K-d Trees for Semidynamic Point Sets

Jon Louis Bentley

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

A *K*-d tree represents a set of *N* points in *K*-dimensional space. Operations on a *semidynamic* tree may delete and undelete points, but may not insert new points. This paper shows that several operations that require $O(\log N)$ expected time in general *K*-d trees may be performed in constant expected time in semidynamic trees. These operations include deletion, undeletion, nearest neighbor searching, and fixed-radius near neighbor searching (the running times of the first two are proved, while the last two are supported by experiments and heuristic arguments). Other new techniques can also be applied to general *K*-d trees: simple sampling reduces the time to build a tree from $O(KN \log N)$ to $O(KN + N \log N)$, and more advanced sampling builds a robust tree in the same time. The methods are straightforward to implement, and lead to a data structure that is significantly faster and less vulnerable to pathological inputs than ordinary *K*-d trees.

1. Introduction

A *K*-dimensional binary search tree (abbreviated *K*-d tree) represents a set of points in *K*-dimensional space. Many kinds of searches can be performed on a *K*-d tree, including exact-match, partial-match, and range queries. The discussion of *K*-d trees in many textbooks concentrates on these query types, which are important in database applications (see, for instance, Mehlhorn [1984, Section 2.1], Preparata and Shamos [1985, Section 2.3.2], and Sedgewick [1988, Chapter 26]).

This paper is concerned with *K*-d trees that support two kinds of proximity queries:

Nearest neighbor query. Report the nearest neighbor to a given query point, under a specified metric.

Fixed-radius near neighbor query. Report all points within a fixed radius of the given query point, under a specified metric.

Additionally, we will focus on *semidynamic* point sets. An original set of *N* points is known when the *K*-d tree is built; subsequent operations may *delete* points from the set or *undelete* points that have been previously deleted. New points may not be inserted.

Restricting *K*-d trees to proximity queries on semidynamic point sets leads to a particularly efficient implementation. Because the points are known in advance, bottom-up algorithms reduce the expected time for operations on a tree from $O(\log N)$ to $O(1)$. Bentley [1990b] uses the data structure to solve a number of closeness problems in computational geometry, such as computing minimum spanning trees, matchings, and many kinds of traveling salesman tours.

Section 2 of this paper reviews previous work on *K*-d trees. Section 3 defines the abstract data type implemented by the algorithms described later. Section 4 introduces bottom-up algorithms for the simple delete and undelete operations, and Sections 5 and 6 describe bottom-up searches. The algorithms in Section 7 for building *K*-d trees are also applicable to general *K*-d trees. Conclusions are offered in Section 8.

2. Previous Work

The multidimensional binary search tree introduced by Bentley [1975] generalizes the standard one-dimensional binary search tree. The first part of this section will review the variant described by Friedman, Bentley and Finkel [1977], which distinguishes between two kinds of nodes: internal nodes partition the space by a cut plane defined by a value in one of the *K* dimensions, and external nodes (or *buckets*) store the points in the resulting hyperrectangles of the partition. Figure 2.1 shows the partition given by a 2-d tree representing 300 points drawn uniformly from the unit square (each bucket has at most 3 points).

A node in a *K*-d tree can be represented by the following C++ structure (for information on C++, see Stroustrup [1986]):

```
struct kdnode {
    int     bucket;
    int     cutdim;
    float   cutval;
    kdnode  *loson, *hison;
    int     lopt, hipt;
};
```

The variable `bucket` is 1 if the node is a bucket and zero if it is an internal node. In an internal node, `cutdim` gives the dimension being partitioned, `cutval` is a value in that

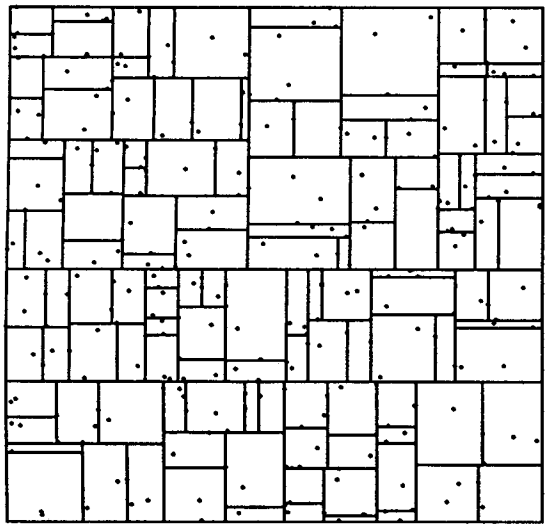

Figure 2.1. A 2-d tree of 300 points.

dimension, and `loson` and `hison` are pointers to its two subtrees (containing, respectively, points not greater than and not less than `cutval` in that dimension). In a bucket node, `lopt` and `hipt` are indices into the global permutation vector `perm[n]`; the $hipt - lopt + 1$ integers in `perm[lopt..hipt]` give the indices of the points in the bucket (thus `perm` is just a convenient set representation).

As the tree is being built, two pointers into `perm` represent a subset of the points. The tree is built by this code:

```
for (i = 0; i < n; i++)
    perm[i] = i;
root = build(0, n-1);
```

The main work is done by the recursive function `build`, shown in Program 2.1. The code is written in C++; for brevity, declarations have been omitted. The function

```
kdnode *build(int l, int u)
{   p = new kdnode;
    if (u-l+1 <= cutoff) {
        p->bucket = l;
        p->lopt = l;
        p->hipt = u;
    } else {
        p->bucket = 0;
        p->cutdim = findmaxspread(l, u);
        m = (l + u) / 2;
        select(l, u, m, p->cutdim);
        p->cutval = px(m, p->cutdim);
        p->loson = build(l, m);
        p->hison = build(m+1, u);
    }
    return p;
}
```

Program 2.1. Building a K-d tree.

```
extern int    nntarget, nnptnum;
extern float  nndist;

int nn(int j)
{   nntarget = j;
    nndist = HUGE;
    rnn(root);
    return nnptnum;
}

void rnn(kdnode *p)
{   if (p->bucket) {
        for (i = p->lopt; i <= p->hipt; i++) {
            thisdist = dist(perm[i], nntarget)
            if (thisdist < nndist) {
                nndist = thisdist;
                nnptnum = perm[i];
            }
        }
    } else {
        val = p->cutval;
        thisx = x(nntarget, p->cutdim);
        if (thisx < val) {
            rnn(p->loson);
            if (thisx + nndist > val)
                rnn(p->hison);
        } else {
            rnn(p->hison);
            if (thisx - nndist < val)
                rnn(p->loson);
        }
    }
}
```

Program 2.2. Nearest neighbor searching.

