ORSA Journal on Computing
Vol. 4, No. 4, Fall 1992

0899-1499/92/0404-0387 $01 25

Fast Algorithms for Geometric Traveling Salesman Problems

JON JOUIS BENTLEY / *AT & T Bell Laboratories, Murray Hill, NJ 07974, EMAIL: jlb@research.att.com*

(Received: December 1990; accepted: December 1991)

This paper describes efficient algorithms for computing approximate traveling salesman tours in multidimensional point sets. We describe implementations of a dozen starting heuristics (including Nearest Neighbor and Farthest Insertion) and three local optimizations (including 2-Opt and 3-Opt). Experiments indicate that most of the algorithms run in $O(N \log N)$ time on uniform data sets, and many run almost as fast on very nonuniform data. The program that implements the algorithms is able to solve uniform planar million-city traveling salesman problems to within a few percent of optimal in several midicomputer CPU hours. The algorithms and program apply to many distance metrics and dimensions.

The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization. The book edited by Lawler, Lenstra, Rinnooy Kan and Shmoys[29] is devoted to the TSP; the first chapter traces the history of the problem and shows the central role it has played in several disciplines over the last half century. Because the TSP is NP-complete, considerable effort has been devoted to approximation algorithms that produce good, if not optimal, tours. Johnson[23] provides a recent survey of TSP algorithms that use local optimization.

This paper concentrates on the geometric TSP. The input is a set of N points in K-space, and the output is a permutation of the points that represents a TSP tour. The cost of the tour is the sum of the distances between adjacent points under a specified metric, such as the Minkowski L_1, L_2 or L_∞ metrics. Geometric versions of the TSP remain NP-hard; see Garey and Johnson.[21]

In this paper we consider the problem of efficiently computing approximate TSP tours for K-dimensional sets under several metrics. We describe algorithms for a dozen different starting heuristic tours and for several local optimization heuristics. The best known worst-case bound for most of the algorithms is $O(N^2)$, or worse. All of the algorithms have been implemented (in the C++ language) and studied empirically. Although we are unable to analyze the algorithms mathematically, heuristic arguments and experimental data suggest that most have a run time on uniform data that grows as $O(N \log N)$. Experiments show that many of the algorithms are robust enough to process nonuniform data sets efficiently.

The program that implements the heuristics is able to find good approximations for million-city planar TSPs on a midicomputer (a VAX-8550) in a small amount of time. As a benchmark for the machine's performance, a distance calculation costs about 20 microseconds; four hours of CPU time corresponds to roughly $40N \lg N$ distance calculations. In a few minutes, the program computes a million-city tour within 60% of a lower bound; in a few hours, it computes a tour within 4% of the bound. An algorithm that uses $2N^2$ distance calculations would require over a year.

This paper describes algorithms with unproven time complexity. There are several compelling reasons for studying fast algorithms whose speed is supported only by experiment and heuristic arguments. Johnson[23] cites several applications that require approximate tours for N up to 1.2 million cities; the algorithms in this paper are likely candidates for such applications (some algorithms have been used in Bell Laboratories applications with $N \approx 10^5$). The algorithms have already served as tools for studying the lengths of the various TSP heuristics. These heuristic algorithms also allow us to conjecture that algorithms with provably good expected run time exist for these problems.

This paper concentrates on efficient implementations of the heuristics and their run times. Bentley[8] provides a more thorough description of the experimental methods used in analyzing these algorithms, with an emphasis on techniques of exploratory data analysis. Bentley[10] describes the software tools used in the experiments and general techniques for the experimental analysis of algorithms. The focus of this paper is on the time required to compute the heuristic tours; the lengths of the tours are mentioned briefly in Section 5. Details on the length of the tours will be described by Bentley, Johnson, McGeoch and Rothberg.[15]

Section 1 of this paper starts by examining the Nearest Neighbor heuristic in detail. That exercise illustrates the supporting data structures and analytic techniques that will be used throughout the paper. The section then studies two more complex heuristics, which allow us to introduce most of the tools that will be used later. Section 2 describes Addition and Insertion heuristics. Section 3 describes three

Subject classifications: Analysis of algorithms traveling salesman

heuristics based on trees: the Minimum Spanning Tree heuristic, Christofides' heuristic, and a variant of Karp's recursive partitioning heuristic.

Section 4 describes the 2-Opt and 3-Opt heuristics for improving tours by local optimizations, these improvements can be applied to any of the starting tours. Section 5 compares the run time and the tour lengths of the various tours; readers who get bogged down in the details of the various algorithms might want to skim ahead to this section. Conclusions and directions for further research are offered in Section 6.

Appendix A1 describes several supporting algorithms and data structures. All of the experiments in the body of the paper are conducted for planar point sets under the Euclidean metric; Appendix A2 describes experiments for other metrics and dimensions. Appendix A3 describes the program that implements the heuristics.

1. Heuristics That Grow Fragments

In this section we will study three heuristics that enlarge fragments of tours. While the tour is being built, each fragment is a path among a subset of the points. Finally, a path among all N points has its two ends connected to yield a tour. The first two heuristics grow a single fragment, while the third heuristic grows many fragments simultaneously.

The Nearest Neighbor Heuristic

The Nearest Neighbor (NN) TSP heuristic starts at an arbitrary point, and successively visits the closest unvisited point. Figure 1 shows an NN tour through a uniform set of 100 points after 10, 50, and 100 edges have been added. An NN tour grows its single fragment only at one end; we will soon briefly consider the Double-Ended Nearest Neighbor (DENN) heuristic, which allows the fragment to grow at both ends. Flood[19] refers to NN as the "next closest city method." Rosenkrantz, Stearns and Lewis[38] analyze the heuristic, and show that the length of a geometric NN tour is never more than $(\lceil \lg N \rceil + 1)/2$ times the length of the optimal tour.

In this section we will study in detail a fast implementation of NN. The algorithm can be described in pseudocode (see Exhibit 1). The algorithm starts at point *StartPt*, and fills the array *Perm* with the points on the tour in NN order. It assumes a data structure with three primary operations: *Build* constructs the data structure from an input point set, *DeletePt*(I) deletes point number I, and *NN*(I) returns the index of the nearest neighbor to point I.

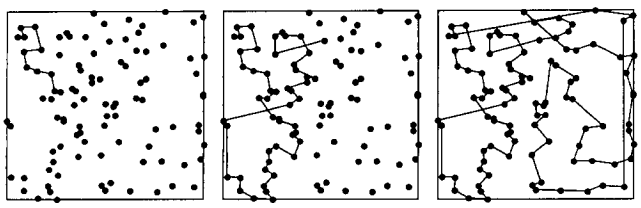

Figure 1. The Nearest Neighbor heuristic.

Exhibit 1.

```
Build(PtSet)
Perm[1] := StartPt
DeletePt(Perm[1])
for I := 2 to N do
  Perm[I] := NN(Perm[I - 1])
  DeletePt(Perm[I])
```

The obvious algorithm implements the point set as an array of points. The *NN* searches require $O(N)$ time, so the total time of the algorithm is $O(N^2)$. Bentley and Saxe[12] derive a data structure for dynamic nearest neighbor searching in planar point sets with the Euclidean metric that allows an NN tour to be computed in $O(N^{3/2} \log N)$ time. They also show that for point sets distributed uniformly over the unit square, a cell-based algorithm computes the NN tour in $O(N \log^2 N)$ expected time.

The K-dimensional binary search tree (or K-d tree) can efficiently implement the operations employed by the NN heuristic; the structure is briefly described in Appendix A1. The data structure can be built in $O(N \log N)$ worst-case time, and the sequence of deletions can be performed by a bottom-up algorithm in $O(N)$ total worst-case time (though some particular deletions might require $\Theta(\log N)$ steps). K-d trees also support a bottom-up nearest neighbor search procedure; Bentley[9] presents data and heuristic arguments that imply that the procedure runs in constant expected time for uniform point sets from which no points have been deleted. In constructing an NN tour, though, the points that have been visited and deleted might leave an unbalanced tree with expensive searches.

The cost of nearest neighbor searching in K-d trees is characterized by two quantities: the number of tree nodes visited and the number of distance calculations. Figure 2 summarizes an experiment to measure the number of nodes visited by the NN heuristic. Ten tours were built for each power of 2 between 32 and 32768; the top panel in the graph describes the average number of nodes visited in each nearest neighbor search. The K-d tree parameters were tuned to ease analysis of the data; the specific parameters are described in Appendix A1. Heuristic arguments in Bentley[9] indicate that the number of nodes visited in a search tree with no deletions should grow as $A + BN^C$, where $A > 0$, $B < 0$ and $-1 < C < 0$; a weighted least-squares regression of that model to the data gave the constants $A = 20.55$, $B = -31.69$ and $C = -0.347$. Figure 2 therefore plots the function along with the observations. The lower panel plots the weighted residuals, which are well behaved. A simple experiment in extrapolation examined five tours of a 262,144-point set (a factor of 8 beyond the maximum in Figure 2); the model predicts an average 20.13 nodes visited, while the observed results were 20.09, 20.10, 20.10, 20.11, 20.13. The second main cost of K-d tree searching is the number of distance calculations; a similar analysis gave the fit $4.22 - 6.0N^{-0.50}$. These analyses together suggest that the average cost of nearest neighbor

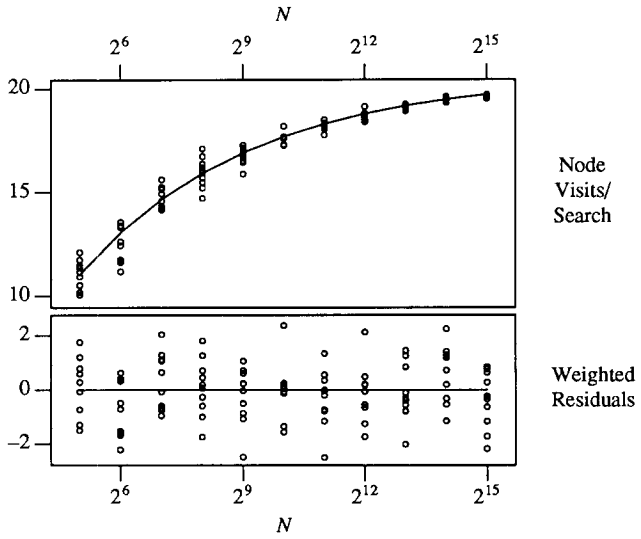

Figure 2. *K*-d tree performance on uniform data.

searching in computing NN tours with *K*-d trees remains $O(1)$. (Bentley[9] describes additional experiments that support this conjecture.)

If we accept the constant-expected-time hypothesis for nearest neighbor searching, the total cost of computing the NN tour is $O(N \log N)$ for building the tree, then $O(N)$ time for $N - 1$ searches, for a total of $O(N \log N)$. A common approach to the TSP, however, is to compute NN tours from M different starting points, and return the shortest tour found. To implement this approach, we build the tree once (at a cost of $O(N \log N)$), and then construct the M tours at $O(N)$ cost each. The total cost of the approach is therefore $O(N \log N + MN)$.

So far we have considered performance only on uniform data. We would like to know how the data structure performs on other data sets, but it seems difficult to prove worst-case theorems. For instance, Zolnowsky[49] presents a variant of *K*-d trees where the worst-case cost of searching for all nearest neighbors in a static set of N points in *K*-space is $O(N \log^K N)$, and gives a point set that realizes this time bound. It seems difficult to extend Zolnowsky's analysis to semidynamic sets. We will therefore study an experiment using the ten nonuniform input distributions described in Table I. (The abbreviation $U[0,1]$ signifies data uniform on the closed interval $[0,1]$ and *Normal*(*S*) signifies normally distributed data with mean 0 and standard deviation *S*.)

Many of the distributions have served as counterexamples to show that certain algorithms have poor worst-case run times. Other distributions model various applications: census tracts, for instance, are modeled by grid distributions in densely populated cities, by clustered normal distributions in sets of small cities with suburbs, and by several other of these distributions in other geopolitical contexts.

Figure 3 describes a set of experiments on point sets drawn from those distributions. The experiments were at

Table I. Ten Input Distributions

No.	Name	Description
0	uni	Uniform within the unit square ($U[0,1]^2$)
1	annulus	Uniform on a circle (width zero annulus)
2	arith	$x_0 = 0, 1, 4, 9, 16, \ldots$ (arithmetic differences); $x_1 = 0$
3	ball	Uniform inside a circle
4	clusnorm	Choose 10 points from $U[0,1]^2$, then put a *Normal*(0.05) at each
5	cubediam	$x_0 = x_1 = U[0,1]$
6	cubeedge	$x_0 = U[0,1]$; $x_1 = 0$
7	corners	$U[0,1]^2$ at (0, 0), (2, 0), (0, 2), (2, 2)
8	grid	Choose N points from a square grid that contains about $1.3N$ points
9	normal	Each dimension independent from *Normal*(1)
10	spokes	$N/2$ points at $(U[0,1], 1/2)$; $N/2$ at $(1/2, U[0,1])$

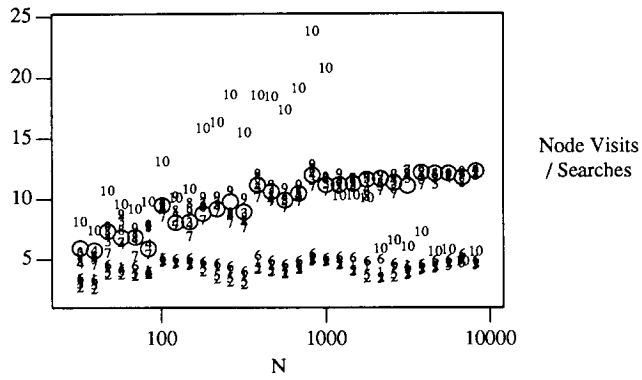

Figure 3. *K*-d tree performance on nonuniform data.

30 values of N, spread between 32 and 8192. At each value, a set of N points was generated from each of the eleven distributions. The *K*-d trees used in the experiments on uniform data have their parameters adjusted to ease the data analysis; in this experiment, and all later experiments that report run times, the parameters are adjusted to minimize CPU times. Details on these parameters are in Appendix A1. Circles denote the baseline uniform distribution; numbers designate distributions 1 through 10.

Figure 3 tells several tales. All of the functions show an oscillation at powers of two that is inherent in binary tree structures; there is also an oscillation at powers of eight because bounds arrays are stored at every third level of the tree. We see that the distributions fall into major classes: distributions 0, 3, 4, 7, 8 and 9 all display similar behavior, while distributions 1, 2, 5 and 6 make fewer node visits. The latter distributions are essentially one-dimensional, so *K*-d trees adaptively become more efficient one-

dimensional binary search trees; the other distributions all display local uniformity, which is discovered and exploited by the K-d trees. Distribution 10 is weird: it is more expensive than the other distributions up to $N = 1000$, at which point it becomes cheaper and approaches the linear distributions in cost. This is an artifact: a K-d tree parameter was set to invoke a sophisticated partitioning algorithm at $N = 1000$.

Figure 4 plots the CPU times (on a VAX 8550) for the same set of experiments shown in Figure 3. The plot symbols are placed at a value corresponding to the total CPU time divided by N, or the unit CPU cost. The time is measured in "ticks" of sixtieths of a second; this quantization produces artifacts in the graph. Runs that require no ticks have zero values, runs that require a single tick yield the hyperbola of points at the left of the graph, runs that require two ticks yield the next hyperbola of points, etc. After $N = 1000$, the points fall naturally into the two classes we observed above, and both appear to lie on straight lines (which correspond to a growth rate of $O(N \log N)$). A weighted least-squares regression shows that the CPU time on uniform data grows as $171N + 25N \lg N$; more details on the times can be found in Section 5.

