RC 17680 (#77873) 2/11/92
Computer Science 15 pages

Research Report

Rectilinear Steiner Tree Minimization on a Workstation

Clark Thomborson

Computer Science Department
University of Minnesota at Duluth
Duluth, MN 55812

Bowen Alpern and Larry Carter

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Rectilinear Steiner Tree Minimization on a Workstation

Clark Thomborson*
Computer Science Department
University of Minnesota at Duluth
Duluth, MN 55812
cthombor@ub.d.umn.edu

Bowen Alpern and Larry Carter
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598
carterl@watson.ibm.com, alpern@watson.ibm.com

Abstract: We describe a series of optimizations to Dreyfus and Wagner's dynamic program for finding a Steiner minimal tree on a graph. Our interest is in finding rectilinear Steiner minimal trees on k pins, for which the Dreyfus and Wagner algorithm runs in $O(k^2 3^k)$ time. The original, unoptimized, code was hopelessly I/O-bound for $k > 17$, even on a workstation with 16 megabytes of main memory. Our optimized code runs twenty times faster than the original code. It is not I/O-bound even when run on a fast 8-megabyte workstation with a slow access path to a remote disk. Our most significant optimization technique was to reorder the computation, obtaining locality of reference at all levels of the memory hierarchy. We made some improvements on the Dreyfus-Wagner recurrences, for the rectilinear case. We developed a special-purpose technique for compressing the data in our disk files by a factor of nine. Finally, we found it necessary to repair a subtle flaw in random(), the 4.3bsd Unix random number generator.

*Research supported by the National Science Foundation, through its Design, Tools and Test Program under grant number MIP 9023238.

1 Introduction

The Steiner problem has a long history in applied mathematics, dating from Fermat in the early 17th century. It also has a direct application to contemporary VLSI design. In its VLSI application, the Steiner problem is to find a minimal-length set of rectilinear edges joining a set of k pins in the plane. Such a set of edges is called a rectilinear Steiner minimal tree, or RSMT.

We use the convention of Dreyfus and Wagner throughout this paper, in which the Steiner problem is posed on an arbitrary edge-weighted graph of n nodes. In this formulation, the Steiner problem is to find the set of graph arcs of minimum total weight needed to connect a specified set of k of the n nodes in the underlying graph. We refer to the distinguished nodes of the graph as "pins," by analogy to the VLSI application.

Most variants of the Steiner problem are NP-hard. For example, given an arbitrary weighted graph, a set of k pins on that graph, and an integer L, one might ask if there exists a tree of weight at most L that contains all the given pins. This decision problem is known as the Steiner problem on graphs; it is NP-complete [15]. The problem remains NP-complete even if the edge weights are obtained, as they are in this paper, from the rectilinear distances in a planar embedding of the pins [15].

Given the NP-completeness of the problem, there is little hope of finding an exact algorithm for the RSMT that runs in subexponential time. Instead, most researchers have concentrated on developing heuristics [33, 32, 23, 19, 20, 27, 26, 25, 17, 11, 10], probabilistic analyses [8], provably-good approximation schemes[21], algorithms with guaranteed worst-case performance [7, 34], and fast algorithms for special-case sets of pins [5, 1].

The performance of a RSMT heuristic is typically measured by its expected percentage reduction over the length of the (easily-computed) minimal spanning tree, where random problem instances are obtained from pins uniformly distributed over the unit grid. If a Prim- or Kruskal-style greedy heuristic is used to grow a Steiner tree, the result is 9% shorter than the MST, on average [9]. The largest average reduction reported for any polynomial-time RSMT heuristic is 10% [17].

One of the goals of our research into exact algorithms for the RSMT is to determine the ratio between the average length of a minimal spanning tree (MST) on a random pinset and the average length of the RSMT on a pinset drawn from the same distribution. For the case of pins uniformly distributed in the unit square, our preliminary data indicates that the average RSMT is 10.7% shorter than the average MST. We tentatively conclude that there is little room for improvement over the Ho-Vijayan-Wong heuristic [17]. We intend to publish our findings in a separate paper, when our analysis and experimentation is complete.

The topic of this paper is the efficient computation of exact rectilinear Steiner trees, using the Dreyfus-Wagner method, on a Sun-4 or RS-6000 workstation with a 500-megabyte disk. Such a workstation is capable of solving any rectilinear Steiner problem of moderate size in a matter of weeks (for a 23-pin problem), or in less than a second (for problems with 10 pins or less). We believe that the techniques described in this paper are interesting in their own right, that they can be applied to other problem areas, and that our RSMT code can be used to obtain otherwise-unavailable experimental data on optimal Steiner trees.

At the outset of this project, we faced a choice between three methods for computing optimal Steiner trees on k pins. A branch-and-bound method, due to Yang and Wing[32],

runs in $O(2^{k^2})$ worst-case time. The average-time performance of the Yang and Wing algorithm is unknown. A divide-and-conquer method, due to Thomborson, Deneen and Shute [28], runs in $O(c^{\sqrt{k}\log k})$ time, for some unknown $c < 5$. Finally, a dynamic program due to Dreyfus and Wagner[12] runs in $O(k^2 3^k)$ time. We shied away from the Yang and Wing approach, fearing that its runtime would be excessive for $k > 10$. We chose not to implement the asymptotically-superior divide-and-conquer method because we believe it will not be competitive for $k < 30$. Instead, we chose to try to improve on a reasonably-efficient C-language implementation of the Dreyfus and Wagner algorithm written in 1988 by Deneen and Shute.

