Optimal(almost) edit distance "1" dictionary

Djamal Belazzougui
Ecole national Superieure d’informatique, Algiers, Algeria
The edit distance between two strings $x$ and $y$ is the minimal number of edit operations needed to get string $y$ from string $x$ (which is the same as number of edit operations needed to get string $x$ from string $y$).

Usually considered edit operations are: insertion, deletions and substitution.

- Insertion: insert a character $c$ at some position in the string.
- Deletion: delete some character from the string.
- Substitution: substitute some character of the string with another character.
Problem definitions (Approximate dictionary with edit distance ”1”)

- We work in the word RAM model with word length \( w \). All standard operations including multiplication, division and shift take constant time.
- We have a set \( S \) of \( n \) strings.
- Total number of characters in all strings is \( m \).
- Each character is encoded using \( b \) bits.
- The size of alphabet is noted by \( \alpha \) where \( \alpha = 2^b \).
- We have to build a data structure on the set \( S \) so that we can answer to an approximate queries that asks for all strings of the set \( S \) that are at edit distance at most ”1” from a query string \( q \).
Applications

- Spell checking in word processors.
- Data-cleaning in databases. The same name is spelled differently in different databases.
- Optical character recognition. Correct the mis-recognized characters or complete the unrecognized characters.
- Information retrieval (Search engines): correct for user’s typing errors.
- Bio-informatics.
Problem definitions (Approximate full-text indexing with edit distance ”1”)

- We work in the word **RAM** model.
- We have a text **T** of length **n** characters.
- We have to build a data structure on the set **S** so that we can answer to an **approximate** query that asks for all **sub-strings** of the text **T** that are at edit distance at most **1** from a query string **q**.
A dictionary that occupies $O(mb)$ bits space.

Space is optimal up to a constant factor.

Query time of the dictionary is $O(k)$, where $k = |q|$ is the length of query string.

Query time is almost optimal.

Application: a full text index that occupies space $O(n(\lg(n) \lg \lg(n))^2 b)$ bits with query time $O(|q|)$. 
Related work (approximate dictionary for edit distance”1”)

General case:

- Recall that $\alpha$ is alphabet size and $\alpha = 2^b$ (each character is represented using $b$ bits).
- Length of query string $q$ is $k = |q|$

<table>
<thead>
<tr>
<th>Method</th>
<th>Query time</th>
<th>Space usage</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG96(1)</td>
<td>$O(k)$</td>
<td>$O(\alpha m)$ words</td>
<td>N.A</td>
</tr>
<tr>
<td>BG96(2)</td>
<td>$O(bk + \lg(n))$</td>
<td>$O(m)$ words</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>BG96(3)</td>
<td>$O(\alpha k)$</td>
<td>$O(m)$ words</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>CGL04</td>
<td>$O(k + \lg(n) \lg \lg(m))$</td>
<td>$O(m + n \lg(n))$ words</td>
<td>$O(m \lg(m))$</td>
</tr>
<tr>
<td>New result</td>
<td>$O(k)$</td>
<td>$O(mb)$ bits (optimal)</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>

Constant sized alphabets case:

- We have $b = O(1)$ and hence $\alpha = 2^b = O(1)$.
- Recall that a word occupies $w$ bits.

<table>
<thead>
<tr>
<th>Method</th>
<th>Query time</th>
<th>Space usage</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG96(1)</td>
<td>$O(k)$</td>
<td>$O(mw)$ bits</td>
<td>N.A</td>
</tr>
<tr>
<td>BG96(2)</td>
<td>$O(k + \lg(n))$</td>
<td>$O(mw)$ bits</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>BG96(3)</td>
<td>$O(k)$</td>
<td>$O(mw)$ bits</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>CGL04</td>
<td>$O(k + \lg(n) \lg \lg(m))$</td>
<td>$O((m + n \lg(n))w)$ bits</td>
<td>$O(m \lg(m))$</td>
</tr>
<tr>
<td>New result</td>
<td>$O(k)$</td>
<td>$O(m)$ bits(optimal)</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>
We have to index a text $T$ of length $n$ characters. Each character is chosen from a set of $\alpha = 2^b$ possible characters.

Queries: report all sub-strings of the text $T$ that are at edit distance ”1” from query string $q$.

We have length of $q$ is $k = |q|$.