We will now describe a more quantitative measure of robustness of run time (discussions of tour length are postponed to Section 5). For each of the eleven distributions, we define its *representative* run time to be the mean of the runtimes with the five largest values of N. (Because the observations at small N are unreliable due to their wide variance, the representative run time prunes those values. We are justified in taking means because the underlying function is near linear—hence means do not give undue weight to large values.) For the 10 nonuniform distributions, we define its *robustness ratio* to be the ratio of its representative time to the uniform representative. For the data in Figure 4, the maximum robustness ratio is 1.06, corresponding to distribution 4 (clustered normals). Thus even these ten nasty distributions are able to increase the run time of the NN implementation by only six percent. More details on this robustness measure can be found in Section 5. The experiments shown in Figures 3 and 4 thus show that the algorithm is fairly robust; they also show that counts of key operations are a good predictor of CPU times.

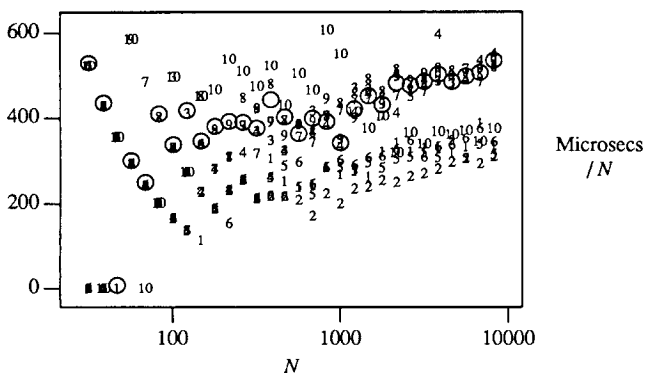

Figure 4. CPU times on nonuniform data.

The Double-Ended Nearest Neighbor Heuristic

An NN tour starts at a point and grows the fragment at only one end. The Double-Ended Nearest Neighbor (DENN) heuristic allows the fragment to grow at both ends, which we will call its two *tails*. A trivial implementation of the DENN heuristic performs two nearest neighbor searches to add each point, and adds the point nearer a tail. The run time of this implementation is roughly double that of the NN heuristic.

A more clever implementation of DENN reduces the run time to just slightly more than that of NN. This algorithm uses "links," a structure that will play an important role in later algorithms. The algorithm keeps a link from each tail to a point that once was its nearest neighbor outside the tour. At each stage of the algorithm, we examine the shorter link. If the link is to a point not already on the tour, it is said to be *valid* so we add that point to the tour and compute its nearest neighbor (which is now a valid link). Sometimes, though, an invalid link points to a vertex that was previously added to the tour. In that case we perform another nearest neighbor search, set that to be the current link, and continue the loop.

We will now consider the correctness of this algorithm. A valid link gives the nearest neighbor of its tail. Although an invalid link does not give a proper nearest neighbor, it provides a lower bound on the distance from that tail to a point currently in the tour. At each stage, the algorithm correctly adds a point closest to the tails.

A simple experiment on uniform point sets up to size 8192 counted the number of "excess" nearest neighbor searches beyond the $N - 1$ used by NN; all observed values were less than 10. Section 5 presents summary data on the performance of DENN, which is quite close to NN. The length of the DENN tours is also quite close to the length of NN tours, so the DENN algorithm does not appear to be of great interest; we will therefore devote no further space to it.

The Multiple Fragment Heuristic

The NN and DENN tours are simple greedy algorithms. In this section we will study the more interesting greedy TSP heuristic of matroid theory (see, for instance, Chapter 12 of Papadimitriou and Steiglitz[34]). This heuristic can be viewed as considering the edges of the graph in increasing order of length, and adding any edge that will not make it impossible to complete a tour (that is, edges that make early cycles or vertices of degree three). While the NN and DENN tours maintain a single TSP fragment, this heuristic employs a set of fragments; we will therefore refer to it as the *Multiple Fragment* or *MF* heuristic. Steiglitz and Weiner[40] describe a heuristic that is quite similar to MF. Bentley and Saxe[12] analyze the performance of MF using the name "Greedy," and show that its worst-case performance is similar to that of the NN heuristic: an MF tour through N points in the unit square has worst-case length $O(\sqrt{N})$.

At the start of the MF algorithm, each point is a fragment by itself. The closest fragments are then repeatedly connected. Figure 5 shows MF at work on the same 100-point set illustrated in the previous section, after 50, 90 and 100 edges have been added.

The implementation sketched above requires $O(N^2 \log N)$ time; we will now study a faster algorithm. The algorithm uses the same approach as Bentley and Friedman's[11] algorithm for computing Minimum Spanning Trees (MSTs), which can be viewed as a lazy evaluation of the edges examined by Dijkstra's[18] algorithm. The key insight is that most of the edges likely to be added to a tour are not affected by the addition of a new edge; we can therefore maintain a great deal of state from one addition to another. The MF algorithm uses several data structures:

- An array, *Degree*, in which *Degree*[*I*] contains the number of tour edges currently adjacent to vertex *I*.
- A *K*-d tree, *Tree*, for performing nearest neighbor searching. The tree contains only points that are eligible for edges, that is, with degree 0 or 1.
- An array, *NNLink*, of nearest neighbor links in which *NNLink*[*I*] contains the index of the nearest neighbor to *I* that (when originally computed) is not in the same fragment that contains *I* (though subsequent operations might invalidate that condition). These links are the edges that are eventually inserted into the tour.
- A priority queue, *PQ*, that contains the nearest neighbor links. The queue (implemented as a heap) supports the operations of inserting an arbitrary link and extracting the link of minimum length. Appendix A1 describes the priority queue operations in detail.
- An array, *Tail*, for representing the tails of the current fragments. If point *I* is a degree-1 tail of a fragment, then array element *Tail*[*I*] contains the index of the point that is the other tail of the fragment.

The complete MF algorithm is shown in Exhibit 2.

The first loop initializes the *Degree* array, the *NNLink* links, and the *PQ* priority queue. The second loop adds the $N - 1$ primary edges to the tour. Finally, the last edge connects the two remaining tails of the single *N*-city fragment.

It is the job of the inner loop to find an edge that can be added to the tour. The *GetTop* operation extracts from the priority queue the vertex, *X*, with minimum link distance (the float *ThisDist* and the integer *X* are set by the call). If *X* already has degree two, then its edge cannot be added to the tour, so it is removed from the priority queue and the loop continues. Otherwise, *Y* is set to the vertex linked to *X*, and we consider adding edge (*X*, *Y*) to the tour. If the degree of *Y* is less than two and it is not the tail of *X*'s fragment, then we can add the edge (so we break the loop); we will call this a *valid* link. Otherwise, we have to update the invalid link. To do this we perform a search for *X*'s nearest neighbor (other than its own tail), and update the priority queue. When the inner loop is completed, the edge is added to the tour and the *Degree* and *Tail* structures are updated.

Exhibit 2.

```
for I := 1 to N
  Degree[I] := 0
  NNLink[I] := Tree → NN(I)
  PQ → Insert(Dist(I, NNLink[I]), I)
  Tail[I] := 0
loop N − 1 times
  loop
    PQ → GetTop(ThisDist, X)
    if Degree[X] = 2 then
      PQ → RmTop( )
      continue
    Y := NNLink[X]
    if Degree[Y] < 2 and Y!= Tail[X] then
      break  // Y is a valid neighbor to X
    if Tail[X]!= 0 then  // Hide own tail
      Tree → DeletePt(Tail[X])
    NNLink[X] := Tree → NN(X)
    if Tail[X]!= 0 then  // Put tail back
      Tree → UnDeletePt(Tail[X])
    PQ → SetTop(Dist(X, NNLink[X]), X)
  AddEdge(X, Y)
  increment Degree[X]
  if Degree[X] = 2 then
    Tree → DeletePt(X)
  increment Degree[Y]
  if Degree[Y] = 2 then
    Tree → DeletePt(Y)
  update Tail[ ] array
add remaining Tail edge to tour
```

To prove the correctness of the program we observe that each (*X*, *Y*) edge added to the tour is a shortest edge that can be added while preserving the possibility of completing a tour. The key observation is that even when an *NNLink* from vertex *X* becomes invalid (because the vertex *Y* it points to has degree two or is in the same fragment as *X*), it still provides a lower bound on the length of the closest valid point to *X*.

There are two key questions in analyzing the cost of the MF algorithm: how many nearest neighbor searches are performed, and what is the cost of each? We will consider the former question first. Figure 6 plots the number of nearest neighbor searches per point, for 10 uniform point sets with *N* at powers of 2 from 32 to 8192. The number of

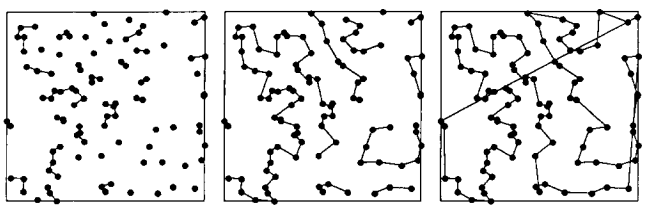

Figure 5. The Multiple Fragment heuristic.

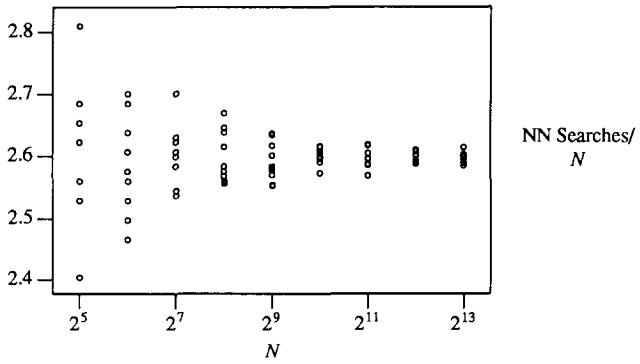

Figure 6. Average number of nearest neighbor searches for MF.

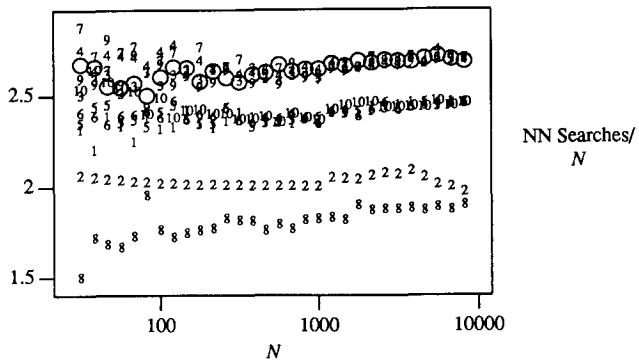

Figure 7. Nearest neighbor searches on nonuniform data.

searches per point seems to approach a constant between 2.4 and 2.8. A weighted least-squares regression of this data to the model $A + B \lg N$ gave the results $A = 2.60$ and $B = -0.00024$. The standard error associated with B was 0.00052, or twice its estimated value, so we may safely assume that the number of nearest neighbor searches is roughly $2.6N$. Since the initialization loop makes $N - 1$ nearest neighbor searches, the inner loop is executed only about 1.6 times per iteration of the outer loop, on the average.

The same set of experiments also measured the cost of K-d tree operations during the nearest neighbor searches. The two key function counts (per search) were fitted to the model $A + BN^C$. The number of nodes visited per search gave the results $A = 24.6$, $B = -35.0$, $C = -0.35$; the number of distance calculations per search gave $A = 5.27$, $B = -7.9$, $C = -0.53$. These functions are discussed in more detail in Appendix A1.

In summary, for uniform data the algorithm appears to perform $O(N)$ nearest neighbor searches, each at a cost of $O(1)$. It also requires $O(N \log N)$ time to build the tree and to perform $O(N)$ operations on a priority queue. The algorithm therefore appears to run in $O(N \log N)$ time for uniform inputs.

We also performed a robustness study of the performance of the MF algorithm on the ten nonuniform distributions. Figure 7 shows the average number of nearest neighbor searches per node. We see that no distribution uses substantially more searches than the uniform distribution, while some (notably numbers 2 and 8) use fewer. Data on K-d tree node visits and distance calculations displayed similar behavior. This MF algorithm appears to be robust for nonuniform inputs.

A Moment of Reflection

This section has introduced most of the techniques that we will use throughout the rest of the paper. The first and third techniques are algorithmic, while the second and fourth are analytic.

- We exploit the locality inherent in the problem to solve it through a series of proximity searches. In this section we used only nearest neighbor searching; in the next section we will also employ fixed-radius near neighbor searching and ball searching.
- The first key cost of the algorithms is that of locality searching. That cost is usually accurately modeled by a function of the form $A + BN^C$, where $A > 0$, $B < 0$, and $-1 < C < 0$.
- The MF algorithm uses a priority queue of links to nearest neighbors, which might be invalid. Several later algorithms use a similar structure.
- The second key cost of the algorithms is the number of locality searches performed. The MF algorithm appears to use roughly $2.6N$ nearest neighbor searches. For the MF algorithm, the number of priority-queue operations is linearly related to the number of nearest neighbor searches.

As we study later algorithms, we will analyze an expanded set of critical operations that are together sufficient to characterize the performance of the algorithm.

2. Heuristics That Grow Tours

In this section we will consider a class of six related heuristics. The heuristics are the cross product of three selection rules (Nearest, Farthest and Random) and two expansion rules (Addition and Insertion). The first three subsections describe, in turn, Nearest Addition (NA), Farthest Addition (FA), and Random Addition (RA). Each heuristic deals with a partial tour that begins as a single point. It then grows a tour that is at all times a cycle of points; adding a new point involves deleting an edge and then adding two new edges connecting the new point into the cycle. The tour is expanded by choosing a new point (by either the Nearest, Farthest, or Random selection rule), and then choosing where to add it to the partial tour by the Addition expansion rule. In the final subsection we will consider the same three selection rules with the Insertion expansion rule. Both expansion rules and all three selection rules are discussed by Rosenkrantz, Stearns and Lewis.[38]

The Nearest Addition Heuristic

Figure 8 shows an NA tour of our favorite 100-point set after 10, 30 and 100 edges have been inserted. The Nearest selection ruler chooses as the next point to be inserted the point that is closest to a tour point (which is exactly the order in which the points are inserted by Prim's[36] Minimum Spanning Tree algorithm).

To describe expansion rules we will need some terminology. Expanding the tour by the point Y at edge XZ involves deleting edge XZ and adding edges XY and YZ. If D is the distance function between points, the cost of that operation is denoted by $Cost(Y, XZ)$ and is equal to $D(X,Y) + D(Y,Z) - D(X,Z)$. An expansion rule must state how we insert the new point Y into a tour when its nearest neighbor in the tour is point X. Rosenkrantz, Stearns and Lewis[38] (their Section 6) use the simple Addition rule of expanding the tour by Y at edge XZ, where Z is the neighbor of X with cheaper expansion cost. They show that under that rule, the length of the NA tour is never more than twice the length of the optimal tour.

The Addition rule that we will use is slightly more sophisticated: if the distance between the new point Y and its nearest neighbor X is $D(X,Y) = R$, we consider all points within distance $AddRad \times R$ of Y, and place Y next to the point that gives the cheapest cost. The experiments in this paper all use $AddRad = 2$, but the parameter represents a tradeoff: larger values consume more search time to achieve better tours (any value less than one gives the simple Addition rule of Rosenkrantz, Stearns and Lewis[38]). The inspection of points when $AddRad > 1$ is accomplished by a fixed-radius near neighbor search in an additional K-d tree.

