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.

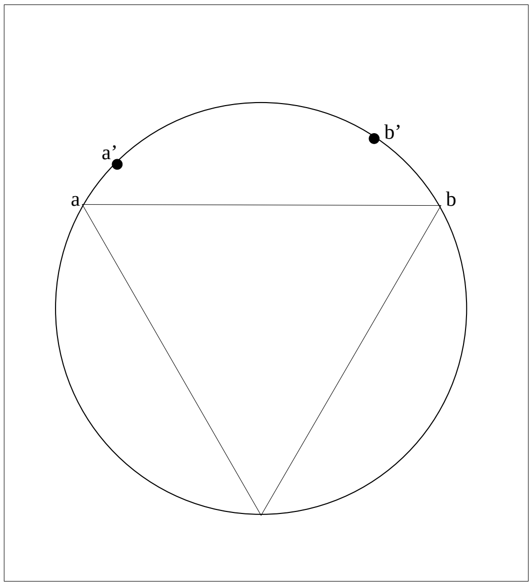

Figure 15. An illustration of Winter's pushing.

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.

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

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

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

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

STEP 5 /FSTs WITH q* AS ROOT/:
   If MIN(q* = 0 and |AQ| > 0 and LF(q*) < LF(R(q_1) < ...
   < LF(R(q_v)) < LF(R(q*)) where q_1,..., q_v are central nodes between
   LF(q*) and q* (H(q_1) < ...< H(q_y)) with H(q*) - H(q_u) = H(R(q_u)),
   u = 1, ...v,
   then (subject to tests described in Sections 4 and 5)
   SROOT(t) := j; ROOT(t) := q*;
   t := t + 1 for every j ∈ AQ with LF(R(q*)) < j;

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

STEP 7 /DETERMINATION OF SMT/:  Form unions of FSTs saved in TLIST
   (subject to tests described in Section 6); The shortest union is
   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 */
Construct_Steiner_Polygon();
Construct_Minimal_Spanning_Tree();
for(i=0; i<n; i++){ /* for all fixed points */
   LF(q) = q;
   H(q) = MIN(q) = 0;
   if(Member_of_Steiner_Polygon(q))
        LSP(q) = RSP(q) = q;
   else
        LSP(q) = RSP(q) = 0;
   L(q) = R(q) = NULL;
}
for(t=0; t<n-1; t++){ /* for all edges in MST */
   Sroot(t) = edge_i(t);
   Root(t) = edge_j(t);
}
q = n;
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 */
for(p=0; p<q; p++){
    AP = A(p);
    /* Step 3 */
    for(r=0; ((H(p) > H(r)) AND (r!=q)); r++){
        if((H(p) == H(r)) AND (r<p))
            r = p;
        if(Subset(V(r), AP)){
            p_star = p;
            r_star = r;
            for(Label = PLUS; Label <= MINUS; Label++){
                /* Step 4 */
                AQ = A(q);
                if(Construct(p_star,r_star,&(E(q)))){
                    L(q) = p;
                    R(q) = r;
                    LBL(q) = Label;
                    LF(q) = LF(p);
                    H(q) = H(p) + 1;
                    /* next line is different */
                    Min(q) = max(Min(p)-1,H(r));
                    if(Lsp(p) != 0)
                        Lsp(q) = Lsp(p)
                    else
                        Lsp(q) = Lsp(r)
                    if(Rsp(r) != 0)
                        Rsp(q) = Rsp(r)
                    else
                        Rsp(q) = Rsp(p)
                    q_star = q;
                    q++;
                    /* Step 5 */
                    if(Proper_to_Add_Tree_to_Tlist(q_star)){
                        for_all(j in AP with Lf(R(q_star)) < j){
                            SRoot(t) = j;
                            Root(t) = q_star;
                            t++;
                        }
                    }
                }
                /* Step 6 */
                p_star = r;
                r_star = p;
            }
        }
    }
}
```

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.