EECS 405 Final Report

for

Improving Efficiency of String Similarity Searches through Indexed Pruning

Prepared by

James Fitzpatrick
Kyle Patterson

Version 1.0
May 2, 2014

Contents

1 Abstract **2**

2 Introduction **2**

3 Problem Definition **3**

4 Related Work **3**
4.1 String Similarity 3
4.2 Top-K Search 3
4.3 Positional VGRAM Search 4

5 Methodology **5**
5.1 Data preparation 5
5.2 Development Machines 5
5.3 Cluster identification 5
5.4 Testing 6

6 Evaluation **6**

7 Results **7**
7.1 Cluster generation 7
7.2 Trie generation 7
7.3 Query executions 7

8 Conclusions **8**

9 References **9**

1. Abstract

Numerous solutions exist to deal with the problems caused by data set inconsistencies, such as k-gram similarity and top-k edit distance. However, many of these solutions are limited by data size, as k-gram computation can be both time and resource intensive. With the decrease in digital storage space cost, the average size of data sets has seen rapid growth in the past decade, and frequently trumps the limitations of existing string similarity solutions. This paper proposes merging the realms of data mining and similarity queries as a valid means of reducing search times in large data sets while maintaining high accuracy in similarity query results.

2. Introduction

String similarity search is a rapidly growing area of research within the computer science community due to its numerous overlapping applications in the modern world. From simple database joins to biological queries (such as *'Are these two genes from the same family?'*) to online social presence (*'Do these two accounts belong to the same user'*), being able to identify two objects that are highly similar to each other has proven necessary to understand the current world.

With increases in digital storage, the average query space has increased from kilobytes of space to gigabytes and, in some cases, terabytes. This query space can now contain tens of thousands to millions of entries. While exhaustive searching can be conducted in linear time, an exhaustive similarity search would require computing the edit distance between the query and every entry in at-best pseudo-quadratic time, which is no longer feasible with such large data sets.

The problem becomes much more prevalant when we consider more advanced similarity search techniques, such as k-edit pruning and k-gram searching. The former prunes the search space by removing the candidate from the search space once known edits reach some threshold k, but otherwise operating as an exhaustive search. In k-gram searching, grams of length k are constructed and stored for reference later; words with more than a predefined threshold of grams in common are considered similar and returned. While

gram searching is much more efficient than exhaustive search, it is more expensive in overhead as these grams must be constructed and stored, leading to high access cost on query execution.

One particular field of computer science, data mining, was established specifically to deal with such large data searching problems, and has produced several techniques that could be helpful in large-scale similarity searching. Cluster analysis, or clustering,

3. Problem Definition

Let S be a complete data set consisting of strings from alphabet Σ. For each string s_i in S, we define the following:

- length(s_i) = number of characters in s_i.
  For example, length($'run'$) = 3.
- LCS(s_i,s_j) = longest common substring between s_i and s_j.
  For example, LCS('run','fun') = 'un'.

We define the InvertedList I as follows:

- For each element i in I, every element of i is either an InvertedList or a string
- Every InvertedList i in I has a key that equals the longest common substring of all elements in i

4. Related Work

TODO: Merge together literature review

4.1. String Similarity

4.2. Top-K Search

Top-K Search, a search utilized in String Similarity, computes the top k results by comparing a query string to a dataset and finds the k elements that

have the smallest edit distance from this string. However, this algorithm can become computationally expensive as the algorithm requires comparing the query to all strings in a dataset. To reduce the number of necessary computations, we implement a version of the pivotal Top-K Search as described by Deng et al. A trie-based approach can take advantage of the similarity strings. Each entry in the trie represents a character, save the root which is the null character. The path from the root to a leaf is a inorder representation of that string. This structure redcudes the number of strings we need to store in full by taking advantage of strings with similar prefixes. By using this trie, we can incrementally search the entire search space simultaneously utilizing edit distance measurements. Once k strings have been found, we have our strings to return.