The Deneen-Shute RSMT code solves a non-degenerate problem on ten pins in twenty seconds on a Sun SPARCstation 1. (Degeneracy speeds things up somewhat by reducing the size of the underlying graph.) Eleven-pin problems take a minute. Thirteen-pin problems take about nine minutes. Fifteen-pin problems take eighty minutes, if sufficient main memory is available to avoid swapping. If less than eight megabytes is available, the workstation "thrashes," spending more than 99% of its time waiting for a data page to be read from disk.

Our first insight was to rearrange the order of computation of partial results in the Dreyfus-Wagner recurrences. The revised computation enjoyed a great deal of locality of reference. The Dreyfus-Wagner recurrence, and our rearrangement, is explained in Section 2.

Having rearranged the computation in order to solve the disk-thrashing problem, we obtained an unxpected benefit: our rearrangement was general enough to provide locality of reference at all levels of the memory hierarchy. We thus obtained a significant reduction in cpu time, due to less waiting for main memory fetches into cache, and less waiting for cache memory fetches into register. When our optimization work was complete, we were pleasantly surprised to find a factor-of-twenty speedup: we can solve any ten-pin problem in less than one second on our 20 MHz SPARCstation 1. Fifteen-pin problems take about four minutes; every added pin multiplies the time by three. Our code runs two to three times faster on a 25 MHz RS-6000/530.

A large part of our speedup is attributable to an algorithmic refinement on the Dreyfus and Wagner algorithm. We took advantage of the rectilinear distance matrix to save a factor of n in a sub-dominant term in the Dreyfus-Wagner asymptotic runtime. This sub-dominant term was, in fact, dominant for all $k \leq 16$, so our algorithmic improvement gave us more than a two-fold speedup for $k = 16$, and even larger speedups for smaller k. For example, at $k = 10$, we obtained nearly a five-fold speedup. Another, relatively minor, speedup was obtained after Shmuel Winograd mentioned that we could reduce our Hanan grid [16] from $n = k^2$ to $n = (k-2)^2$ points. This reduced our runtimes, and more importantly our disk space, by approximately $400\%/k$. We describe our algorithmic refinements in Section 3.

In Section 4, we discuss a special-purpose data compression algorithm that reduced the size of our disk files by a factor of nine. This sped up disk accesses markedly: our code runs at 98% cpu utilization, and spends at most 20% of its time doing file I/O, even if it is limited to about 8 megabytes on a SPARCstation. Our data compression method may be applicable to the data tables produced by other dynamic programs.

In Section 5, we describe a subtle flaw in a 4.3bsd Unix utility for generating a pseudo-random sequence by the additive congruential method. In brief, the problem is that there is a small amount of "hidden" state in the generator, so it is difficult to save the generator

state to a disk file for later restart. We devised a method for saving and restoring the complete state of the generator, creating a version called mrandom(). Section 5 discusses random() and mrandom() more fully, as well as the general problem of generating a long series of pinsets, independently and uniformly distributed in the plane.

Finally, in Section 6, we summarize our techniques and identify some areas for future work on Steiner tree codes, on codes for other combinatorial optimizations, and on optimizing compilers for workstations.

2 Reordering the Dreyfus-Wagner Recurrences

The Dreyfus and Wagner algorithm is, in essence, a pair of recurrence relations over all subsets d of the first $k-1$ pins. They suggest computing a vector of length n for each such subset d. Recall that the k pins of a Steiner problem are distinguished nodes of a weighted graph of size n.

In the case of a rectilinear Steiner problem, the underlying graph may be taken as the Hanan grid [16]: the $n \leq k^2$ nodes whose x- and y-coordinates match those of any one or two of the pins' coordinates, the 4-neighborhood edges on the n nodes, with the edge weight function being the rectilinear distance.

The value of $S[d,i]$ in the Dreyfus-Wagner recurrence is the length of a minimal Steiner tree on a subset d of the first $k-1$ pins, where the tree is constrained to include node i of the graph.

We adopt the convention that a set d is indexed by the binary value of its bit-vector. Thus $d=1$ is the singleton consisting of the first pin, $d=2^i$ is the singleton consisting of the $(i-1)$st pin, and $d=2^{k-1}-1$ is the set consisting of all but the kth pin. We write node(p) to denote the Hanan grid node corresponding to pin number p, $0 \leq \text{node}(p) < n$. Finally, we write dist$(i,j)$ to denote the rectilinear distance between two nodes i and j, $\text{dist}(i,j) = |x(i)-x(j)| + |y(i)-y(j)|$, where $x()$ and $y()$ are the x and y coordinates of a node.

With these conventions, the length of the minimal Steiner tree on the given k pins is given by the value of $S[2^{k-1}-1, \text{node}(k)]$.

For analysis and discussion, we break down the Dreyfus-Wagner computation into three phases.

1. The "initialization" phase: $T[d,i] = +\infty$.
2. The "addterm" phase updates elements of T by the recurrence $T[d,i] = \min(T[d,i], S[e,i] + S[d-e,i])$, where the minimization is taken over all proper partitions $(e, d-e)$ of d. Note that no S values are required to perform this minimization if d is a singleton.
3. The "completion" phase computes a row of S from a row of T, using the distance function on the underlying graph: $S[d,i] = \min_{j \leq n}(T[d,j] + \text{distance}(j,i))$.

