SUFFIX TREES FOR DOCUMENT RETRIEVAL

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ryan Reck

June 2012

COMMITTEE MEMBERSHIP

TITLE:	Suffix Trees for Document Retrieval
AUTHOR:	Ryan Reck
DATE SUBMITTED:	June 2012

Committee Chair	Franz Kurfess, Dr.
Committee Member	John Clements, Dr.
Committee Member	Alex Dekhtyar, Dr.

Abstract

Suffix Trees for Document Retrieval

by

Ryan Reck

This thesis presents a look at the suitability of Suffix Trees for full text indexing and retrieval. Typically suffix trees are built on a character level, where the tree records which characters follow each other character. By building suffix trees for documents based on words instead of characters, the resulting tree effectively indexes every word or sequence of words that occur in any of the documents. Ukkonnen's algorithm is adapted to build word-level suffix trees. But the primary focus is on developing Algorithms for searching the suffix tree for exact and approximate, or fuzzy, matches to arbitrary query strings. A proof-of-concept implementation is built and compared to a Lucene [6] index for retrieval over a subset of the Reuters RCV1 data set [8].

Acknowledgements

Thank you to my thesis advisor Franz Kurfess for his patience in helping me see this through. To my committee for their help and guidance; and to my wife Martha for the support and motivation to finish.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Information Retrieval often needs to provide a user with a list of documents relevant to some search query [11]. A lot of research goes into how to determine the best documents to return. To provide results quickly even on large data sets some sort of index is required to map query terms to documents, as well as appropriate algorithms for building the index ahead of time and scoring documents at search time. Different kinds of indexes are used for the different qualities they have. Some indexes work for different kinds of data but most are designed for indexing words in plain-text documents.

Reverse indexes are the standard way of determining relevant documents in response to a query [15]. A reverse index tracks all the words in all indexed documents along with which documents they occur in. Additional information, such as where in each document the word occurs, can also be stored and then used when needed to match documents to a query. I propose using a suffix tree based reverse index instead of the standard flat reverse index. A suffix tree index maintains more structural information than a flat index, and is therefore more suited to answering certain types of queries.

Different kinds of indexes require different searching and scoring algorithms. Searching a reverse index involves looking up what documents contain the words in the query and then computing a score for each document. The scoring algorithm is important [2] since it determines what documents are considered more relevant to the user and thus presented first.

In addition to creating a proof of concept suffix tree based search index implementation, I explore the various alternative algorithms for building such a suffix tree and algorithms for finding and scoring documents with respect to a search query.

1.1 Thesis Question

The driving question here is how does a suffix tree-based search index compare to a typical reverse index. The comparison primarily focuses on their fuzzy and non-fuzzy query capabilities by evaluating the relevance of the returned results, and includes measures of their run-time performance and storage space requirements.

To answer this question, a proof-of-concept suffix tree-based search index is implemented along with a reverse index implemented with standard open source tools. The two indexing systems are then compared and evaluated.

1.2 Background Definitions

Before proceeding much further there are some background definitions to take care of.

Suffix Tree - Gusfield [5] defines suffix tree as follows:

> A suffix tree T for an m-character string S is a rotted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty sub-string of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that start at position i. That is it spells out $S[i..m]$.

Additionally to guarantee every suffix is distinct, and thus ends at a leaf node, an extra term is concatenated to the end of the string. Here we will use $\$n$ where n is the unique id number of the document.

Generalized Suffix Tree - A Suffix Tree containing multiple strings in a single tree. Generally a unique suffix is added to the end of the string to distinguish strings.

Word-Level Suffix Tree - A Suffix Tree showing sequences of words instead of sequences of characters. Generalized Word-Level Suffix Trees are what I am using for my search indexes.

Reverse Index - An index from terms to documents they occur in and, optionally, where the terms occur in the documents. Reverse indexes are also known as inverted indexes.

Exact Matching Algorithm - An algorithm to determine if a query string is a proper sub-string of a target string or any document in the search index.

Exact Sub-String Matching Algorithm - An algorithm to determine the longest common sub-string between a query string and a target string or any document in the search index.

Fuzzy Matching Algorithm - An algorithm that determines approximate matches

between a query string and a target string or any document in the search index. Fuzzy matched documents will contain some extra words inserted into the query string, omit some words from the query string, or both.

Chapter 2

Previous Work

Relevant related work falls into three basic categories: suffix tree construction algorithms, suffix tree uses, and established indexing designs. Related work in each of these areas follows.

2.1 Suffix Tree Construction Algorithms

There have been three big suffix tree papers each proposing an algorithm for their efficient construction. This section provides an overview of these papers, followed by an in-depth look at Ukkonen's construction algorithm which provides the basis for the construction algorithm developed here.

2.1.1 1973 Weiner - Linear Pattern Matching Algorithms

Suffix trees were first introduced by Weiner in 1973 [14] although he called them *position trees*. Unfortunately his paper was considered extremely difficult to understand and garnered little follow-on work.

2.1.2 1979 McCreight - A Space-Economical Suffix Tree Construction Algorithm

McCreight came along in '79 and introduced another algorithm with better space efficiency [9]. Like Weiner's algorithm McCreight's built the tree by working backwards from the longest suffix to the shortest. Building the tree in this manner is simple and allows the tree to be built in a single scan of the string. But this method is more difficult to extend for generalized suffix trees.

2.1.3 1992 Ukkonen - Constructing Suffix Trees On-Line in Linear Time

Ukkonen came along in '92 with the first new suffix tree paper in over a decade [13]. Ukkonen's algorithm conceptually builds the tree differently than either of the previous algorithms by starting with an empty tree and extending it for each character in the string in a single forward scan. By building the tree in this manner it becomes an on-line algorithm: so that after inserting the Nth character the suffix tree is complete for the first N characters of the input string. Since publication, Ukkonen's algorithm has generally been the preferred suffix tree construction algorithm, mostly due to the paper being considered easier to understand than the previous offerings.