`findmaxspread` returns the dimension with largest difference between minimum and maximum among the points in `perm[l..u]`. The function `px(i,j)` accesses the j-th coordinate of point `perm[i]`. The function `select` permutes `perm[l..u]` such that `perm[m]` contains a point that is not greater in the `p->cutdim`-th dimension than any point to its left, and is similarly not less than the points to its right. Function `build` runs in $O(KN \log N)$ time.

The `nn` function in Program 2.2 computes the nearest neighbor to a point. The external (global) variables are used by the recursive procedure `rnn`, which does the work. At a bucket node, `rnn` performs a sequential nearest neighbor search. At an internal node, the search first proceeds down the closer son, and then searches the farther son only if necessary. (The function `x(i,j)` accesses the j-th dimension of point i.) These functions are correct for any metric in which the difference between point coordinates in any single dimension does not exceed the metric distance; the Minkowski L_1, L_2 and L_∞ metrics all display this property.

The performance of searching algorithms in K-d trees has proven difficult to analyze. Lee and Wong [1977] give a worst-case bound on the cost of region searching. Friedman, Bentley and Finkel [1977] experimentally investigate the run time of nearest neighbor searching in K-d trees. Section 5 presents several of the experiments. They observe that the expected run time, for fixed K, grows as $O(\log N)$ for many input distributions, and give heuristic arguments suggesting why that should be the case. Zolnowsky [1978] presents a variant of K-d tree where the worst-case cost of searching for

all nearest neighbors in a set of N points in K-space is $O(N \log^K N)$, and gives a point set that realizes this time bound. Zolnowsky's result implies that the amortized cost of a nearest neighbor search is $O(\log^K N)$.

Sproull [1988] describes several ways in which the K-d tree algorithms and data structure can be improved. The nearest neighbor search function `rnn` already incorporates one of Sproull's ideas: it removes the bounds array that consumed most of the CPU time in Friedman, Bentley and Finkel's [1977] program (we will return to this topic in Section 5). For the common case of the Euclidean metric, Sproull also observes that one need not compute the (expensive) true distance to the nearest neighbor. Computing only the square of the distance is clearly sufficient within a bucket. This code fragment shows how the square suffices at internal nodes:

```
diff = x(nntarget, dim) - p->cutval;
if (diff < 0) {
    rnn(p->loson);
    if (nndist2 >= diff*diff) rnn(p->hison);
} else {
    rnn(p->hison);
    if (nndist2 >= diff*diff) rnn(p->loson);
}
```

The new global variable `nndist2` represents the square of `nndist`.

Sproull describes several other improvements to K-d trees. While standard K-d trees use partition planes that are orthogonal to a coordinate axis, Sproull uses arbitrary partition planes. In particular, he uses a plane found from the principal eigenvector of the covariance matrix of the current subset of points. Sproull also offers a number of "coding tricks" that speed up a program implementing K-d trees (two of his suggestions were mentioned earlier).

3. The Abstract Data Type

The basic operations on a semidynamic K-d tree for proximity problems are defined in this C++ class:

```
class kdtree {
public:
            kdtree(pointset *ptset);
    void    deletept(int ptnum);
    void    undeletept(int ptnum);
    int     nn(int ptnum);
};
```

The first operation creates a tree given a point set. Function `deletept` removes point `ptnum` from the set; its dual `undeletept` returns it to the set. Function `nn` returns the index of the nearest neighbor to its argument (not the argument itself); ties are broken arbitrarily. In later sections we will add other operations to the class.

Many closeness problems in computational geometry can be solved using these primitive operations. Here, for instance, is a C++ function to store a nearest neighbor traveling salesman tour of a set in the vector `tour[n]`:

```
void nntour(pointset *ptset,
            int startpt,
            int *tour)
{   tree = new kdtree(ptset);
    tour[0] = startpt;
    tree->deletept(tour[0]);
    for (i = 1; i < n; i++) {
        tour[i] = tree->nn(tour[i-1]);
        tree->deletept(tour[i]);
    }
    delete tree;
}
```

Here is a (partial) list of other closeness problems that Bentley [1990b] solves using the primitive operations in this class:

> Minimum Spanning Trees: true MST, degree-constrained MST.
>
> Matchings: greedy matching, 2-opting a matching.
>
> Traveling salesman problem heuristics: nearest neighbor, minimum spanning tree, approximate Christofides' heuristic, multiple fragment (greedy), nearest insertion, farthest insertion, random insertion, nearest addition, farthest addition, random addition.
>
> Local improvements to traveling salesman tours: two-opt, two-and-a-half-opt, three-opt.

Most of these algorithms have (experimentally observed) running times of $O(N \log N)$; the K-d trees of this paper play a crucial role in the rapid implementations.

4. Bottom-Up Deletion and Undeletion

To support semidynamic point sets, we will add the new field `empty` to each node in the tree. The field is 1 if all points in the subtree have been deleted and is 0 otherwise. Search procedures are modified to return immediately when they visit an empty node. To delete a node from the tree we first delete it from its bucket and then (as needed) turn on empty bits on the path to the root. To delete a point from a bucket we swap that point in the `perm` vector with the current `hipt`, and decrement the latter. This strategy does not attempt to rebalance the tree as elements are deleted; we will see in Section 5 that deletions do not have a strong negative effect on search time.

A recursive top-down deletion algorithm starts at the root, searches down to the proper bucket, swaps and decrements, and (as needed) turns on `empty` flags on the way up. There is, however, the bothersome complication that a point with a value equal to the discriminator might be in either son. Not only does this clutter the code, but deleting one of M identical points must sometimes involve investigating all M points.

A bottom-up version of the function is cleaner and faster. It requires two changes to data structures: each node in the tree contains a `father` pointer, and the array element `bucketptr[i]` points to the bucket that contains node `i`. Both of these changes are straightforward to incorporate into the `build` function. Here is the resulting deletion code:

```
void delete(int pointnum)
{   p = bucketptr[pointnum];
    j = p->lopt;
    while (perm[j] != pointnum)
        j++;
    permswap(j, (p->hipt)--);
    if (p->lopt > p->hipt) {
        p->empty = 1;
        while ((p = p->father)
               && p->loson->empty
               && p->hison->empty)
            p->empty = 1;
    }
}
```

Any single deletion requires at most $O(\log N)$ time, because the tree has that depth. The total cost of sequentially deleting every point in a set is $O(N)$; the analysis is isomorphic to the time of building a heap bottom-up (see Theorem 3.5 of Aho, Hopcroft and Ullman [1974]). The amortized cost of a deletion is therefore $O(1)$. (The linear-time function `deleteall` removes all points in a set with less overhead by traversing the tree.)