<table>
<thead>
<tr>
<th>Method</th>
<th>Query time</th>
<th>Space usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGL04</td>
<td>$O(k + \lg(n) \lg \lg(n))$</td>
<td>$O(n \lg^2(n))$ bits</td>
</tr>
<tr>
<td>New result</td>
<td>$O(k)$</td>
<td>$O(n(\lg^2(n) \lg \lg^2(n))b)$ bits</td>
</tr>
</tbody>
</table>
Naive solution (using exact dictionary)

- Using a standard hash based dictionary occupying space $O(mb)$ bits.
- Query time for exact queries is $O(|s|)$ for a string $s$.
- For approximate queries on a string $q$, we can simply generate all strings at edit distance "1" from $q$ and query exact dictionary for each string.
- We have $k, k2^b$ and $(k - 1)2^b$ candidate strings that can be obtained from $q$ using deletion, substitution and insertion respectively.
- Total query time becomes $O(k2^b \cdot k) = O(k^22^b)$. 
Overview of the new solution (Reducing number of candidate strings)

- number of candidate strings for deletions is $k$.
- Reduce the number of candidate strings for substitutions and insertions to $k$ and $k + 1$ respectively instead of $k2^b$ and $(k + 1)2^b$. For each possible position for insertion or substitution we have a list of candidate characters. Hence we have to explore $k$ and $k + 1$ lists in total for substitutions and insertions.
- Sufficient to check just for first character of the each list.
Overview of the new solution (Reduce query time for each candidate string to $O(1)$)

- Look-up for a candidate string involves two steps: compute a hash function on candidate string and compare the candidate string with the string from the dictionary pointed by the hash function.
- Use a preprocessing step that takes $O(k)$ time.
- Each subsequent hash function evaluation takes $O(1)$ time.
- Each subsequent string comparison takes $O(1)$ time as well.

Djamal Belazzougui Ecole national Superieure d’informatique, Algiers, Algeria
Optimal(almost) edit distance "1" dictionary
We make use of the following components:

- Two **succinctly** encoded tries.
- **Injective** hash functions from sets of strings into sets of integers.
- Minimal perfect hash functions.
- Succinctly encoded **sequences** of integers.
- **Dictionary** based on Minimal perfect hashing.
- Dictionary that stores Succinctly encoded lists, where each list is associated with a key.
Succinctly encoded trie. A trie can be encoded using space $O(mb)$ bits.

Time to construct the trie is $O(m)$.

Time to traverse the trie is $O(k)$ for a string of length $k$.

At each step of the traversal, we get a unique identifier in interval $[0, m - 1]$. 
Minimal perfect hash function for integers

- A minimal perfect hash functions (mphf) maps a set of $n'$ integer keys to interval $[0, n - 1]$.
- Time to construct the perfect hash function if $O(n')$.
- Query time is $O(1)$.

Minimal perfect hash function for strings

- For a set of $n'$ strings having a total of $m'$ characters, we first apply a simple hash function that maps the set of strings to integers of $O(\lg(n))$ bits.
- We can now build the mphf on the set of integers.
Goal: reduce each of the variable length strings to integers occupying $O(w)$ bits each, so that each string is mapped to a distinct integer.

The hash functions is a polynomial over prime field modulo a prime $P$, where $P \geq 2^b$ and $P \geq mn^2$.

For a string $s$, the hash function is evaluated as

$$h_t(s) = \sum_{i=1}^{|s|} s[i] \otimes t^i$$

where $t$ is a number from interval $[1, P]$ that characterizes the hash function.

All additions and multiplications are done modulo $P$.

With a randomly chosen $t$, the hash function $h_t$ will be injective over interval $[0, P - 1]$ with probability at most $1/2$.

Keep choosing a new value for $t$ until all strings are mapped to a distinct integer.
We have a sequence of $n$ integers.
The sum of the integers in the sequence is $m$.
We can encode the integers to use space $n(2 + \lg(m/n))$ bits.
The encoding permits to retrieve the sum of integers $\text{sum}_k = x_0 + x_1 + \cdots + x_k$ for any $k$ in constant time.
Retrieving the integer $x_k$ can be deduced in $O(1)$ time using formulae $x_k = \text{sum}_k - \text{sum}_{k-1}$. 
Components: mphf based dictionary for variable length strings

- We use mphf to map $n$ strings to integers in the range $[0, n - 1]$.
- Store the lengths of strings using a succinctly encoded sequence of integers.
- Store the strings in array in the order given by the mphf.
- Constant access to the start position of every string.
Very similar to the dictionary for variable length strings.

Instead of storing a set of strings, we store a set of lists where each list is associated with a key (we do not store the key itself).

returns size of a list associated with a key in constant time.

Constant time access any element of a list.

