Nearest Neighbor Searching in Metric Spaces: Experimental Results for sb(S)

Kenneth L. Clarkson
Bell Laboratories, Lucent Technologies
Murray Hill, New Jersey 07974
clarkson@research.bell-labs.com
http://cm.bell-labs.com/who/clarkson/

December 5, 2002

Abstract

Given a set S of n *sites* (points), and a distance measure d, the *nearest neighbor searching* problem is to build a data structure so that given a query point q, the site nearest to q can be found quickly. This paper gives a data structure for this problem; the data structure is built using the distance function as a "black box". The structure is able to speed up nearest neighbor searching in a variety of settings, for example: points in low-dimensional or structured Euclidean space, strings under Hamming and edit distance, and bit vector data from an OCR application. The data structures are observed to need linear space, with a modest constant factor. The preprocessing time needed per site is observed to match the query time. The data structure can be viewed as an application of a "kd-tree" approach in the metric space setting, using Voronoi regions of a subset in place of axis-aligned boxes.

1 Introduction

Given a set S of n *sites* (points), and a distance measure d, the *nearest neighbor searching* problem is to build a data structure so that given a query point q, the site nearest to q can be found quickly.

This paper gives a data structure, denoted sb(S), for this problem, in a setting where the sites and queries are in a metric space. Algorithms in a such a setting have been considered for some time[FS82]; a recent survey of work in this area is given by Chavez *et al.*[CNBYM01]. This paper is a continuation of previous theoretical work[Cla97], but the emphasis here is on empirical results. A recent paper gives an data structure akin to that in[Cla97], with provable results in the setting of *growth-restricted* metric spaces.[KR02] Such a property is roughly comparable to the assumption of a uniform distribution of data points in $\mathbb{R}^d$.

The nearest neighbor problem has a vast literature, with threads of work in many different fields. Here we are seeking a general, practical approach, where the goal is to produce an analog, for nearest neighbor searching, of the role played by `qsort` as a sorting routine: a code that works pretty well, most of the time, so that more specialized procedures are not often needed.

The procedure given here satisfies some basic requirements for such a goal. First, it does no harm: that is, its space requirements are modest, and its worst-case behavior is to take about the same amount of time as brute-force search. Its preprocessing time seems to behave roughly like its search time, so that in settings where the data structure doesn't help, that can be discovered at preprocessing time. Finally, it

is general, both with respect to its application to general metric spaces, and with respect to the searching tasks that can be done: it has been adapted to fixed-radius, k-nearest-neighbor, and inverse nearest-neighbor searching, as well as nearest. Its preprocessing solves the all-nearest-neighbor problem for the sites, and in one version, produces the results of a well-known clustering algorithm[Gon85].

The closest comparable software is due to Arya and Mount[AM]; their code provides a similar capability for point sets in $\mathbb{R}^d$, using a variant of kd-trees. It is almost certainly faster in that setting than the code given here, but plainly is more limited. Moreover, kd-trees are clearly inappropriate for some of the high-dimensional pointsets tested here, without at least some moderately expensive preprocessing to reduce dimensionality, such as principal components analysis. A notable, and novel, result here is shown in Figure 12. This illustrates modest, but definite, speedups for some high-dimensional bitvector datasets arising in practice, and for strings under hamming and edit distance.

2 The Data Structure

The sb(S) data structure is somewhat analogous to kd-trees, and can be motivated by considering a simple version of them: suppose we have a collection of orthogonal rectangles whose union contains S, and for each rectangle $\mathcal{R}$, we have recorded the list $\mathcal{R} \cap S$. (See Figure 1.) Given a query point q, we could look in each $\mathcal{R} \cap S$, looking for a nearest site. Suppose the nearest site found so far is p^*, so that the current "nearest neighbor ball" centered at q has radius equal to the distance of q to p^*. Consider rectangle $\mathcal{R}$. It's easy to check if the current nearest neighbor ball for q meets $\mathcal{R}$; if not, the nearest site to q does not lie in $\mathcal{R}$. Otherwise, we must examine the sites in $\mathcal{R}$. The basic idea is that by doing a constant-time test, we can exclude a number of sites from consideration.