The NA algorithm is quite similar to Bentley and Friedman's[11] implementation of Prim's[36] single-fragment MST algorithm. (The NA algorithm adds nodes in exactly the order in which they are added by Prim's[36] MST algorithm. We can therefore efficiently implement NA (using the simple Addition rule of Rosenkrantz, Stearns and Lewis[38]) in certain dimensions and metrics. We first efficiently compute the MST (see Section 3), and then use a priority queue to add the edges of the tree in order of the Nearest selection rule.) It uses many of the same structures employed by the MF algorithm in the previous section: a K-d tree for searching, an array of links to nearest neighbors outside the current tour (now called $NNOut$), and a priority queue of links (organized with the minimum value on top). There is a link from each point currently in the fragment to a point that was (originally) its nearest neighbor outside the fragment. The pseudocode is displayed in Exhibit 3.

The initialization code deletes the starting point from the tree, computes its nearest neighbor outside the fragment, and inserts that link into the priority queue. The loop then adds $N - 1$ points to the tour.

The job of the inner loop is to find the point Y to be added to the tour, and X, its nearest neighbor already in the tour. It starts by examining the minimum link, which is from a point X known to be in the tour to a point Y. If Y is currently outside the tour, the postcondition is achieved and the loop is broken; otherwise, X's link is updated and reinserted into the priority queue. When a valid (X, Y) edge is found, the point Y is inserted into the tour, Y is deleted from the tree, and Y's link is computed and inserted into the priority queue. Note that links are never deleted from the priority queue; instead, they acquire large values, drift down the queue, and are never seen again.

We are left with the problem of inserting the point Y into the current tour, given its nearest neighbor X. To do this we use a doubly-linked list of the points in the tour and a K-d tree that represents the points already in the tour (the other K-d tree represents the points not in the tour). If the distance between Y and X is R, we perform in the K-d tree a fixed-radius near neighbor search centered at Y with radius $AddRad \times R$. As we visit each point Z, we consider adding Y on both sides of Z (using the doubly-linked list), and store the position we discover with minimum expansion cost. At the end of the search, we insert Y at the appropriate position in the doubly-linked list and add it to the K-d tree.

We turn now to the computational cost of the NA algorithm. We will start with the cost of nearest neighbor searching on uniform data. Figure 9 shows the number of nearest neighbor searches per point (which is the average number of times the inner loop is executed). Bentley and Friedman[11] observed a mean value typically between 2 and 3, and therefore assumed that the value was constant. Figure 9 shows that the function grows with N; the residu-

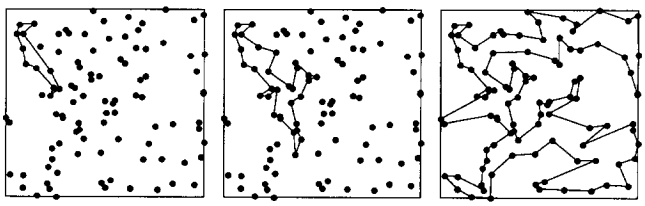

Figure 8. The Nearest Addition heuristic.

Exhibit 3.

```
Tree → DeletePt(StartPt)
NNOut[StartPt] := Tree → NN(StartPt)
PQ → Insert(Dist(StartPt, NNOut[StartPt]),
  StartPt)
loop N - 1 times
  loop
    PQ → GetTop(ThisDist, X)
    Y := NNOut[X]
    if Y is not in the tour then
      break
    NNOut[X] := Tree → NN(X)
    PQ → SetTop(Dist(X, NNOut[X]), X)
  Add point Y to tour; X is its nearest
    neighbor in the tour
  Tree → DeletePt(Y)
  NNOut[Y] := Tree → NN(Y)
  PQ → Insert(Dist(Y, NNOut[Y]), Y)
```

als in a weighted least-squares fit to a logarithmic model showed that the function grows more slowly. The function plotted in the figure is the fit to the model $A + BN^C$, which yields $A = 3.14$, $B = -2.94$ and $C = -0.45$. The average number of nearest neighbor searches, although growing, appears to be bounded above by a constant near three.

There are several other costs associated with the NA algorithm. The same set of experiments on uniform data showed that the number of K-d tree nodes visited during nearest neighbor searching grows as $32.4 - 45.6N^{-0.25}$ and the number of distance calculations grows as $4.96 - 13.4N^{-0.63}$ (Appendix A1 contains more details on these functions). We must also consider the cost of fixed-radius near neighbor searching in the tree that represents the points in the tour. For those searches (which are used to find an edge with minimum expansion cost), the average number of nodes visited grows as $30.8 - 39.5N^{-0.28}$ and the number of distance calculations is $7.96 - 5.95N^{-0.30}$.

Figure 10 shows that the NA algorithm is not robust: the CPU time used by distributions 4 and 9 grows much more quickly than that of other distributions. Distributions 4 and 9 both have normal data, which is sparse at the periphery and dense in the center. The NA algorithm typically climbs quickly to the dense center, and then updates a large number of invalid links as it moves to the sparse edges (Bentley and Friedman[11] observe exactly this slowdown in their implementation of Prim's single-fragment MST algorithm). The experiments in Figure 10 are truncated at $N = 1000$ because the rapid growth rates on those distributions make larger values too expensive to compute.

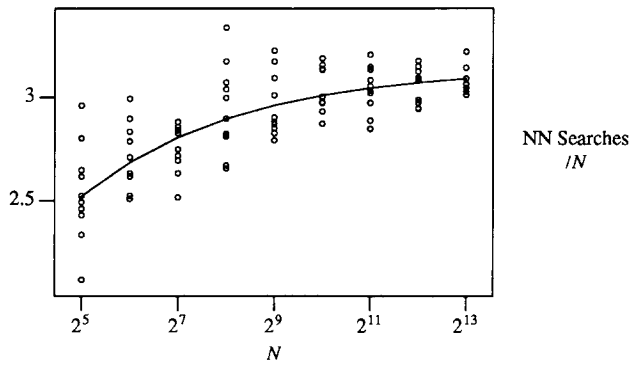

Figure 9. Average number of nearest neighbor searches of NA.

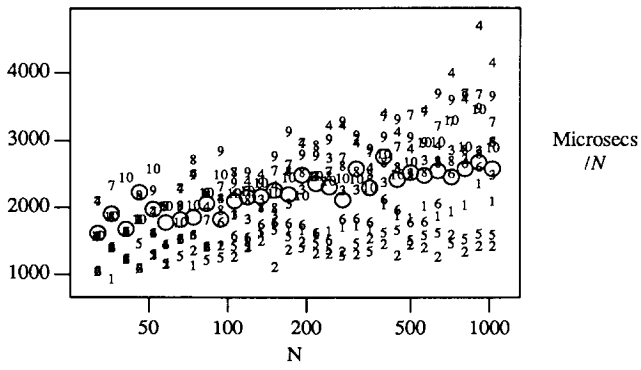

Figure 10. NA run time on nonuniform data.

The Farthest Addition Heuristic

We will now turn our attention to the Farthest Addition or FA heuristic. While NA adds the nearest point that is not already in the tour, FA adds the point that is farthest from any point currently in the tour. Figure 11 shows an FA tour after inserting 10, 30 and 100 edges. Adding a farthest point is at first counterintuitive, but the figure shows why it works: the first few farthest points sketch a broad outline of the tour, and successive points refine that gross outline.

The algorithm uses a strategy similar to the NA algorithm, but several key operations are inverted:

- The K-d tree represents points that are in the tour.
- The links go from a point Y not in the tour to Y's nearest neighbor in the tour fragment. The links are implemented by the *NNIn* array, for Nearest Neighbor In the fragment.
- The priority queue is inverted so that the maximum element is now on top.

The basic operation of the algorithm is similar to the NA algorithm, and the Addition mechanism for expanding the tour is exactly the same (see Exhibit 4).

The initialization removes all nodes from the tree with *DeleteAll*, then reinserts (*UnDeletes*) the starting point, and computes all links and inserts them in the priority queue. The main loop is similar to that of the NA algorithm. The difference is in the inner loop. When it examines a link from node Y to node X, it doesn't know whether X is still Y's nearest neighbor in the fragment. It therefore performs a new search for Y's nearest neighbor; if X is unchanged then the link is valid, the loop is broken, and the algorithm correctly adds the farthest point. Otherwise, the new link is reinserted into the priority queue.

The first question to ask about the performance of FA is the number of nearest neighbor searches (which is the same as the number of priority-queue operations). Figure 12 plots the average number of searches per node on a log-linear scale; the data points lie on a straight line. A weighted least-squares regression to a logarithmic model gave the excellent fit of $0.11 + 0.45 \lg N$. Thus the total number of searches appears to grow as $\Theta(N \log N)$.

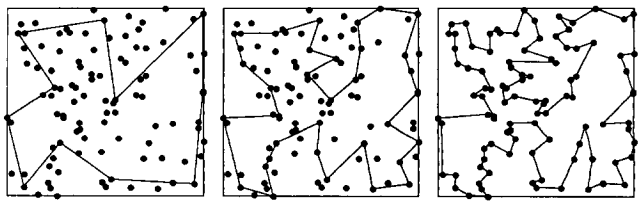

Figure 11. The Farthest Addition heuristic.

Exhibit 4.

```
Tree → DeleteAll( )
Tree → UnDeletePt(StartPt)
for I := 1 to N do
  if I!= StartPt then
    NNIn[I] := StartPt
    PQ → Insert(Dist(StartPt, I), I)
loop N − 1 times
  loop
    PQ → GetTop(ThisDist, Y)
    OldX := NNIn[Y]
    X := NNIn[Y] := Tree → NN(Y)
    if X = = OldX / / X is current
      break
      PQ → SetTop(ThisDist, Y)
  Add point Y to tour; X is its nearest
    neighbor in the tour
  Tree → UnDeletePt(Y)
  PQ → RmTop( )
```

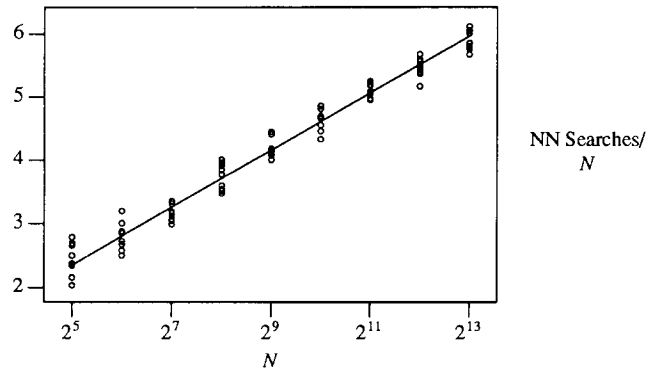

Figure 12. Average number of nearest neighbor searches of FA.

Not only does FA make more searches than the previous algorithms, each search is more expensive. (Because points are only added to the search structure, we could use one of the dynamic nearest neighbor search structures described by Mehlhorn[33] to implement the N searches and deletions in $O(N \log^2 N)$ worst-case time for planar Euclidean problems.) The function $A + BN^C$ (with the typical constraints of $A > 0$, $B < 0$, $-1 < C < 0$) grew too slowly to model the observed number of nodes visited. The fit to a logarithmic model was $4.1 + 1.68 \log N$, but the residuals indicated that the data didn't grow quite that quickly. The mean number of distance calculations per search grew up to a maximum of about 2.5 near 128, and then decreased to near 2.3. The combination of many node visits and few distance calculations shows that the K-d tree is being searched many times when it is in a very ragged state (for instance, $N - 1$ searches take place when there is a single node in the tree). Determining more precise growth rates for these functions will require more data or insight (or both).

The FA algorithm uses the same tree for nearest neighbor searching (for selecting the points) and for fixed-radius near neighbor searching (for expanding the tour). The cost of the latter searches appears to grow as $47.2 - 93.6N^{-0.33}$ for node visits and as $8.06 - 18.3N^{-0.50}$ for distance calculations. Thus the cost of the entire algorithm seems to be dominated by performing $O(N \log N)$ searches and priority-queue operations at logarithmic cost; the algorithm therefore appears to run in $O(N \log^2 N)$ time.

Figure 13 is part of the robustness study for the FA heuristic, which appears to be more robust than NA. The most expensive input is number 8 (the grid distribution, which has many ties), with a robustness ratio of 1.22; the next most expensive input has a ratio of just 1.04.

The Random Addition Heuristic

The Random Addition (RA) heuristic adds the points to a tour in a random order. Figure 14 shows an RA tour after adding 10, 30 and 100 edges. The FA heuristic approximates a good tour of the entire set by adding as the point that is farthest from the current tour; the RA heuristic approximates a good tour by a random sample of the points.

The RA heuristic is relatively easy to implement. To achieve a random order, we shuffle an array of indices of points, using a method such as Knuth's[28] (his Section 3.2.2) Algorithm M. A single K-d tree stores the points already in the tour. To insert a new point, we first perform a nearest neighbor search and then the fixed-radius near neighbor search.

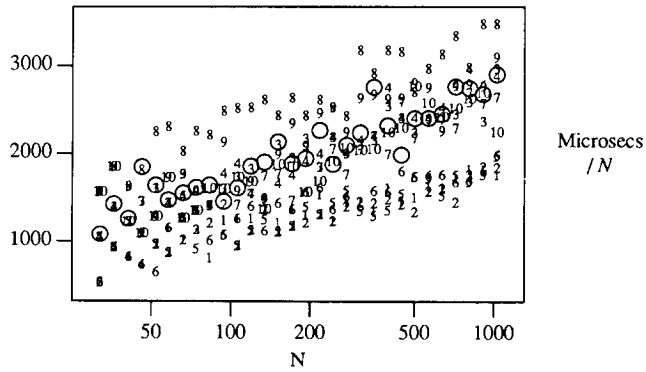

Figure 13. FA run time on nonuniform data.

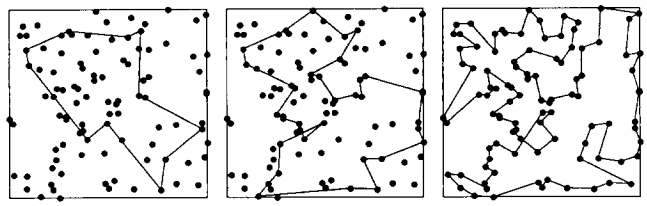

Figure 14. The Random Addition heuristic.

On uniform data, the cost of nearest neighbor searching appeared to require $22.1 - 32.4N^{-0.32}$ node visits and $3.94 - 3.64N^{-0.42}$ distance calculations. Fixed-radius near neighbor searching consumed $42.2 - 67.8N^{-0.29}$ node visits and $8.45 - 11.7N^{-0.40}$ distance calculations. The robustness study showed that the grid distribution required about 35% more CPU time than the uniform distribution; all other distributions were quite well behaved.

Insertion Heuristics

In this subsection we will consider the Insertion expansion rule in conjunction with the Nearest, Farthest and Random selection rules. Most of the algorithms remain unchanged; only the definition and implementation of point expansion differ.

The simple Addition rule of Rosenkrantz, Stearns and Lewis[38] inserts the new point, Y, to be adjacent to its nearest neighbor in the tour, X. We studied a slightly more sophisticated Addition rule that considers as potential neighbors for Y all points within a ball centered at Y. The Insertion expansion rule is even more thorough: it places Y next to the point X that gives the minimum expansion cost over all points currently in the tour. Rosenkrantz, Stearns and Lewis[38] show that any Insertion tour has a length no more than $\lceil \log N \rceil + 1$ times the length of the optimal tour. They also show that the Nearest Insertion tour is never more than twice the length of the optimal tour. Johnson and Papadimitriou[24] conjecture that the length of the Farthest Insertion tour is never more than 3/2 the length of the optimal tour.

We will now consider the implementation of the Insertion expansion rule. Our job is to find the minimal-cost edge at which to insert the new point Y. Previous algorithms in this paper have solved problems like this by a nearness search centered at point Y (either nearest-neighbor or fixed-radius near neighbor). The Insertion heuristic will also use such a search, which will solve part of the problem: the best edge might have a point that is relatively near Y. On the other hand, both vertices of the min-cost edge might be far away from Y, but in that case, the edge itself passes relatively near Y (in this case, relative to the length of the edge).

