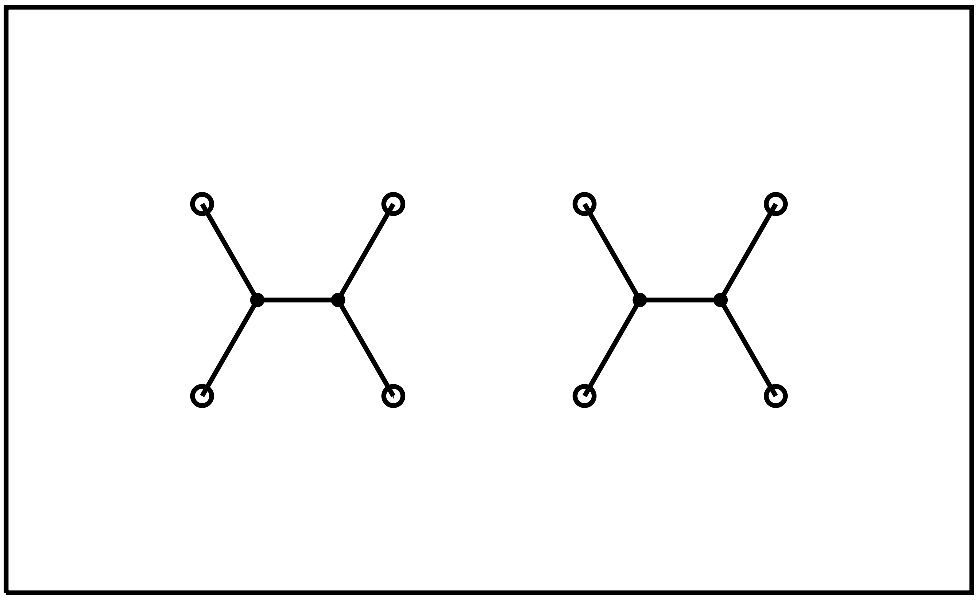

Figure 35. 4 x m pattern B.

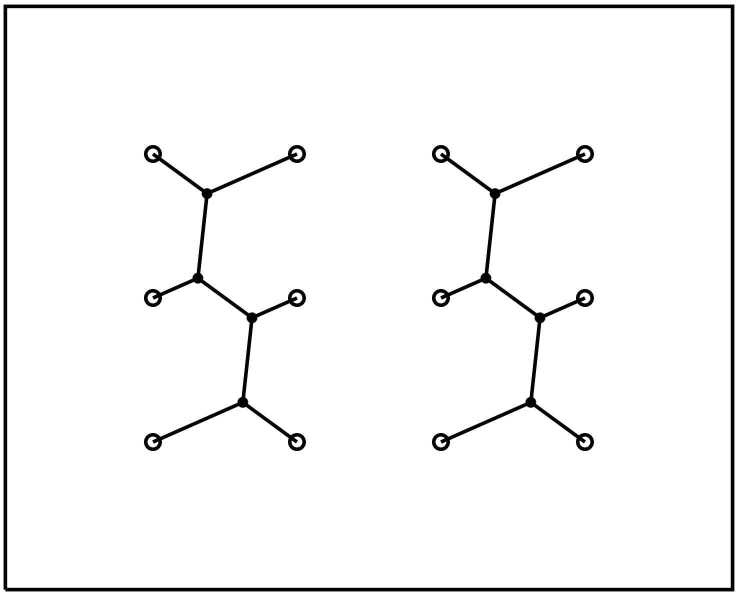

Figure 36. 4 x m pattern C.

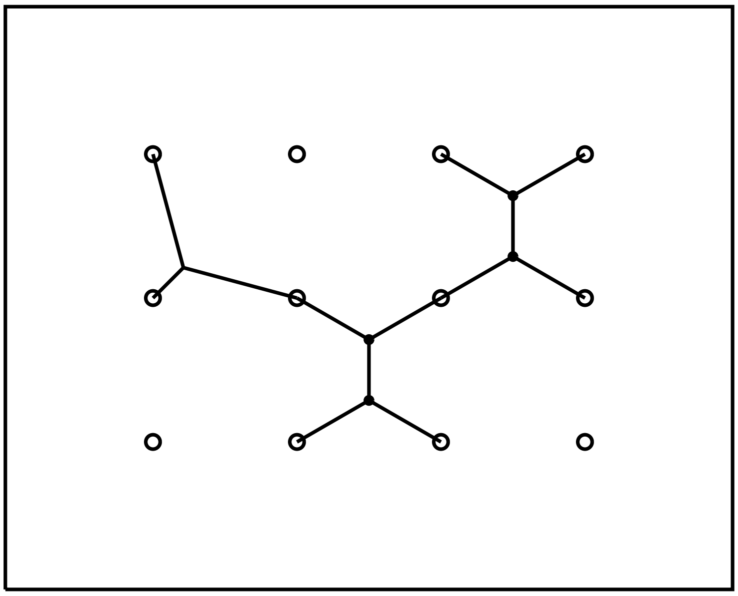

Figure 37. 4 x m pattern D.

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