The edges in the minimal Steiner tree are implicit in the completed $S[d,i]$ array. One need only examine the terms in the recurrence that generated each $S[d,i]$ value to discover a node j and a set e that acheived the minimum length tree. The node j is then part of a minimal Steiner tree whose subtrees contain pin sets e and $d-e$. A simple recursive

backtracking routine can then print out an optimal Steiner tree in $O(n2^k)$ time. We will not discuss this backtracking procedure further, as it is described adequately in [12] and we have made no effort to accelerate it. Solving the recurrences takes $O(n3^k)$ time, so the $O(n2^k)$ time required for the backtracking phase is irrelevant in theory and, we found, in practice.

Dreyfus and Wagner suggest saving space by storing both S and T in the same array. They rely on the control structure of their computer program to perform all "addterms" on a row $T[d,*]$ before overwriting that row with the "completed" values $S[d,*]$. Their control structure also ensures that every row $S[d,*]$ is completed before it is referenced by another row's addterm process.

We adopt Dreyfus and Wagner's suggestion, so our computation requires $2^{k-1}-1$ rows of n values. To save space in main memory, we represent S values as 16-bit short integers. Since most compilers do not support unsigned arithmetic on 16-bit shorts, we are thus limited to computing Steiner trees of length at most $2^{15}-1 = 32767$. Our preliminary analysis shows that this limitation poses no difficulty for any Steiner problem with less than 30 pins drawn from a 1000×1000 grid. Furthermore, we will be able to extrapolate our results on the 1000×1000 results grid to the standard model, pinsets drawn uniformly from the unit square. Finally, a slight modification to our program allows S values to be 32-bit long integers, with only a modest increase (perhaps 30%) in runtime and in disk space (less than a factor of 2).

For the case $k = 23$, the complete S array would thus occupy $2 \cdot 23^2 \cdot 2^{22} = 4.5$ gigabytes. This not only beyond the virtual memory capacity of a contemporary workstation, it would even strain the addressing capabilities of a Cray. So we are forced to write our own memory manager, swapping in portions of the S array from disk whenever these are needed.

We decided to store S in row-major format: the elements of each row $S[d,*]$ are contiguous, both on disk and in main memory. As will be seen below, this allows an efficient coding of the inner loop of our computation. Row $S[d,*]$ is stored in disk file number $\lfloor d/M \rfloor$, where $M = 2^{12}$ for our Sun-4 code and $M = 2^{13}$ on the RS-6000. Our RS-6000 has twice as much main memory (48 megabytes) as our Sun-4 (24 megabytes), so we cut our I/O overhead by bringing in larger blocks of S.

Note that the "completion" step is inherently I/O-efficient, given our row-major storage format. Care must be taken, however, in the "addterm" step to avoid excessive I/O. The Deneen-Shute control structure, shown in Figure 1, suffers from this defect. Note, however, the elegance and simplicity of their iteration over all 2-partitions $(e, d-e)$ of a given bitvector set d.

The cpu will be idle for 10 milliseconds or more whenever a page (4 or 8 Kbytes on most workstations) is paged into memory. Thus the disk time overhead per element is at least 2.5 microseconds, even if the page is full of elements that we will reference. A modern workstation can do dozens of operations in this amount of time.

As a rule of thumb, then, if you want good cpu utilization (in order to get good real-time performance), your code should use a data element dozens of times for each I/O operation on that element.

For most $d > 2^{k-2}$, the Deneen-Shute code will page through most of the S array to compute a single row $S[d,*]$. Thus if the S array does not fit into main memory, the cpu utilization will drop dramatically during the latter part of the calculation. In fact, both $S[e,i]$ and $S[d-e,i]$ will be used less than once, on average, each time they are read from

```
# define   NextSubset(e,d)   ( (d) & ((e) - (d)) )
# define   FirstElement(d)   NextSubset(0,d)
for (d=1; d < (1<<k); d++) {
    Sinit(d);
    for (e=FirstElement(d); e<d-e; e=NextSubset(e,d)) {
        Saddterm(d,e);
    }
    Sfinish(d);
}
```

Figure 1: The Deneen-Shute control structure.

disk, so the computation slows down by a factor of about 100.

The control structure suggested by Dreyfus and Wagner is only a little better in its I/O efficiency. They suggest completing $S[d,*]$ for all subsets d of cardinality c, before performing any addterms on subsets of cardinality $c+1$. Furthermore, they suggest completing each $S[d,*]$ before performing any addterms on other rows of S. The Dreyfus and Wagner control structure will therefore thrash whenever main memory is not large enough to hold about $2^k/\sqrt{k}$ rows of S, that is, the S values for all subsets d of cardinality c, $\lfloor c/2 \rfloor$, and $c - \lfloor c/2 \rfloor$.

We discovered a recursive control structure that guarantees locality of reference at all levels of the memory hierarchy. Up to now, we have only been concerned with disk memory and main memory, but modern workstations also have a data cache that can be accessed many times faster than main memory. Also, some of our files could be stored remotely on a slower disk or tape drive. Finally, a cpu might have a small fast on-chip cache, as well as a slower off-chip cache memory. It is possible to tune an algorithm to take optimal advantage of each level of the hierarchy – indeed, significant speedups can be obtained when you have good estimates for memory size and access times [29]. However, in this case we were able to design a non-parametrized control structure that does well on every level of the hierarchy, in the following sense:

Theorem 1 *The Scalc() algorithm of Figure 2 will perform $O(3^m/2^m)$ Saddterm operations, on average, on each row of S in a memory with a total capacity of 2^m rows, if the memory is managed with a least-recently-used replacement strategy.*

Since the full computation requires $O(n3^k)$ elemental operations and $O(n2^k)$ space, this theorem describes the best-possible uniform bound on operations per unit space. We will discuss this optimality property more fully in the workshop version of our paper.

We will prove the theorem below, after a little more discussion of the underlying issues and a description of the operation of Scalc().