The main difference between McCreight's and Ukkonen's algorithms turns out to be mostly conceptual. Conceptually Ukkonen's algorithm builds in a single forward pass by building the tree for the previous $N-1$ characters of the string and extending that tree by one character each step to get the tree representing the first N characters of the string, while McCreight's builds backwards by inserting

the longest suffix first then dividing that edge and inserting nodes to get to the shortest suffix. The actual construction process proceeds similarly in both algorithms, but this conceptual difference probably leads to Ukkonen's algorithm being easier to understand. The difference does mean that Ukkonen's algorithm is better suited for building Generalized Suffix Trees.

For building a Generalized Suffix Tree an on-line algorithm is capable of extending the previous suffix tree with the new suffixes unique to the recently added string. Without an on-line algorithm the previous tree could not be extended to create the new generalized tree so it would be necessary to build the suffix tree for the new string and then merge the new tree into the generalized suffix tree.

2.1.4 Ukkonen's Algorithm In-Depth

Ukkonen's algorithm builds the suffix tree for string s in a single pass over the characters in the string. A simple suffix tree, with one node for every character in every suffix, explodes with n^2 space efficiency. To avoid that, Ukkonen reduces the number of nodes necessary by defining *explicit* and *implicit* nodes. If a node has only one child node, they get joined into a single explicit node, and the absorbed parent's node becomes an implicit node, since it is no longer represented in the tree explicitly. With this change, a node no longer represents a single character, but a sequence of characters. For efficiency these sequences are not duplicated from the original string, but represented as start and end indexes into the original string. This indirection removes the need for having multiple copies of the string. Additionally, storing indexes, instead of actual sub-strings, allows leaf nodes to be left open-ended during the construction process, ending at the most recently processed character. Having leaf nodes be open-ended serves to allow every leaf to

be extended without an explicit update, which helps obtain a linear time bound.

Each step of Ukkonen's algorithm extends every suffix in the tree by one additional character. In other words, for the text t and tree $tree$ where $tree_i$ is the suffix tree built from the sub-string $t[0, i]$, $tree_i$ is obtained by extending the previous tree, $tree_{i-1}$. The nodes in $tree_{i-1}$ that need updating for $tree_i$ make up what is known as the *boundary path*. Now the nodes on this path can be classified into three groups: leaf nodes, branching nodes without a branch on $tree[i]$ and branching nodes with a branch on $tree[i]$. The leaf nodes only need to have their edge/sub-string extended by one character for them to represent the new suffix, and since the leaves are left open-ended this is already done. Next we need to update the branching nodes that represent the suffixes that are getting extended. For any branching node without a $tree[i]$ branch, a new branch and leaf node must be added. For branching nodes with a $tree[i]$ branch, no action needs to be taken, since the current suffix is already represented in them. Therefore, the only nodes that must be updated on each iteration are the non-branching nodes on the boundary path.

If the parent node of the new branch is an implicit node, it must be made explicit. To make the node explicit, the explicit node that encompasses it must be split into two explicit nodes. Then a new new branch is added to the parent node and a new open leaf node is created.

The nodes that need new branches fall between the *active point* and the *end point*. For each of these nodes, the node must be made explicit, if it is not already, and a new t_i transition added to our new leaf node.

Changes

Ukkonen's algorithm was designed around character-level suffix trees, and was optimized for those constraints. Building suffix trees using words instead of characters is conceptually identical, but varies somewhat in the actual implementation. I still use indexes into the word list to avoid copying out sub-sequences of words, but comparison of words, or strings, is not as simple as comparing character values, so the efficiency constraints might not hold.

The other significant difference is using generalized suffix trees instead of just simple ones. Ukkonen's algorithm works almost unmodified for constructing generalized suffix trees, but needed to be slightly modified to add extra document indexes, see Section 3.1.1.

2.2 Suffix Tree uses

Suffix trees have been used in several niches. They are incredibly useful in DNA and gene sequencing applications where they can efficiently find common sub-sequences in long DNA sequences. They can also be used for text editor auto-completion and similar tasks, but I am unaware if they are. Gusfield mentions several applications in his book *Algorithms on Strings, Trees and Sequences* [5].

Recently suffix trees have seen some uses in information retrieval fields. Here they have been mostly used for computing document similarity and related tasks. In their 2007 paper [3] Chim and Deng use suffix trees to cluster documents in on-line forum communities.

Little research has been done on using suffix trees for document retrieval. Though Eissen et. al. [4] did an in-depth comparison of the suffix tree document

model with more common models, there hasn't been any research on using suffix trees for document retrieval directly, as opposed to clustering the results after using another means to determine the results to return.

2.3 Accepted search means

There are two fundamental ways to search a set of documents, with a pre-built index or without. For pre-built indexes, reverse indexes are the most used due to their efficiency, simplicity, and effectiveness [15]. Various other types of indexes, such as document-term matrices, associative memory models [1][12], or ngram indexes, could be used but aren't generally competitive with reverse indexes and don't extend easily into phrase searches.

2.3.1 Reverse Index - Lucene

A reverse index is basically a list of terms and which documents each term occurs in. This type of index works incredibly well for simple searches, the terms are looked up and the list of documents returned. With a few minor tweaks they can handle much more complex types of queries: for example if you store position info, you can use the positions of matched terms in a document for phrase based queries.

