

Generating Candidate Spelling Corrections

presented by Dustin Boswell

May 2, 2004

Typical Spell Checking Steps

- We have a word q that needs to be corrected.
 - 1) Candidate Generation
 - Come up with “close” possibilities
 - 2) Ranking
 - Sort them by likelihood
- This talk is about fast algorithms for candidate generation.

Problem Statement

- We have a set of words D (our “Dictionary”).
- We are given a query string q
- Find a subset $close(q) \subset D$ such that
 - $close(q)$ **must** contain the “true” spelling correction
 - $close(q)$ is as small as possible

“Close” means small Edit Distance

$EditDist(q, q')$ - the smallest number of
{insertions, deletions, substitutions, transpositions}
needed to transform q into q' .

$$\underline{EditDist(\text{drag}, \text{car}) = 3}$$

darg (transpose ra \rightarrow ar)
carg (substitute d \rightarrow c)
car (delete g)

-Can be solved in $O(|q| \cdot |q'|)$ using dynamic programming

Problem Re-Statement

- Find a subset $close(q) \subset D$ that contains all $q' \in D$ such that $EditDist(q, q') \leq \frac{|q|}{3}$
 - $close(hipopawtamous) =$ words within EditDist of 14/3
 - $close(thier) =$ words within EditDist of 5/3
- 3 is an arbitrary reasonable constant.

Why this task is non-trivial

- Consider a dictionary of size $|D| = 100K$
(the set used by a search engine would be even larger)
- Computing EditDist() isn't that fast
(takes about $10\mu S$ per call, on average)
- Brute force solution: try all words.
($100K$ words \cdot 10μ Secs/Word = 1 Second)
- We're hoping for a solution in the milliseconds.

K-Nearest-Neighbor Approach

- Consider other problems like this:
 - Find pictures that “look like” this one
 - Find sound clips that “sound like” this one
 - Find words that are “spelled like” this one
- All problems are the same, where “likeness” is some task-specific function.
- In the case when this distance function is a metric there are many documented techniques for this problem.

Tree Data-Structures

- We'll be looking at the following approaches:
 - Burkhard-Keller Tree's (BKT)
 - Vantage Point Tree's (VPT)
 - Bi-Sector Tree's (BST)

- All the approaches recursively break up the dictionary into subsets (subtrees) based on how far words are from the "root word"

- During a search, we will be able to eliminate certain subtrees

Burkhard-Keller Tree's (BKT) - Construction

- randomly pick a “root word” x^* .
- define each subtree as $X_i = \{x : d(x^*, x) = i\}$
- recursively build each subtree

Example: { cat , crab , dog , duck , goat }

- Let $x^* = \text{cat}$
 - $X_1 = \{ \}$
 - $X_2 = \{ \text{goat} , \text{crab} \}$
 - $X_3 = \{ \text{dog} \}$
 - $X_4 = \{ \text{duck} \}$
 - ...

Burkhard-Keller Tree's (BKT) - Searching

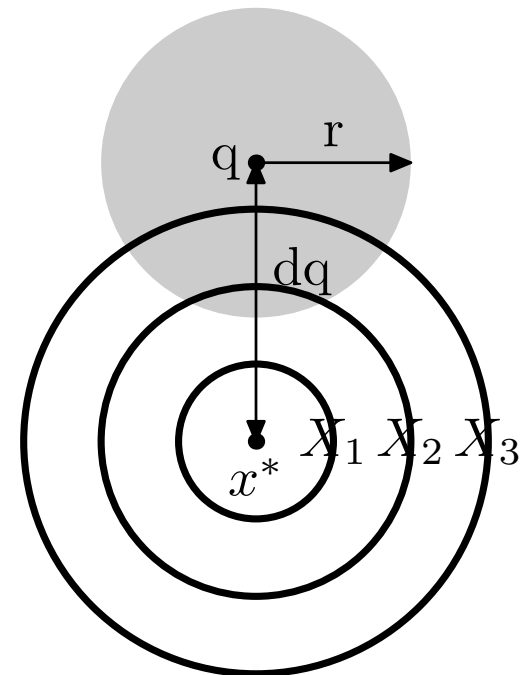
A search for $close(q, r)$ proceeds as follows:

Compute $dq = d(q, x^*)$.