Algorithms for metric spaces, and using such a "kd-tree" approach, have been proposed, typically by partitioning using spheres.[Bri95, Uhl91, Yia93] Here we use, instead, Voronoi regions for a subset $R \subset S$. That is, the sites of $S \setminus R$ are grouped according to their nearest neighbor in R. For $p \in R$, call such a set the *S-Voronoi set* $V_p \cap S$ of p. For a given query q, distance $d(q, R)$, and site $p \in R$, we can do some simple checks that can sometimes rule out every site in the S-Voronoi set of p from being a nearest neighbor to q in S. To describe one such constant-time check, it is time to give a few definitions:

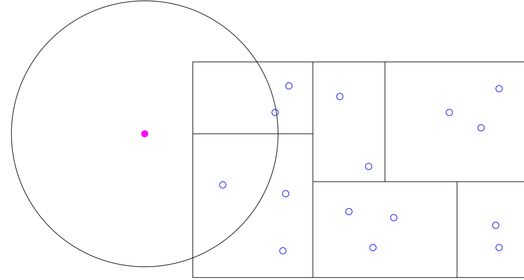

Figure 1: Excluding sites quickly with rectangles.

Definition: Nearest Neighbors. For metric space (V, d), set $R \subset V$, point $q \in V$, the *nearest neighbor distance* of q with respect to R is

$$d(q, R) \equiv \min_{p \in R \setminus \{q\}} \{d(q, p)\}.$$

A point $p \in R$ realizing that distance is a *nearest neighbor* of q in R.

Definition: γ-Nearest Neighbors. Say that $p \in V$ is a *γ-nearest neighbor* of $q \in V$ with respect to R if $d(q, p) \leq \gamma d(q, R)$.

We have the following simple condition:

Claim 2.1 *If the S-Voronoi set of p meets the nearest neighbor ball $N(q, R)$ of q with respect to R, then p is a 3-nearest neighbor of q with respect to R.*

Proof: (See Figure 2.) Suppose site a is both in the S-Voronoi set of p, and in $N(q, R)$; that is, there is a site $a \in V_p \cap S$, and $d(a, q) < d(p', q)$,

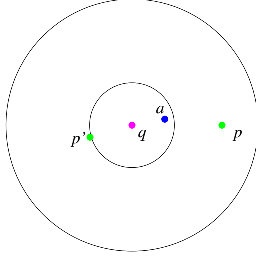

Figure 2: The Voronoi set of p etc.

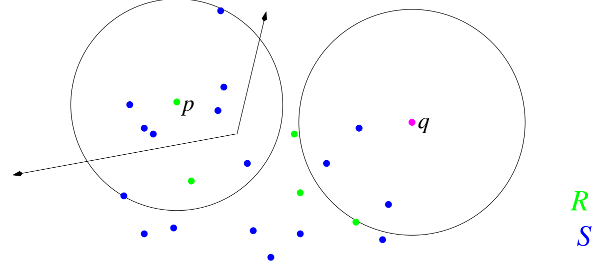

Figure 3: M_p bound

where p' is nearest to q in R. Then

$$d(a,p) < d(a,p'),$$

since $q \in V_p$, and

$$d(a,p') \leq d(a,q) + d(q,p') \leq 2d(q,p'),$$

and so $d(q,p) \leq d(q,a) + d(a,p) \leq 3d(q,p')$. ■

With the following definition, a slightly different bound is easily shown.

Definition: M_p**.** Let M_p denote the farthest distance to $p \in R$ from a site in its S-Voronoi set.

Referring to Figure 3, we have the following constant-time test.