This situation is illustrated in Figure 15. The point Y is to be inserted into the current tour, which is noted by the lines. The nearest neighbor to Y is point C, and the edge CD is a fine candidate for replacement (it is the edge that would be used by the Addition expansion rule). In this case, though, edge AB gives minimum cost. Even though points A and B are far from Y (relative to the distance to Y's nearest neighbor, C), Y is near to A relative to the length of the edge AB.

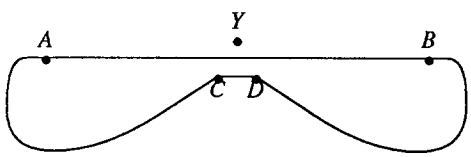

Figure 15. Inserting the new point Y.

To make these notions of relative distance more precise, we will require the following definitions.

A *sphere of influence* at tour vertex A with scale S is a ball centered at A with radius S times the length of the longer edge adjacent to A.

A *Nearest-Neighbor-ball* or *NN-ball* with scale S at point Y is a ball centered at Y with radius S times the distance from Y to its nearest neighbor among points in the tour.

The following theorem allows us to search for the min-cost edge at which to insert the new point Y.

Theorem 2.1. *Let Y be a point not in a tour, let C be Y's nearest neighbor in the tour, and let D be C's neighbor in the tour that minimizes $Cost(Y, CD)$. Then the tour edge with minimal expansion cost is one of the following:*

- *The nearest-neighbor edge CD.*
- *An edge AB such that A is in Y's NN-ball with scale* 1.5.
- *An edge AB such that Y is in A's sphere of influence with scale* 1.5.

Before we prove the theorem, we will explain how we will use it to implement the Insertion heuristics. Our algorithm is given the new point Y and its nearest neighbor in the tour, C; a simple calculation computes its neighbor D. A fixed-radius near neighbor search finds all points A that are in Y's NN-ball with scale 1.5, and the costs of inserting both edges adjacent to each point are examined. To find whether Y is within the sphere of influence of any point A, we use the K-d tree *SetRad* and *BallSearch* operations described in Appendix A1, which associate a radius with each point in the K-d tree. When we change an edge adjacent to tour vertex V, a *SetRad* operation sets the radius associated with V to be 1.5 times the length of the longer edge adjacent to V. We then use a *BallSearch*(Y) to report all balls that contain the point Y, which are those vertices whose spheres of influence contain Y. When we add the point at the min-cost edge, we update the doubly linked list of tour points, and change the radii associated with any of the three points whose spheres of influence changed.

Our proof of Theorem 2.1 will use the following lemma, which bounds the cost of inserting a single point into a tour.

Lemma 2.2. *The cost of inserting a new point into a tour is never more than twice the distance from the point to its neighbor in the tour. That is, $Cost(Y, CD) \leq 2D(Y, C)$.*

The Triangle Inequality states that

$$D(Y, D) \leq D(Y, C) + D(C, D)$$

which we can rewrite as

$$D(Y, D) - D(C, D) \leq D(Y, C).$$

The definition of *Cost* (with atypical bracketing) is

$$Cost(Y, CD) = D(Y, C) + [D(Y, D) - D(C, D)].$$

We now apply the previous inequality inside the brackets to yield

$$Cost(Y, CD) \leq D(Y, C) + [D(Y, C)] = 2D(Y, C). \quad \blacksquare$$

To prove Theorem 2.1 we will prove the stronger Theorem 2.3, which immediately implies Theorem 2.1. This theorem characterizes how min-cost edges are discovered: "long" edges are found by searching spheres of influence, while NN-balls reveal "short" edges.

Theorem 2.3. *Let Y be a point not in a tour, let C be Y's nearest neighbor in the tour, and let D be C's neighbor that minimizes $Cost(Y, CD)$. Suppose that there is a tour edge AB such that $Cost(Y, AB) < Cost(Y, CD)$. Then at least one of the following must be true, based on the relative lengths of the edges AB and YC:*

- *Case* 1 [*Long AB*]: *If $D(A, B) \geq D(Y, C)$, then Y is in A or B's sphere of influence with scale* 1.5.
- *Case* 2 [*Short AB*]: *If $D(A, B) \leq D(Y, C)$, then A or B is in Y's NN-ball with scale* 1.5.

We will use a proof by contradiction for both cases. For Case 1, assume that $D(A, B) \geq D(Y, C)$ and that Y is neither in A's nor B's sphere of influence with scale 1.5; we will show that $Cost(Y, AB) > Cost(Y, CD)$. Assume (without loss of generality) $D(A, B) = 1$. Figure 16 shows points A, B, C, D and Y and several edges among them; the circles centered at A and B with radius 1.5 represent their spheres of influence. By assumption, point Y is outside both spheres of influence, so $D(A, Y) > 1.5$ and $D(Y, B) > 1.5$. By definition,

$$Cost(Y, AB) = D(A, Y) + D(Y, B) - D(A, B)$$

which we combine with the above inequalities to yield

$$Cost(Y, AB) > 1.5 + 1.5 - 1 = 2.$$

Also by the assumption, Y's nearest neighbor C satisfies $D(Y, C) \leq D(A, B) = 1$. Lemma 2.2 therefore implies that $Cost(Y, CD) \leq 2D(Y, C) \leq 2$. Thus we have $Cost(Y, CD) \leq 2 < Cost(Y, AB)$, the desired inequality.

For Case 2, we will assume that $D(Y, C) \geq D(A, B)$ and that points A and B are not in Y's NN-ball with scale 1.5; we will show that $Cost(Y, AB) > Cost(Y, CD)$. Without loss of generality, we will assume that $D(Y, C) = 1$. Figure 17 shows Y's NN-ball with scale 1.5; both A and B lie outside that ball. By Lemma 2.2, we know $Cost(Y, CD) \leq 2D(Y, C) = 2$. Since $D(A, B) \leq D(Y, C) = 1$, $D(Y, A) > 1.5$, and $D(Y, B) > 1.5$, we know

$$Cost(Y, AB) = D(Y, A) + D(Y, B) - D(A, B)$$
$$> 1.5 + 1.5 - 1 = 2.$$

Thus we have $Cost(Y, CD) \leq 2 < Cost(Y, AB)$, the desired inequality. $\blacksquare$

The figures illustrate the proof of Theorem 2.3 for planar point sets under the Euclidean metric. The figures show that neither bound of 1.5 can be tightened. The proof, however, uses only simple properties of the underlying distance function D: the Triangle Inequality and the definition of balls in the space. The algorithm for the Insertion heuristic based on Theorem 2.1 employs Fixed-Radius Near Neighbor searching and Ball searching. Since K-d trees support these operations, the implementation of the Insertion heuristics performs correctly for the Minkowski L_1, L_2 and L_∞ metrics in any K-dimensional space.

We now have the tools necessary to implement the Insertion heuristics; we will start with Nearest Insertion. NI tours are very similar to NA tours: they tend to be slightly shorter (data in Section 5 indicates about a tenth of a percent), and indistinguishable to the eye (the NI tour of the point set in Figure 8 is exactly the NA tour shown there). For that reason, most of the costs of the NI algorithm are exactly the same as the costs of NA.

The substantial difference between NA and NI is the cost of ball searching, which has two components: setting the radii and searching to see which balls contain the query point. Experiments on uniform data sets showed that both of those costs are accurately modeled by functions of the form $A + BN^C$. A robustness analysis showed that the program performed quite well on eight of the ten nonuniform distributions. However, the heuristic displayed quadratic behavior on distributions 1 and 10 (annulus and spokes), each of which involves continually adjusting one edge whose sphere of influence contains all points.

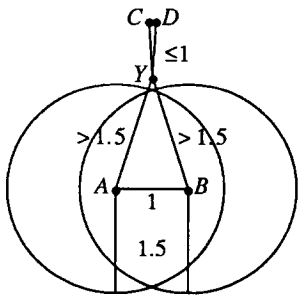

Figure 16. Case 1 [Long AB]: $D(A, B) > D(Y, C)$.

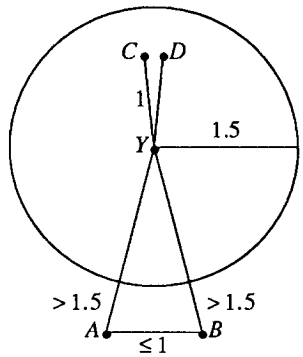

Figure 17. Case 2 [Short AB]: $D(A, B) \leq D(Y, C)$.

The FI and RI algorithms are quite similar to FA and RA. On uniform data, both costs of ball searching appear to grow faster than $A + BN^C$ where $A > 0$, $B < 0$ and $-1 < C < 0$, but slower than $O(\log N)$; the overall contribution to the algorithm is at most $O(N \log N)$. Both algorithms are robust for all ten input distributions.

3. Heuristics Based on Trees

In this section we will study three heuristics based on trees. The first two are derived from the minimum spanning tree of the point set, while the third is built directly from the underlying K-d tree.

The Minimum Spanning Tree Heuristic

The Minimum Spanning Tree (or MST) heuristic first constructs the MST of the point set and then traverses the tree to yield the tour. The points are appended to the tour in the depth-first, inorder traversal. Figure 18 illustrates the heuristic: the left panel shows the MST, and the right panel shows the resulting tour. The MST tour is easily seen to be within a factor of two of the length of the optimal tour (because the length of the MST provides a lower bound on the length of the optimal tour, and the MST tour is within a factor of two of the length of the MST).

The MST tour belongs to the antiquity of computing. Rosenkrantz, Stearns and Lewis[38] describe it as "widely known but unpublished," while Johnson and Papadimitriou[24] label the heuristic "often-rediscovered" and part of the "folklore." Computation of the MST dominates the time of the MST heuristic, so MST tours can always be computed in $O(KN^2)$ time, and more quickly in many special cases (see Preparata and Shamos,[35] Bentley, Weide and Yao,[14] Gabow, Bentley and Tarjan,[20] or Agarwal, Edelsbrunner, Schwarzkopf and Welzl[1]). Our implementation of the heuristic uses Bentley and Friedman's[11] single-fragment MST algorithm; the structure and performance of the algorithm is quite similar to the NA and NI implementation described in Section 2. Section 5 contains details on the performance and robustness of the algorithm. After the MST is computed, the algorithm performs a recursive Depth-First Search to list the points in the tour order.

(The order in which nodes adjacent to a given vertex are visited during the Depth-First Search might affect tour length. Section 5 states that the length of MST tours grows roughly as $1.03\sqrt{N}$, while experiments in Bentley[7] indicate that MST tours have length roughly $0.95\sqrt{N}$. I believe that the difference is due to the traversal orders. An artifact in the implementation of Bentley[7] caused the neighbors of a point to be visited in clockwise order, while the current implementation visits the point in an arbitrary order. The geometric order of the older implementation appears to yield shorter tours, but is hard to generalize to higher dimensional spaces.)

Christofides' Heuristic

Christofides'[16] Heuristic (CH) makes even more sophisticated use of the minimum spanning tree:

1. Construct an MST of the point set.
2. Construct a matching of all vertices that have odd degree.
3. Combine the matching edges with the MST edges, so that all vertices in the graph now have even degree.
4. Compute an Eulerian tour through the graph.

If Step 2 computes the optimal matching using an $O(N^3)$ algorithm, then it is easy to show that the length of this heuristic is never more than 3/2 times the length of the optimal tour.

Our implementation of CH is an approximation. It uses the same MST algorithm in Step 1 as the MST heuristic. To reduce the cost of matching, we use the approximate Greedy matching followed by 2-Opting described in Appendix A1. That algorithm appears to run in $O(N \log N)$ time, even on the odd-degree subsets induced by MSTs. Step 4 uses the linear-time Eulerian traversal algorithm in Section 17.2 of Papadimitriou and Steiglitz.[34] Figure 19 shows (left-to-right) the MST, the approximate matching of odd-degree points, and the resulting tour. Section 5 contains details on the performance of the algorithm.

The Fast Recursive Partitioning Heuristic

The final heuristic we will discuss is inspired by the approximation algorithm of Karp,[25] which is also described by Karp and Steele.[26] Karp's scheme recursively subdivides the plane until the final subdivisions (or buckets) have a small number of points, computes optimal tours within the buckets, and then patches the tours together. Karp's algorithm uses a fair amount of CPU time to produce very good tours; we will now study a poor cousin that uses very little CPU time to produce longer tours. The

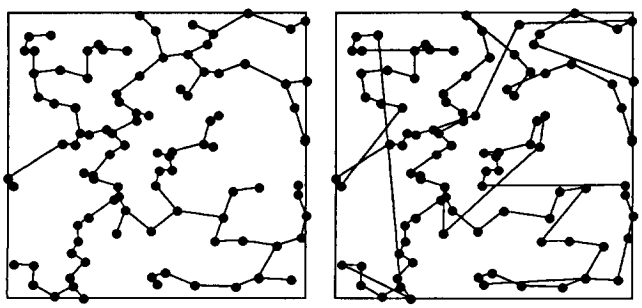

Figure 18. The Minimum Spanning Tree heuristic.

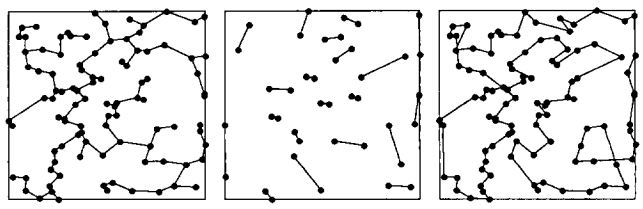

Figure 19. Christofides' heuristic.

Fast Recursive Partitioning (or FRP) heuristic behaves similarly: it uses K-d tree routines to divide the space into buckets, and then computes a Nearest Neighbor tour within each bucket. The traversal of the tree always visits first the son that is closer to the last point in the last bucket visited, and moves from the last point visited to the closest point in the next bucket. Figure 20 shows the progress of the FRP tour.

The size of the buckets is controlled by a parameter B. The run time of the FRP algorithm is provably $O(N \log N)$ for building the K-d tree, and then $O(N/B \times B^2) = O(NB)$ for quadratically computing in the buckets roughly N/B NN tours of size roughly B. Thus the parameter B provides a tradeoff: larger values give shorter tours but take more CPU time to compute. The value of B was set to 15 for all experiments in this paper. The K-d tree code was tuned using several of the techniques described by Bentley.[9]

4. Local Optimization Heuristics

In this section we will study heuristics that make local improvements to existing tours. The first subsection starts with the straightforward 2-Opt heuristic, and the next subsection considers more complex heuristics.

Two-Opt

Figure 21 illustrates a single 2-Opt swap in a tour: we remove the two edges AB and CD and replace them with the two edges AC and BD. That operation maintains the tour and reduces its length. Notice that we must be careful not to use the replacement edges AD and BC, which would break the tour into two disjoint cycles. If we think of the tour as a circular sequence of cities, a 2-Opt swap corresponds to reversing a subsequence of the cities. Croes[17] therefore refers to a 2-Opt swap as an "inversion"; Flood[19] describes 2-Opt swaps as removing intersections (he generalizes the notion beyond geometric graphs).

A tour that cannot be improved by any 2-Opt swap is said to be 2-Optimal. The 2-Opt algorithm, in most general terms, starts with an arbitrary tour, applies 2-Opt swaps as long as possible, and ends when no further 2-Opt swaps can be made, at which time the tour is 2-Optimal. Figure 22 shows the 2-Opting of an MF tour of 32 points; it makes a total of 6 2-Opt swaps, then becomes 2-Optimal.