Figure 2 shows our recursive control structure, called from the main program by Scalc(0, 0, dlim/2). Most recursive routines are difficult to understand at first, and this one is no exception. The design principle of Scalc() is to recurse "uniformly" over the d and e indices into S. In particular, it is an inefficient use of memory to hold d fixed while letting e and $d-e$ range widely. Our objective is to perform Saddterms in an order that maximizes the similarities among the sets d and e.

```
# define    IsSingleton(d)    ( (d) == firstelement(d) )
void Scalc(d, e, PinVal)
long d, e, PinVal;
{
    if (PinVal == 2 && e != 0) { /* compute 9 "addterm" leaves */
        Sadd9terms(d, e);
    } else if (PinVal == 0) { /* leaf node, general case */
        if (d != 0) { /* note: S[0] is never referenced */
            if (IsSingleton(d-e)) {
                Sinit(d);
            }
            if (e == 0) { /* no more addterms are needed */
                Scomplete(d);
            } else { /* add term (e,d-e) to row d */
                Saddterm(d, e);
            }
        }
    } else { /* internal node: 3-way recursion except on left spine */
        Scalc(d, e, PinVal/2); /* (0,0) subtree */
        if (d != 0) {
            Scalc(d+PinVal, e+PinVal, PinVal/2); /* (1,1) subtree */
        }
            Scalc(d+PinVal, e, PinVal/2); /* (1,0) subtree */
        }
    }
}
```

Figure 2: Our recursive control structure Scalc().

Note that d, e, and $(2^{k-1}-1)-d-e$ form a 3-way partition over the set $\{0,1,...k-2\}$ of pins. Our recursive control structure is thus applicable to any combinatorial algorithm that is based on an enumeration of all 3-way partitions of a set.

A typical "leaf" of our recursion is an Saddterm for a (d,e) pair, where e is a proper subset of d that contains highest-numbered pin of d. Special-case leaves, separated out by the if-then testing of the non-recursive prefix of Scalc(), cause rows of S to be initialized or completed at appropriate times. Nine related Saddterm leaves may be computed at once, saving on register loads and stores. More on this optimization later.

The subsetting of d and e is accomplished by the 2- or 3-way recursion of the last nine lines of Scalc(). The "(0,0) subtree" corresponds to leaving a pin out of d, and therefore out of e. The pin currently under consideration has value "PinVal" in the binary representation of a subset. The "(0,1) subtree" corresponds to including the current pin in d, but not in e. This subtree is not generated unless d already has some pins in it (dlow != 0), thereby insuring that e will contain the highest-numbered pin of d. Finally, the "(1,1)" subtree corresponds to the inclusion of the current pin in both d and e.

Proof of Theorem 1. The reference-locality property of our Scalc() recursion is a reflection of the fact that each subtree is completed before the next is started. Just three blocks of storage are referenced in the $O(3^h)$ leaves of a subtree of height $h = \log_2(\text{PinVal})$, namely those rows $S[x,*]$ with x in one of the three ranges $d \le x \le d+\text{PinVal}$, $e \le x \le e+\text{PinVal}$, and $d-e \le x \le d-e+\text{PinVal}$. If $3 \cdot 2^h \le 2^m$, then all three blocks will fit in a memory capable of holding 2^m rows. During the computation of a subtree of height h, no reference will be made to any row of S outside of these blocks, so a least-recently-used replacement policy will not discard any row within the blocks until all rows outside the blocks have been overwritten. □

We take explicit advantage of the memory access pattern of Scalc() by keeping just three blocks of S in memory at any time. File reads and writes occur only when the recursion variable "PinVal" is equal to the number of rows of S in a file.

We note, in passing, that we could avoid using file I/O if we had sufficient virtual memory in our workstations. Our recursive control structure, and the demand-paging strategy of a typical workstation operating system, would do a good job of optimizing its use of all data pages currently available in main memory. However, our use of explicit file I/O leads to significant practical advantages. First, we gain a constant-factor speedup because we can predict the pattern of file accesses, reading a large file as a single block instead of accessing portions of it upon demand. More importantly, since we compile our Sun4 program to use at most 8 megabytes of main memory (16 megabytes on the RS-6000), we can run our programs in the background with a "nice" cpu priority, without disturbing other uses of the workstation. In particular, our code will not cause other programs to be paged-out, so we do not lose the good response-time on multiple windows that one comes to expect from a workstation.

As mentioned above, the uniform reference-locality property of our recursion also allows our program to make good use of all lines of data cache that are available to it, at any given time.

Locality of reference at the register level is a little harder to arrange. Early versions of our program had no test for PinVal = 2; all Saddterms were computed one row at a time with a literal translation of the Dreyfus-Wagner recurrence: "for $(i=0; i<n; i++)$ if $(S[d,i] > S[e,i] + S[d-e,i])$ $S[d,i] = S[e,i] + S[d-e,i]$);".

Optimized Sun4 code for the Saddterm inner loop required 12 statements (that is, 12

clock ticks) for each min-add operation, in the overwhelmingly-common case that $S[d,i]$ is not updated. The Sun compiler (cc -O4) did not unroll this loop. Even if it had done so, at a minimum we would need three memory loads, one addition, one compare, and one branch, for a total of six clocks per min-add. If, instead, we interleave addterm operations on adjacent rows $S[d,i]$, $S[d+1,i]$, $S[d+2,i]$, and $S[d+3,i]$, for any $d == 0 \bmod 4$, we can amortize the memory references to the eight data elements $S[e,i]$, $S[e+1,i]$, $S[e+2,i]$, $S[e+3,i]$, $S[d-e,i]$, $S[d-e+1,i]$, $S[d-e+2,i]$, and $S[d-e+3,i]$, over nine min-add steps. An ideal Sun4 compiler would then issue code requiring just 4-1/3 clocks per min-add, since just $12/9 = 4/3$ memory reads are sufficient, on average, for a min-add step.