Claim 2.2 *Suppose that the S-Voronoi set of p meets $N(q,R)$, so there is a site $a \in V_p \cap S$, and $d(a,q) < d(p',q)$, where p' is nearest to q in R. then*

$$d(q,p) \leq d(q,a) + d(a,p) \leq d(q,p') + M_p.$$

Proof: Trivially follows using the triangle inequality, the assumptions, and the definition of M_p. ■

Construction of sb(S). The data structure sb(S) is based on an incremental construction: a permutation $(p_1 \ldots p_n)$ of S yields subsets $R_i \equiv \{p_1 \ldots p_i\}$ for $i = 1 \ldots n$; we can think of these subsets as being built by adding sites one by one. The search algorithm for sb(S) will maintain the nearest neighbor of a query point q in R_i, and a set of *pending* sites $p \in R_i$, such that the S-Voronoi set of p with respect to R_i may contain a nearer site to q. The invariant maintained is that the nearest site to q is either a pending site, or in the S-Voronoi set with respect to R_m of a pending site, as m passes from 1 to n.

Of course, we don't want to consider each m from 1 to n; it is only necessary to consider those p_i that *touch* a pending site p_j, where touching means that p_i changes $V_{p_j} \cap S$ when p_i is added. That is, p_i touches p_j if there is some p_k so that: p_j is closest to p_k in R_{i-1}, but p_i is closest to p_k in R_i. For each p_j, such touching sites are put into a list $\mathcal{E}_j$ in preprocessing, in the same order as in the permutation. The list $\mathcal{E}_j$ also has, for each touching site p_i, the maximum distance of p_j to a site in its Voronoi set with respect to R_i. (That is, the value of M_{p_j} with respect to R_i.) The latter distance value can be used to check the pending status of p_j when the touching site p_i is processed.

The lists $\mathcal{E}_j$, together with the associated distance data, comprise the sb(S) data structure. Before describing the preprocessing algorithm for building sb(S), we describe how to use it.

Searching. One method of using sb(S) for nearest neighbor searching is shown in Figure 4. We will explain the procedure, but not every detail of the code.

The value `alpha` is between 0 and 1, and allows a tradeoff between search speed and accuracy. If `alpha` is set to 1, then searching always returns the correct answer; for smaller `alpha`, searching is faster but not necessarily correct.

The value `sbt->end_thresh` is discussed in the next section; for now, we can consider its value to be n; that is, the test on line 6 always succeeds.

```
size_t search_sb_d_order(sb* sbt, size_t q,
                         float alpha)
{
 size_t pj=0, touch=0;
 float d=0;
 SD q_nn = {0, FLT_MAX};
 heap *ph = sbt->psh;

 do {
    stat_inc_search_loop;
 1  if (!sbt->seen_for[pj]
 2    && sbt->nn[touch].max.old+alpha*q_nn.key > d)
 3  {
 4    d = sbt->dist(q,pj);
 5    if (d<q_nn.key) {q_nn.key=d; q_nn.point=pj;}
 6    if (sbt->ins_num[pj] < sbt->end_thresh)
 7    {
 8      MARK_SEEN(sbt, pj);
 9      sbt->cur_touching[pj] = sbt->offset[pj];
10      insert_heap(sbt->psh, pj, -d);
11    }
12   }
13   while (sbt->psh->num_entries>0)
14   {
15     size_t pm = topp(sbt->psh).point;
16     d = -topp(sbt->psh).key;
17     touch = (sbt->cur_touching[pm])++;
18     if (touch<sbt->offset[pm+1]
19       && sbt->dDL[touch] + alpha*q_nn.key > d)
20     {
21       pj = sbt->DL[touch];
22       break;
23     }
24     delete_heap(sbt->psh, 0);
25   }
    } while (sbt->psh->num_entries>0);

  reset_aux(sbt);

  return q_nn.point;
}
```

Figure 4: Search procedure for sb(S).

During a search, the set of pending sites is kept in a heap, using the simple data structure attributed to Floyd.[Flo64] The key for p_m in the heap is its distance to the query point q, or rather, the negation of its distance, so that the top of the heap corresponds to the pending site whose distance to q is minimum.

Also maintained for each pending site p_m is a site p_j in the list $\mathcal{E}_m$ for p_m. (This site is indexed by `sbt->cur_touching[pm]`, where the index is into the array `sbt->DL`, between `sbt->offset[pm]` up to `sbt->offset[pm+1]`, which is the representation of $\mathcal{E}_m$. The basic operation of the search algorithm is to determine whether the touching site p_j must be made pending, and for the pending site p_m with the minimum distance to q, to check if p_m is still pending when the next site on $\mathcal{E}_m$ is considered. These two main steps are done in lines 1-12 and 13–25 of the figure. Next we consider these two steps in more detail.