Lucene is a popular open source search engine that uses reverse indexes [6]. Lucene can store positions for performing phrase queries as well as term frequency info for better results, compared to the simplest occurs/does-not-occur boolean scoring.

Using suffix trees to index documents still qualifies as a reverse index, since it

still has a set of terms tied to what documents they occur in. However, instead of a flat term index the index is a suffix tree. This should provide improved performance for phrase queries since the phrases do not have to be reconstructed after the terms and their document-locations are looked up in the index. Additionally the same suffix tree used to index the documents could be used for similarity comparisons and clustering without requiring extra storage space for both an index and a suffix tree.

2.3.2 On-Line Search

The index-less alternative search method is on-line search. Instead of building an index ahead of time, every search involves scanning every document for an occurrence of the search term. Obviously this is not feasible for many IR applications but works just fine on a small enough scale (like grep "suffix tree" thesis/*.tex)

Chapter 3

System Design & Implementation

To examine the effectiveness of suffix tree indexes, I built a simple search tool to retrieve documents using either a suffix tree based index, or a standard Lucene index. The results were later evaluated for the quality of the returned results.

The system has two main components: a servlet interface for searching existing indexes, and a command line interface for creating new indexes and updating existing ones.

The indexes are additionally abstracted through the SearchEngine interface. The interface defines all the methods needed to search and update existing indexes. The SuffixTreeEngine and LuceneEngine both implement this interface, so the two engines are completely interchangeable once loaded from disk.

3.1 Suffix Tree Component

The Suffix Tree component builds a suffix tree-based document index. The model for representing a suffix tree was fairly straight-forward and most of the

work went into the various search algorithms. The suffix tree is modeled by TreeNodes, which each store a reference to the words they represent and a HashMap for their children, indexed by the child's sub-string's first word. In a suffix tree no two children of any node can begin with the same word, so that first word is guaranteed to uniquely identify a single child node.

Each node stores a list of the words that node represents. This is not grossly inefficient since all Java List implementations return a list that refers to the same backing word list merely with different offsets, in other words the data in the list is not duplicated across every node, but each node refers to its portion of the original list. However, most standard Java lists' sub-list method returns a sub-list that cannot be written to disk. So a custom list class was required to wrap an ArrayList, for fast random access and to provide start/end pointers into the backing array list, as well as return serializable sub-lists.

The various algorithms involved absorbed most of the development time. The following subsections detail all of the significant algorithms involved in building suffix trees, searching for matches, and scoring the results.

3.1.1 Building Suffix Trees

Suffix trees are built with Ukkonen's algorithm, modified to build generalized word-level suffix trees. Each document is given a unique suffix "$docId$" to ensure that every suffix is unique.

In a suffix tree constructed with Ukkonen's algorithm, any node only contains information about its first occurrence in the document. This becomes a problem for returning search results from non-leaf nodes, since they only store the information about the first document they occurred in, not all of them. This limitation

could be worked around by traversing the entire sub-tree at a search result's terminus since there will be at least one leaf node for each document that contains the matched suffix. However, walking entire sub-trees would add significant time to every search. If the occurrence information were moved into every node, and not just leaves, you could eliminate this search-time cost but with an increased space requirement. This change results in a higher space requirement and increases the time required to build the index but eliminates substantial work from the search process so the increased space requirement brings substantial benefits in being able to return search results more quickly.

Moving the position information up from the leaf nodes has two obvious alternatives: recursively walk the tree adding positions to any node based on the positions of its children, or look-up each suffix of the original string in the suffix tree and mark every node the look-up traverses. The first method requires traversing the entire tree and calculating the positions based on each node's length and the positions of its children. With the second method, the suffix tree must be searched for each suffix of the inserted string and the position of the suffix recorded in the nodes traversed.

While neither of these methods are particularly efficient, the second approach is guaranteed to involve fewer nodes than the first. The first must examine every node in the tree while the second only examines every node which is part of the most recently added string, or document. Clearly, in any suffix tree with more than one distinct document, the second will involve fewer nodes.

3.1.2 Searching

Different search strategies can be used depending on the search results desired. These can be as simple as determining if a query string occurs in any document in the collection exactly, whether the query or any of the its sub-strings occur, or if any strings sufficiently similar to the query string occur. Each one of these employs a different search strategy and involves a different amount of complexity.

Exact Matching

Searching a suffix tree for an exact match is a simple look-up, and very efficient. The search becomes a linear complexity look-up traversing the tree nodes. Starting at the root of the tree and the first word in the query, look-up the word in the node's child map, if it is found, compare the child's sub-string to the string. If the child's string is a proper prefix of the search string, continue the look-up using the child node and the next word in the query after the prefix as the next word to look-up. If the query string is a prefix or exact match of the child's string, the search terminates successfully. If at any node the next query word is not found, the search terminates unsuccessfully. The complexity is linear since searching for a string involves at most n word look-ups or comparisons, where n is the size of the string. More interesting look-ups, with more interesting results than whether or not the string is found, add more complexity.

The exact matching algorithm is shown in Algorithm 3.1. The algorithm is implemented as a recursive method called $rSearch$ that looks up the queried words in the given node. If there are words to look up, it calls $commonPrefixLength$ to determine the length of the match, and if the words the next node represents were all matches then the search continues by search $next$ for the remainder of

the search terms. If the terms were all matched as a prefix of the next nodes words, matches are also constructed for all documents in the next node.

