Giving suggestions to Misspelled Words: An Application of Nearest Neighbor Search on Strings (Paper for CSE 203: Prof. Russell Impagliazzo)

Dustin Boswell (dboswell [at] cs ucsd edu)

March 24, 2004

1 Introduction - The Nearest Neighbor Problem

We have a database $X = \{x_1 \dots x_n\}$ where each element x_i is from some universe U. There is also a distance function $d : U \times U \mapsto R^+$ that computes a non-negative distance between any pair of elements. Given a query $q \in U$, the *Nearest-Neighbor Search* task is to find one or more $x_i \in X$ that are *nearest neighbors*. Different variants include:

$$\begin{array}{rcll} NN(q) & = & arg \min_{x \in X} d(q,x) & \text{find one nearest neighbor} \\ kNN(q,k) & = & C \subset X, \quad |C| = k, \quad \forall x \in C, y \notin C : d(q,x) \leq d(q,y) & \text{find } k \text{ nearest neighbors} \\ rNN(q,r) & = & C \subset X, \quad C = \{x : d(q,x) \leq r\} & \text{find neighbors within radius } r \end{array}$$

The brute-force solution to each of these problems is to scan through X and compute $d(q,x)$ for all $x \in X$. Instead, we are interested in techniques that require fewer than n distance computations. To do this, X will have to be *preprocessed* so that at *query-time* fewer computations are needed.

2 The Special Case of Metric Spaces

In the very special case when U is a fixed $D-$dimensional vector space (and $d()$ is usually a Euclidean distance), there are many specialized techniques such as kd-trees. Unfortunately, most of these techniques do not scale well to large dimensions. Moreover, certain data types like strings are not naturally represented by fixed-dimensional vectors. Hence it is often useful to work in a *general metric space*, where the only thing assumed is that the distance function $d()$ is a *metric*:

- $d(x_i, x_j) = 0 \iff x_i = x_j$
- $d(x_i, x_j) = d(x_j, x_i)$ symmetry
- $d(x,z) \leq d(x,y) + d(y,z)$ triangle inequality

In this paper we will review three common data structures for the nearest neighbor problem in general metric spaces, and compare their performance on a dataset of 100,000 words, where the intended application is to give suggestions to a misspelled word.

3 Spelling Correction

We have a "dictionary" (X) of words, and for a given query word q would like to find other words in X whose spelling is similar to q. (While front-end applications only present a handful of respelling suggestions, the "candidate list" is often much larger; this paper does not address the second process of ranking and narrowing the candidate list.) In our experiments we used a set of 100,000 words taken from various corpora and internet sources. While it contains many misspelled tokens itself, we used it (rather than a typical 10,000 word dictionary) to test performance on large data sets. Also, "respelling" is not limited to natural words - these techniques could be used to find file names or url's.

The next step is to define a distance measure between strings. The typical metric used is the *edit distance*: the minimum number of insertions and deletions required to change one string into the other. One variation is to allow "change" edits (which has the effect of charging only 1 edit for an insert & delete to the same location). Another is to allow "transpositions" (swapping two adjacent characters in a string). While not completely obvious, all of these edit distances are in fact metrics.

For our experiments we chose the edit distance to include changes and transpositions as single operations. (This distance is more natural for the task of misspellings. Unfortunately, it brings all strings "closer together" which will make the task more difficult.) We also assume that the number of edits needed to correct a word is proportional to the length of that string, and that at most a third of the characters need changing. And so we define our nearest-neighbor search problem as:

$$rNN(q) = \{ \text{ Find all strings in } X \text{ within edit distance of } r = \left\lceil \frac{|q|}{3} \right\rceil \}$$

4 Nearest Neighbor Search Techniques

We will review three common data structures used for nearest neighbor search in metric spaces. All of them make heavy use of the triangle inequality and represent the dataset as a tree, where during a search, branches can be completely eliminated based on previous distance calculations.