For the first step: when p_j is added, some sites in the S-Voronoi set of p_m are removed, and become part of the S-Voronoi set of p_j. Denoting such sites as S_{mj}, note that if S_{mj} contains a site a closer to q than some given upper bound `q_nn.key`, then

$$\begin{aligned} d(p_m, q) &\leq d(p_m, a) + d(a, q) \\ &< \max_{b \in S_{mj}} d(p_m, b) + \texttt{q_nn.key}. \end{aligned}$$

This provides the test on line 2, which must be satisfied if p_j touching p_m is to become pending. Here $\max_{b \in S_{mj}} d(p_m, b)$ corresponds to `sbt->nn[touch_index].max.old`. When $\alpha = 1$, the test on line 2 must hold if p_j should be made pending. When $\alpha < 1$, the test is more strict, fewer sites are made pending, and the returned answer is only approximately the nearest neighbor. (The test on line 1 checks that p_j was not already seen for q, that is, it was not already made pending.)

For the second main step (lines 13–25), the top entry of the heap `sbt->psh` is examined, corresponding to the pending site p_m closest to q, and the next site p_j touching p_m is considered, corresponding to `sbt->DL[touch_index]`. When p_j

is added, the S-Voronoi set of p_m shrinks, and it may be that a simple test can show that the S-Voronoi set of p_m can't possibly contain a site nearer to q than the current upper bound on the nearest neighbor distance. Such a test is given in line 19, where `sbt->dDL[touch_index]` is the pre-computed farthest distance of p_m to the sites in its S-Voronoi set, after p_j is added.

Preprocessing. The preprocessing for sb(S) involves considering each p_i, for $i = 1 \ldots n$, and finding those sites not in R_i for which p_i is nearest in R_i. The former nearest neighbors of such sites are the ones that p_i touches. The preprocessing also finds, for such a touched site p_j, the distance of the farthest site to p_j in its current Voronoi region. The preprocessing is aided by keeping, for each p_j, its S-Voronoi set in a heap, with key for a site equal to its distance to p_j, and with the farthest site at the top of the heap.

When adding p_i, its processing can be accelerated by using both the data structure, as constructed up to that point, and the auxiliary heaps for each p_j. The idea is to use these structures to speed up the "inverse nearest neighbor" searching needed for p_i. That is, rather than examine every p_k for $k > i$, we use a searching procedure, akin to that for answering nearest neighbor queries, to find those p_k with p_i nearest in R_i.

The inverse searching procedure maintains a set of pending sites; the invariant holds that any site p_k with p_i nearest in R_i has a pending site nearest in R_m, where p_m is the current site being considered for pending status. Only those sites that touch a pending site need be considered for pending status. Moreover, the following allows some sites to be discarded as pending.

Lemma 2.3 *If $p \in R$ is at most M_p from any site in $V_p \cap S$, then any site p' closer to some $a \in V_p \cap S$ than p has $d(p', p) \leq 2M_p$.*

Proof: For any such a, $d(a,p) \leq M_p$, and so

$$d(p,p') \leq d(p,a) + d(a,p') \leq 2p(p,a) \leq M_p.$$

■

That is, any pending p_m must have $d(p_i, p_m) \leq 2M_{p_m}$. The inverse searching procedure is thus: for each pending site p_j, walk down its $\mathcal{E}_j$ list, considering each site on the list for pending status, until the corresponding M_{p_j} value is smaller than $d(p_j, p_i)/2$. Note that since the check that $M_{p_j} \leq d(p_j, p_i)/2$ does not depend on any other pending site, or change of status for p_i, so the pending sites need not be kept in a heap, and a list $\mathcal{E}_j$ can be traversed independently of others.