Algorithm 3.1: Exact Match

```
def exactRSearch(node, words):
  result = dict()
  if len(words) == 0:
    for doc in node.docs:
      result[doc] = node.getFirstLocation(doc)
  else:

    if words[0] in node.children:
      next = node.children[words[0]]
      cpl = commonPrefixLength(words, next.words)
      if cpl == len(next.words):    # all words in next matched
        result2 = rSearch(next, words[cpl:])
        for doc in result2:
          preLength = 0
          if node != root:
            preLength = len(node.getFirstLocation(doc))
          postLoc = result2[doc]
          result[doc] = Location(postLoc.start - preLength,
              postLoc.end)
      elif cpl == len(words):       # all words found in next
        for doc in node.docs:
          preLength = 0
          if node != root:
            preLength = len(node.getFirstLocation(doc))
          postLoc = next.getFirstLocation(doc)
          result[doc] = Location(postLoc.start - preLength,
              postLoc.start + cpl-1)
  return res

def commonPrefixLength(list1, list2):
  n=0
  while n < min(len(list1), len(list2)) and list1[n] == list2[n]:
    n+=1
  return n
```

To illustrate the algorithm, let's walk through an example. Take the tree from Figure 3.1, and we'll search for the string "suffix tree search algorithm." Starting at the root we look-up the first word in the root's children and find the node representing the string "suffix tree." We calculate the $commonPrefixLength$ and since it's equal to the number of words this node represents we continue the search by recursively calling $exactRSearch$ with the child node and the remaining

search terms: "search algorithm." Looking up "search" in the next node yields no result so the search terminates with no matches found.

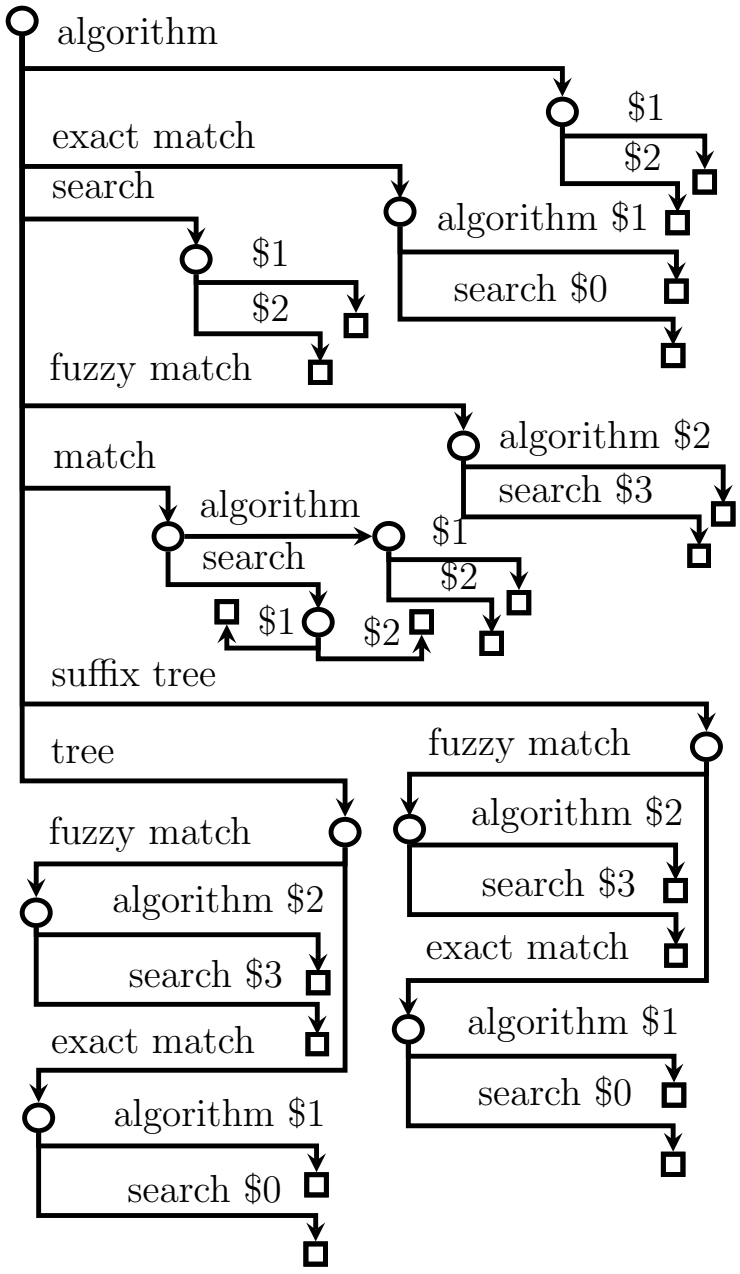


Figure 3.1: Example Suffix Tree

Exact Sub-String Matching

Checking for exact sub-string matching is not significantly more complicated. Instead of a single look-up for the entire string, each suffix of the query string must be looked up. Therefore the complexity becomes (n^2) but since query strings are generally short, the added overhead is not bad in practice.

Additionally searches are not simply terminated when the next word fails to be found but gets recorded as a partial match up to the last matched word. This change adds no time complexity for finding the results but does complicate the gathering and ranking of results as well as requiring more time to deal with the greater number of returned results.

The exact sub-string match algorithm in Algorithm 3.2 is nearly identical to the exact match algorithm. There are two changes to the search algorithm to capture partial matches: lines 9-11 are added to return partial matches when no next node can be found, and the loop starting on line 17 is rewritten to handle partial matches that don't continue in the next node or match the next node's word list completely. One other change is required to match sub-strings correctly, the method which invokes *rSearch* must invoke it multiple times omitting query terms to find partial matches that do not include the first word.