An obvious implementation of 2-Opt repeats the basic step of considering all $N(N-1)/2$ pairs of edges in the tour, and performing a 2-Opt swap if one is found. Each 2-Opt swap therefore takes $O(N^2)$ time, and the algorithm might require many swaps to reach the final 2-Optimal tour (see Johnson and Papadimitriou[24]).

The basic 2-Opt algorithm can be implemented in a dazzling variety of ways. For instance, in which order do we inspect potential edges to be considered in swaps? Do we consider the edges in some static order (say, corresponding to the point numbers), or traverse around the tour? And if we traverse the tour, how do we change the traversal order after inverting a fragment of the tour? Given that we have chosen the first edge to consider in a swap, how do we choose the second edge to consider? (We will shortly see that we can exploit the geometric structure in this step.) Suppose that we have considered many "second" edges and have discovered several possible 2-Opt swaps; which one do we take? We might choose the swap with maximum gain (a greedy approach), or with minimum gain (a more cautious approach to avoid local minima), or the first swap we see (for run time efficiency), or choose one arbitrarily (randomization). The possibilities go on and on.

To find an efficient implementation of geometric 2-Opting, we must identify a central searching problem. The key to our algorithm is the following simple observation:

At least one vertex in a 2-Opt swap has an edge that decreases in length.

If both new edges increase in length, the 2-Opt swap could not reduce the length of the tour. Thus the search for the "second edge" in a 2-Opt swap can be confined to a ball centered at a vertex of the first edge with radius equal to the edge length. This operation can be implemented by a fixed-radius near neighbor search. (Stieglitz and Weiner[40] use a similar observation to speed up 2-Opt in general graphs.)

We will now sketch an efficient geometric version of 2-Opt, making reference to Figure 21. We proceed around the tour, visiting each point A in order; at each A, we

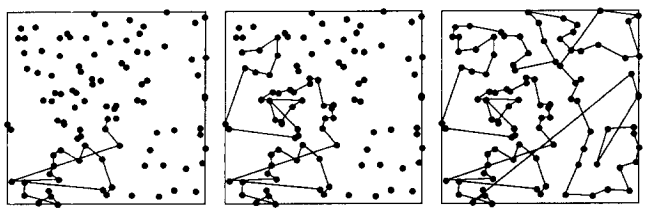

Figure 20. The Fast Recursive Partitioning heuristic.

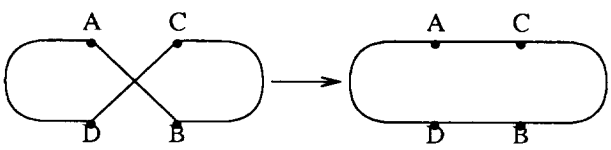

Figure 21. A 2-Opt swap.

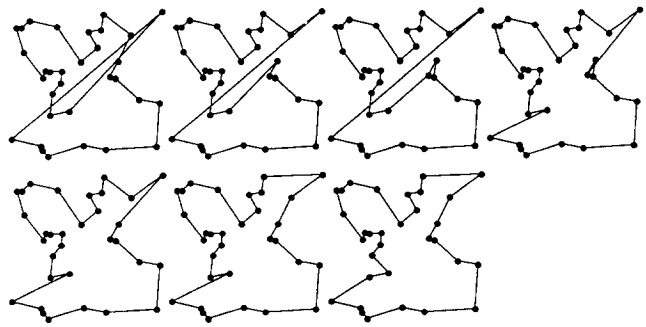

Figure 22. 2-Opting a 32-city tour.

consider both of its neighbors as point B, which defines the edge AB. We perform a fixed-radius near neighbor search centered at A with radius $D(A, B)$. For each point C we find during the search, we consider its single appropriate neighbor D (the other neighbor would break the tour into two disjoint cycles). If the 2-Opt swap shown in Figure 21 would reduce the tour length (deduced by comparing the sums of distances), we make the 2-Opt swap and continue traversing the tour. If we ever make one complete rotation around the tour without making a 2-Opt swap, then we know that the tour is 2-Optimal and halt. The algorithm's correctness depends on the observation made above: if a pair of edges can be improved by a 2-Opt swap, we find them during a fixed-radius near neighbor search.

The program provides both true and approximate 2-Opting. The approximate version associates with every node a bit that is originally 1. As the algorithm traverses the tour, it considers the edges adjacent to a vertex for 2-Opting only if the bit is 1. If it finds no 2-Opt swap adjacent to that vertex, it turns the bit to 0. If it does make a 2-Opt swap, it sets to 1 the bits associated with all four vertices in the swap. The approximate 2-Opt algorithm makes a tradeoff: it is much faster than the true version, and the tours it produces are only very slightly longer (details are in Bentley[8, 10]).

Using the notation in Figure 21, when the algorithm is at node A and considering the edge AB, it searches for a point C that is nearer to A than B. If we know, however, that B is A's nearest neighbor, then no point C could shorten the edge AB. Thus if B is A's nearest neighbor, then we need not perform the fixed-radius near neighbor search centered at A.

We have mentioned many variants of 2-Opting. We will now specify some of the key design choices made in the 2-Opt algorithm whose performance will be described shortly.

- The tour is implemented as an array of points. The main loop of 2-Opt iterates through that array in increasing order; thus points are visited in tour order. A 2-Opt swap reverses a subsequence of the array. After a swap, the main loop backs up one step in the array then continues searching in the same direction. The tour is declared 2-Optimal when the search proceeds through the entire array without making a 2-Opt swap.
- We use the approximate 2-Opt that associates a bit with each node in the tour.
- The algorithm performs a fixed-radius near neighbor search at point A with radius $D(A, B)$ only if B is not A's nearest neighbor.
- When the search discovers a swap that reduces tour length, it immediately makes the swap and halts the fixed-radius near neighbor search.

The performance of 2-Opt varies as a function of the starting tour to which it is applied. We will now briefly describe the performance of 2-Opt applied to MF tours. Since the graphs for these experiments closely resemble previous graphs, we will only sketch the results. Bentley[8] contains a more detailed analysis of the performance of 2-Opt applied to NN starting tours, and Bentley[10] briefly analyzes 2-Opt applied to CH tours.

- The total number of nodes visited during 2-Opting is approximately $0.3N \lg N + N$. Thus for $N = 10^6$, the algorithm makes only roughly 7 revolutions around the tour.
- The approximation scheme of keeping a bit with each node is very effective; the bits are 1 at only about $1.5N$ nodes of those visited. At the nodes whose bits are 1, the average number of fixed-radius near neighbor searches performed is 1.15. Thus the total number of searches is approximately $1.7N$.
- Both primary costs of fixed-radius near neighbor searching in the K-d tree are bounded above by constants. The number of nodes visited is accurately modeled by $36.1 - 56N^{-0.39}$, and the number of distance calculations grows as $12.3 - 18N^{-0.52}$.
- Apart from the K-d tree searching, the total number of other distance calculations in 2-Opting appears to be roughly $22N$.
- For N in the range $32 \ldots 100000$, the total CPU time used by 2-Opting is just slightly superlinear: roughly $1430N + 31N \lg N$.
- 2-Opting makes roughly $0.2N$ 2-Opt swaps. A random reversal touches $N/2$ array elements, on the average; the reversal routine therefore always reverses the shorter of the two subsequences defined by the 2-Opt. The total number of array elements touched during reversing was observed to grow as roughly $0.037N^{1.7}$. At $N = 10^4$, this is an average of only 23 array accesses per point, and represents a tiny fraction of the CPU time. By the time $N = 10^6$, though, this is 590 array accesses per point, or about 40% of the CPU time. Bentley[8] describes how Johnson, McGeoch and Ostheimer have implemented the tour with a balanced tree to reduce each operation on the tour to logarithmic cost (the tree code, though, is over 1000 lines of C).

This 2-Opting algorithm is quite robust. None of the above key parameters vary widely for nonuniform distributions, and the CPU time is never more than 17% more than the time for uniform data. Section 5 and Appendix A2 contain data on the run times and tour lengths for 2-Opting from all twelve starting tours.

The program described in Appendix A3 contains many other options for 2-Opting that we have not considered. Here are a few of the more interesting variants.

- An alternative implementation of approximate 2-Opting does not associate a bit with each node, but rather puts all nodes originally into a queue. It then repeatedly extracts a node from the queue, checks it for a potential 2-Opt swap, and if a swap is found, puts all four associated vertices back into the queue (unless they are already present). This process continues until the queue is empty.

An alternative uses a stack instead of a queue. These approaches use slightly less time than the bit method to achieve slightly longer tours. Appendix A1 describes how these techniques were applied more successfully in 2-Opting a greedy matching.

- A *MinGain* parameter states that a swap should be made only if it reduces the tour length by the stated amount. This parameter avoids the infinite loops observed when inaccuracy of floating-point operations caused both a swap and its inverse to reduce tour length.
- A *MaxNodes* parameter halts the fixed-radius near neighbor search after visiting the specified number of nodes; *MaxSwaps* halts the search after finding the specified number of potential swaps.

Other Local Optimizations

In general, a λ-Opt step improves a tour by deleting λ existing edges and inserting λ new edges. The obvious λ-Opt algorithm requires $O(N^\lambda)$ time to check for a single λ-Opt swap. In this subsection we will examine fast implementations for several λ-Opt algorithms.

We'll begin with the limited form of 3-Opt shown in Figure 23. While a general 3-Opt swap deals with 6 points this special case deals with only 5 points; we will therefore refer to this as a Two-and-a-Half-Opt swap, or 2H-Opt swap (using Tukey's[45] "half" notation). Bellmore and Nemhauser[6] describe this heuristic by viewing the tour as a sequence of points: while a 2-Opt swap reverses a subsequence of the tour, a 2H-Opt swap moves a single point from one place in the sequence to another. A tour is said to be 2H-Optimal if it cannot be improved by any 2-Opt or 2H-Opt swaps.

We could implement the 2H-Opt heuristic using the techniques developed in Section 2 for the Insertion expansion rule: that section provides us with a complex mechanism for inserting a point into its best possible place in an existing tour. We will now study a simpler and faster implementation of 2H-Opt that "piggybacks" the 2H-Opt algorithm onto 2-Opt. The 2-Opt algorithm performs a fixed-radius near neighbor search centered at A with radius $D(A, B)$. If it finds the point C in that ball, it checks the appropriate neighbor D of C to see if we can perform the 2-Opt swap shown in Figure 21. The 2H-Opt algorithm adds two additional checks: the first is for the 2H-Opt swap shown in Figure 23, and the second is for a dual check in which point A is moved to a different place in the sequence. (This is similar to the two cases handled by the Insertion expansion rule in Section 2.)

When applied to MF tours of uniform point sets, the 2H-Opt algorithm yields substantially shorter tours than 2-Opt, but at only a slight increase in CPU time (from $1430N + 31N \lg N$ microseconds to $1430N + 54N \lg N$ microseconds). Most of the performance measures of 2-Opt remain substantially unchanged; the only exception is the number of distance calculations (the two additional checks raise that from $22N$ to $50N$). The 2H-Opt algorithm is only slightly less robust than 2-Opt: distribution 7 (corners) requires about 40% more time than uniform data.

Lin[30] describes the more powerful 3-Opt heuristic. A 3-Opt swap is shown in Figure 24: it decreases the tour length by deleting the edges AB, CD and EF and replacing them by AC, BF, and DE. This can be viewed as removing the subpath $C..F$ from the tour and moving it (possibly with reversal) to between vertices A and B. Figure 25 shows a 3-Optimal tour through a set of 1000 points distributed uniformly over the unit square; the original MF tour had length 26.21, and 3-Opting reduced the length by about 10 percent to 23.58.

We implement 3-Opt by two locality searches, each of which looks for a "short enough" edge. The first search is like the 2-Opt search: we perform a fixed-radius near neighbor search centered at A, with radius $D(A, B)$. That search locates any point C that reduces the edge length AB, and we identify C's single appropriate neighbor D. We now consider deleting the edges AB and CD, and adding the edge AC; we thus do a second fixed-radius near

Figure 24. A 3-Opt swap.

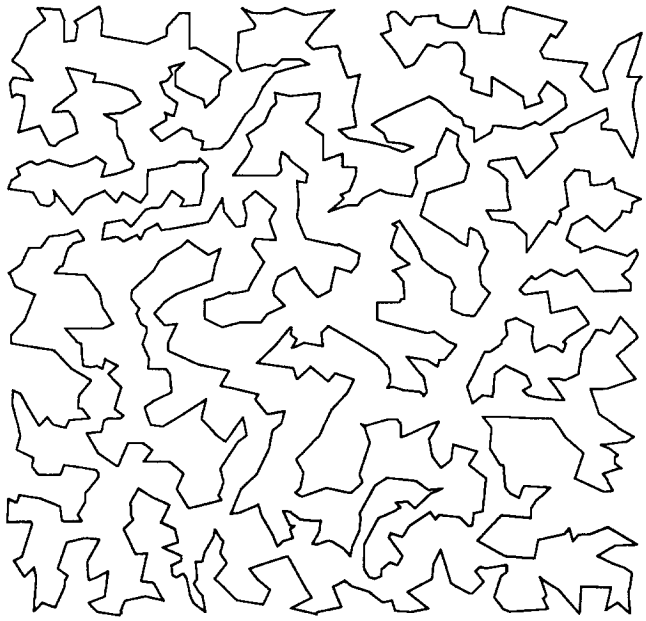

Figure 25. A 1000-city MF + 3-Opt tour.

Figure 23. A 2H-Opt swap.

neighbor search centered at C, with radius $D(A, B) + D(C, D) - D(A, C)$. That search reports any point E that is a potential 3-Opt swap, so we identify E's appropriate neighbor F, and compute the cost of deleting the edges AB, CD and EF and adding AC, BF, and DE.

When applied to MF tours of uniform point sets, the 3-Opt algorithm yields substantially shorter tours than 2H-Opt, at a significant increase in CPU time (from $1430N + 54N \lg N$ microseconds to $4008N + 65N \lg N$ microseconds). Most of the performance measures of 3-Opt are of a similar character to those of 2H-Opt; the number of distance calculations, for instance, grows from $50N$ to $79N$. Unfortunately, 3-Opt is not robust; it displays almost quadratic slowdowns on the linear distributions 2, 5, 6 and 10.

Figure 26 illustrates why we cannot generalize the efficient implementations of 2-Opting and 3-Opting to 4-Opting. 4-Opting is no longer a local phenomenon: two of the edges in a 4-Opt swap can be arbitrarily far away from the two other edges. Johnson,[23] however, uses K-d trees in an alternative method in an efficient implementation of Lin and Kernighan's[31] "variable-depth" Opting: the algorithm precomputes a subgraph of the point set that contains the (say) 20 nearest neighbors of each point, and then applies sophisticated algorithms to the small (nongeometric) subgraph.

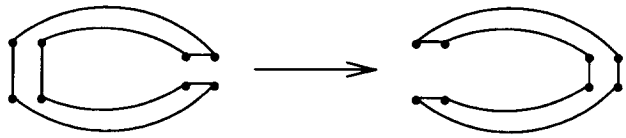

Figure 26. A 4-Opt swap.

5. Comparison of the Heuristics

So far we have studied the TSP heuristics individually. In this section we will compare their run times in some detail, and briefly examine the length of the tours.

We begin by summarizing the robustness studies of the heuristics. Table II gives the representative robustness ratio defined in Section 1 for each of the 10 nonuniform distributions, and the maximum of those 10 ratios. (As a reminder, the names of the distributions are, in order, annulus, arith, ball, clusnorm, cubeedge, cubediam, corners, grid, normal, and spokes; distribution 0 is the baseline uniform distribution.) The default maximum value of N for the experiments was 8192, with these exceptions: NA, FA and RA used a maximum N of 1024, NI used 2000, MST and CH used 3000, FRP used 32768, and 3-Opt used 1000. These experiments use the K-d tree parameters tuned for speed described in Appendix A1. A plus sign after the maximum ratio indicates that the ratio was increasing as a function of N.