Some pending sites p_j will "survive," that is, will have their $\mathcal{E}_j$ lists traversed to the end with their pending status preserved. Such a site p may have some members of $V_p \cap S$ with p_i nearest. To find such members, we need not examine all of $V_p \cap S$; only those members a with $d(p_i, p_j) \leq 2d(a, p_j)$ can have p_i nearest, as in the lemma above. To find such a, we can recursively examine the heap of sites in $V_p \cap S$; this procedure need not examine the children of a node in the heap if the site corresponding to the node has distance to p_j smaller than $d(p_i, p_j)$.

2.1 Variations on sb(S)

One improvement to the basic data structure is to run the construction only up to some $R_{\epsilon n}$, and then keeping in the data structure the S-Voronoi sets of the sites of $R_{\epsilon n}$. The search procedure would then search the S-Voronoi sets of sites that are still pending at the end of the basic search procedure. The value of `sbt->end_thresh` in Figure 4 equals ϵn.

Another variation mentioned above for the search procedure involves the use of the parameter α, which trades accuracy for speed.

One significant improvement involves a change in the order of insertion of sites into R_i. While the motivation of these algorithms is from randomization, it seems to be more efficient here to construct a packing: that is, the next site to add to R_i is the one whose minimum distance to sites in R_i is maximum. Such a site will be the one realizing the bound M_p; this value and site are already maintained by the algorithm, so the change to finding the maximum M_p is not difficult.

Although the results proven about this variation are limited so far, it seems to be about 30% faster for observed problems than the randomized version. Moreover, this packing construction is useful in itself: such packings have been proposed as a method of cluster analysis.[Gon85]

While the search algorithm above considers pending sites according to their distance from the query point, a more natural scheme considers sites in their insertion order. This scheme is more amenable to analysis, but appears to be slower.

2.2 Performance

This section gives some data for tests for $\mathrm{sb}(S)$, using the packing variation and search procedure given above.

2.2.1 The datasets and conditions

The data structure was tested on Euclidean data in a cube, in various dimensions. Following the work of Arya and Mount, it was tested for uniform, normal, clustered normal, correlated normal, Laplacian, and correlated Laplacian distributions.[AM]

Strings from a dictionary wordlist were tested, under hamming distance and a version of string edit distance.

Two sets of 256-dimensional data were also considered, from an OCR application, of size 1606 and 6791.

We also considered three bit-vector datasets, taken from an optical character recognition application. They are two sets of about 9000 sites in 81 dimensions, and 14520 sites in 2304 dimensions.

In the figures below, labels are generally:

- the dimension, for Euclidean data
- 's' for strings under edit distance
- 'h' for strings under Hamming distance
- 't' for the 256-dimensional data
- 'o' for the bit-vector data

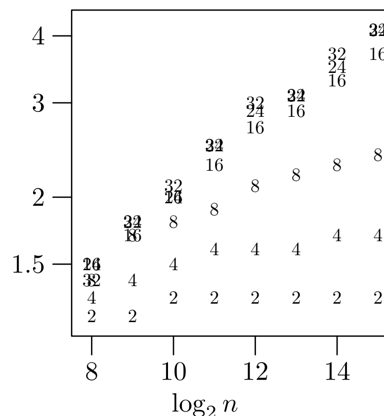

Figure 5: Storage for synthetic data; label is dimension.

In these results, the `end_thresh` parameter described above is set at $n/10$. All query results are the average over 500 queries.

2.2.2 Storage and Preprocessing

First, it's convenient to quickly verify that the storage requirements are modest, and in particular do not rise sharply with dimension. In Figure 5, the number of entries in $\mathrm{sb}(S)$ per site are recorded, as a function of the log of the number of sites. For each dimension and number of sites, the storage shown is the maximum over all distributions. A similar graph is shown for the other datasets in Figure 6. For each entry, a site number and a small number of distance values are needed.

The number of distance evaluations per site seems to behave similarly to the query time; Figure 7 shows the ratio of such per-site preprocessing to the number of distance evaluations per query, over all datasets. These results are, of course, dependent on the `end_thresh` parameter; when `end_thresh` is equal to n, the preprocessing time rises, while the query time falls slightly.