The `undelete` function has a similar structure: it performs a sequential search within the bucket, increments `hipt` and performs a `permswap`, then finally moves up the tree, turning off as many `empty` flags as necessary. As with deletion, a single undeletion requires at most $O(\log N)$ time, and the total cost of sequentially undeleting every point in a set is $O(N)$, for constant amortized cost.

Some sequences of $O(N)$ deletions and undeletions can require time proportional to $N \log N$. The first $N/2$ operations delete all the elements in one subtree of the root in $O(N)$ time, then N subsequent pairs of operations delete and undelete an arbitrary element of that subtree; each of the operations must proceed from the node to the root, so the cost of each is proportional to $\log N$. Fortunately, many sequences of operations that occur in geometric algorithms can be shown to require $O(N)$ time.

5. Bottom-Up Searching

Before we consider bottom-up search algorithms, we will briefly study the run time of the top-down nearest neighbor search algorithm (given by functions `nn` and `rnn` in Section 2). Figure 5.1 summarizes 100 experiments with N ranging from 100 to 100,000, uniformly on a logarithmic scale. For each value, we generated N points uniformly on the unit square, built a 2-d tree for the point set (with the bucket cutoff set to 5), and performed a search to find the nearest neighbor of each point in the set. The circles in the graph plot the average number of distance calculations per search made within the buckets, and the crosses plot the average number of internal nodes visited by each search.

Figure 5.1 shows several aspects of top-down nearest neighbor search. Both variables display a cyclic behavior, periodic at powers of two. Because the bucket cutoff is fixed (5 in this case), the average number of nodes in a bucket increases and then decreases periodically at powers of two, which is the rate at which new levels are added to the tree. Notice the tradeoff between distance calculations and

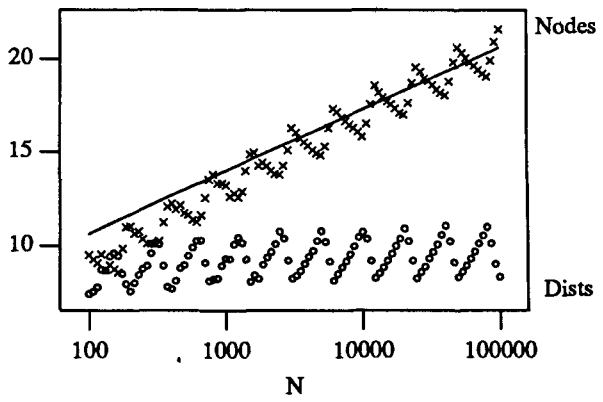

Figure 5.1. All nearest neighbors, top-down search, bucket cutoff 5.

nodes visited: when there are more points in each bucket, the algorithm makes more distance calculations and visits fewer nodes. The line at $\lg N + 4$ shows that the number of nodes visited is growing logarithmically. The number of distance calculations appears to be approaching oscillation around a constant near ten.

Our next set of experiments uses an inefficient variant of K-d tree to simplify analysis of the data: the bucket cutoff is set to one. For each N a power of two from $2^5 = 32$ to $2^{17} = 131072$, we generated 10 sets of N points at random on the unit square, built a 2-d tree, and then searched for all nearest neighbors in the point set. The bucket cutoff of one decreases the number of distance calculations but increases the number of nodes visited. Figure 5.2 shows that the average number of distance calculations appears to approach a constant near 5. The number of nodes visited appears to approach the line on the graph at $\lg N + 14$.

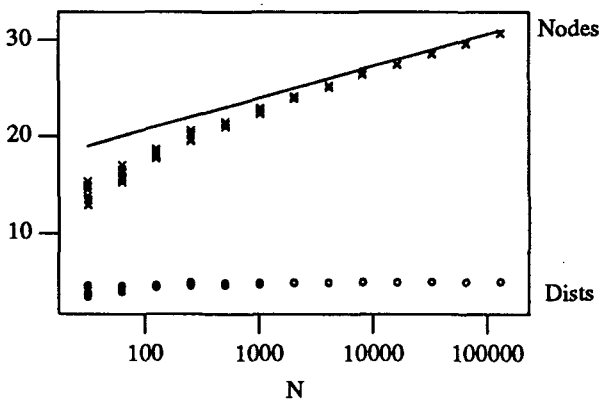

Figure 5.2. All nearest neighbors, top-down search, bucket cutoff 1.

```
int nn(int j)
{   nntarget = j;
    nndist2 = HUGE;
    p = bucketptr[nntarget];
    rnn(p);
    while (1) {
        lastp = p;
        p = p->father;
        if (p == 0) break;
        diff = x(nntarget, p->cutdim)
                    - p->cutval;
        if (nndist2 >= diff*diff) {
            if (lastp == p->loson)
                rnn(p->hison);
            else
                rnn(p->loson);
        }
        if (ball2_in_bounds(p->bnds,
                    nntarget, nndist2))
            break;
    }
    return nnptnum;
}
```

Program 5.1. Bottom-up nearest neighbor searching.

These experiments show us how a top-down search typically proceeds: it visits exactly lg N internal nodes on the way to the correct bucket, rummages around a constant number of neighboring buckets, then recursively returns to the root. Friedman, Bentley and Finkel [1977] give probabilistic arguments to support this description. The bottom-up search algorithm in Program 5.1 reduces the $O(\log N)$ time to $O(1)$ by starting at the correct bucket. It then works its way up the tree, using the recursive `rnn` function to search as many "farther" sons as needed. To halt the upward climb we store at each node in the tree a bounds array `bnds` that describes the hyperrectangle that the node represents (defined by cut planes above it in the tree; details are in Friedman, Bentley and Finkel [1977]). We stop the search when the bounds for the node contain the current nearest-neighbor ball (defined to be a ball centered at the search point with radius equal to the distance to the nearest neighbor). Function `ball2_in_bounds` is passed a bounds array, a point index, and a distance (squared). In $O(K)$ time, it returns 1 if the ball centered at the point with that (squared) radius is contained in the hyperrectangle represented by the bounds array.