Six of the 12 heuristics are quite robust for all 10 distributions: NN, DENN, MF, FI, RI, and FRP all have run times within 15% of their uniform performance, and FA and RA are just slightly behind. The 2-Opt and 2H-Opt heuristics are fairly robust, while 3-Opt quickly degrades to quadratic performance.

So far we have ignored the length of the tours produced by the heuristics. Beardwood, Halton, and Hammersley[5] first proved that the length of the optimal tour through N points uniform over the unit square grows as $\alpha\sqrt{N} + o(N)$. Since then, researchers have shown that many other geometric structures on uniform sets display similar behavior; Steele[39] is the source of the techniques for most of the proofs. In our experiments, all of the heuristics produced tours with lengths accurately described by a function of the form $\alpha\sqrt{N} + \beta$; the observed values are reported in Table III. The table also describes the CPU time, both as a function of N (the cost per point in microseconds), and the

Table II. Robustness of the Heuristics

Heuristic Name	Distribution No. 1	2	3	4	5	6	7	8	9	10	Maximum Ratio
nn	0.68	0.60	1.01	1.06	0.64	0.70	1.00	1.01	1.03	0.73	1.06
denn	0.68	0.60	1.05	1.01	0.64	0.68	0.98	1.07	1.02	0.76	1.07
mf	0.64	0.48	1.01	1.05	0.62	0.65	1.01	0.71	1.03	0.64	1.05
na	0.82	0.56	1.07	1.56	0.61	1.03	1.36	1.11	1.43	1.23	1.56 +
fa	0.65	0.64	0.96	1.04	0.65	0.67	0.94	1.22	1.03	0.93	1.22
ra	0.56	0.61	1.01	1.01	0.57	0.60	0.98	1.35	1.06	0.86	1.35
ni	2.94	0.55	1.00	1.42	0.63	0.82	1.27	1.07	1.25	1.83	2.94 +
fi	0.51	0.60	1.02	1.11	0.64	0.73	1.09	1.10	1.07	0.68	1.11
ri	0.45	0.60	0.99	1.14	0.61	0.70	1.13	1.07	1.11	0.69	1.14
mst	1.02	0.56	1.01	1.90	0.65	1.41	1.44	0.91	1.71	1.27	1.90 +
ch	0.79	0.41	1.04	1.96	0.51	0.99	1.41	1.01	1.57	1.04	1.96 +
frp	1.11	0.85	1.00	1.01	0.97	0.97	1.00	0.98	0.99	1.02	1.11
mf-2	0.30	0.43	0.98	1.12	0.49	0.53	1.17	1.06	0.93	0.57	1.17
mf-2h	0.31	0.49	0.96	1.27	0.54	0.55	1.40	1.21	0.99	0.74	1.40
mf-3	0.22	37.43	1.03	3.40	36.45	36.89	11.93	0.97	1.60	28.34	37.43 +

Table III. Functional Behavior of the Heuristics

Heuristic Name	Uniform Length	CPU Time		Robustness Ratio
		Unit cost	$N = 10{,}000$	
nn	$0.879\sqrt{N} + 0.91$	$171 + 25 \lg N$	5.0	1.06
denn	$0.880\sqrt{N} + 0.83$	$133 + 31 \lg N$	5.4	1.07
mf	$0.819\sqrt{N} + 0.99$	$326 + 84 \lg N$	14.4	1.05
fa	$0.805\sqrt{N} + 0.30$	$-412 + 315 \lg N$	37.7	1.22
na	$0.902\sqrt{N} + 0.39$	$1117 + 147 \lg N$	30.7	1.56 +
ra	$0.817\sqrt{N} + 0.38$	$920 + 82 \lg N$	20.1	1.35
fi	$0.805\sqrt{N} + 0.28$	$-1935 + 772 \lg N$	83.2	1.11
ni	$0.901\sqrt{N} + 0.41$	$548 + 335 \lg N$	50.0	2.94 +
ri	$0.815\sqrt{N} + 0.37$	$-767 + 546 \lg N$	64.9	1.14
mst	$1.03\sqrt{N} + 0.26$	$220 + 127 \lg N$	19.1	1.90 +
ch	$0.812\sqrt{N} + 0.65$	$653 + 149 \lg N$	26.3	1.96 +
frp	$1.11\sqrt{N} - 0.18$	$78 + 6 \lg N$	1.6	1.11

value given by that function (in total seconds) at $N = 10{,}000$. In these experiments, the K-d trees used the bucket cutoff parameter of 5. (Even though the FA, FI, RA and RI heuristics apparently require time asymptotically proportional to $N \log^2 N$, their run time is accurately described by the $O(N \log N)$ functions in the table for $1{,}000 \leqslant N \leqslant 100{,}000$.)

Table IV summarizes an experiment in which each of the twelve starting heuristics was run with (approximate versions of) each of the three local optimizations. Five runs were made for each start/opt pair; all runs used $N = 10{,}000$ uniform points. We measured both the time in seconds and the tour length for each run; the various runs were all closely clustered, so we report only the means. Johnson[23] observes that the Held-Karp lower bound for the TSP on uniform sets of this size tends to be quite close to 71.5. The table reports the run time in seconds and the tour length as the percent over Johnson's lower bound (the percentage X may be converted to an absolute length by $71.5 \times (1 + X/100)$). The run times of the local optimizations include the run time of the starting tours.

Figure 27 summarizes the above table. The plot symbol "nn" describes the mean run time and tour length of the NN heuristic, while "nn-2h" describes NN followed by 2H-Opt. (Because the two heuristics are so similar, the "nn" and "denn" symbols overprint one another.) Many of the combinations are dominated by a pair that produces a shorter tour in less time. The main contenders in this set are, in increasing run time and decreasing tour length: FRP, NN, MF, MF-2, MF-2H and MF-3. All of these pairs are quite robust, except the last.

Observe that 2-Opt following an FI start is much different than 2-Opt following an MF start: the former takes just 13 seconds to reduce excess length from 13.0% to 11.9%, while the latter takes 16 seconds to reduce excess length from 15.7% to 5.8%. Careful study of the graph and Table IV shows that all of the local optimizations are very sensitive to their starting tours; Appendix A2 presents similar

Table IV. Local Optimization Applied to Each Heuristic

Heuristic Name	Percent Over Lower Bound				CPU Seconds			
	Start	2-Opt	2H-Opt	3-Opt	Start	2-Opt	2H-Opt	3-Opt
nn	24.2	8.7	6.8	4.5	4	27	28	54
denn	24.2	8.6	6.7	4.6	4	28	28	57
mf	15.7	5.8	4.7	3.5	14	30	33	55
na	26.9	16.9	11.2	6.9	26	45	47	83
fa	13.2	11.8	9.6	6.9	38	52	52	78
ra	15.2	12.0	9.8	6.8	16	31	31	56
ni	26.8	16.9	11.3	6.8	46	65	67	101
fi	13.0	11.9	9.7	6.9	76	89	89	112
ri	14.8	12.3	9.9	7.0	57	72	72	97
mst	44.5	12.8	9.3	5.6	16	44	45	80
ch	14.9	6.7	4.9	3.8	24	40	40	60
frp	55.2	14.9	10.5	5.8	2	35	34	73

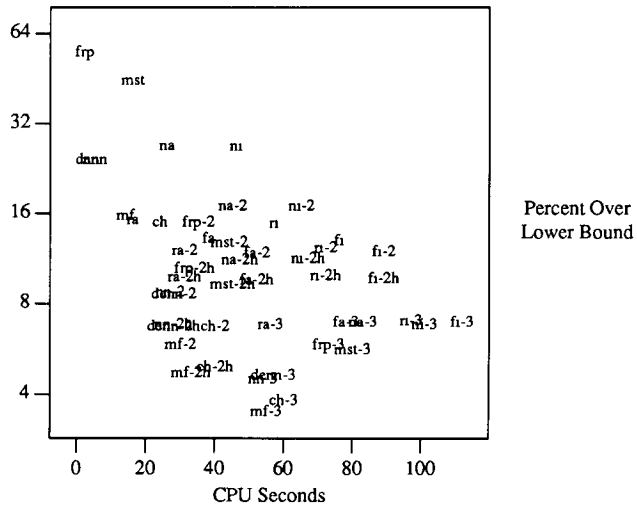

Figure 27. Starting groups and local optimizations, $N = 10{,}000$.

tables for other metrics and dimensions. In all cases, though, 2H-Opt produces much shorter tours than 2-Opt, and is only a little slower.

As a final experiment, we computed million-city tours for the contending algorithms (the results are summarized in Table V). David Johnson supplied the lower bound of 709, extrapolated from lower bounds computed for point sets up to size 10^5; the tour lengths are given in both absolute units and in percent over the lower bound. By way of comparison of CPU speed, the system-supplied quicksort for a million floating point numbers requires 165 seconds, a tuned quicksort requires 30 seconds, and building a K-d tree for 10^6 planar points requires 255 seconds.

6. Conclusions

In this paper we have described several new algorithms: implementations of a dozen starting heuristics, three heuristics for local optimizations, and the matching algorithms in Appendix A1. The algorithms have all been implemented; Appendix A3 describes the implementation. Several of the TSP algorithms are fast, robust, and produce short tours: the main contenders for practical applications are FRP, NN, MF, and 2-Opt and 2H-Opt applied to MF tours. Appendix A2 shows that these algorithms remain efficient for other metrics and higher-dimensional spaces.

Table V. Million-City Tours

Heuristic Name	CPU Time		Tour Length	
	Seconds	Common	Absolute	Percent
frp	194	3.2 min	1110.4	56.6
nn	578	9.6 min	874.3	23.3
mf	1689	28.1 min	809.9	14.2
mf-2	6582	1.8 hr	748.4	5.6
mf-2h	8385	2.3 hr	741.0	4.5
mf-3	13709	3.8 hr	733.5	3.5

The algorithms share a common structure: they solve a problem by a sequence of proximity searches (nearest neighbor or fixed-radius near neighbor) in a semidynamic point set. Many of the algorithms also employ a priority queue of links. These techniques were first used by Bentley and Friedman[11] to compute Minimum Spanning Trees; they appear to be of general utility for solving closeness problems for many metrics and dimensions. These techniques might be able to solve other closeness problems, including computing additional TSP tours (such as the Nearest Merger heuristic), Matchings (such as 3-Opting a matching), and Greedy Triangulations.

An obvious problem raised by this paper is to prove theoretical bounds on the complexity of computing the various heuristics. Here are some of the more enticing open questions.

- These algorithms show the utility of data structures that support searching in semidynamic point sets that allow deletions and undeletions, but not insertions. Standard "dynamization" techniques, such as those in Bentley and Saxe[13] and Mehlhorn[33] (his Section VII.1), do not handle deletions efficiently. Is it possible to build an efficient worst-case data structure that supports the mix of operations used by these algorithms?
- An orthogonal problem is to prove bounds on the number of proximity searches required by the various algorithms. For instance, can one prove that for uniform point sets, the MF or NI algorithms make only $O(N)$ nearest neighbor searches? For instance, Bentley and Saxe[13] use sphere-touching arguments to show that an implementation of Greedy matching uses only $O(N)$ nearest neighbor searches in the worst case.
- The bounds implied by the experimental results and heuristic arguments in this paper may be interpreted as conjectures about the expected time required for these tasks. Bentley and Saxe,[13] for instance, use a cell-based method to compute Nearest Neighbor tours of uniform planar sets in $O(N \log^2 N)$ expected time; can similar bounds be found for other tours?

Appendix A1. Supporting Algorithms

This appendix briefly reviews some of the data structures and algorithms that are used in the body of the paper. The algorithms are described in a pseudocode based on Pascal that incorporates parts of the class mechanism of C++; readers unfamiliar with classes should see Stroustrup.[41]

Priority Queues

The priority queues used by the algorithms can be implemented by the heaps described by Knuth[27] (his Section 5.2.3) and Aho, Hopcroft and Ullman[2] (their Section 4.10). The heaps operate on pairs of elements: a floating point *Value* and an associated integer *Signature*. (Typically, the *Value* is the cost of a link and the *Signature* is the number of the point from which the link originates.) At initializa-

tion, the heaps can be declared to have either the minimum or maximum elements on top. The heaps provide the following operations:

Insert(Val, Sig): inserts the pair into an N-element heap in $O(\log N)$ time.

GetTop(Val, Sig): places the value and signature of the top heap item in the variables *Val* and *Sig* in constant time (*Val* and *Sig* are passed by reference).

SetTop(Val, Sig): changes the pair at the top of the heap (and readjusts the heap) in $O(\log N)$ time.

RmTop( ): removes the pair at the top of the heap in $O(\log N)$ time.

K-d Trees

K-dimensional binary search trees represent point sets in K-space. Semidynamic trees represent point sets that support deletions and undeletions, but new points may not be inserted. The semidynamic K-d trees described by Bentley[9] are summarized in Exhibit 5.

The first two operations are the C++ constructor and destructor that build and free a tree. A tree that represents N points in K-space can be built in $O(N \log N + KN)$ worst-case time. The *DeletePt* function deletes a single point in constant (amortized) time, while *DeleteAll* deletes all the points in the tree in $O(N)$ time. The *UnDelete* function negates the effect of a previous *Delete* operation (observe that new points may not be inserted into the tree), and *UnDeleteAll* returns a tree to its original state. Both *UnDelete* functions have the same run time as the corresponding *Delete* functions.

Exhibit 5.

```
class kdtree{
public:
//Maintenance
  KDTree(PointSet*P);
  ~KDTree( );
voic DeletePt(int PointNum);
voic DeleteAll( );
voic UndeletePt(Int PointNum);
void UndeleteAll( );
//Searches
int  NN(int PointNum);
void FRNN(int PointNum, float Radius,
  PFIV F);
void SetRad(int PointNum, float Radius);
void BallSearch(int PointNum, PFIV F);
}
```

Bentley[9] describes many variations of K-d trees; van Kreveld and Overmars[48] give a more theoretical view of the structure. All experiments in this paper use bottom-up operations for deletion, undeletion, and searching. Experiments that report CPU times are tuned for speed and robustness: they use a bucket size of 5, sampling cut planes, and store bounds arrays at every third level of the tree. Experiments that describe node visits and distance calculations are tuned for reproducibility and ease of analysis: they use a bucket size of 1, simple cut planes, and store bounds arrays at every level of the tree. The squared distance was used in most K-d tree operations; when true Euclidean distances were needed, they were computed by a loop-unrolled Newton iteration; a distance cache was used by the Insertion and Addition heuristics and by 2H-Opting and 3-Opting.

The *NN* function uses a bottom-up search to return the index of the nearest undeleted neighbor to *PointNum* (other than *PointNum* itself). There are two key operations in the search: visiting nodes in the tree and performing distance calculations. All other operations require time that is (at most) proportional to those two. Section 5 of Bentley[9] presents heuristic arguments and experimental data to support the conjecture that the expected costs of both operations on uniform data grow as a function of the form $A + BN^C$, where $A > 0$, $B < 0$ and $-1 < C < 0$. Table VI summarizes the cost of nearest neighbor searching in the various algorithms, which have different deletion patterns. The standard errors were estimated by the least-squares regression routine. Multiplying the values by 1.96 gives the 95% confidence intervals for the values A, B and C, respectively.