4.3. Positional VGRAM Search

Gram based appraoches to string similarity have been discussed for quite some time. By assembling a inverted list of grams, we can divide a query into grams of size n to compare it to the strings of a dataset. The strings that share the most grams with this string should be considered to be "closer" to the query than others. Many string comparison algorithms consider grams of set lengths. However, there are many advantages and disadvantages to having any particular length of n-gram. Rather than to have a one-for-all solution to the strings, the authors propose a solution to generate grams of variable length between a minimum and maximum length based on a set gram dictionary, which is constructed based on the data collection of strings prior to the query. When computing the gram dictionary, we should only store longer grams that have a frequency above a certain threshold. The algorithm utilizes a trie to reduce the computations of grams that are substrings of longer grams, since the subgrams will necessarily occur when the larger grams are present. Since smaller grams are present in larger ones, the grams that survive pruning will more often than not be shorter grams. Longer grams will be present in data sets only if they are very frequent. By this pruning, we can reduce our gram set to the grams that are most helpful in indentifying desired strings. Further, the position of these grams may be of importance to certain datasets. In some sets, a particular gram may occur at the beginning of the ending of a string, which will identify strings more accurately. By adding positional information, we can also reduce our search space.

5. Methodology

5.1. Data preparation

Data was mirrored from the Database and Logic Programming bibliography (DBLP) and International Movie Database (IMDB) public sources to a MySQL server hosted on a micro instance of Amazon Elastic Cloud Computing (EC2). Data was cleaned to remove any incomplete entries, and trimmed to remove any entries that contained any non-standard ASCII characters (i.e, characters whose value is greater than 127).

In total, the IMDB database contained 55,743 movie entries while the DBLP database contained 2,502,896 publication entries.

5.2. Development Machines

Tries and tests were constructed and executed on a 64-bit Dell Latitude computer with 8 GB of RAM and an i7-2720QM quad-core processor operating at 2.20 GHz. Algorithms were developed using C# and executed in the Visual Studio 2013 v12 Runtime Environment.

5.3. Cluster identification

Algorithm 1 shows how to generate an inverted list index for a given set of strings S.
We begin by computing the difference matrix of S, which is an $O(n^2)$ operation, and storing the matrix in L. We then loop through the matrix, seeking entries of decreasing size. These entries are stored as coordinates of string pairs, where the ID of each string would yield the matrix position of the LCS. For each pair, we check if the key (i.e, the longest common substring of the string pair) already exists within our inverted list; if it does, we append each string of the pair into the proper list. If the key is not yet present, we create a new List for the string pair and insert it into the inverted list using the computed key.

Upon completion of Algorithm 1, an index has been created that can be stored for future reference in queries and for pruning purposes.

Algorithm 1 Generate inverted list of LCS(S)

```
1: L = GenerateDifferenceMatrix(S);
2: M = InvertedList();
3: for i = max(length(s_1), ..., length(s_n)) to 0 do
4:    l = FindEntriesOfLength(L,i);
5:    for all (s_x, s_y) in l do
6:       if LCS(s_x, s_y) exists in M then
7:          M.Add(LCS(s_x, s_y), s_x, s_y);
8:       else
9:          M.Add(LCS(s_x, s_y), new InvertedList(LCS(s_x, s_y), s_x, s_y));
10:      end if
11:   end for
12: end for
13: return M;
```

5.4. Testing

Tries were evaluated using the following method:

1. Generate query structure for a randomized subset of data
2. Select random values from data subset and randomly edit values within k edits
3. Query trie structure for edited values

Tests were executed for data subsets in increments of 10,000 from 10,000 elements to the full data set, with 30 queries executed per trie. Query structures used include VGRAM and Top-K Tries.

6. Evaluation

TODO: Explain what we measured in the testing phase and how we'll interpret the results.

7. Results

7.1. Cluster generation

TODO: Clustering is bad, mmkay?