The data structure is retrieval only: the data structure returns the correct list for an existing key, but returns an arbitrary list for a non-existing key.
Our data structure will contain the following elements:

- A trie \( Tr \) that stores the strings of \( S \).
- A reverse trie \( \overline{Tr} \) that stores the strings of \( \overline{S} \) where the set \( \overline{S} \) is the set of strings of \( S \) written in reverse order.
- A variant of mphf based dictionary.
- Dictionary of lists.
Our data structure will contain the following elements:

- Construction of $T_r$ and $\overline{T_r}$ takes time $O(m)$.
- The trie $T_r$ and reverse trie $\overline{T_r}$ are succinctly encoded.
- Space usage of $T_r$ and $\overline{T_r}$ is $O(mb)$ bits.
- traversal of the trie and reverse trie returns a unique identifier at each step.
A dictionary for variable length strings augmented with signatures for long strings.

Short strings are stored unmodified.

A string is considered as a long one if its length exceeds $w$ bits.

Each long string will be stored in triple its original size.

We use a parameter $u = \lg(m)/b$.

For a long string $s$ we store signatures of prefixes of $s$ of lengths $u, 2u, 3u, \ldots$.

We also store signatures of suffixes of $s$ of lengths $u, 2u, 3u, \ldots$. 
Each signature occupies $\lg(m)$ bits.

Signatures of prefixes of $s$ are obtained by traversing trie $Tr$ for the string $s$.

Signatures of suffixes of $s$ are obtained by traversing trie $\overline{Tr}$ for the string $\overline{s}$ (string $s$ written in reverse order).

The signature of a prefix of length $iu$ is the unique identifier returned by the trie $Tr$ at step $iu$ of traversal for the string $s$.

The signature of a suffix of length $iu$ is the unique identifier returned by the trie $\overline{Tr}$ at step $iu$ of the traversal for the string $\overline{q}$.

Total space used by signatures does not exceed $2|s|b$ bits.
For each string \( s \) from \( S \) of length \( k = |s| \), we do the following pre-processing:

- We traverse the trie \( Tr \) and store in an array \( L[0..k] \), the labels encountered at each step of the traversal.
- Likewise we traverse the trie \( Tr \) for the string \( \overline{s} \) and store the labels encountered at each step of traversal in array \( R[0..k] \).
- In total we store exactly \( k \) characters in the lists for the string \( s \).
- Total space used by all lists is the same as the space occupied by the strings themselves.
We have a string $q$ of length $k$.

Traverse the trie $Tr$ for the string $q$ and store in an array $L[1..k]$, the labels (integers) returned at each step of the traversal.

Traverse the trie $\overline{Tr}$ for the string $\overline{q}$ ($q$ written in reverse order), and store the returned labels in an array $R[1..k]$.

Time for traversal of $Tr$ and $\overline{Tr}$ is $O(k)$. 

Compute an array $A_t[0, k + 1]$ of powers of $t$ (recall that $t$ is the number that characterizes the hash function). This takes time $O(k)$: first set $A_t[0] = 1$, then set $A_t[i + 1] = A_t[i] \cdot t$ for every $i \in [1, k])$.

Compute an array $F[0..k]$ of the hash values of all prefixes of $q$: first set $F[0] = 0$, then set $F[i] = F[i - 1] + (q[i] \cdot A_t[i])$ for each $i \in [1, k]$. Total time is $O(k)$.

Compute the array $G[1..k]$ of hash values of all suffixes of $q$: first set $F[0] = 0$, then set $F[i] = F[i - 1] + (q[i] \cdot A_t[i])$ for each $i \in [1, k]$. Total time $O(k)$. 
Queries (Substitution)

For \( i \in [1, k] \) such that \( L[i - 1] \neq \perp \) and \( R[k - i] \neq \perp \) we do the following:

- Query the retrieval list for every pair \((L[i - 1], R[k - i])\).
- \( L[i] \) is the identifier of \( q[1..i - 1] \) (prefix of \( q \) of length \( i - 1 \))
- \( R[k - i] \) is the identifier of \( q[i + 1..k] \) (suffix of \( q \) of length \( k - i \)).
- First element of list associated with a pair \((L[i - 1], R[k - i])\) is a character \( c \) that could be substituted at position \( i \) in string \( q \).
- We now have to look for the string \( q' = q[0..i - 1] \cdot cq[i + 1..k] \) in the mphf based dictionary. If we have a match we continue to report all remaining elements (characters) of the list.
Look up in the dictionary should not take more than \(O(1)\) time.

We can compute \(h(q')\) in constant time using formulae

\[
h(q') = F[i - 1] \oplus (c \oplus G[i + 1]) \otimes A_t[i]
\]

We can now use the hash value \(h(q')\) to compute \(mphf\) which will point to a string \(s\) in the dictionary. If \(s\) is a short string of length \(\leq w\), we can compare it with \(q'\) in constant time. Otherwise we compare \(s\) with \(q'\) using the array \(s_l\) of left signatures and the array \(s_r\) of right signatures.