Algorithm 3.2: Exact Sub-String Match

```
1  def rSearch(node, words):
2    result = dict()
3    if len(words) == 0:
4      for doc in node.doc_positions:
5        result[doc] = node.getFirstLocation(doc)
6    else:
7      if words[0] not in node.children:  # no next node to search
8        if node != root:
9          for doc in node.doc_positions:      #record partial matches
10           result[doc] = node.doc_positions[doc][0]
11     else:                              # next is not null
12       next = node.children[words[0]]
```

```
13          cpl = commonPrefixLength(words, next.words)
14          result2 = None
15          if cpl == len(next.words):    # all words in next matched
16            result2 = rSearch(next, words[cpl:])
17          for doc in next.doc_positions:
18            preLength = 0
19            if node != root:
20              preLength = getLength(node.doc_positions[doc][0])
21
22            if result2 and doc in result2:
23              start, end = result2[doc]
24              result[doc] = (start - preLength, end)
25            else:                         #record partial matches
26              start, _ = next.doc_positions[doc][0]
27              result[doc] = (start - preLength, start + cpl-1)
28    return result
```

To continue our previous example searching for "suffix tree search algorithm" the search starts the same, but differs in how non-matches are handled. Instead of terminating when "search" is not found in the "suffix tree" node, a partial match is returned instead. The other difference is repeating the search for each of the suffixes of the search string, ie "tree search algorithm," "search algorithm," and "algorithm"; though in this case the additional single term matches are overshadowed by the previous matches for "suffix tree."

Fuzzy String Matching

Fuzzy matching allows greater variance between the query string and matched documents and subsequently adds more complexity. Fuzzy matching matches strings within a given Levenshtein distance [7] of the query string. The Levenshtein distance, or edit distance, of two strings is the number of insertions, omissions, or replacements needed to turn one string into the other. Two strings are said to match if their edit distance is less than the maximum allowed fuzz, or k as it is referred to here.

The key difference between inexact and exact match searching of suffix trees

lies in how children nodes to search are chosen. Exact matching only searches the nodes whose string exactly matches the next part of the query string, and these nodes can be efficiently looked up in a hashtable of their first word. Inexact matching searches every child whose string's edit distance is within k of the next part of the query string. So inexact matching must determine for every child of every node on the search path whether that child's string is within k of the search string, or where the edit distance between the child node's sub-string and the current prefix of the search query is less than k, before continuing with that node's children. The fuzzy match then proceeds to search each of the suitable child nodes, with a decreased allowed edit distance based on the edit distance of the child node's sub-string to the current prefix of the search query. Once the allowable edit distance reaches zero, matching is done with the exact match search previously discussed.

Gusfield presents an algorithm for finding all matches within edit distance k of a search string in $O(km)$ time using suffix trees. However, the extra processing required, proportional to m or the sum of the lengths of every document in the index, would have performed poorly. Instead a theoretically less efficient algorithm was used that works out better in practice since k is generally much smaller than m.

Calculating a complete edit distance between the remaining query text and a node's string is unnecessary, the only thing needed is the length of the longest possible match with no more than k differences and without continuing past the end of either string. To calculate the length of the longest match, whenever a mismatch is encountered, try each of the three possible edits (insertion, omission, and replacement) and keep the longest obtained length with the fewest number of edits. The resulting algorithm has exponential complexity on the number of

edits k, but since this is generally small, performance is still good.

There are two additional cases where a child node will be searched: if there is no match between a node's text and the remaining search string and the length of the node's text is less than k, or if the length of the remaining portion of the node's text, subtracting the length of the best possible match, is less than the remaining fuzz after subtracting the fuzz used in the match; in other words, when $len(text) - len(match) < k - k_{match}$ evaluates to true. In either of these cases the node must still be scanned since a successful match could continue with any of its children. However, to accurately compute the edit distance, the node must be searched several times with varying remaining search strings representing the alternatives for fuzzy matching the remaining portion of the child node's text. For each iterative search the fuzz consumed is the same but the remaining search string differs depending on how many words in the search text should be passed on to the child search to generate the best match. This variance means these child nodes must each be searched $len(text) - len(match) + 1$ times to cover all the possibilities for how many words are inserted verses replaced.

Scanning every child of a node is generally not too bad, since most nodes have a relatively low branching factor. The root, on the other hand, has a very high branching factor so it is handled differently. Optimal matches always begin with a match, since any match starting with a mismatch will always score lower than the same match starting one word later. So since matches only start from the root, fuzzy matching every child node would be less efficient than just fuzzy matching the one child that begins with the same word as the query string. Additionally to bring back matches that might skip the first word in the query text, the root is searched for every sub-string in the query text identical to the Exact Sub-String Matching algorithm above.

As the fuzzy match algorithm descends to a new node, the maximum allowable edit distance is decreased by the edit distance from the query string to the new node. Once the edit distance is zero, calculation reverts to the previously mentioned exact search algorithm.