If $dq \leq r$: output x^*

For each i in the range $[dq - r, dq + r]$:

 Recursively search X_i



Vantage Point Tree's (VPT) - Construction

- randomly pick a “root word” x^* .
- break the words up into two equal-sized subsets:
 - the closest 50% go into X_1
 - the other 50% go into X_2
- recursively build each subtree

Example: { cat , crab , dog , duck , goat }

- Let $x^* = \text{cat}$
 - $X_1 = \{ \text{goat} , \text{crab} \}$
 - $X_2 = \{ \text{dog} , \text{duck} \}$

This is a binary tree - we can generalize VPT^m to be m -ary

Vantage Point Tree's (BKT) - Searching

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

(A 3-ary VPT is shown here.)

Compute $dq = d(q, x^*)$.

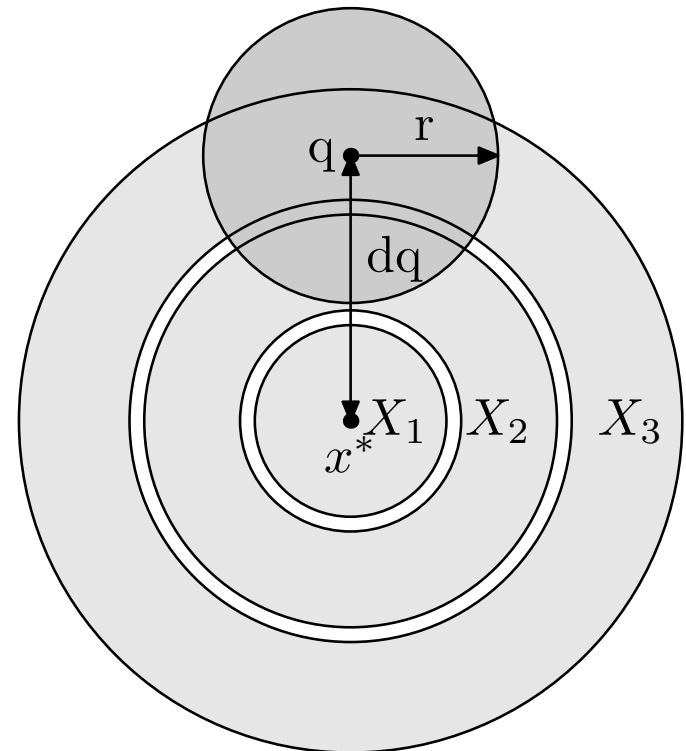
If $dq \leq r$: output x^*

For each $i = 1 \dots 3$:

 If $[dmin_i, dmax_i] \cap [dq - r, dq + r]$:

 Recursively search X_i

($[dmin_i, dmax_i]$ is the inside/outside
"radius" of X_i .)



Bi-Sector Tree's (BST) - Construction

- randomly pick two “parents” x_1^* , x_2^* .
- break the words up into two subsets:
 - words closer to x_1^* go into X_1
 - words closer to x_2^* go into X_2
- recursively build each subtree

Example: { cat , crab , dog , duck , goat }

- Let $x_1^* = \text{cat}$, $x_2^* = \text{dog}$
 - $X_1 = \{ \text{crab} , \text{goat} \}$
 - $X_2 = \{ \text{duck} \}$

This is a binary tree - we can generalize BST^m to be m -ary

Bi-Sector Tree's (BST) - Searching

A search for $close(q, r)$ proceeds as follows:
(A 3-ary BST is shown here.)

For each $i = 1 \dots 3$:

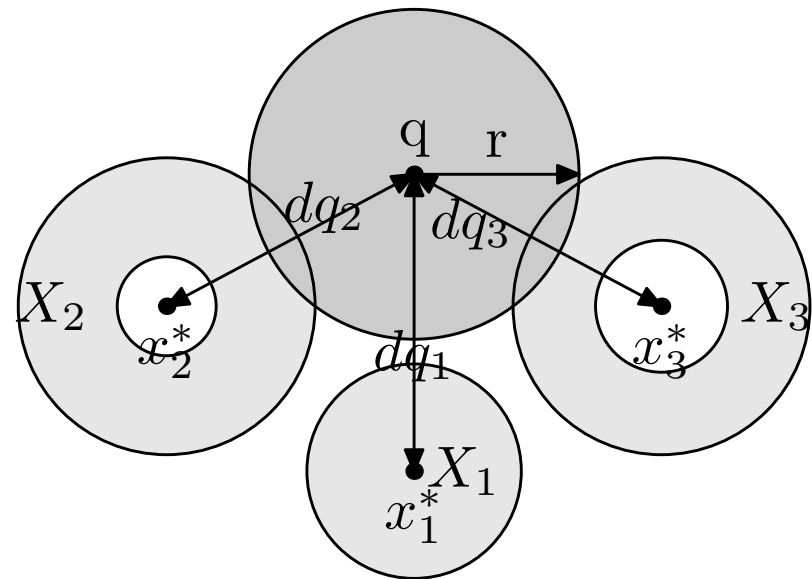
 Compute $dq_i = d(q, x_i^*)$.