2.2.3 Query time

Running time vs. distance evaluations. In the results below, the "query time" is assumed

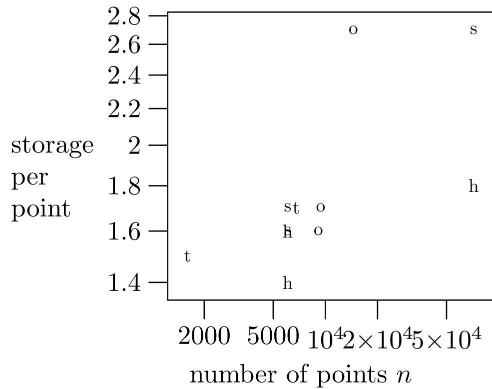

Figure 6: Storage for all non-synthetic datasets

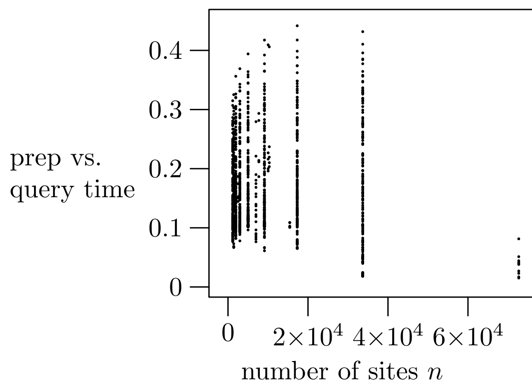

Figure 7: Ratio of preprocessing time to query distance evaluations, all datasets

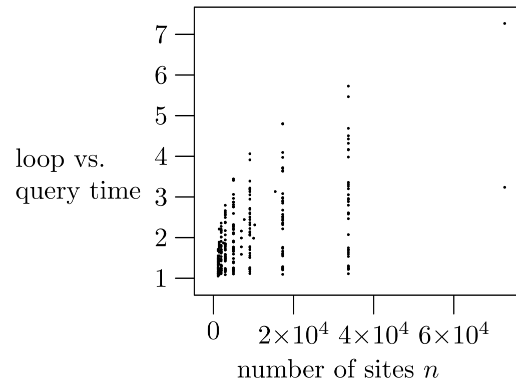

Figure 8: Ratio of loop iterations to query distance evaluations

to be synonymous with the number of distance evaluations done for a query. This is justifiable in cases where distance evaluations are very expensive, but it is also useful to check that the bookkeeping in the query procedure is not too far out of range of the distance evaluations. Consideration of the query procedure of Figure 4 shows that the work done, except for heap operations, is proportional to the number of iterations of the main loop. The work done for heap operations is proportional to the maximum heap size, M_h, times the logarithm of M_h. Figure 8 shows the ratio of the number of iterations of that main loop to the number of distance evaluations done during a query, while Figure 9 shows the ratio of $M_h \log M_h$ to the distance evaluations. These figures use the results for all datasets. The results give some confidence that distance evaluations are a good indicator of the total work done.

Figure 10 shows the number of distance evaluations for exact searching of two-dimensional uniform Euclidean data, using several variations of the search method. The most notable of these methods use labels of “d”, for the nearest-distance method described above, and “i”, for a search method where the sites that are touching currently pending sites are considered in insertion (permutation) order. The other labels are “s”, for a method that uses a stack of pending sites in place of a heap; “h”, for a method that