7.2. Trie generation

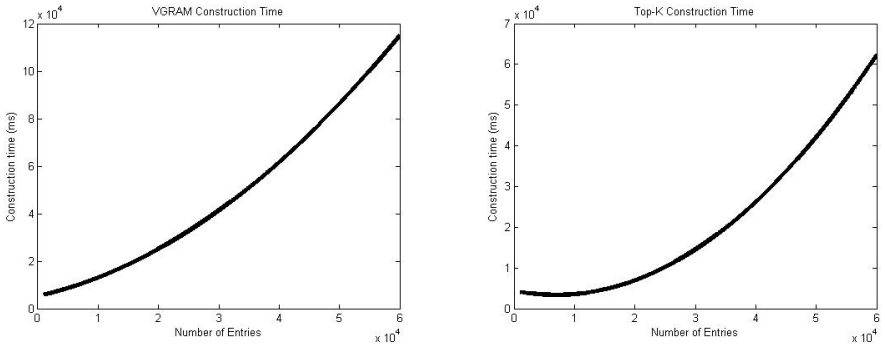

Figure 1: Construction times for VGRAM and Top-K per data set sizes.

7.3. Query executions

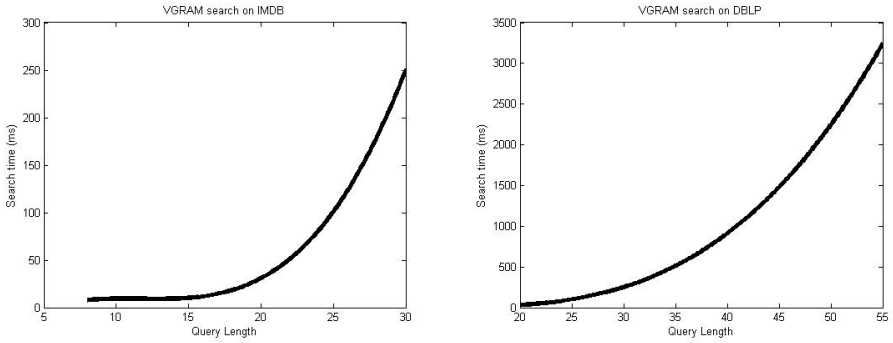

Figure 2: Query times per IMDB and DBLP data sets using the VGRAM Trie.

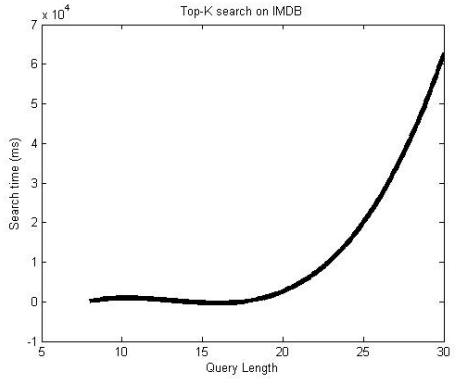

Figure 3: Query times for IMDB data set using the Top-K Trie. The full DBLP data set was too large for Top-K queries.

8. Conclusions

In this paper, we presented a pruning technique that relies on common data mining and indexing techniques

9. References

1. Don Deng, Guoliang Li, Jianhua Feng, Wen-Syan Li, Top-k String Similarity Search with Edit-Distance Constraints, ICDE 2013.

2. C. Li, B. Wang, X, Yang. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. VLDB, September 23-27, 2007.

3. M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. *SIGMOD Conference*, pages 429-440, 2009.

4. A. Andoni and K. Onak. Approximating edit distance in near-linear time. STOC, pages 199-204, 2009.

5. W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18-31, 1980.

6. T. Kahveci and A. K. Singh. Efficient index structures for string databases. VLDB, pages 351-360, 2001.

7. Z. Yang, J. Yu and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. AAAI, 2010.

8. R. Zafarani and H. Liu. Connecting users across social media sites: a behavioral-modeling approach. KDD, pages 41-49, 2013.