Despite extensive experimentation with various C-language constructs, we are still unable to persuade Sun's "cc -O4" optimizer to produce code that took less than 8 clocks per min-add in Sadd9terms(). In contrast, it did not take us long to write specialized C code for Sadd9terms() on the RS-6000 that runs at approximately 5 clocks per min-add. The limiting speed of the RS-6000 on the Sadd9terms() inner loop is just 3-1/3 clocks per min-add: 1-1/3 clocks for the memory accesses (the same as the Sun4), one clock for the addition, and one for the comparison. The branches add no latency on an RS-6000, if the comparison result is available sufficiently far in advance of the branch instruction.

Given enough incentive and/or inspiration, we believe we could write machine-specific versions of Sadd9terms() that would run close to the limiting speed on both the Sun4 and the RS-6000, thereby speeding up our code appreciably (perhaps by 60%). Additional speedups could then be obtained by interleaving even more addterms: we might try to write an Sadd27terms() routine that would perform $3^3 = 27$ min-add steps with only $3 \cdot 2^3 = 24$ data reads, in an effort to get below 3 clocks per min-add on the RS-6000 and below 4 clocks per min-add on the Sun4.

3 Algorithmic Improvements

Dreyfus and Wagner proposed their dynamic program as a method for solving the Steiner problem on graphs. In our application, the graphs have special structure which we were able to exploit for significant speedup.

First we realized that the Dreyfus-Wagner "completion" recurrence $S[d,i] = \min_{j \le n}(T[d,j] + \text{distance}(j,i))$ is inefficient for a graph of bounded degree, or indeed for any sparse graph. It would be better to perform a breadth-first search, or some other "wavefront propagation" scheme, calculating an entire row $S[d,*]$ in time proportional to the number of edges in the underlying graph. This way, we could calculate an entire row of S in $O(n)$ time, rather than the $O(n^2)$ time of the Dreyfus-Wagner recurrence.

Further consideration revealed that our grid graphs have another special property. A minimal-length path between any two nodes i and j can always be formed from some rotation of an L-shaped set of edges: either going left then up, or up then right, or right then down, or down then left. We thus rewrite the Dreyfus-Wagner completion recurrence as a four-step process:

```
for (i=0;   i<n ;i++) S[d,i] = (S[d,i], S[d,left(i)]  + ldistance[i]);
for (i=n-1;i>=0;i--) S[d,i] = (S[d,i], S[d,right(i)] + rdistance[i]);
for (i=0;   i<n ;i++) S[d,i] = (S[d,i], S[d,up(i)]    + udistance[i]);
for (i=n-1;i>=0;i--) S[d,i] = (S[d,i], S[d,down(i)]  + ddistance[i]);
```

Here left(i) is the left neighbor of node i, if any, otherwise it is the index of a special node at distance ∞ from every other node. The other node adjacency functions right(), up(), and down() are defined similarly. We index the nodes so that left(i) $= i - 1$ and up(i) $= i + k_x$, where k_x is the number of distinct x coordinates in our Hanan grid. This adjacency function is not accurate at the boundary of the grid, however in this routine we need only to have very large values stored in the various distance matrices to represent the length of an edge to a non-existent neighbor. Also note that we must iterate in different node orders, depending on the direction of "wavefront propagation."

As can be seen from the above, our revised Sfinish() code updates each array element $S[d, i]$ at most four times, with a very low amount of overhead. Thus Sfinish() never dominates the runtime of our code, except possibly for very small k.

As mentioned in the introduction, we made a second algorithmic improvement at the suggestion of Shmuel Winograd. He thought that his idea was probably well-known, and indeed it did not surprise us — but we hadn't thought of doing it on our own. The idea is simple. Without increasing the Steiner tree length, the pin at the maximal x-coordinate can be connected by a horizontal edge to the Hanan grid node of next-smaller x-coordinate and identical y-coordinate. We thus run the Dreyfus-Wagner recurrence over a smaller Hanan grid, with the x-maximal column of nodes removed, and with the x-maximal pin mapped onto its x-neighbor Hanan grid point. A simple post-processing step can add the eliminated horizontal edge to the Steiner tree produced by the Dreyfus-Wagner recurrences.

By similar reasoning, we can strip the x-minimal column from the Hanan grid, as well as the y-extremal rows. In the end, we are left with just $n = (k-2)^2$ nodes for the Dreyfus-Wagner recurrences, reducing our time and space by a factor of $1 - k^2/(k-2)^2$, or approximately $400\%/k$.

Our grid reduction has a non-trivial chance of reducing the number of pins in our problem, even for non-degenerate point sets. Two pins will coincide after the grid reduction if they were among the four nodes forming a square at the upper-left corner of the Hanan grid, or indeed if they are in the four-node square at any of the corners of the grid. More formally, if we represent the pins by listing their y-ranks, after sorting the pins by their x-rank, any of the $k!$ permutations of the values 1 through k are equally likely to appear if the pins' x- and y-coordinates are drawn from independent distributions. There are eight patterns of y-ranks that will lead to a reduced number of pins after the grid is reduced, namely $(1, 2, ...)$, $(2, 1, ...)$, $(k, k-1, ...)$, $(k-1, k, ...)$, $(..., 1, 2)$, $(..., 2, 1)$, $(..., k, k-1)$, and $(..., k-1, k)$. The probability of observing at least one of these patterns is thus $8(k-2)!/(k)! = 8/k^2$. For pinsets of cardinality $k = 20$, we therefore have a 2% chance of needing to solve a Dreyfus-Wagner recurrence on 19 or fewer pins. Our computation speeds up by a factor of 3 whenever a pin is removed. On average, then, the speedup due to "cornerpoint pin reduction" is $(3-1)8/k^2$, or 4% for $k = 20$. For $k = 10$, the speedup is much more dramatic: 16% on average.