Figure 5.3 shows the next set of experiments: bottom-up nearest neighbor searches on exactly the trees summarized in Figure 5.2. A simple analysis (which we'll see shortly) indicated that the number of nodes visited might grow as $a - bN^{-1/2}$, for appropriate constants a and b. A weighted non-linear least squares program was used to fit the number of nodes visited to the model $a + bN^c + d \lg N$, where the logarithmic term can account for work going up the tree. The estimates were $a=19.17$, $b=-25.88$, $c=-.386$, and $d=-.0016$. Because the standard error of the estimate of d is .0546 (an order of magnitude larger than the estimate), we may safely assume that d is zero. A second fit with $d=0$ showed that the number of nodes visited is accurately described by the function $19.14 - 26.01N^{-0.39}$, which is plotted on the graph. A similar weighted least squares fit

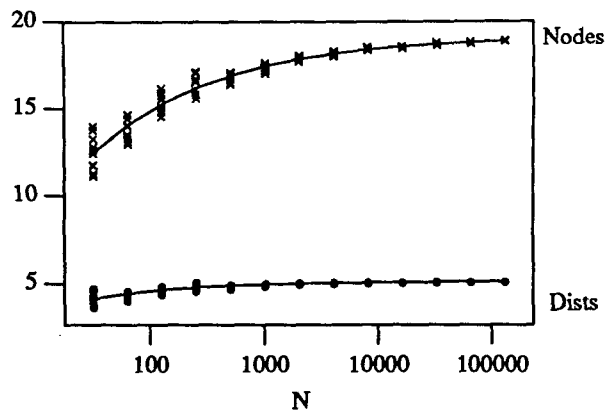

Figure 5.3. All nearest neighbors, bottom-up search.

showed that the number of distance calculations is approximately $5.11 - 6.18N^{-0.53}$, which is also plotted in Figure 5.3 (the values are about one percent greater than the number used by the top-down algorithm).

The data supports this conjecture.

Conjecture 5.1: The expected running time of a bottom-up nearest neighbor search in a K-d tree for a point set uniform on the unit square is $O(1)$.

The conjecture is also supported (but not proved) by several analytical arguments.

1. Bentley, Weide and Yao [1980] show that nearest neighbor search in a cell data structure requires constant time for many distributions of points. As Figure 2.1 suggests, K-d trees yield a cell-like partition for uniform data; thus their theorem supports the conjecture.
2. Consider the root of the K-d tree. Arguments in Section 3 of Bentley, Weide and Yao [1980] show that with high probability, bottom-up searches for at most $O(N^{1/2} \log N)$ of the points will have to proceed as high as the root of the tree. If those searches require $O(\log N)$ time, then the total cost of performing all N searches obeys the recurrence

   $$T(N) = 2T(N/2) + O(N^{1/2} \log^2 N)$$

   which has solution $T(N) = O(N)$, and the amortized cost of a search is constant.
3. We turn now to a more precise analysis of nodes visited. Other arguments of Bentley, Weide and Yao [1980] suggest that, on the average, bottom-up searches for only $2\sqrt{N}$ of the points will proceed as high as the root. If we assume that the cost of each of those searches is lg N, then the total cost of N searches is given by the recurrence

   $$T(N) = 2T(N/2) + 2\sqrt{N} \lg N$$

   with the boundary condition $T(1)=1$. We can investigate the behavior of this recurrence at powers of two by defining $C_i = T(2^i)/2^i$; C_i can be viewed as the average cost of a bottom-up nearest neighbor search in a tree of height i. The recurrence for T becomes

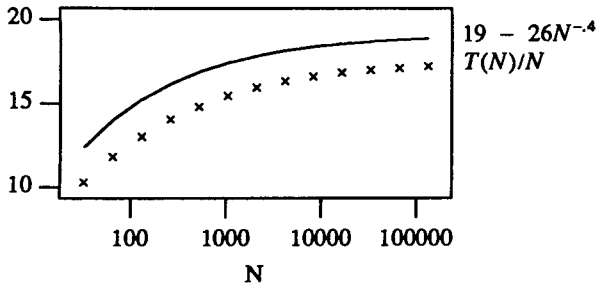

Figure 5.4. Solution of the recurrence $T(N) = 2T(N/2) + 2\sqrt{N} \lg N, T(1) = 1$.

$C_i = C_{i-1} + i2^{1-i/2}$ with the boundary $C_0 = 1$. This recurrence can be telescoped into the sum

$$C_n = 1 + 2 \sum_{1 \le i \le n} i(2^{-1/2})^i$$

Using the identity $\sum_{i \ge 1} ix^i = x/(1-x)^2$, the series converges to

$$C_\infty = 1 + \sqrt{2}/(1 - 1/\sqrt{2})^2 \approx 17.4853$$

Figure 5.4 shows the growth of the recurrence, together with the predictor function $19.14 - 26.01N^{-0.39}$ from Figure 5.3, which is shown as a solid line. The recurrence is consistently about two less than the predictor, which accurately describes the experimental data. The difference of two could be adjusted by using a different boundary condition, such as $T(32) = 12.3$.

These arguments are only heuristic, but they reinforce the experimental observations.

All experiments reported so far have been conducted on static point sets. To study the effect of deletions, Figure 5.5 shows the performance of the bottom-up search when applied to the nearest neighbor tour (function `nntour`) of Section 3. The nearest neighbor tour was computed for ten point sets uniform on the unit square at each power of two from 32 to 131072. The bucket cutoff was one. Both curves in the graph have the same character as those in Figure 5.3, and the differences are indeed slight: the nearest neighbor tour visits about 6% more K-d tree nodes than computing all nearest neighbors, but uses about 20% fewer distance calculations. Weighted least squares regressions showed that the number of nodes visited was $20.41 - 37.87N^{-0.38}$ and the number of distance calculations was $4.22 - 8.70N^{-0.55}$; both functions are plotted in the graph. Bentley [1990b] observes that similar functions characterize the cost of nearest neighbor searching in many geometric algorithms.

Calling the `ball2_in_bounds` function at every node visited during the search is relatively expensive. We therefore store a pointer to a bounds array only at nodes in the tree with depth congruent to zero modulo the variable `bndslevel` (three was discovered to be an effective choice, and is the value used in experiments henceforth in this paper). We then modify function `nn` to perform the `ball2_in_bounds` test only if the `bnds` pointer is nonzero. Because a bounds array is represented by $2K$ floating-point numbers, this also reduces the space required by the tree.

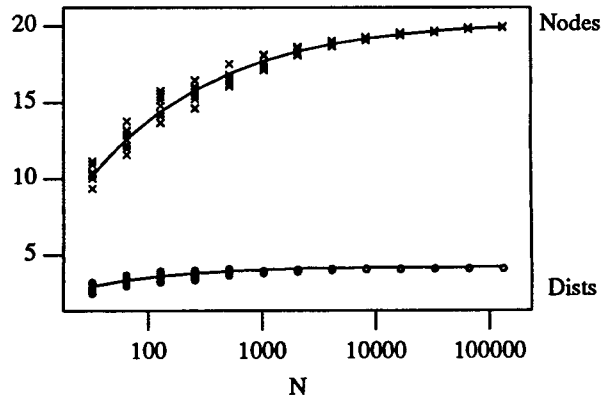

Figure 5.5. Nearest neighbor TSP tour, bottom-up search.