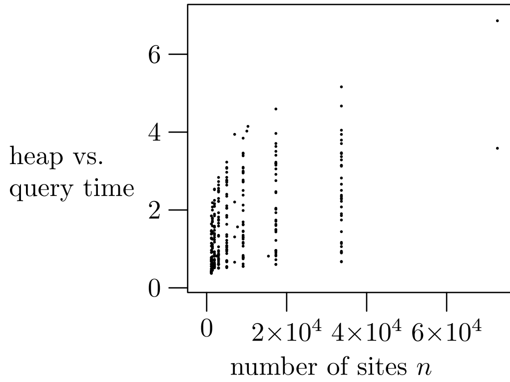

Figure 9: Ratio of $M_h \log_2 M_h$ to query distance evaluations

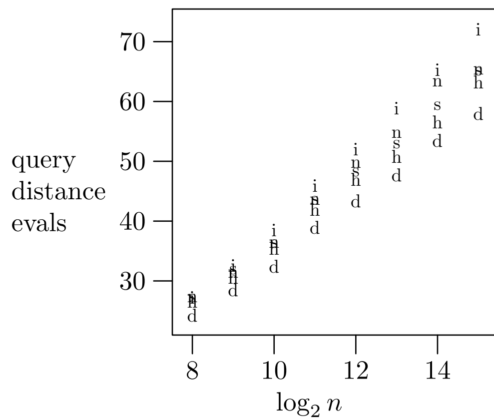

Figure 10: Query distance evaluations, $d = 2$, uniform, different search methods

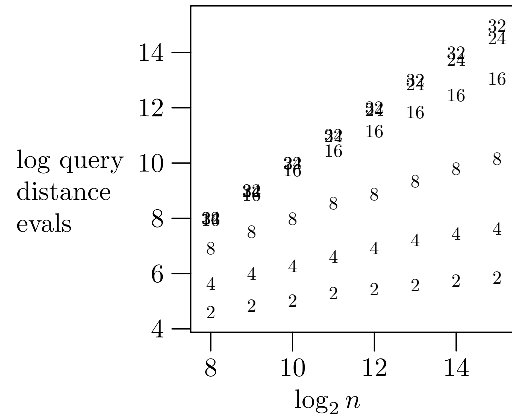

Figure 11: Query distance evaluations, uniform

reduces but does not eliminate the use of the heap; and "n", for a recursive method that does not use a heap. The main observation here is that the "i" method is uniformly worst, and "d" is uniformly best. This seems to hold for other problem instances, as well.

The logarithmic dependence of the query time on the number of sites is apparent here also.

Figure 11 shows search times for uniform Euclidean data, in different dimensions, and Figure 12 shows the speedup (number of sites divided by distance evaluations) for the various non-synthetic datasets tested. For the latter, note that although the speedups are generally modest, there is always at least some speedup in distance evaluations, even a quite substantial one in some cases.

Approximation. Figures 13 through 16 show results are given for variation of the parameter α. Note that even with $\alpha = 0.1$, where the speedup over exact queries is considerable, the number of exact (truly nearest) answers is between twenty to forty percent, and even when not exact, the error ratio is typically fifteen to twenty-five percent.

2.2.4 Other Euclidean distributions

In Figure 17 are shown the query times for a variety of probability distributions, for $d = 16$.