It may well be possible to obtain more dramatic speedups, on average, by additional reductions to the Hanan grid. Near the boundary of the grid, one might be able to assert the existence of more than the four "Winograd" edges in a minimal Steiner tree.

Another possible avenue for algorithmic improvements is suggested by the research of Eppstein *et al.* into the structure of dynamic programming [13]. We see no direct application of the Eppstein-Galil-Giancarlo ideas to the Dreyfus-Wagner recurrences, but of course this does not prove that such an application does not exist.

4 Data Compression

Data compression on our temporary files afforded us notable savings in space and time. Our research in this area was mostly *ad hoc*. We merely applied standard techniques until we obtained good compression ratios.

Our earliest code formed a Unix pipe to the 4.3bsd system utility "compress." This is a Ziv-Lempel compression method that does a good job of finding near-field correlations on a byte-by-byte basis. Our $S[d,i]$ data is, however, organized on a much grander scale: the individual elements are 2-byte integers, not single bytes; an $S[d,i]$ element has significant correlations not only to its (near-field) right- and left- neighbors $S[d,i+1]$ and $S[d,i-1]$, and to its (distant) up- and down- neighbors $S[d,i+kx]$ and $S[d,i-kx]$, but also to its (very distant) "set-neighbors" $S[d',i]$ where d' is at unit Hamming distance to d. Thus we were pleasantly surprised to obtain a 3:1 compression ratio from the "compress" utility.

We took our next idea from picture compression: coding an S value by its difference from a linear predictor. Our first predictor was derived from the familiar 2×2 Laplacian difference operator on the four values in its down-left neighborhood, namely $S[d,i] = S[d,i-1] + S[d,i-k_x] - S[d,i-k_x-1] + \Delta$, where k_x is the number of columns in our reduced Hanan grid. If there are fewer than four S values in a down-left neighborhood, as occurs for nodes in the first row and the first column of the reduced Hanan grid, we filled in zeros for the missing S terms.

Since most Δ values were zero, we transmitted the length of any sequence of successive zeros with a one- or two-byte sequence. A one-byte code sufficed for all runs of at most 16 zeros, Runs of up to 256 zeros required two code bytes, and longer runs were decomposed into runs of length at most 256. To obtain longer runs, we adjusted the transmission order of the Hanan grid, transmitting the first row and column of the grid (i.e., the Δs with incomplete predictor functions) before sending the rest of the grid. We sent non-zero Δ values with one or more code bytes, depending on the magnitude of the value.

Our compression ratios improved dramatically when we incorporated set-neighborhood correlations into our scheme. For computational convenience, we stored the array in Gray code order, $d = d' \oplus (d'/2)$, so that row $S[d',*]$ was stored on disk immediately after row $S[d'-1,*]$. By a well-known property of Gray codes, the set d corresponding to the (d')th row of a disk file is at unit Hamming distance (*i.e.* differs by just one pin) from the set corresponding to the $(d'-1)$th row of the file. It was thus easy to take a second-order difference, transmitting only the difference between highly-correlated, file-row-adjacent, Δ values: $\Delta^2(d',i) = \Delta(d',i) - \Delta(d'-1,i)$. The first row in each disk file is encoded with our previous scheme, as a string of encoded Δ values.

Our compression ratios vary somewhat from file to file within a given computation, from random pinset to random pinset drawn from the same distribution, and (most dramatically) when the x-range or y-range of the distribution is varied. Typically, we see 9:1 compression for files of 2^{12} rows when the pins are drawn from a 1000×1000 grid. For this reason, we believe that a 500-megabyte disk should be sufficient for a 23-pin problem. (Unfortunately, we have access to only 300 megabytes of scratch space on disk, so we are only able to run 22-pin problems at the present time.)

Better compression ratios could no doubt be acheived with more complicated, or more clever, schemes. Our present scheme has the virtues of speed and simplicity, requiring only a few dozen instructions on average to compress or decompress an average element of S. The 9:1 ratio we typically observe is sufficient to remove disk access time as a

consideration in total runtimes. With uncompressed files of 2^{12} rows, our Sun4 ran at only 80% cpu utilization. With compressed files of 2^{12} rows, we run at 98% cpu utilization, even though it takes almost a second to open a file, and even though we can only read files at 100 or 200 Kbytes/second over our Ethernet connection to a remote fileserver.

5 Generating Random Pinsets

Originally, we generated pinsets uniformly distributed in a 1000×1000 grid by the following method. We initialized the 4.3bsd random() additive congruential generator by calling srandom(seed), where we took the value for seed from a disk file. We called the generator twice for each pin, once for each coordinate, extracting the coordinate values from the most-significant-bits of the pseudorandom integer:

```
for (i=0; i<k; i++) {
    x[i] = (long)( (float)random() / (float)MAXLONG * 1000.0);
    y[i] = (long)( (float)random() / (float)MAXLONG * 1000.0);
}
```

Then we called random() one more time, writing the return value into the seed file for use in reseeding the generator on the next program run.