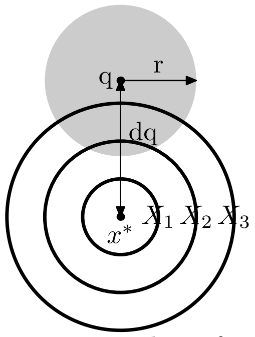
4.1 Burkhard-Keller Tree's (BKT)

This technique was introduced in [BK73]. A random element $x^* \in X$ is chosen as the root of the tree. The other elements in $X - x^*$ are partitioned into sets $X_1, X_2, \ldots X_{dmax}$ where $X_i = \{x : d(x^*, x) = i\}$. (Note: this assumes that the distance function is discrete and bounded, which is true for edit distance assuming the maximum string length is bounded.) For each X_i a branch is created from x^* to the BKT subtree (recursively built) on X_i.

A search $rNN(q, r)$ proceeds as follows:

```
Compute $dq = d(q, x^*)$.
If $dq \le r$ :  output $x^*$

For each $i$ in the range $[dq - r, dq + r]$:
  Recursively search $X_i$
```



[The figure shows a visual example of how search proceeds. We are searching for any points within the shaded region. Since the distance function is discrete, all of the points live on concentric rings, in terms of their distance from the root x^*. In this case, we can exclude X_1 immediately, but X_2 and X_3 must be searched. The exclusion of branches outside the range $[dq - r, dq + r]$ is based on the triangle inequality.]

BKT's work most efficiently when the set $\{d(x^*, x') : x' \in X - x^*\}$ is "spread out" so that many branches can be eliminated during a search. Accordingly, constructing subtrees by selecting roots x^* at random may not be the best strategy. It may improve performance to sample different roots and select the root with the highest estimated variance $\sigma^2(d(x^*, x'))$.

4.2 Vantage Point Trees (VPT)

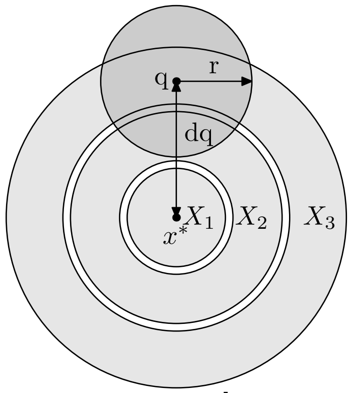
The VPT was presented in [Yia93]. Like BKT's, a random element $x^* \in X$ is selected as the root node. The other elements are partitioned into m equal-sized groups $X_1, \ldots X_m$ where X_1 contains the closest $|X|/m$ items, X_2 contains the next $|X|/m$ closest, etc... and X_m contains the $|X|/m$ furthest elements from x^*. (m is a small fixed parameter. We will say VPT^m when needed. Also, note that in the event of ties or rounding issues, each of the X_i won't necessarily be the same size.) A VPT is recursively built from each branch X_i.

For each of the subtrees X_i of a node x^*, we compute the closest and furthest distance $d(x^*, x' \in X_i)$. That is, for $i = 1 \ldots m$, we compute the pair $dmin_i = \min_{x' \in X_i}(d(x^*, x'))$, $dmax_i = \max_{x' \in X_i}(d(x^*, x'))$. (Naturally, they have the relation $dmin_1 \leq dmax_1 < dmin_2 \leq dmax_2 \ldots$.) These values will help us prune branches during a search.

A search $rNN(q, r)$ proceeds as follows:

```
Compute $dq = d(q, x^*)$.
If $dq \leq r$ :  output $x^*$

For each $i = 1 \ldots m$:
  If $[dmin_i, dmax_i]$ intersects $[dq - r, dq + r]$:
    Recursively search $X_i$
```



[For VPT's each of the X_i is a set of points that fall within a distance *region* $[dmin_i, dmax_i]$ from x^*. VPT's are designed so that the number of points in each region is roughly the same. In the case shown, we can exclude X_1 since it falls entirely outside of q's radius. X_2 and X_3 must be searched.]

Like BKT's, performance may be improved by constructing the tree with roots for which $\sigma^2(d(x^*, x' \in X - x^*))$ is large.

Notice that in the limit as m gets large, the sets X_i will only contain elements that are the same distance from x^*. That is, we can say VPT$^\infty \equiv$ BKT.