We will now turn our attention to CPU times. Figure 5.6 shows the results of an experiment on 100 uniform point sets, with N varying from 100 to 100000. The programs were implemented in C++ on a VAX 8550. The circles show the CPU time (microseconds per point) for constructing a nearest neighbor tour using bottom-up deletion and searching; the crosses show the CPU time for the top-down operations (they do not include the time to build the K-d tree). For all experiments, the bucket cutoff was set to 5. The least squares regression line through the top-down operations is $34.2 \lg N + 51$ microseconds. A least squares regression for the the bottom-up times fit the data to the model $a + bN^c$; the resulting fit of $363 - 538N^{-0.29}$ is also plotted.

So far we have studied the performance of the bottom-up search algorithm only on data uniform on the unit square. Some algorithms that perform well on uniform data sets perform poorly in real applications; K-d trees were designed to avoid this problem by adapting to the input data. To see how well they accomplish this, we will study an experiment

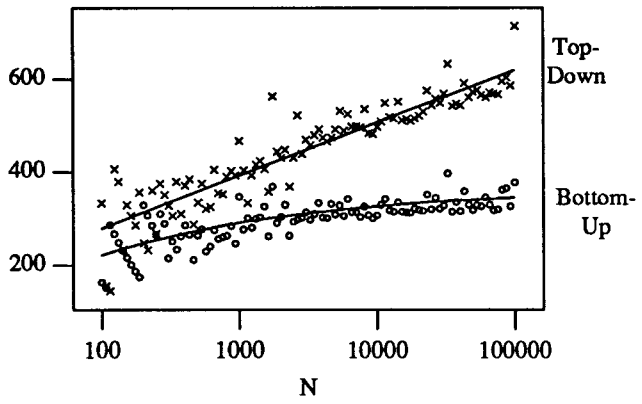

Figure 5.6. Nearest neighbor TSP tour, microseconds per point.

using ten input distributions that are very nonuniform. The distributions are described in this table (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):

NAME	DESCRIPTION
uni	uniform within the unit square ($U[0,1]^2$)
annulus	uniform on the edge of a circle (width zero annulus)
arith	$x_0 = 0, 1, 4, 9, 16, \ldots$ (arithmetic differences); $x_1 = 0$
ball	uniform inside a circle
clusnorm	*Normal*(0.05) at 10 points from $U[0,1]^2$
cubediam	$x_0 = x_1 = U[0,1]$
cubeedge	$x_0 = U[0,1]$; $x_1 = 0$
corners	$U[0,1]^2$ at (0,0), (2,0), (0,2), (2,2)
grid	N points chosen from a square grid of $1.3N$ points
normal	each dimension from *Normal*(1)
spokes	$N/2$ points at ($U[0,1]$, 1/2); $N/2$ at (1/2, $U[0,1]$)

Some of these distributions have served as worst-case counterexamples in papers on geometric algorithms, while others model inputs in various geometric applications.

Figure 5.7 shows the efficiency of bottom-up searching in computing nearest neighbor tours on point sets drawn from these distributions. Five sets of size 10,000 were drawn from each distribution, and the nearest neighbor tour was computed for each. For all experiments, the bucket cutoff was set to 5. Figure 5.7 reports the average number of distance calculations per search (circles) and the average number of nodes visited (crosses) per search. The uniform distribution (leftmost on the graph) provides a benchmark: most distributions have very similar performance, the one-dimensional distributions are more efficient (annulus, arith, cubeedge and cubediam), while one distribution, spokes, is noticeably slower. In Section 7 we'll see how a modified K-d tree gracefully handles the spokes distribution. It is comforting, though, to see that the performance of the algorithms is not substantially slower on nonuniform data. Bentley [1990b] reports that bottom-up searching displays similar performance when employed in a variety of geometric algorithms.

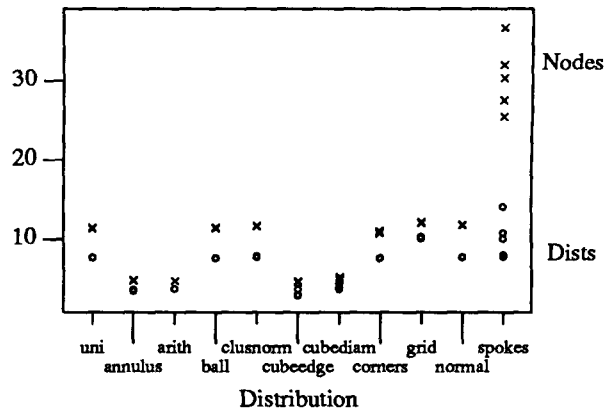

Figure 5.7. Nearest neighbor TSP tour, N = 10,000.

6. Spherical Queries

In this section we will examine two types of queries that deal with spheres. The first type is a fixed-radius near neighbor query: it asks which points in the set are contained in a given sphere. The second type is the dual query: each point in the set has an associated radius (and thereby represents a sphere), and a query asks which spheres contain a given point.

A fixed-radius near neighbor search is defined by an additional member of the C++ `kdtree` class:

```
typedef void (*PFIV)(int i);
void     frnn(int ptnum, float rad, PFIV f);
void     setfrnnrad(float smallerrad);
```

The `frnn` function performs the search: it calls the function `f` for each point within radius `rad` of point `ptnum`. The function parameter uses the declaration `PFIV`; the abbreviation stands for Pointer to Function of Integer returning Void. The function `setfrnnrad` shrinks the radius in the middle of a search; shrinking the radius to zero stops the search.

Program 6.1 gives the recursive function `rfrnn` for top-down near neighbor searching; it is very similar to the `rnn` function in Section 2. The function always visits the nearer son first, and thus tends to report the points in increasing order from the target. The bottom-up algorithm `frnn` in Program 6.1 uses the `rfrnn` function; it is quite similar to the bottom-up nearest neighbor search function in Section 5. The global variables are the integer `nntarget`, the floats `nndist` and `nndist2`, and the function `nnfunc`.

The running time of the search algorithm depends, of course, on many factors. In some applications, though, the query spheres tend to include some constant number of points. Bentley [1990a] describes how this condition holds for 2-opting a traveling salesman tour; it also holds for searching for the M nearest neighbors to a query point. In such applications the algorithm appears to require constant time; its operation is similar to the bottom-up nearest neighbor search in the previous section.

We will turn now to the dual problem of "ball searching": each point in the set has an associated radius (and thereby represents a sphere), and a query asks which spheres contain a given point. We augment the `kdtree` class definition with these functions:

```
void     setrad(int ptnum, float rad);
void     ballsearch(int ptnum, PFIV f);
```

Bentley [1990b] uses these function to implement the "Nearest Insertion", "Farthest Insertion" and "Random Insertion" traveling salesman heuristics.