Algorithm 3.3: Fuzzy Match

```
def fuzzyRSearch(node, words, restrict, fuzzFactor):
  result={}
  if len(words) == 0:
    for doc in node.doc_positions:
      result[doc] = node.getFirstLocation(doc)
    return result

  if fuzzFactor <= 0:
    return wrapFuzzy(rSearch(node,words), node)

  nodes = []
  if node == root:
    if words[0] not in root.children:
      return dict()
    nodes = [ root.children[words[0]] ]
  else:
    nodes = node.children.values()

  for child in nodes:
    straightMatch = fuzzyCommonPrefixLength(
      words, child.words, fuzzFactor, restrict)

    matchesToTry = []

    if straightMatch.matchCount:
      matchesToTry.append(straightMatch)

    if fuzzFactor > 0 and straightMatch.length < len(child.words)
       and len(child.words) - straightMatch.length <= fuzzFactor -
       straightMatch.fuzz and len(child.children) > 0:
      size = len(child.words)
      difference = size - straightMatch.length
      cutoff = min(difference, len(words) -
         straightMatch.wordsConsumed)
      for i in range(0, cutoff+1):
        preFuzz = size
        if straightMatch.wordsConsumed:
          preFuzz = straightMatch.preFuzz
        matchesToTry.append(FuzzyMatch(
          fuzz=straightMatch.fuzz + difference,
          length=size,
          preFuzz=preFuzz,
          matchCount=straightMatch.matchCount,
```

```
41            wordsConsumed=straightMatch.wordsConsumed + i,
42            restrict=UNRESTRICTED))
43
44      for match in matchesToTry:
45        result2={}
46        if match.length == len(child.words) and \
47           len(words) - match.wordsConsumed > 0:
48          result2 = fuzzyRSearch(
49            child,
50            words[match.wordsConsumed:],
51            match.restrict,
52            fuzzFactor - match.fuzz)
53
54        for doc in child.doc_positions:
55          preLength = 0
56          if node != root:
57            preLength = getLength(node.doc_positions[doc][0])
58          newMatch = None
59          if doc in result2 and result2[doc].matchCount > 0:
60            newMatch = result2[doc].extend(preLength, match)
61          elif match.matchCount > 0:
62            start, end = child.doc_positions[doc][0]
63            newMatch = FuzzyMatchLocation(
64              start= start,
65              end= end,
66              offset= match.preFuzz,
67              fuzz= match.fuzz - match.preFuzz,
68              matchCount= match.matchCount)
69          if newMatch and (doc not in result or
70                           newMatch.betterThan(result[doc])):
71            result[doc] = newMatch
72    return result
73
74  def fuzzyCommonPrefixLength(pattern, text, maxFuzz, restrict):
75    minLength = min(len(pattern), len(text))
76    pre = commonPrefixLength(pattern, text)
77
78    best = None
79    if pre < min and maxFuzz > 0:
80      if pre or restrict != NO_INSERT:          # try insert
81        match = fuzzyCommonPrefixLength(
82          pattern[pre:],
83          text[pre+1:],
84          maxFuzz-1,
85          NO_OMIT)
86        if match.matchCount > 0:
87          best = match
88          best.preFuzz += 1
89          best.length += 1
90      if pre or restrict != NO_INSERT:          # try omit
91        match = fuzzyCommonPrefixLength(
92          pattern[pre+1:],
```

```
93            text[pre:],
94            maxFuzz-1,
95            NO_INSERT)
96          if ((best == None and match.matchCount > 0) or
97              (best and match.matchCount > best.matchCount)):
98            best = match
99            best.wordsConsumed += 1
100       if pre or restrict != NO_INSERT:          # try substitute
101         match = fuzzyCommonPrefixLength(
102           pattern[pre+1:],
103           text[pre+1:],
104           maxFuzz-1,
105           UNRESTRICTED)
106         if ((best == None and match.matchCount > 0) or
107             (best and match.matchCount > best.matchCount)):
108           best = match
109           best.preFuzz += 1
110           best.length += 1
111           best.wordsConsumed += 1
112     #end if pre < min nad maxFuzz > 0
113     if best:
114       best.fuzz += 1
115       if pre:
116         best.matchCount += pre
117         best.wordsConsumed += pre
118         best.length += pre
119         best.preFuzz = 0
120     else:
121       best = FuzzyMatch(
122         fuzz=0,
123         length=pre,
124         preFuzz=0,
125         matchCount=pre,
126         wordsConsumed=pre,
127         restrict=UNRESTRICTED)
128     return best
```

Revisiting our example tree in Figure 3.1 and our query string "suffix tree search algorithm" again the search begins the same, with a look-up of "suffix" and recursive call on the child node for the string "suffix tree." Now for a fuzzy match though we scan all of the children of this node and calculate the best match possible for each node. Scanning either child node here results in a zero length best match, so the match must be expanded to the length of the child node (2 words) before the search can continue with the child node. Here we'll

need to recursively search the "exact match" and "fuzzy match" nodes twice, once where both words are considered insertions and once where one word is considered an insertion and the other a substitution which consumes one word from the query text as well. Normally an additional search with 2 substitutions would be attempted however our query text is too short for that. Recursing on either node with two insertions, or with the query text "search algorithm" will find a match for search and return a match. Recursing with one substitution will have the query text "algorithm" which will also find a match. In all we'll find four approximate matches for "suffix tree search algorithm": "suffix tree exact match algorithm," "suffix tree exact match search," "suffix tree fuzzy match algorithm," and "suffix tree fuzzy match search."

Similar Document Searches

Since documents with common sub-strings have similar contents and likely similar topics, it makes a good basis for a document similarity metric. Suffix trees can find common sub-strings quickly, so adding a similar document query makes a natural addition. Similar documents are found by counting the number of common strings between the given document and all the other documents. The common sub-strings are found by walking the entire tree, or at least every branch that matches the given document and at least one other document.

3.1.3 Scoring

A scoring algorithm must be used to rank the search results. The algorithm used depends on the search strategy. The scoring algorithms used here are quite simple. For Exact Sub-String Matching, the score is simply the percentage of

search terms found, or $\frac{matchCount}{\mathsf{query}}$ where $matchCount$ is the number of query terms found and query is the length of, or number of words in, the query. This metric provides adequate ranking since few documents match equally well with any long queries.

For Fuzzy Sub-String Matching the algorithm gets more complicated as it must incorporate a fuzziness factor to rank more accurate matches higher. The equation used is currently $\frac{(K_{max}+1)*matchCount+K_{max}-K}{(K_{max}+1)*\mathsf{query}+K_{max}}$, where K_{max} is the maximum allowed edit distance, K is the actual edit distance between the matched sub-string and the query text. Basically this ranks primarily on how many terms were matched and secondarily on the edit distance between the strings, so all n term matches rank higher than any $n-1$ term matches and for any n the strings closer to the query term rank higher than strings further away, in terms of edit distance.