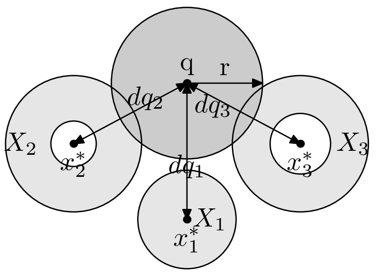
4.3 BiSector Trees (BST)

Bisector Trees were proposed in [KM83]. In this tree, there are m (typically 2) elements $(x_1^* \ldots x_m^*)$ associated with a root node. The rest of the elements are partitioned into sets $X_1 \ldots X_m$ where X_i is the set of elements closest to x_i^*. Similar to VPT's, a pair $dmin_i = \min_{x' \in X_i}(d(x_i^*, x'))$, $dmax_i = \max_{x' \in X_i}(d(x_i^*, x'))$ is computed for each i. Again, a BST is recursively built for each X_i.

Searching is similar to VPT's:

```
For each $i = 1 \ldots m$:
  Compute $dq_i = d(q, x_i^*)$.
  If $dq_i \leq r$ :  output $x_i^*$.

For each $i = 1 \ldots m$:
  If $[dmin_i, dmax_i]$ intersects $[dq_i - r, dq_i + r]$:
    Recursively search $X_i$
```



[For BST's, there are multiple x_i^* at a node, and each dq_i is computed. In the figure shown, $dmin_1 = 0$, so it is a "closed donut". Again, we can exclude X_1 but not X_2 or X_3.]

Ideally the "representatives" $x_1^* \ldots x_m^*$ would each be the center of some tight cluster, so that the "radii" $dmax_i$ are small (this increases the likelihood that an X_i can be pruned from the search). Accordingly, it may reduce query-time if the sets $x_1^* \ldots x_m^*$ are chosen such that $\sum dmax_i * |X_i|$ is low. This favors groupings with small radii (so that the likelihood of recursing to it is small) and small population (so that the cost of recursing there is small). Admittedly, this formula is an ad-hoc proposal of this author. However, optimizing the set of roots is a research problem in its own, so we don't address it further.

4.4 Monotonous Bisector Trees (MBST)

One variation on BST's is to have each x_i^* be a parent in the construction of its subtree X_i. Hence the tree is "monotonous" since when a parent x_i^* is introduced, it is repeated in its child's subtree, and that child's subtree etc... This has the drawback that space is wasted. On the other hand, there are two benefits. First, if distance calculations are properly cached, at a given node we only need to compute $m-1$ new distance calculations (since one of the distances was already computed at the node above). Second, when an x_i^* acts as a root again for the subset X_i, its radius $dmax$ can only get smaller. And as we mentioned before, smaller $dmax_i$ lead to more pruning.

5 Experiments

We gathered a set (X) of 100,000 words from dictionaries, and common words found on the internet (mostly proper nouns).

We then constructed 10 random instances for each of the following trees:

- BKT
- VPT^2, VPT^4, VPT^{16}
- BST^2, BST^4, BST^{16}
- $MBST^2$, $MBST^4$, $MBST^{16}$

We then constructed a set of 50 test queries as follows: select a random word q from the set X. For each letter in q, with probability 1/5 we changed the letter to a random letter of the alphabet. Since our "search radius" is $|q|/3$, the original word q will typically be in the search results - although this isn't necessary, it does make the experiment as realistic as possible.

For each query, we performed the $rNN(q)$ search on all data structures and measured:

- the number of results found near q
- the number of distance calculations performed during the search

The number of results found was the same for all trees (as they should be if they are correct). The number of distance calculations is the performance metric - smaller is better. We also measured the CPU-time for each query - as expected, the edit-distance calculation (an $O(nm)$ time procedure for strings of length n and m) dominated the search time. For brevity, we omit the CPU-time (but it is equal to about 10μs per distance calculation).

Data Structure	# distance evals (avg.)
BKT	42319
VPT^2	47124
VPT^4	43845
VPT^{16}	42626
BST^2	71735
BST^4	68019
BST^{16}	69737
MBST^2	50459
MBST^4	54637
MBST^{16}	63909

Figure 1: The table shows the average number of distance evaluations used for a query on a given tree (averaged over the 10 instances of that tree, and over 50 queries). The average query had 92 results. There were 100,000 words in the data set.