To implement a ball search we will add to each bucket node a linked list of all balls that intersect it. The `ballsearch` function proceeds to its bucket (in constant time) and checks all balls on the list to see whether they intersect the query point; it calls the function `f` on all balls that do. The `setrad` function makes two passes: the first pass deletes the old radius from the lists that contain it, and the second pass inserts the new radius in the appropriate

```
void frnn(int ptnum, float rad, PFIV f)
{   nntarget = ptnum;
    nndist = rad;
    nndist2 = rad*rad;
    nnfunc = f;
    p = bucketptr[nntarget];
    rfrnn(p);
    while (1) {
        lastp = p;
        p = p->father;
        if (p == 0) break;
        diff = x(nntarget, p->cutdim)
                 - p->cutval;
        if (lastp == p->loson) {
            if (nndist >= -diff)
                rfrnn(p->hison);
        } else {
            if (nndist >=  diff)
                rfrnn(p->loson);
        }
        if ((p->bnds != 0) &&
                ball_in_bounds(p->bnds,
                    nntarget, nndist))
            break;
    }
}
void rfrnn(kdnode *p)
{   if (p->empty) return;
    if (p->bucket) {
        for (i = p->lopt; i <= p->hipt; i++)
            if (distsqrd(perm[i], nntarget)
                    <= nndist2)
                (*nnfunc)(perm[i]);
    } else {
        diff = x(nntarget, p->cutdim)
                 - p->cutval;
        if (diff < 0.0) {
            rfrnn(p->loson);
            if (nndist >= -diff)
                rfrnn(p->hison);
        } else {
            rfrnn(p->hison);
            if (nndist >=  diff)
                rfrnn(p->loson);
        }
    }
}
```

Program 6.1. Bottom-up fixed-radius near neighbor searching.

lists. The work is done by the top-down recursive procedure in Program 6.2. The bottom-up version of this function is similar to the bottom-up versions of previous functions.

The running time of these algorithms depends on many factors. In applications in which the spheres tend to include a constant number of points, both the `setrad` and `ballsearch` functions appear to require constant time. Several alternative schemes were tested, but increased the running time of the functions. One technique, for instance, tested nodes to ensure that their bounds intersected the ball in the current `setrad` operation. This decreased the total number of nodes visited, but increased the cost of visiting each node; the overall run time was slightly higher.

```
void rsetrad(kdnode *p)
{   if (p->bucket) {
        // insert or delete point
        //  number in p->balllist
    } else {
        diff = x(setptnum, p->cutdim)
                 - p->cutval;
        if (diff < 0.0) {
            rsetrad(p->loson);
            if (setradius >= -diff)
                rsetrad(p->hison);
        } else {
            rsetrad(p->hison);
            if (setradius >=  diff)
                rsetrad(p->loson);
        }
    }
}
```

Program 6.2. Top-down radius adjustment.

7. Building The Tree

In this section we will apply sampling techniques to algorithms for building K-d trees. The first two applications reduce the time for building the tree, while the third application leads to a tree that is more efficient to search. These techniques apply to general K-d trees, as well as to trees for semidynamic point sets.

The recursive `build` function in Section 2 uses the function `findmaxspread` to find the dimension with largest spread among the points in `perm[l..u]`. When the `build` function is called on N points, `findmaxspread` takes $O(KN)$ time while the rest of `build` requires $O(N)$ time. The function `build` therefore has the recurrence

$$T(N) = 2T(N/2) + O(KN)$$

with the boundary $T(1)=O(K)$; the solution is $T(N) = O(KN \log N)$. We will reduce the time of `build` by finding the dimension of maximum spread in a sample of the point set of size $\sqrt{N}$; the cost of partitioning after the sample is $O(N)$. The recurrence then becomes

$$T(N) = 2T(N/2) + O(K\sqrt{N}) + O(N)$$

with the same boundary $T(1)=O(K)$; the solution is therefore $T(N) = O(KN + N \log N)$.

To test the effect of this change, an experiment built a K-d tree for ten sets of 100,000 points generated uniformly on the unit square. The average time required to build the tree dropped from 32 seconds to 20 seconds (a decrease of 35%, which would be larger for larger values of K). Profiling showed that in the original program, 14.0 seconds were spent in computing spreads, 15.5 seconds were spent in selecting the median, and 2.5 seconds were spent the `build` function itself. Computing the spread on a sample reduced the 14.0 seconds to 2.0 seconds. (The average time for a bottom-up nearest neighbor search in the slightly less balanced tree increased by about 1 percent.)

The bulk of the time of building a tree is now spent in the `select` function (which uses a median-of-three partition, which is a sample of size 3). We could reduce that time by using the selection algorithm of Floyd and Rivest

[1975], which uses a sample of size (roughly) $\sqrt{N}$ to find the median in (roughly) $3N/2$ comparisons. Instead, we will compute the true median of a sample of size $\sqrt{N}$ elements, and partition around that value (which approximates the median of the set) in just N comparisons. In building ten trees for uniform point sets of size 100,000, this reduces the time to build a tree from 20 seconds to 12 seconds. (The experiments also showed, however, that the average search time increased by about 2 percent; since many applications spend much more time searching than building the tree, the default behavior of the program is not to use this kind of sampling.)

So far we have concentrated on faster ways of building the same kind of K-d tree; we will now consider building a better K-d tree. The trees are adaptive in choosing the dimension in which to cut the point set; they always, however, choose to cut near the median. The "spokes" distribution of the Section 5 showed why median cuts can be a poor idea. In that distribution, $N/2$ points are uniform between $(1/2,0)$ and $(1/2,1)$ and $N/2$ points are uniform between $(0,1/2)$ and $(1,1/2)$. In other words, the points are distributed uniformly over a plus sign "+" centered in the unit square. If the root cut of the K-d tree is a vertical line (that is, $cutdim = 0$ and $cutval = 1/2$), then the $N/2$ nearest neighbor searches for points along that line must all proceed to the root; a similar situation occurs for a horizontal cut line. Each of the $N/2$ searches has logarithmic cost, and the average search cost for the point set grows as log N.

To find a good cut plane we will use a technique inspired by Bentley and Shamos [1976] (also described by Preparata and Shamos [1985, Section 5.4]). Their algorithm finds all pairs of points within δ of one another in a sparse point set, using the following definition and theorem.