For computing the document similarity score, I counted the number of occurrences of each length of common sub-strings from $1-n$, and then computed the score as the sum of the number of common sub-strings of length times their length and divided by a constant (100) to bring the range to approximately $[0-1]$. This scoring metric does not guarantee similarity scores are between 0 and 1, but it seems to give decent scores in practice. The data I had did not have large numbers of documents with large common sub-strings, so scaling it so only an exact match got a score of 1 pushed all the other scores below 0.01.

Algorithm 3.4: Similar Document Search

```
1 def simSearch(node, tgt):
2   scores = dict()
3   for child in node.children.values():
4     if len(child.doc_positions) > 1 and tgt in child.doc_positions:
5       recScores = simSearch(child, tgt)
6       for doc in child.doc_positions.keys():
7         if doc == tgt:
```

```
8             continue
9           score = RawSimScore()
10          if doc in scores:
11            score = scores[doc]
12          else:
13            scores[doc] = score
14          if doc in recScores:
15            score.add_scores(recScores[doc], len(child.words))
16          score.add(len(child.doc_positions[doc]), len(child.words))
17    return scores
18
19  class RawSimScore():
20    def __init__(self):
21      self.scores = dict()
22
23    def add(self, count, length):
24      if length in self.scores:
25          count += self.scores[length]
26      self.scores[length] = count
27
28    def add_scores(self, other, prefix):
29      for length, count in other.scores.items():
30        self.add(count, length + prefix)
31
32    def get_score(self):
33      return sum( l*c for l,c in self.scores.items() )
```

Using the same suffix tree from Figure3.1 to illustrate the similar document search algorithm we'll search for documents similar to "suffix tree exact match search" also known as document 0 in the tree. We begin by calling *simSearch* with the root, the document to search for, and 0 for the matched depth so far. At each node we scan its children for nodes referencing the target document. For each node referencing the target and at least one other node, we recursively search it then pad the scores it returned to account for the length of the current child's sub-string and account record matches for any other matched documents as well. Long matches end up being weighted fairly heavily since their sub-matches are also counted.

3.2 Lucene Component

The Lucene search component adapts Lucene's components to the SearchEngine interface, as well as providing a simple document store apart from Lucene's index. It is possible to store the full text of the documents entirely within the Lucene index but it had negative effects on its size and query speed, as well as going against accepted Lucene best practices.

3.3 Caveats

Since Lucene is a mature project no attempt was made to compete in terms of speed or storage efficiency. Most significantly along this line I made no attempt to find an efficient serialized form for a suffix tree instead utilizing Java's built-in serialization mechanism. This decision results in the suffix tree needing to be small enough to be able to be serialized/de-serialized which is smaller than the largest tree representable in the same amount of memory. The comparison with Lucene is primarily to compare the results returned, not how quickly they are returned or how efficiently the index can be stored.

Chapter 4

Evaluation

To investigate the effectiveness of the suffix tree based search index two different evaluation methods were used. First we demonstrated the proof of concept implementation to a group of users to gather their opinions. Second we conducted a more objective evaluation using the Reuters RCV1 data set [8].

The Reuters corpora volume 1 (RCV1) is a collection of Reuters news wire articles from August 20th, 1996 through August 19th, 1997. It contains about 810,000 news stories.

4.1 User Evaluation

To evaluate how well the suffix tree search index works, experiments were set up where it could be directly compared against a Lucene search index. A small number of users were asked to use both, and give their impressions of how well the suffix tree search engine worked for them. The document set was a week's worth of Slashdot stories pulled via their RSS feed.

The results were fairly predictable, namely keyword search seems to work better for most people most of the time. The users surveyed have years of experience with standard keyword based search engines, like Lucene, and are accustomed to how they work. Many expressed difficulty even thinking up phrase queries to make against the data set, that had interesting results. Though the size of the document set and fairly short length of the documents in it limits the number of phrases that have any results beyond single word partial matches.

4.2 Objective Evaluation

The objective evaluation was done with the Reuters data set. The plan was to compare the results with other algorithms and accepted benchmarks, but no comparable uses of the Reuters data set could be found. It is used for many text retrieval tasks but efficacy of document indexing algorithms is not one of them. It still provided a good document set for direct comparison against a Lucene index for a more objective evaluation of the suitability of suffix tree based search indexes.

Here, the proof-of-concept implementation met some setbacks. Due to loading the entire suffix tree into memory, the number of documents that could be stored in a single index was limited by the Java heap size. Interestingly, heap usage peaks during de-serialization, resulting in being able to construct and save suffix trees that could not be loaded again. With a 2GB heap the suffix tree would become too large around 2,500 Reuters articles. This restriction meant only a small subset of the Reuters data set could be used, so separate indexes were built per category, the testing here was done using the index for the science category which contained 2,410 articles.

The suffix tree for the science category still managed some impressive stats. It contained 801,062 nodes with 182,972 non-leaf nodes and 618,090 leaf nodes. 618,090 leaf nodes means there are 256 leaves for each document, or 256 unique suffixes per document, or that the documents averaged 256 words in length (not counting *stop-words*, or small insignificant words like *to*, *or*, or *the*, which were filtered from the documents before they were indexed.) A break down of nodes by depth is presented in Table 4.1 which provides a quick view of the shallow but very broad tree that a suffix tree tends towards.