The functions describing number of nodes visited in

Table VI. Nearest-Neighbor Search Costs

Heuristic Name	Nodes Visited				Distance Calculations			
	Function	Standard Errors			Function	Standard Errors		
nn	$20.6-31.7N^{-0.35}$	0.09	1.74	0.013	$4.22-6.0N^{-0.50}$	0.012	1.14	0.044
denn	$20.5-29.0N^{-0.34}$	0.19	2.53	0.024	$4.24-3.7N^{-0.42}$	0.078	2.39	0.016
ra	$22.1-32.4N^{-0.32}$	0.20	1.88	0.017	$3.94-3.6N^{-0.42}$	0.023	0.74	0.050
ri	$22.3-30.3N^{-0.30}$	0.25	1.75	0.018	$3.94-4.2N^{-0.44}$	0.017	0.62	0.035
mf	$24.6-35.0N^{-0.35}$	0.16	2.06	0.017	$5.27-7.9N^{-0.53}$	0.011	0.80	0.025
ch	$32.2-39.7N^{-0.21}$	0.81	1.37	0.018	$5.01-7.9N^{-0.49}$	0.050	2.06	0.065
mst	$34.6-40.7N^{-0.19}$	2.33	2.11	0.039	$5.01-8.4N^{-0.51}$	0.011	4.97	0.162
na	$32.4-45.6N^{-0.25}$	0.46	1.35	0.012	$4.96-13.4N^{-0.63}$	0.056	6.51	0.133
ni	$36.5-40.2N^{-0.17}$	1.34	0.55	0.017	$5.05-7.0N^{-0.46}$	0.064	1.70	0.072
fa	$< 4.1 + 1.7 \lg N$				< 2.5			
fi	$< 3.8 + 1.7 \lg N$				< 2.5			

Table VI are summarized graphically in Figure 28 (similar functions are combined; the function plotted is the first in the list of tour names).

The *FRNN* function performs a fixed-radius near neighbor search; it calls the function *F* on all points within *Radius* of *PointNum*. *PFIV* is an abbreviation for Pointer to Function of Integer returning Void. The expected runtime of *FRNN* for spheres that contain a constant number of points also appears to grow as $A + BN^C$, where A, B and C obey the constraints mentioned previously.

The final two functions implement ball searching. In this problem we have a set of balls in space, where each ball is centered at a point in the set and has an associated radius. *SetRad* sets the radius associated with a point, and *BallSearch* calls the function *F* with the indices of all points whose balls contain point *PointNum*. When the balls contain a constant number of points, the expected bottom-up search time appears to be bounded by a constant.

Matching Algorithms

Christofides' heuristic in Section 3 calls for computing a minimum-length matching among N points in K-space. Tarjan[44] (his Chapter 12) describes the general matching problem; algorithms compute optimal matching in dense graphs with N vertices in $O(N^3)$ time. Vaidya[47] computes an optimal matching among points in the plane in $O(N^{5/2} \log^4 N)$ time. Avis[3] surveys the problem of computing approximate matchings in geometric sets; that paper describes, among others, algorithms due to Supowit, Reingold and Plaisted[43] and algorithms and experimental results due to Iri, Murota and Matsui.[22] Vaidya[46] gives an approximation algorithm that finds a matching of length at most $1 + \varepsilon$ times the length of the optimal matching for fixed L_q metric in worst-case time $O(\varepsilon^{-1.5K}\alpha(N, N)^{0.5}N^{1.5} \log^{2.5} N)$; he also gives an algorithm with $O(N \log^3 N)$ time that finds a matching at most $3 \log_3 1.5N$ times the optimal length.

We will turn our attention now to the Greedy heuristic for matching. The heuristic iteratively matches the closest pair of points in the set, deletes those two points, and continues. The left panel of Figure 29 shows a Greedy matching (the right panel shows the same matching after 2-Opting, which we will study shortly). Reingold and

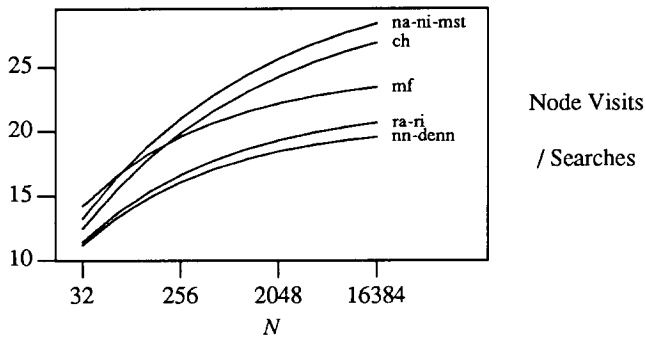

Figure 28. Nearest neighbor search costs.

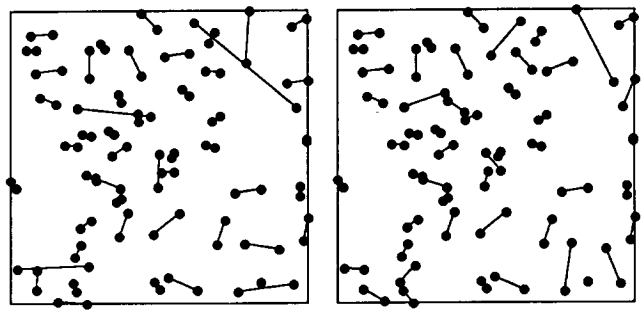

Figure 29. A Greedy matching, before and after 2-Opting.

Tarjan[37] prove that in the worst case, the length of the Greedy matching is never more than $(4/3)N^{\log 1.5}$ times the length of the minimal matching, where $\log 1.5 \approx 0.585$.

Bentley and Saxe[13] describe a complicated algorithm to compute planar Greedy matchings in $O(N^{3/2} \log N)$ time. Supowit[42] gives an algorithm to compute Greedy matchings in K-space in $O(N \log^K N)$ time. Manacher and Zobrist[32] describe a "stage" algorithm for planar point sets. At each stage, the algorithm computes all nearest neighbors in the point set. Any pair of mutual nearest neighbors are added to the matching and deleted from the set. Experiments indicate that for uniform point sets some constant fraction of points are discarded at each stage, so with a K-d tree algorithm to find nearest neighbors, the algorithm runs in $O(N \log N)$ observed time.

We will now describe a matching algorithm based on links and a priority queue. The links point from a point to its nearest neighbor (at one time) not in the matching. The priority queue is organized with the minimum value on top. The details of the algorithm are similar to the other algorithms (see Exhibit 6).

We next briefly sketch the performance of the algorithm. On uniform data, the algorithm makes approximately $1.5N$

Exhibit 6.

```
for I := 1 to N do
  NNLink[I] := Tree → NN(I)
  PQ → Insert(Dist(I, NNLink[I]), I)
loop N/2 times
  loop
    PQ → GetTop(ThisDist, X)
    if NNLink[X] = -1 then // X already in
      PQ → RmTop( )
      continue
    Y := NNLink[X]
    if NNLink[Y]!= -1 then // Y outside
      break
    NNLink[X] := Tree → NN(X)
    PQ → SetTop(Dist(X, NNLink[X]), X)
  AddEdge(X, Y)
  PQ → RmTop( )
  NNLink[X] := NNLink[Y] := -1
  Tree → DeletePt(X)
  Tree → DeletePt(Y)
```

nearest neighbor searches, and the observed average search cost is bounded above by a constant. The run time appears to be robust for the 10 nonuniform distributions. The link-based algorithm appears to be slightly slower than Manacher and Zobrist's[32] stage algorithm on uniform data: it makes slightly few nearest neighbor searches, but pays additional time for the priority queue operations. The stage algorithm, however, displays quadratic behavior for the "arith" distribution, for instance, so it is far from robust.

Avis, Davis and Steele[4] show that for every dimension $K \geq 2$, the expected length of the Greedy matching grows as $\alpha N^{1-1/K} + o(N^{1-1/K})$, which is within a constant factor of optimal. Iri, Murota and Matsui[22] experimentally investigate the constant α for several heuristics on planar point sets. The observed constants range from a high of 0.637 to a low of 0.474; they cite a conjecture of Papadimitriou that the constant for the optimal matching is 0.35, though in their experiments they observed that the optimal value "seems to lie between 0.32 and 0.33." We performed experiments on the Greedy heuristic for N ranging from 32 to 8192, and least-square regressions showed that the Greedy length grows as $0.385\sqrt{N} + 0.47$. Even if we take the low estimate for the length of the optimal matching, Greedy seems to perform within 20%, on the average. The run time and the matching length on the sets of odd vertices in the CH algorithm are quite close to the performance on uniform data.

The matchings produced by the Greedy algorithm can usually be improved by 2-Opting. A 2-Opt swap in a matching reduces the matching length by interchanging two edges associated with four points; a matching is 2-Optimal if no further 2-Opt swaps can be made. To reduce 2-Opting to a problem of locality searching, we observe that in any 2-Opt swap, an edge adjacent to one of the vertices decreases in length.

A simple implementation of 2-Opting a matching repeatedly scans through all N points in the set. At each point it performs a fixed-radius near neighbor search with radius equal to the distance to its matched point. The search checks all points in the ball to see whether any yields a 2-Opt swap, and performs the first swap that is encountered.

Experiments on uniform data show that the searches have constant average cost, but the main loop performs searches at roughly $0.21N^{1.56}$ points (that is, it traverses the matching about $\sqrt{N}/5$ times). We therefore use an approximate version of 2-Opting that associates a bit with each node, similar to the approximate 2-Opting of TSP tours in Section 4. This reduces the number of searches to $O(N)$, but we are left with about $O(N^{3/2})$ bit tests. An alternative implementation keeps the points to be searched in a queue: it initially contains all points, we continue searching for the point at the head of the queue, we add all four points back to the queue after a 2-Opt swap (unless some are already in the queue), and stop when the queue is empty. The resulting approximate 2-Opting yields matchings that are close in length to the true 2-Opt matchings, yet it performs only about $1.4N$ fixed-radius near neighbor searches to make about $0.16N$ 2-Opt swaps. This version of 2-Opting a matching appears to require $O(N)$ time.

The length of a Greedy matching after 2-Opting appears to grow as $0.326\sqrt{N} + 0.24$ for point sets distributed uniformly over the unit square. This is quite close to the length of optimal matchings observed by Iri, Murota and Matsui,[22] who found that the constant associated with the $\sqrt{N}$ term "seems to lie between 0.32 and 0.33."

Appendix A2. Other Metrics and Dimensions

The animations and experiments in the body of this paper deal with planar point sets using the Euclidean metric. The algorithms, however, are more general: they can be applied to K-dimensional point sets under several different metrics. This section describes experiments that sketch the performance of the algorithms in different dimensions and metrics.

The first experiment we will consider examines the performance of the algorithms under the L_∞ metric on planar sets. The experiment is similar to one we saw in Section 5: for five different sets of $N = 10{,}000$ uniform points we construct twelve different starting tours and apply three different local optimizations to each starting tour. Table VII reports the mean tour length (now in absolute units, because we no longer have a good lower bound on the length) and CPU times in seconds.

The CPU times are typically a little less than the CPU times reported in Section 5 for the Euclidean metric. One possible reason that the Euclidean times might be longer is the expensive square root inherent in that metric (2-Opting cannot use the common speedup of dealing only with squares of distances). The program avoids much of the expense, though, by a highly tuned Euclidean distance function (including loop-unrolled Newton iteration from a close starting point) and a cache to store distances once they are computed. Other factors that might change the run time include the influence of the shape of balls on the cost of locality searches, and the number of swaps made by the local optimization heuristics.

The L_∞ tours are shorter than the L_2 tours. One reason is that L_∞ distances are shorter than L_2 distances: consider the L_∞ distance from the origin to a point (Euclidean) distance 1 away. If the point is at 0 radians, its L_∞ distance is also 1; if the point is at $\pi/4$ radians, its L_∞ distance is $1/\sqrt{2}$. If we take the trivial probabilistic model that all angles are equally likely and integrate between those two angles, we find that the L_∞ distance is on the average only $4/(\pi\sqrt{2}) \approx 0.900316$ times the L_2 distance. In fact, the ratios of the length of the L_∞ starting tours to the L_2 starts in Section 5 vary between 0.892 and 0.924, and the mean ratio is 0.9046. The ratios of tour lengths vary from this constant; this is due to other structural differences among the various tours.

The next experiment deals with the L_1 metric; everything else remains unchanged. The CPU times are now slightly increased, which might be due to the awkward shape of the unit L_1 ball. The tour lengths are also slightly longer. Under the trivial probabilistic model described

Table VII. The Heuristics under the L_∞ Metric

Heuristic Name	Tour Length				CPU Time			
	Start	2-Opt	2H-Opt	3-Opt	Start	2-Opt	2H-Opt	3-Opt
nn	79.9	72.9	72.1	69.7	4	21	23	55
denn	79.9	72.9	72.0	69.7	4	21	24	56
mf	73.8	69.3	69.0	67.7	13	26	30	59
na	82.3	80.2	79.1	75.4	25	39	40	74
fa	73.2	72.8	72.4	71.3	36	48	47	70
ra	75.2	74.1	73.6	71.8	15	28	28	53
ni	82.2	80.2	79.1	75.3	43	56	57	91
fi	73.0	72.7	72.3	71.3	70	81	81	102
ri	74.8	74.0	73.5	72.0	50	62	62	86
mst	92.7	80.0	78.2	72.8	15	33	36	74
ch	73.6	71.8	70.6	69.5	24	36	36	57
frp	102.5	82.4	79.9	73.7	2	23	25	70

above, an L_1 distance tends to be about $4/\pi \approx 1.273$ times an L_2 distance. Excluding the unpredictable FRP tour, the ratios of L_1 starting lengths to L_2 starts vary between 1.260 and 1.291, and the mean ratio is 1.275. The observed lengths and times are listed in Table VIII.

Even though the details of run time and tour length vary, the undominated algorithms are the same for the L_1, L_2 and L_∞ metrics. In increasing order of time and decreasing order of length, the contenders are FRP, NN, MF, MF-2, MF-2H, and MF-3.

So far we have considered only planar applications of the algorithms; the algorithms, though, are applicable to K-dimensional spaces. Figure 30 shows an FI tour among 40 points uniformly distributed on the Euclidean hypercube, after 7, 12, 22 and 40 points have been added. This figure is most easily viewed with stereo viewing glasses. Some readers, however, might be able to view it by holding the figure about one foot from the face and starting past the figure out to infinity.

The next experiment is the same as the previous one, except conducted for the L_2 metric in 3-space (see Table IX).

The run times of the algorithms in 3-space tend to be between 2 and 3.5 times longer than in 2-space. We also observe the 2H-Opt is almost always faster than 2-Opt. This is an artifact: 2-Opt was run before 2H-Opt, and it computed and cached all nearest neighbor pairs; 2H-Opt retrieved them from the cache, so was not charged for computing them. Likewise, the cost of building the K-d tree was charged only to the starting tour, not to the local optimizations.

Tours in 3-space have length proportional to $N^{2/3}$, so the tours are now significantly longer than their 2-dimensional counterparts. The contenders remain the same as in the planar case: FRP, NN, MF, MF-2, MF-2H and MF-3. In this case, though, MF is even stronger than before: MF without 2-Opting now beats all other starting tours with 3-Opting, except NN and DENN.