> **Definition 7.1**: A point set is (δ, C)-sparse if no rectilinearly oriented hypercube of edge length 2δ contains more than C points.
>
> **Theorem 7.2**: [Bentley and Shamos] Given a (δ, C)-sparse set of N points in K-space, there exists a cut plane P perpendicular to a coordinate axis with these properties:
>
> 1. At least $N/(4K)$ points are on each side of P.
> 2. There are at most $KCN^{1-1/K}$ points within distance δ of P.

Bentley and Shamos's divide-and-conquer algorithm uses the theorem to find a cut plane, recursively finds near neighbor pairs that are on the same side of the plane, and then finds the relatively few pairs that are on opposite sides of the plane. Its run time is $O(N \log N)$, for fixed K.

We will now use their technique to build K-d trees that are more robust to pathological inputs. Papadimitriou and Bentley [1980] show that some point sets do not have good nearest neighbor cut planes, so this technique does not have worst-case guarantees; it does, however, perform well on the pathological point sets described by Bentley and Shamos [1976] and Zolnowsky [1978]. We will choose a cut plane by processing a sample, S, of M points in the original set of size N. We first recursively build a K-d tree for S, then find for each point in S its nearest neighbor in S; this defines a collection of M balls in space. We then consider each of the K dimensions, and try to choose a cut plane that intersects relatively few balls. Each ball is represented in the projection by three scalars: its center and two endpoints (plus and minus the radius). After sorting the $3M$ values, we scan through them, keeping track of the number of balls currently intersected; this requires $O(KM \log M)$ time.†

To test variable-cut planes, an experiment generated ten uniform point sets and ten spoke point sets, each of size 10,000. This table reports the average CPU times (in seconds) for building and searching the trees.

Input	Median Cuts			Variable Cuts		
	Build	Search	Total	Build	Search	Total
Uniform	1.01	2.97	3.98	1.68	3.03	4.71
Spokes	1.00	6.49	7.49	1.61	1.75	3.36

Variable-cut trees require about 65% more CPU time to build than median cut planes. For uniform distributions, the search time in a variable-cut tree is, as expected, the same as in a median tree. For the spokes distribution, though, the search time drops from being very bad for median trees to being very good for variable-cut trees.

8. Conclusions

The algorithms in this paper are easy to implement and are efficient in practice. Although the descriptions in the body of the paper have been for the planar case ($K=2$), Appendix 2 describes experiments for higher dimensions. The techniques underlying the algorithms appear to be of general interest.

Semidynamic Data Structures. Point sets that support deletions and undeletions (but not insertions) are general enough to be useful in a broad class of applications but specialized enough to allow efficient operations.

Bottom-Up Operations. Both searches and maintenance operations (deletion and undeletion) can be performed bottom-up by augmenting the tree data structure with a `bucketptr` vector and `father` pointers.

Sampling. Section 7 describes three applications of sampling in building trees. The run time was reduced by finding the spread of a sample and by partitioning around the median of a sample. The tree was made more robust by finding a cut plane that effectively divides a sample. All the techniques are applicable to general trees, not just trees that represent semidynamic sets. The technique in Section 5 of storing the bounds array only every few levels in the tree can be viewed as choosing a sample of levels.

† A few implementation details about variable-cut planes. The planes are found only if the subset is sufficiently large ($N \geq 1000$); the sample is of size $M = 10N^{1/4}$. The cut plane is chosen so that at least $N/(4K)$ points are on each side of it. We choose to cut at the point with minimum score, where the score of a point is defined to be the number of balls intersected plus the number of points between that point and the median times the penalty factor of $N^{-1/K}$.

Caching. Appendix 1 describes how caches reduce the run time of two operations that are associated with *K*-d trees: building the trees and computing distances.

Acknowledgments

I am grateful for the helpful comments of Ken Clarkson, David Johnson, Brian Kernighan, Colin Mallows, Doug McIlroy, Sally McKee, Ravi Sethi, and Chris Van Wyk.

References

Aho, A. V., J. E. Hopcroft and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

Bentley, J. L. [1975]. "Multidimensional binary search trees used for associative searching", *Communications of the ACM 18*, 9, September 1975, pp. 509-517.

Bentley, J. L. [1990a]. "Experiments on traveling salesman heuristics", *Proceedings First Symposium on Discrete Algorithms*, pp. 91-99.

Bentley, J. L. [1990b]. "Fast algorithms for geometric traveling salesman problems", in preparation.

Bentley, J. L. and M. I. Shamos [1976]. "Divide and conquer in multidimensional space", *Proceedings Eighth ACM Symposium on the Theory of Computing*, pp. 220-230.

Bentley, J. L., B. W. Weide and A. C. Yao [1980]. "Optimal expected-time algorithms for closest point problems", *ACM Transactions on Mathematical Software 6*, 4, pp. 563-580.

Floyd, R. W. and R. L. Rivest [1975]. "Expected time bounds for selection", *Communications of the ACM 18*, 3, March 1975, pp. 165-172.

Friedman, J. H., J. L. Bentley and R. A. Finkel [1977]. "An algorithm for finding best matches in logarithmic expected time", *ACM Transactions on Mathematical Software 3*, 3, pp. 209-226.

Lee, D. T. and C. K. Wong [1977]. "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees", *Acta Informatica 9*, pp. 23-27.

Mehlhorn, K. [1984]. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag.

Papadimitriou, C. H. and J. L. Bentley [1980]. "A worst-case analysis of nearest neighbor searching by projection", in Carnegie-Mellon University Computer Science Report CMU-CS-80-109.

Preparata, F. P. and M. I. Shamos [1985]. *Computational Geometry*, Springer-Verlag.

Sedgewick, R. [1988]. *Algorithms*, Second Edition, Addison-Wesley.

Sproull, R. L. [1988]. "Refinements to nearest-neighbor searching in k-d trees", Sutherland, Sproull and Associates SSAPP #184, July 1988.

Stroustrup, B. [1986]. *The C++ Programming Language*, Addison-Wesley.

Zolnowsky, J. E. [1978]. Topics in Computational Geometry, Ph. D. Thesis, Stanford University, May 1978, STAN-CS-78-659, 53 pp.

Appendix 1: Caching

This appendix describes two kinds of caching that are useful in many programs that operate on *K*-d trees. For concreteness, we will assume that a point set is represented by a C++ class with (at least) these operations:

```
class pointset {
public:
            pointset(char *filename);
            pointset(pointset *ptset,
                     int subsetn,
                     int *subsetvec);
    float   x(int i, int j);
    float   dist(int i, int j);
    float   distsqrd(int i, int j);
    kdtree  *grabtree();
    void    releasetree();
};
```

A `pointset` is created either by reading from a file (specified by a string) or by taking a subset of an existing `pointset`. Note that once a `pointset` is created, there is no way to change any coordinate of any point. The function `x(i,j)` accesses the *j*-th coordinate of point *i*. The function `dist(i,j)` returns the distance from point *i* to point *j*, and `distsqrd` returns the square of the distance.