 If $dq_i \leq r$: output x_i^* .

For each $i = 1 \dots 3$:

 If $[dmin_i, dmax_i] \cap [dq_i - r, dq_i + r]$:

 Recursively search X_i



Experimental Setup

- Use a dictionary of 100,000 words.
- Randomly build the following structures:
 - BKT
 - VPT^2 , VPT^4 , VPT^{16}
 - BST^2 , BST^4 , BST^{16}
- Create random queries by picking real words, and randomly editing

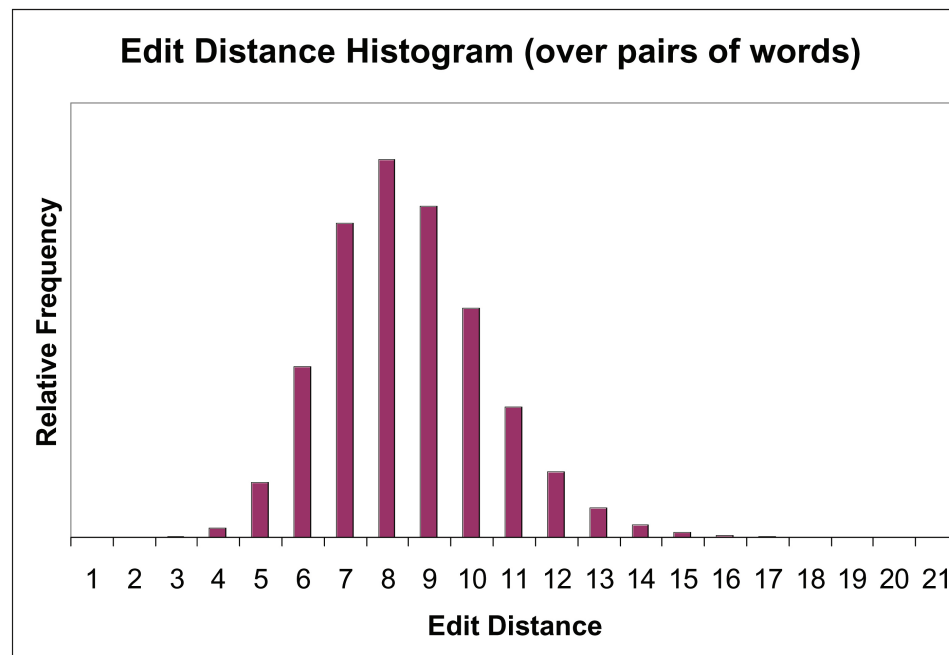
Performance Results

Data Structure	Space Explored (avg.)	CPU time (avg.)
BKT	42.3%	423ms
VPT ²	47.1%	471ms
VPT ⁴	43.8%	438ms
VPT ¹⁶	42.6%	426ms
BST ²	71.7%	717ms
BST ⁴	68.0%	680ms
BST ¹⁶	69.7%	697ms

- Figures were averaged over random instances of each tree, and each query.
- The average query had 92 results.
- There were 100,000 words in the data set.

Are these trees a good fit for this problem?

- The “intrinsic dimensionality” of this data is high (most words are equally far away from each other)
- The triangle inequality doesn't help prune subtrees very often.