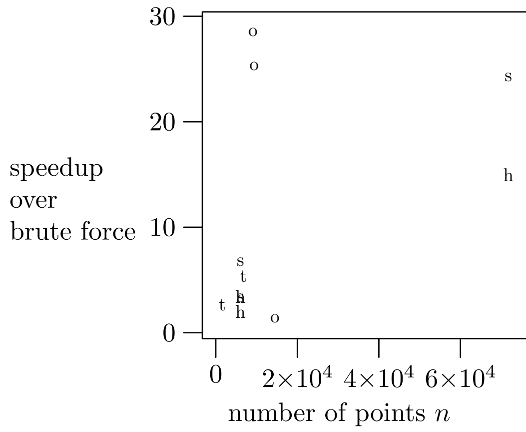

Figure 12: Query speedup over brute force, non-synthetic data

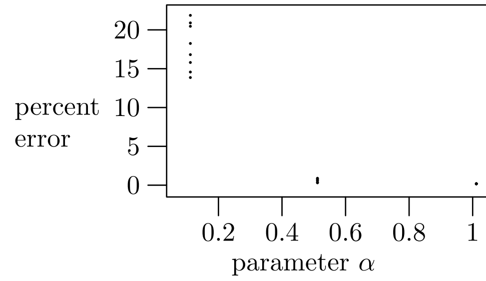

Figure 15: Error ratio (for queries with non-zero error), vs. α

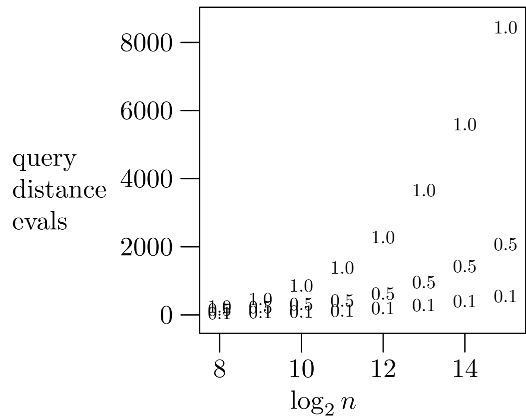

Figure 13: Query distance evalutions, vs. approximation parameter α

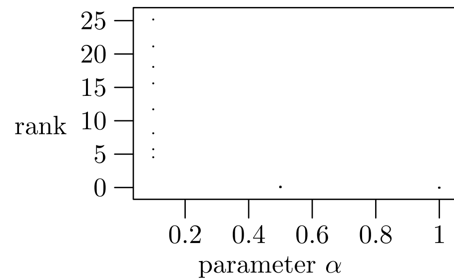

Figure 16: Rank (in distance) vs. α

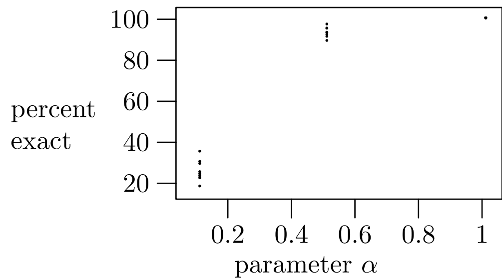

Figure 14: Proportion of queries with no error, vs. α

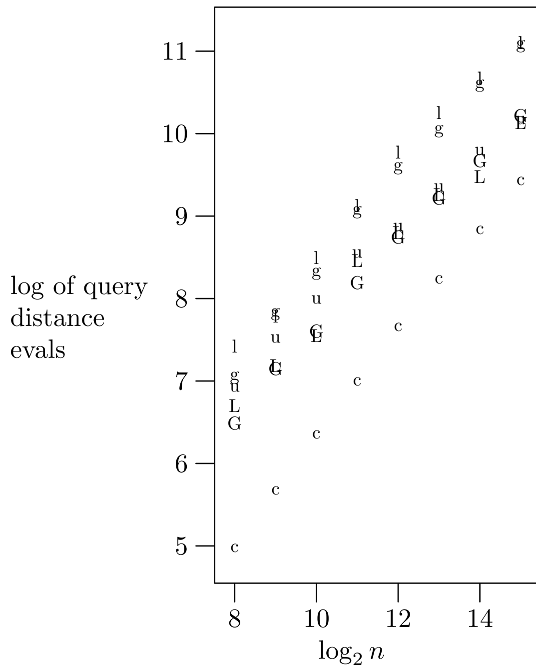

Figure 17: Query times, $d = 16$, various distributions

Here the labels are:

- 'c' for clustered Gaussian data
- 'l' for Laplacian data
- 'L' for correlated Laplacian data
- 'g' for Gaussian data
- 'G' for correlated Gaussian data

3 Concluding remarks

We've described a data structure for nearest neighbor searching, and showed useful or at least interesting behavior for it in a variety of settings.

References

[AM] S. Arya and D. Mount. Ann: Library for approximate nearest neighbor searching. http://www.cs.umd.edu/~mount/ANN.

[Bri95] M. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Int. Conf. on Very Large Data Bases*, 1995.

[Cla97] K. L. Clarkson. Nearest neighbor queries in metric spaces. In *Proc. 29th Symp. Theory Comp.*., 1997.

[CNBYM01] Edgar Chavez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and Jose L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[Flo64] R. Floyd. Algorithm 245: Treesort3. *Comm. ACM*, 7:701, 1964.

[FS82] C.D. Feustel and L. G. Shapiro. The nearest neighbor problem in an abstract metric space. *Pattern Recognition Letters*, 12 1982.

[Gon85] T. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[KR02] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. of the 34th Annual ACM Symp. on Theory of Comp*, pages 741–750, 2002.

[Uhl91] Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inform. Proc. Letters*, 40, 1991.

[Yia93] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 311–321, 1993.