Straightforward implementations of many algorithms rebuild the same *K*-d tree several times for a given point set. For instance, a traveling salesman program might use one tree to compute a starting tour and then use another tree to improve the tour by 2-opting. A `pointset` associates a `kdtree` with each point set. The tree is seized by the `grabtree` function (which ensures that all points are undeleted) and is returned by `releasetree`. This cached tree reduces the time to rebuild a tree from $O(KN + N \log N)$ to $O(N/B)$, where B is the bucket size.

A bottleneck in many applications is computing a distance function. Because there are $N(N-1)/2$ interpoint distances, it is usually impractical to store them all. Let M be the smallest power of 2 greater than or equal to N; note that $M < 2N$. This cached distance function uses M integers and M floating point numbers:

```
int     cachesig[m];
float   cacheval[m];

float dist(int i, int j)
{   if (i > j) swap(&i, &j);
    ind = i ^ j;
    if (cachesig[ind] != i) {
        cachesig[ind] = i;
        cacheval[ind] = computedist(i, j);
    }
    return cacheval[ind];
}
```

The values i and j are first normalized so that $i \le j$, and then the value of i is exclusively or-ed with j (represented in C++

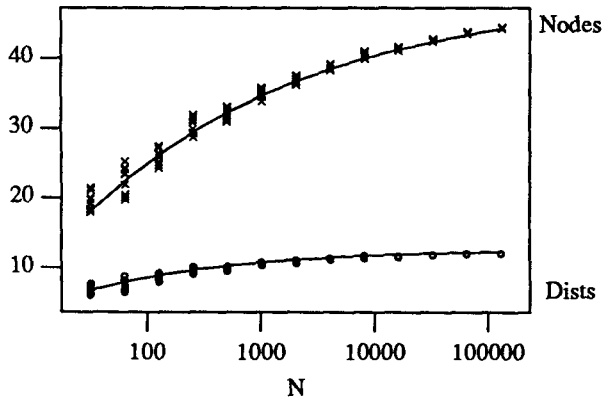

Figure A2.1. All nearest neighbors, bottom-up search, $K = 3$.

by the ^ operator) to produce the hash index *ind*. The integer *cachesig*[*ind*] is a signature of the (i,j) pair represented in *cacheval*[*ind*] (because $ind = i \hat{} j$, we can recover j by $ind \hat{} i$). The cache algorithm therefore checks the signature entry *cachesig*[*ind*] and, if necessary, updates *cachesig*[*ind*] and recomputes the value entry *cacheval*[*ind*]. It finally returns the correct value.

This cache will prove effective if the underlying algorithm displays a great deal of locality in its use of the distance function. In 2-opting large traveling salesman tours, for instance, the cache hit rate is typically near 75%. Accessing the cache takes just one third the time of computing the distance from scratch, so the total time spent in computing distances is halved.

Appendix 2: Experiments in Higher Dimensions

All experiments in the body of the paper were for $K=2$; the algorithms have similar performance for higher dimensions. In this section we briefly report on their performance when $K=3$.

Figure A2.1 summarizes the first set of experiments for $K=3$. For each N a power of two from $2^5=32$ to $2^{17}=131072$, we generate 10 sets of N points at random on the unit cube, build a 3-d tree with bucket size of one, and then search for all nearest neighbors in the point set. The circles show the average number of distance calculations and the crosses show the average number of internal nodes visited. Weighted non-linear least squares fits showed that the number of nodes visited is approximately $49.14 - 66.84N^{-0.22}$ and the number of distance calculations is approximately $12.63 - 18.66N^{-0.33}$; both functions are plotted on the graph. Notice that the functions approach their asymptotic values much more slowly for $K=3$ than for $K=2$; the growth is is slower yet for larger K, which is why Figure A2.1 stops at $K=3$.

We will turn next to an experiment on the highly nonuniform distributions described in this table, which are the multidimensional generalizations of the distributions in Section 5:

NAME	DESCRIPTION
uni	uniform within the hypercube $(U[0,1]^K)$
annulus	x_0, x_1 uniform on a circle; $x_2, ..., x_{K-1}$ from $U[0,1]$
arith	$x_0 = 0, 1, 4, 9, 16, ...; x_1 = ... = x_{K-1} = 0$
ball	uniform inside a sphere
clusnorm	*Normal*(0.05) at ten points on $U[0,1]^K$
cubediam	$x_0 = x_1 = ... = x_{K-1} = U[0,1]$
cubeedge	$x_0 = U[0,1]; x_1 = ... = x_{K-1} = 0$
corners	$U[0,1]^2$ at (0,0), (2,0), (0,2), (2,2), $x_2, ..., x_{K-1}$ from $U[0,1]$
grid	N points from a grid hypercube with $1.3N$ points
normal	each dimension from *Normal*(1)
spokes	N/K at $(U[0,1], 1/2, ..., 1/2)$, N/K at $(1/2, U[0,1], ..., 1/2), ...$

Figure A2.2 shows the efficiency of bottom-up searching in computing nearest neighbor tours on point sets drawn from these distributions for $K=3$. Five sets of size $N=10,000$ were drawn from each distribution, and the nearest neighbor tour was computed for each. For all experiments, the bucket cutoff was set to 5 and variable-cut planes were employed. The graph reports the average number of distance calculations (circles) and the average number of nodes visited (crosses). The graph is similar to Figure 5.7, except the nasty behavior of the spokes distribution has been tamed (by variable-cut planes) and the grid distribution suffers more in higher dimensions. Once again, the performance of the algorithms is not substantially slower on highly nonuniform data.

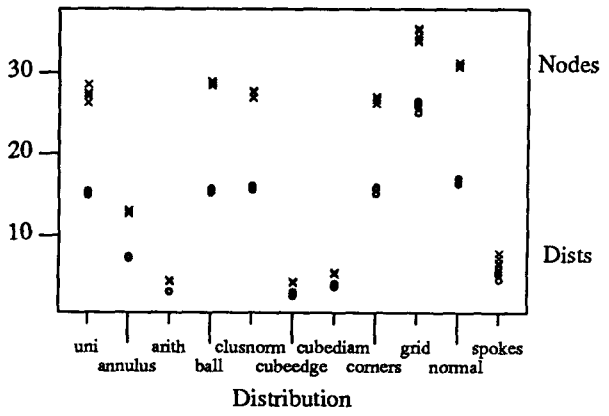

Figure A2.2. Nearest neighbor TSP tour, $N = 10,000$, $K = 3$.