This process worked fine until one day it looped on a series of 16-pin sets: the 33rd number in random()'s sequence when initialized with a seed of 1380895877 is the same number, 1380895877.

We then tried to save random()'s entire state to a disk file, using its setstate() and initstate() calls. This was trickier than it seemed at first, since random() maintains some state information in private local variables. Unless these variables are saved and restored along with the state table, random() will not be restartable. In a pathological case, if only one call to random() is made on each program run, the resulting "random number sequence" is just an arithmetic progression.

We finally resolved our difficulties, within the constraints of the random() interface, when we discovered that random()'s internal state varies in a cyclic fashion, modulo its table size. We thus make sure that random() is called $31j$ times, for some j, on each program run. In this way, we can obtain the same sequence from a restarted generator as we would have seen if the generator were kept running undisturbed. We therefore expect to enjoy the long-period guarantees of the additive congruential scheme.

So far, we have observed no defect in our point generation scheme. The observed distribution and average length of trees for 2-pin sets matches the theoretical distribution (a third-order polynomial with a breakpoint) very nicely. The observed average length of Steiner trees for 3-pin sets matches the theoretically-predicted average length. And the observed number of "corner-pin reductions" for 16-pin sets matches the value derived in Section 3.

Random pinsets could also be obtained from a scheme based on a multiplicative congruential generator, such as the 4.3bsd rand() or rand48(). One should be cautious with such generators, however. Frequently-recommended 32-bit moduli and multipliers (*e.g.* the minimal standard generator [24] RNUN in the IMSL library) produce pinsets with noticeable "stripes" [22]. A combined scheme, based on two multiplicative generators, can be used to overcome this defect [22].

6 Conclusion

We have described a method for computing rectilinear Steiner minimal trees on up to 23 pins on a contemporary workstation with a 500-megabyte disk. Our method is based on the Dreyfus-Wagner dynamic program for solving Steiner problems on arbitrary graphs[12].

To keep the Dreyfus-Wagner algorithm from "thrashing" a workstation's disk, we found it necessary to reorder the computation. Our reordering is compactly expressed by a recursive control program. It is provably the best such reordering for maintaining locality of reference across the entire memory hierarchy. Thus it not only prevents disk thrashing, it speeds up the computation by keeping an appreciable fraction of its data references within whatever portion of the cpu cache is available to the running process.

A theory of computational reordering is emerging: there is a body of previous work on ordering matrix multiplications, Fourier transformations and sorting algorithms for locality [14, 18, 2, 3, 4, 6, 29, 30]. However, no one has previously studied iterations over three-way partitions of sets. Such partitions are encountered in other forms of dynamic programming, so our recursive control structure may well be useful in other combinatorial problems.

Someday, perhaps, the problem of ordering a set of recurrences for efficient computation on a workstation will be well-enough understood to be made part of an optimizing compiler. Also, some future operating system might be able to take runtime "hints" from the code emitted from a re-ordering compiler, causing it to initiate paging activity in advance of a forthcoming demand, and to make more accurate decisions about which data pages to discard or write-back to disk.

As described in Section 3, we obtained significant speedups by rewriting the Dreyfus-Wagner recurrences in view of the underlying rectilinear metric space in our application, and by reducing the Hanan grid. We have thus made a small contribution to the theory of Steiner tree algorithms. One could hope for further reductions in the Hanan grid that would lead to further speedups in the Dreyfus-Wagner recurrences, as well as improvements in the number of terms required to compute the dominant "addterm" recurrence.

We documented some inefficiencies in the code emitted by optimizing compilers for our inner "addterm" loop, indicating that another factor of two speedup on our code is available, either by low-level optimizations or by compiler improvements. We believe that such optimizations can and should be handled by the compiler, so we hope that Section 3 of our paper will be used as a case study by compiler designers.

In Section 4, we illustrated how known techniques in the field of picture compression can be applied to the problem of storing large data arrays on disk. The resulting arrays are not only nine times smaller but, because they are smaller, they can be accessed more rapidly.

Section 5 documented a problem with the 4.3bsd random() utility. This should serve as a cautionary tale for anyone trying to collect experimental data on probabilistically-generated objects, especially if these objects are computed from point sets uniformly distributed in the unit square. Our resolution of our problem with random() should be of interest to anyone using this pseudorandom generator. Unfortunately, we cannot guarantee that random() will not exhibit some other defect in the future. Future research in this area is of vital importance, so that we can make statistically-valid conclusions on the basis of pseudorandom geometric experimentation.

We are currently using our Steiner code to collect data on sets of 2 to 23 pins chosen uniformly over a large grid. Tentatively, we assert that a rectilinear Steiner minimal tree on 10 to 20 pins is about 10.7% smaller than a rectilinear minimal spanning tree, on average, and that the ratio appears to be constant for $k > 10$. We are thus able to assert that the best high-speed heuristics currently available are almost as good as any possible heuristic, in that they deliver about a 10% reduction.

A possible future use of our code is as a subroutine in the Komlos-Shing approximation scheme [21]. In this scheme, one uses an exact Steiner algorithm on $O(\log n)$ pins to solve an n-pin problem with vanishingly-small relative error. It would be interesting to see how the Komlos-Shing approximation scheme behaves in a practical setting, where our code can be called as a subroutine for problems of size k, for some k small enough for rapid computation with our code. Recent results of Deneen and Dezell suggest that the planar dissection method proposed by Komlos and Shing, while effective asymptotically, is ineffective when there are only 5 to 8 points, on average, in the pinsets for which optimal Steiner trees are generated. The resulting Steiner trees are longer, on average, than the minimum spanning tree [10]! Clearly, more algorithmic work is needed if the Komlos-Shing idea is ever to be translated into a practical heuristic.