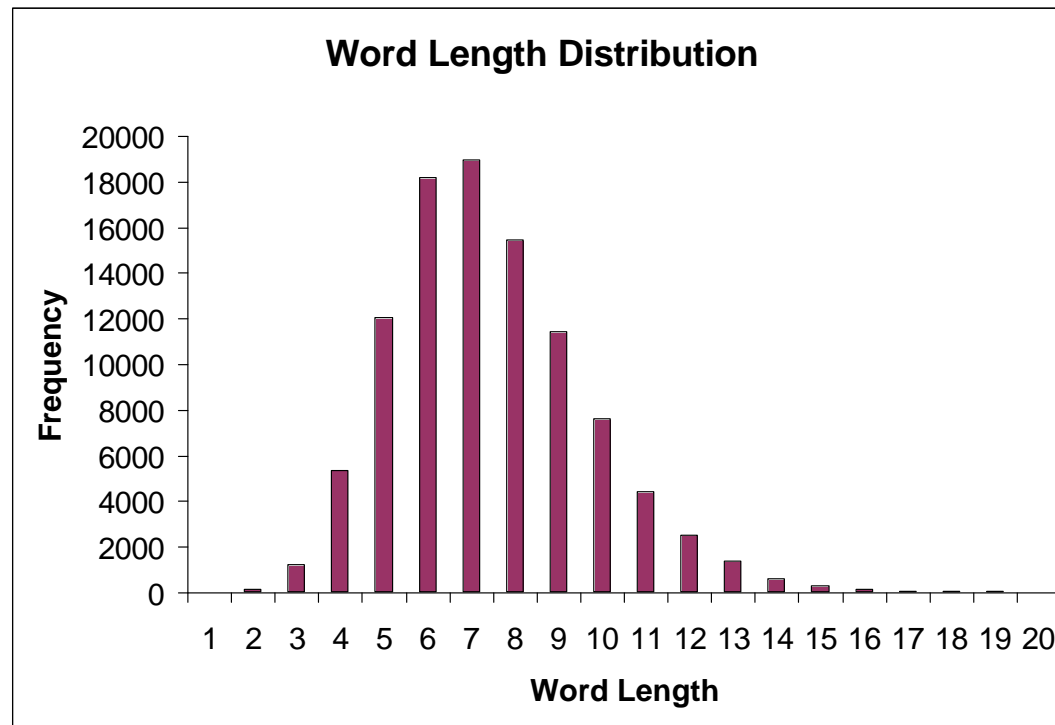
More string-specific Approach

- Exploit our knowledge of string characteristics
- Partition the Dictionary by word length
- Find quick approximations to EditDist()

Partition Dictionary by Length

- Partition $D = \{D_1, D_2, \dots\}$
 $D_i = \{x \in D : |x| = i\}$
- Consider a query word q , and any dictionary word q' .
 $abs(|q| - |q'|) > r \Rightarrow EditDist(q, q') > r$
- For example, if $q = \text{recieve}$, we only consider
 $D_5 \ D_6 \ D_7 \ D_8 \ D_9$

Partition Dictionary by Length



A Quick Lower Bound for *EditDist()*

Consider the 26-dimensional binary vector:

$$L = l_1 l_2 \dots l_{26}$$

There is a bit for each letter 'a' - 'z'.

$$l_i = 1 \quad \text{if a word contains letter } i$$

$$l_i = 0 \quad \text{otherwise}$$

Example: abcdefghijklmnopqrstuvwxyz

$$L(\text{recieve}) = \quad 00101000100000000100010000$$

A Quick Lower Bound for *EditDist()*

Define $LB(q, q') = NumBits(L(q) \oplus L(q'))$.

See that $(LB/2)$ is a lower bound for the EditDist:

- Transposition of two chars \Rightarrow no change in $LB()$.
- Insert one character $\Rightarrow LB()$ changes by at most 1.
- Delete one character $\Rightarrow LB()$ changes by at most 1.
- Change one character $\Rightarrow LB()$ changes by at most 2.

A Quick Lower Bound for *EditDist()* (Example)

```

Example:          abcdefghijklmnopqrstuvwxyz
L(recieve) =     00101000100000000100010000
L(peer)         = 00001000000000010100000000
                  -----
XOR             = --1-0---1-----1-0---1----
Num Bits       = 4
    
```

$$\text{EditDist}(\text{recieve} , \text{peer}) \geq 4/2 = 2$$

- The bit-vectors can be pre-computed
- Computing *LB()* takes $0.1\mu\text{S}$.

Example: recieve

- Find all words within EditDist of $\frac{7}{3} = 2$ from recieve
- Start with 100,000 words
- Eliminate words outside the length range $[7-2, 7+2]$
(75,000 words left)
- Eliminate words for which $LB() > 4$
(818 words left: archive describe eviscerate peer ...)
- Eliminate words whose EditDist is greater than 2
(19 words left : believe, deceive, recede, receive, received, receiver, receives, recipe, reeve, relieve, relieved ...)

Average Performance

- takes **20ms** on average
(compared to 400ms for BKT's)
- I've implemented other "hacks"
that bring this down to **2ms**
- Other possible optimizations:
 - Use even larger bit-vectors
(currently use 64, try 128)
 - Variable-sized bit fields
(eg. 'e' gets 3 bits, 'q' gets 1 bit)

Conclusions

- “Metric trees” (BKT’s, VPT’s, BST’s) aren’t well suited for this task
- The Edit Distance is not as fast as we’d like.
- Methods specifically tailored to strings seem like the way to go.