The FRP algorithm provably runs in $O(N \log N)$ worst-case time for any fixed K, and Bentley[9] presents evidence

Table VIII. The Heuristics under the L_1 Metric

Heuristic Name	Tour Length				CPU Time			
	Start	2-Opt	2H-Opt	3-Opt	Start	2-Opt	2H-Opt	3-Opt
nn	111.9	102.9	101.7	98.3	5	25	26	67
denn	112.0	102.7	101.6	98.4	5	25	27	68
mf	104.3	98.2	97.7	96.0	16	29	34	69
na	116.6	113.7	112.1	106.6	31	49	49	93
fa	103.5	102.9	102.4	100.7	41	56	54	82
ra	106.3	104.8	104.1	101.5	19	34	33	66
ni	116.5	113.6	112.0	106.4	52	69	70	115
fi	103.3	102.8	102.2	100.7	89	103	102	129
ri	105.7	104.7	104.0	101.8	67	82	81	112
mst	131.6	113.4	110.8	102.7	19	42	43	93
ch	104.2	101.5	100.0	98.5	28	42	41	68
frp	134.7	117.2	113.5	103.6	2	26	27	100

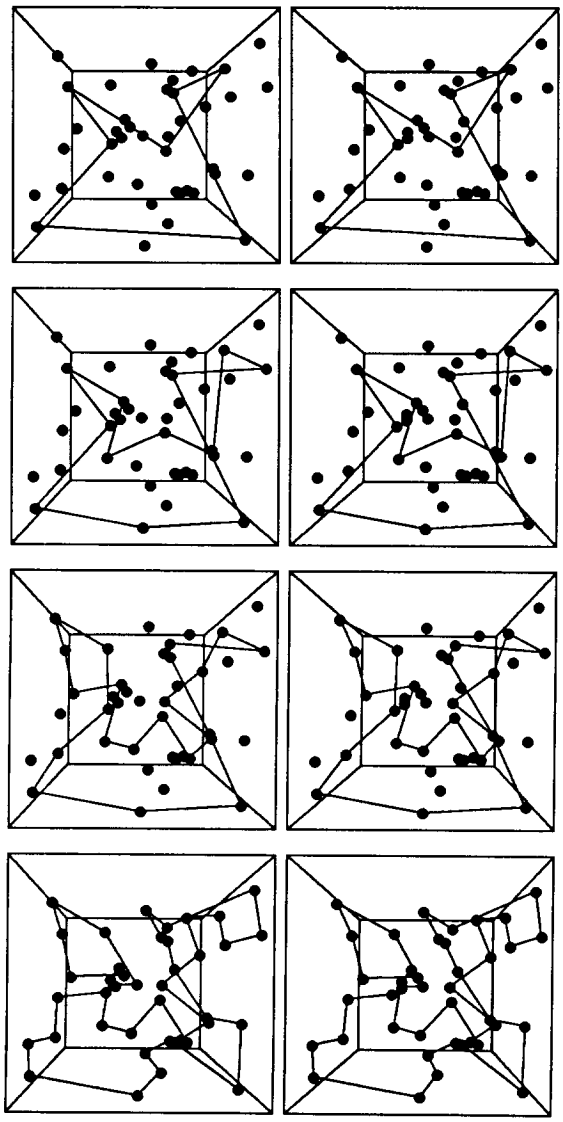

Figure 30. The Farthest Insertion heuristic in 3-space.

that NN runs in $O(N \log N)$ expected time for $K = 3$. We will now describe the results of a simple experiment on the performance of MF in 3-space. In general, the performance of MF in 3-space is quite similar to its performance in 2-space. Its run time on uniform sets is $193N \lg N + 64N$ microseconds. It performs roughly $2.65N$ nearest neighbor searches (compared to $2.60N$ in the plane), and the searches have the same constant-bounded form. The algorithm is quite robust: the maximum robustness ratio among the three-dimensional generalizations of the 10 nonuniform distributions was 1.15 (Bentley[9] describes the 3-d distributions).

The next experiment studied the three-dimensional performance of 2H-Opt following an MF start. The run time of 2H-Opt was $103N \lg N + 1815N$ microseconds. The main loop visits about $0.38 \lg N + 0.28$ nodes; at $N = 10{,}000$, this corresponds to about 5.3 revolutions around the tour. The approximate algorithm finds one bits at about $1.5N$ of those nodes, and makes an average of 1.17 fixed-radius near neighbor searches at those nodes with one bits. The costs of the locality searches have the same constant-bounded form. Apart from fixed-radius near neighbor searching, the algorithm makes about $50N$ distance calculations. The most significant difference is that the algorithm now touches roughly $0.014N^{1.92}$ array elements. At $N = 10{,}000$ this is only $67N$ array accesses, but the balanced tree implementation of Johnson, McGeoch and Ostheimer described in Section 4 becomes more important for larger N. The robustness ratio of 2H-Opting is 1.53 (for N up to 8192), but that ratio appears to be quite flat and not to grow with N.

Appendix A3. The Program

This appendix sketches the program that implements the heuristics. The user's view of the program is described in Bentley.[10] A "detail" parameter controls the output of the program: when it is set to 0 the program performs its job silently, at 10 the program reports summary statistics, at 50 the program produces a simple movie (which are rendered

Table IX. The Heuristics in 3-space

Heuristic Name	Tour Length				CPU Time			
	Start	2-Opt	2H-Opt	3-Opt	Start	2-Opt	2H-Opt	3-Opt
nn	375.3	363.6	361.6	356.0	8	41	38	116
denn	374.4	363.5	361.3	356.0	8	42	39	116
mf	356.3	350.1	349.3	346.8	25	48	52	115
na	393.4	390.7	388.9	379.5	50	80	76	166
fa	377.3	376.3	375.3	371.2	69	96	92	161
ra	378.5	377.1	376.5	371.7	35	63	58	130
ni	393.2	390.6	388.8	379.6	100	130	127	216
fi	377.1	376.2	375.3	371.3	214	242	238	306
ri	378.7	377.6	376.8	372.1	173	201	197	269
mst	452.5	408.7	401.7	375.0	29	74	73	200
ch	374.3	369.5	366.7	361.0	46	74	70	133
frp	476.3	431.2	418.8	384.5	2	54	51	201

as "stills" in this paper), and at 99 the program produces an incredibly detailed movie.

The program was written in the C++ language described by Stroustrup.[41] As the simplest possible measure of size, the program contains roughly 6100 lines of code. Of those lines, though, 500 are blank, 300 are comments, 100 are assertions, 200 fetch global parameters, 150 increment counters and gather statistics, 500 produce animations, and 1200 are declarations; this leaves about 3300 lines of "real" code. The code formatting style is verbose and several techniques use extra code to reduce run time (such as unrolling loops); furthermore, the program implements many experiments on algorithms that are not described in this paper. Here is a brief description of the various files in the program.

- `exper.h` (698 lines). The header file that contains definitions of all the C++ classes (including definitions of all private variables and functions).
- `exper.c` (594 lines). Implements the environment for supporting experiments that is described by Bentley.[10] This includes a little language for describing experiments and C++ classes for gathering and printing statistics.
- `graph.c` (169 lines). Implements the sparse unweighted graphs used by the MST and CH heuristics. It provides maintenance operations to build graphs, and depth-first and Eulerian traversals of graphs.
- `kdbuild.c` (817 lines). Provides routines for building K-d trees and deleting and undeleting points. It contains many options to implement many variations of trees, and also routines for the FRP tour.
- `kdsearch.c` (671 lines). Implements the primary searches in K-d trees: nearest neighbor, fixed radius near neighbor, and ball searches. About one third of the file implements additional kinds of searches not used in this paper: farthest neighbor, multiple nearest neighbor, and almost-sorted fixed-radius near neighbor searches.
- `match.c` (353 lines). Implements the matching algorithms described in Appendix A1: Greedy matching (both the stage and link implementations), and 2-Opting a matching.
- `miscgeo.c` (172 lines). Implements several geometric routines not used in this paper, such as computing multiple nearest neighbors (for instance, the 20 nearest neighbors of each point), and storing the resulting list using a simple data compression scheme.
- `mst.c` (93 lines). Implements a version of Bentley and Friedman's[11] single-fragment algorithm, with the addition of code for computing degree-constrained trees (in which each node has degree at most D).
- `points.c` (682 lines). Implements routines on point sets, including very fast distance calculations for several metrics, and routines to cache distances and trees. Also includes routines for generating sets from the eleven distributions and for reading point sets from files. Approximately 100 lines implement algorithm animation routines for drawing objects in 2-space and 3-space.
- `set.c` (264 lines). Implements ordered set algorithms for priority queues and for linked lists of integers.
- `tour.c` (822 lines). Implements the twelve starting tours described in this paper, and several other heuristics that did not prove effective.
- `touropts.c` (789). Implements three forms of local optimization: 2-Opt, 2H-Opt, and 3-Opt. Includes a variety of options (such as visiting points during fixed-radius near neighbor searchers in an "almost sorted" order).

Acknowledgments

I am grateful for the contributions of Rick Becker, Steve Fortune, David Johnson, Brian Kernighan, Doug McIlroy, Howard Trickey, Chris Van Wyk, and the anonymous referees.

References

1. P.K. AGARWAL, H. EDELSBRUNNER, O. SCHWARZKOPF and E. WELZL, 1990. Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs, pp. 203–210. in *Sixth Annual ACM Symposium on Computational Geometry*, Berkeley, CA (June 1990).
2. A.V. AHO, J.E. HOPCROFT and J.D. ULLMAN, 1974. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
3. D. AVIS, 1983. A Survey of Heuristics for the Weighted Matching Problem, *Networks 13*, 475–493.
4. D. AVIS, B. DAVIS and J.M. STEELE, 1988. Probabilistic Analysis of a Greedy Heuristic for Euclidean Matching, *Probability in the Engineering and Information Sciences 2*, 143–156.
5. J. BEARDWOOD, J.H. HALTON and J.M. HAMMERSLEY, 1959. The Shortest Path through Many Points, *Proceedings of the Cambridge Philosophical Society 55*, 299–327.
6. M. BELLMORE and G.L. NEMHAUSER, 1968. The Traveling Salesman Problem: A Survey, *Operations Research 16*, 538–558.
7. J.L. BENTLEY, 1984. A Case Study in Applied Algorithm Design, *Computer 17:2*, 75–88.
8. J.L. BENTLEY, 1990. Experiments on Traveling Salesman Heuristics, pp. 91–99 in *First Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA (January).
9. J.L. BENTLEY, 1990. K-d Trees for Semidynamic Point Sets, pp. 187–197 in *Sixth Annual ACM Symposium on Computational Geometry*, Berkeley, CA (June).
10. J.L. BENTLEY, 1990. Tools for Experiments on Algorithms, pp. 99–123 in *Proceedings of the CMU 25th Anniversary Symposium*, Pittsburgh, PA (October).
11. J.L. BENTLEY and J.H. FRIEDMAN, 1978. Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces, *IEEE Transactions on Computers C-27:2*, 97–105.
12. J.L. BENTLEY and J.B. SAXE, 1980. An Analysis of Two Heuristics for the Euclidean Travelling Salesman Problem, pp. 41–49 in *18th Annual Allerton Conference on Communication, Control, and Computing* (October).
13. J.L. BENTLEY and J.B. SAXE, 1980. Decomposable Searching Problems I: Static-to-Dynamic Transformations, *Journal of Algorithms 1:4*, 301–358.
14. J.L. BENTLEY, B.W. WEIDE and A.C. YAO, 1980. Optimal Expected-Time Algorithms for Closest Point Problems, *ACM Transactions on Mathematical Software 6:4*, 563–580.
15. J.L. BENTLEY, D.S. JOHNSON, L.A. MCGEOCH and E.E. ROTHBERG, 1992. Near Optimal Solutions to Very Large Traveling Salesman Problems (in preparation).
16. N. CHRISTOFIDES, 1976. Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem, Report 388, Graduate

School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.
17. G.A CROES, 1958. A Method for Solving Traveling Salesman Problems, *Operations Research 6*, 791–812.
18. E.W. DIJKSTRA, 1959. A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik 1*, 269–271.
19. M.M. FLOOD, 1956. The Traveling Salesman Problem, *Operations Research 4:1*, 61–75.
20. H.N. GABOW, J.L. BENTLEY and R.E. TARJAN, 1984. Scaling and Related Techniques for Geometry Problems, pp. 135–143 in *Sixteenth ACM Symposium on the Theory of Computing*.
21. M.R. GAREY and D.S. JOHNSON, 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.
22. M. IRI, K. MUROTA and S. MATSUI, 1983. Heuristics for Planar Minimum-Weight Perfect Matchings, *Networks 13*, 67–92.
23. D.S. JOHNSON, 1990. Local Optimization and the Traveling Salesman Problem, pp. 446–461 in *Proceedings of the Seventeenth Colloquium on Automata, Languages and Programming*, Springer-Verlag, New York.
24. D.S. JOHNSON and C.H. PAPADIMITRIOU, 1985. Performance Guarantees for Heuristics, Chapter 5 of [29].
25. R.M. KARP, 1977. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane, *Mathematics of Operations Research 2*, 209–224.
26. R.M. KARP and J.M. STEELE, 1985. Probabilistic Analysis of Heuristics, Chapter 6 of [29].
27. D E. KNUTH, 1975. *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, Reading, MA.
28. D E KNUTH, 1981. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA.
29. E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN and D.B. SHMOYS, 1985. *The Traveling Salesman Problem*, John Wiley & Sons, New York.
30. S. LIN, 1965. Computer Solutions of the Traveling Salesman Problem, *Bell System Technical Journal 44*, 2245–2269.
31. S. LIN and B.W. KERNIGHAN, 1973. An Effective Heuristic Algorithm for the Traveling Salesman Problem, *Operations Research 21:2*, 498–516.
32. G.K MANACHER, and A.L. ZOBRIST, 1983. Probabilistic Methods with Heaps for Fast-Average-Case Greedy Algorithms, pp. 261–278 in *Advances in Computing Research, Vol. 1: Computational Geometry*, F.P. Preparata (ed.), JAI Press.
33. K. MEHLHORN, 1984. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, New York.
34. C.H. PAPADIMITRIOU and K. STEIGLITZ, 1982. *Combinatorial Optimization*, Prentice-Hall, Englewood Cliffs, NJ.
35. F.P PREPARATA and M.I. SHAMOS, 1985. *Computational Geometry*, Springer-Verlag, New York.
36. R.C. PRIM, 1957. Shortest Connection Networks and Some Generalizations, *Bell System Technical Journal 36*, 1389–1401
37. E.M. REINGOLD and R.E. TARJAN, 1981. On a Greedy Heuristic for Complete Matching, *SIAM Journal on Computing 10*, 676–681.
38. D.J. ROSENKRANTZ, R.E. STEARNS and P.M. LEWIS II, 1977. An Analysis of Several Heuristics for the Traveling Salesman Problem, *SIAM Journal of Computing 6*, 563–581.
39. J.M. STEELE, 1981. Subadditive Euclidean Functionals and Non-linear Growth in Geometric Probability, *Annals of Probability*, 365–376.
40. K. STEIGLITZ and P. WEINER, 1968. Some Improved Algorithms for Computer Solution of the Traveling Salesman Problem, pp. 814–821 in *6th Annual Allerton Conference on Circuit and Systems Theory* (October).
41. B. STROUSTRUP, 1986. *The C++ Programming Language*, Addison-Wesley, Reading, MA.
42. K. SUPOWIT, 1990. New Techniques for Some Dynamic Closest-point and Farthest-point Algorithms, pp. 84–90 in *First Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA (January).
43. K. SUPOWIT, E.M. REINGOLD and D.A. PLAISTED, 1983. The Traveling Salesman Problem and Minimum Matching in the Unit Square, *SIAM Journal on Computing 12*, 144–156.
44. R.E. TARJAN, 1983. *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA.
45. J.W. TUKEY, 1977. *Exploratory Data Analysis*, Addison-Wesley, Reading, MA.
46. P.M. VAIDYA, 1988. Geometry Helps in Matching, pp. 422–425 in *Twentieth ACM Symposium on the Theory of Computing*.
47. P.M. VAIDYA, 1989. Approximate Minimum Weight Matching on Points in *k*-Dimensional Space, *Algorithmica 4:4*, 569–583.
48. M.J. VAN KREVELD and M.H. OVERMARS, 1991. Divided k-d Trees, *Algorithmica 6:6*, 840–858.
49. J.E. ZOLNOWSKY, 1978. Topics in Computational Geometry, Ph.D thesis, Stanford University (May), STAN-CS-78-659.