We did not fully explore all known algorithmic options for computing exact rectilinear Steiner trees. Instead we concentrated our efforts on the Dreyfus-Wagner recurrences[12]. It may well be that a branch-and-bound technique [32] will eventually prove superior in average case runtime, possibly in conjunction with a divide-and-conquer method that takes advantage of the planarity and rectilinearity of the problem [28].

7 Acknowledgements

We are grateful to Linda Deneen and Gary Shute of the University of Minnesota–Duluth for making their Dreyfus and Wagner implementation, and other graph-theoretic source codes, available to us. Much of this research was accomplished while Clark Thomborson was on a one-month appointment as an Academic Visitor at IBM's T. J. Watson Research Center.

References

[1] P. K. Agarwal and Man-Tak Shing. Algorithms for special cases of rectilinear Steiner trees: I. points on the boundary of a rectilinear rectangle. *Networks*, 20:453–485, 1990.

[2] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A Model for Hierarchical Memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, May 1987.

[3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical Memory with Block Transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, October 1987.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[5] A. V. Aho, M. R. Garey, and F. K. Hwang. Rectilinear Steiner trees: Efficient special-case algorithms. *Networks*, 7:37–58, 1977.

[6] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, October 1990.

[7] Piotr Berman and Viswanathan Ramaiyer. An approximation algorithm for the Steiner tree problem. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, October 1991.

[8] Marshall W. Bern. Two probabilistic results on rectilinear Steiner trees. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 433–441, May 1986.

[9] Marshall W. Bern and Marcio de Carvalho. A greedy heuristic for the rectilinear Steiner tree problem. Technical Report UCB/CSD 87/306, Computer Science Division, U.C. Berkeley, 1987.

[10] L. L. Deneen and J. B. Dezell. Using partitioning and clustering techniques to generate rectilinear steiner trees. In *Second Canadian Conference on Computational Geometry*, August 1990.

[11] Linda L. Deneen and Gary M. Shute. A plane-sweep algorithm for rectilinear Steiner trees with deferred connections. Technical Report 90-6, University of Minnesota, Duluth, 1990.

[12] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972.

[13] David Eppstein, Zvi Galil, and Raffaele Giancarlo. Speeding up dynamic programming. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, October 1988.

[14] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum, 1972.

[15] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal of Applied Mathematics*, 32(4):826–834, June 1977.

[16] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal of Applied Mathematics*, 14(2):255–265, March 1966.

[17] Jan-Ming Ho, Gopalakrishnan Vijayan, and C. K. Wong. New algorithms for the rectilinear Steiner tree problem. *IEEE Transactions on Computer-Aided Design*, 9(2):185–193, February 1990.

[18] J. W. Hong and H. T. Kung. I/O complexity: The Red-Blue Pebble Game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, May 1981.

[19] F. K. Hwang. The rectilinear Steiner problem. *Design Automation and Fault Tolerant Computing*, 2:303–310, 1978.

[20] F. K. Hwang. An $O(n \log n)$ algorithm for suboptimal rectilinear Steiner trees. *IEEE Transactions on Circuits and Systems*, CAS-26(1):75–77, January 1979.

[21] János Komlós and M. T. Shing. Probabilistic partitioning algorithms for the rectilinear Steiner tree problem. *Networks*, 15:413–423, 1985.

[22] Pierre L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–774, June 1988.

[23] Jerry H. Lee, N. K. Bose, and Frank Kwangming Hwang. Use of Steiner's problem in suboptimal routing in rectilinear metric. *IEEE Transactions on Circuits and Systems*, CAS-23(7):470–476, July 1976.

[24] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[25] Dana Richards. Fast heuristic algorithms for rectilinear Steiner trees. *Algorithmica*, 4:191–207, 1989.

[26] Michal Servit. Heuristic algorithms for rectilinear Steiner trees. *Digital Processes*, 7:21–32, 1981.

[27] J. MacGregor Smith, D. T. Lee, and Judith S. Liebman. An $O(n \log n)$ heuristic algorithm for the rectilinear Steiner minimal tree problem. *Engineering Optimization*, 4:179–192, 1980.

[28] Clark D. Thomborson, Linda L. Deneen, and Gary M. Shute. Computing a rectilinear Steiner minimal tree in $n^{O(\sqrt{n})}$ time. In A. Albrecht, editor, *Parallel Algorithms and Architectures*, pages 176–183, Berlin, June 1987. Akademie-Verlag.

[29] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory i: Two-level memories. Technical Report CS–90–21, Department of Computer Science, Brown University, September 1990.

[30] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory ii: Hierarchical multi-level memories. Technical Report CS–90–22, Department of Computer Science, Brown University, September 1990.

[31] Y. F. Wu, P. Widmayer, and C. K. Wong. A faster approximation algorithm for the Steiner problem in graphs. *Acta Informatica*, 23:223–229, 1986.

[32] Y. Y. Yang and Omar Wing. Optimal and suboptimal solution algorithms for the wiring problem. In *IEEE International Symposium on Circuit Theory*, pages 154–158, 1972.

[33] Y. Y. Yang and Omar Wing. Suboptimal algorithm for a wire routing problem. *IEEE Transactions on Circuit Theory*, pages 508–510, September 1972.

[34] A.Z. Zelikovsky. An 11/8-approximation algorithm for the Steiner problem on networks with rectilinear distance. unpublished manuscript, March 1991.