- Length of \(s\) is \(3k\).
- \(l_{q'} = 0 \lor s_l[l_{q'}] = L[l_{q'}].\)
- \(q[u \cdot l_{q'} + 1..i - 1] = s_1[u \cdot l_{q'} + 1..i - 1].\)
- \(s_1[i] = c.\)
- \(q[i + 1..k - u \cdot r_{q'}] = s_1[i + 1..k - u \cdot r_{q'}].\)
- \(r_{q'} = 0 \lor s_r[r_{q'}] = R[r_{q'}].\)
Example (construction)

- A set of two strings "wearable" and "hashable".
- Suppose $w = 32$ bits. We use ASCII alphabet: $b = 8$ bits and $\alpha = 256$ (we have 256 distinct characters).
- The two strings are considered as long strings as their length exceeds $w = 32$ bits.
- We have $u = (32/8) = 4$. 
Example (construction)

We build the trie $T_r$ and reverse trie $\overline{T_r}$.

```
0
   / \   / \\
1 H 2 W 3 A 4 E
   / \   / \\
 5 S 6 A 7 H 8 R
   / \   / \\
 9 A 10 A 11 B 12 B
     /   /   /   / \\
13 L 14 L 15 E 16 E
```
- Traverse the trie $Tr$ for the strings "wearable" and "hashable" resulting in sequences $[0, 2, 4, 6, 8, 10, 12, 14, 16]$ and $[0, 1, 3, 5, 7, 9, 11, 13, 15]$.

- Traverse the reverse trie $\overline{Tr}$ for the strings "elbarae" and "elvahsah", resulting in sequences $[0, 1, 2, 3, 4, 6, 8, 10, 12]$ and $[0, 1, 2, 3, 4, 5, 7, 9, 11]$.

- For string "wearable", we store the characters $w, e, a, r, a, b, l, e$ in the lists associated with pairs $(0, 12), (2, 10), (4, 8), (6, 6), (8, 4), (10, 3), (12, 2), (14, 1), (16, 0)$ respectively.

- For string "hashable", we store the characters $h, a, s, h, a, b, l, e$ in lists associated with pairs $(0, 11), (1, 9), (3, 7), (5, 5), (7, 4), (9, 3), (11, 2), (13, 1), (15, 0)$ respectively.
Build the \textit{mphf} based dictionary.

<table>
<thead>
<tr>
<th>Left signatures</th>
<th>\textit{WEARABLE}</th>
<th>\textit{HAS_ABLE}</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>
We have to index a text $n$ characters each encoded using $b$ bits.

We first, index the text using the data structure of CGL04.

Space usage of that data structure is $O(n \lg(n) \lg \lg(n))$ words.

Query time of data structure is $O(k + \lg(n))$ for a pattern of length $k$ which is optimal when $k \geq \lg(n)$. 
To obtain optimal query time for $k < \lg(n)$, we build our dictionary on all substring of the text of lengths below $\lg(n)$.

For each substring stored in dictionary, we store a pointer to its location in the text.

Total number of substrings is about $n \lg(n) \lg \lg(n)$, and each substring is of length at most $\lg(n) \lg \lg(n)$.

Total space is thus $O(n(\lg^2(n) \lg \lg^2(n))b)$ bits.
We have presented a dictionary for approximate edit distance "1" queries that uses optimal space up to constant factor.

Query time for a string of length $k$ characters is $O(k)$, which is optimal for very large alphabets (characters that occupy $w$ bits), but not for smaller alphabets, for which query time is a factor $w$ away from optimal.

Straightforward application of our dictionary permits to build a full-text index that uses space $O(n(lg(n) lg lg(n))^2)$ with query time $O(k)$ for a pattern of length $k$. 
Follow-ups

- We plan to investigate **practical performance** of the dictionary.

- For **constant-sized** alphabets we can improve **query time** of our dictionary from $O(k)$ to optimal $O(k/w)$ (factor $w$ speedup) at the expense of using space that is a factor $\lg(w)$ from optimal (we use space $O(m \lg(w))$ bits instead of $O(m)$ bits).

- For approximate **full-text indexing**, we have an **improved solution** with space usage reduced to $O(n \lg(n) \lg \lg(n))$ bits for **constant sized** alphabets and to $O(n \lg(n) \lg \lg(n))$ words for arbitrary alphabets.
Open problems

- Improving space: reduce constant factors, entropy compression.
- Is there any lower bound on space/time trade-off for the dictionary.
- What about external memory. A straightforward adaptation of our dictionary in external memory would use $O(k) I/Os$. An optimal solution would use $O(k/B) I/Os$.
- Dynamic version of our dictionary.