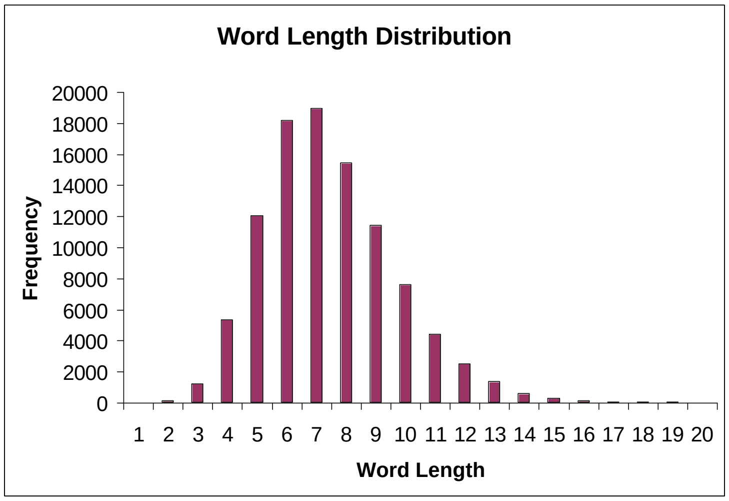
6 Conclusions

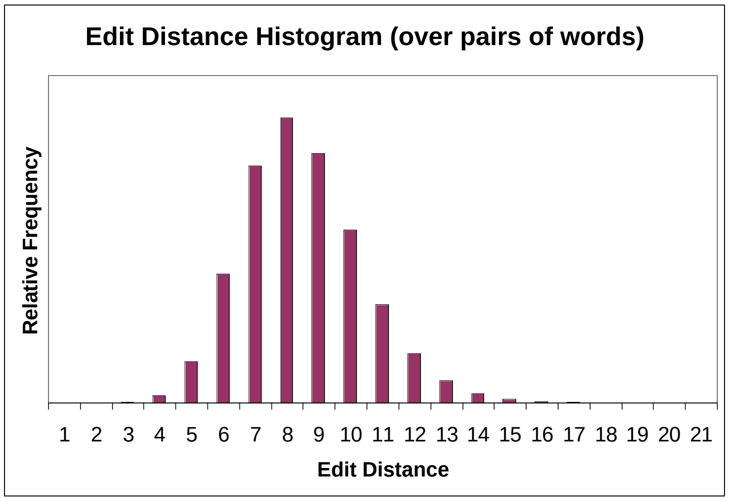
Searching for similar strings is a difficult problem in the general metric setting. The triangle inequality (which is used heavily in all the methods presented here) isn't as useful as one would hope. All the methods shown start by computing the distance from the query to a "root string" (dq). Typically, this value is about $dq = 8$ or so. The search procedure then explores strings that are in the range $[dq - r, dq + r]$. And since r is typically 3, this results in exploring strings in the range $[5, 11]$ from the root string. Unfortunately, this does not eliminate many strings at all. (See figure 2 and 3 for word length and edit distance histograms for our dataset.)

Of the methods shown, there are some clear observations. The variant of BST's called monotonous BST's (MBST's) clearly outperformed regular BST's for this task. However, both of these performed worse than BKT's and VPT's.

It is interesting to note that for VPT's, increasing the "branch-out factor" bettered performance. The BKT performed similarly to the highest-degree VPT - in keeping with our realization that as the degree gets infinite, a VPT becomes a BKT.

However, even the best of the methods performed poorly - performing distance calculations on almost half of the dataset. It is the opinion of this author that methods that rely only on the distance function and the triangle inequality will never perform as well as string-specific methods. (The author has performed other experiments with string-specific methods that outperform these methods by a factor of 20 or more.)





One final mention is that all of the trees were built using random roots during construction. As mentioned previously, all of the trees would benefit from sampling different roots and using ones that best meet some sort of "good root" criteria. Unfortunately, BST's and VPT's are different structures that would require different "good root" criteria, and the results would be highly dependent on how they were implemented and how much extra CPU-time was allowed for constructing these trees. For this reason, we opted to test the pure randomized versions only.

References

[BCMW94] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In M. Crochemore and D. Gusfield, editors, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, Asilomar, CA, 1994. Springer-Verlag, Berlin.

[BK73] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[Bri95] Sergey Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.

[CMBY99] E. Ch'avez, J. Marroqu'in, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces, 1999.

[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.

[HS03] Gisli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[KM83] I. Kalantari and G. Mcdonald. A data structure and an algorithm for recognizing nearest neighbours. *IEEE Transactions on Software Engineering*, SE-9:631–634, September 1983.

[Yia93] Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993.