Depth	Node Count	Non-Leaf Nodes	Leaf Nodes
0	1	1	0
1	31726	18923	12803
2	320467	87542	232925
3	271690	49958	221732
4	118975	18525	100450
5	41104	5811	35293
6	12445	1660	10785
7	3497	430	3067
8	890	99	791
9	212	22	190
10	53	1	52
11	2		2
total	801062	182972	618090

Table 4.1: Nodes by Depth

Some other relevant stats are given in Table 4.2. Of particular interest is the out degree for the root level. Due to large number of nodes directly below the root, the fuzzy match algorithm does not scan the root node's children like it does at other levels. On other levels the out degree is significantly smaller with only 5 nodes with an out degree above 1,000 and a single outlier node in the first level with an out degree of 3,214.

All Node Stats							
Depth	Node Count	Text Length			Out-Degree		
		min	avg	max	min	avg	max
0	1	1	1.00	1	31726	31726.00	31726
1	31726	1	52.20	840	0	10.10	3214
2	320467	1	116.60	884	0	0.85	885
3	271690	1	121.56	879	0	0.44	182
4	118975	1	116.11	804	0	0.35	47
5	41104	1	114.17	660	0	0.30	20
6	12445	1	110.08	604	0	0.28	13
7	3497	1	106.19	499	0	0.25	7
8	890	1	110.62	467	0	0.24	7
9	212	1	99.07	429	0	0.25	7
10	53	1	78.68	289	0	0.04	2
11	2	16	25.50	35	0	0.00	0
total	801062	1	115.37	884	0	1.00	31726

Table 4.2: All Node Stats

Non-Leaf Node Stats							
Depth	Node Count	Text Length			Out-Degree		
		min	avg	max	min	avg	max
0	1	1	1.00	1	31726	31726.00	31726
1	18923	1	4.19	816	2	16.94	3214
2	87542	1	19.90	816	2	3.10	885
3	49958	1	29.55	813	2	2.38	182
4	18525	1	29.65	763	2	2.22	47
5	5811	1	27.62	443	2	2.14	20
6	1660	1	27.30	414	2	2.11	13
7	430	1	23.63	277	2	2.07	7
8	99	1	27.53	251	2	2.14	7
9	22	1	20.86	182	2	2.41	7
10	1	5	5.00	5	2	2.00	2
total	182972	1	22.22	816	2	4.38	31726

Table 4.3: Non-Leaf Node Stats

Leaf Node Stats							
Depth	Node Count	Text Length			Out-Degree		
		min	avg	max	min	avg	max
1	12803	1	123.16	840	0	0.00	0
2	232925	1	152.95	884	0	0.00	0
3	221732	1	142.29	879	0	0.00	0
4	100450	1	132.05	804	0	0.00	0
5	35293	1	128.42	660	0	0.00	0
6	10785	1	122.83	604	0	0.00	0
7	3067	1	117.77	499	0	0.00	0
8	791	1	121.02	467	0	0.00	0
9	190	1	108.12	429	0	0.00	0
10	52	1	80.10	289	0	0.00	0
11	2	16	25.50	35	0	0.00	0
total	618090	1	142.95	884	0	0.00	0

Table 4.4: Leaf Node Stats

The evaluation consisted of generating test queries, querying with the generated queries, and evaluating the results. The test queries were generated manually, trying to find queries with numerous results but without biasing towards one index or another, or picking strings verbatim from the articles. The 20 queries used can be found in Table 4.5.

Querying with the query list was simple to automate and could quickly return results for both the suffix tree index and the Lucene index with varying degrees of fuzz, or tolerance. However, analyzing the results was a time consuming manual process.

Once the results were available, it became immediately apparent that the suffix tree algorithm returned way more results than Lucene and that Lucene's phrase match does not allow missing words from the query string. Due to the latter, the results also compare what a non-phrase Lucene query returns. Table 4.6 shows a comparative summary of the results. Lucene limits its results to 100 items at a time, so it did not return a great number of results.

Number	Query String
1	IBM deep blue super computer
2	human cloning ban
3	ban on human cloning
4	nasa mars mission
5	nasa launches mars global surveyor
6	trouble aboard Mir space station
7	awarded nobel prize
8	dolly the cloned sheep
9	real life jurassic park
10	50th anniversary of roswell alien landing
11	launch manned space flights
12	internet the biggest marketplace on the globe
13	internet the biggest marketplace in the world
14	new top level domains
15	safely dispose of radioactive waste
16	four canisters of radioactive plutonium
17	life on mars
18	russian space probe to mars
19	human form of mad cow disease
20	room temperature superconductor

Table 4.5: Test Queries

Query #	STS		Lucene Phrase		Lucene Normal	
	Total	Length > 1	Total	Not in STS > 1	Total	Not in STS
1	416	11	0	0	100	0
2	475	67	36	2	100	0
3	475	72	35	0	100	0
4	939	179	9	0	100	0
5	923	46	0	0	100	0
6	1118	624	0	0	100	0
7	74	40	8	0	74	0
8	153	101	41	0	100	0
9	522	12	1	0	100	0
10	252	13	0	0	100	0
11	1129	168	1	0	100	0
12	176	3	1	0	100	0
13	604	15	0	0	100	0
14	1240	10	3	0	100	0
15	136	2	1	0	100	0
16	557	33	3	0	100	0
17	508	62	71	9	100	0
18	1147	411	14	0	100	0
19	787	67	5	0	100	0
20	140	3	0	0	100	0

Table 4.6: Comparative Query Results

The Suffix Tree Search results are broken into two categories, the total number of results returned, and the number of results where more than a single word was matched. Both of the Lucene categories were compared against STS to see how many documents Lucene returned that STS did not. Lucene's Phrase query was compared against the STS matches that contained at least two words. In the cases where Lucene found a match that STS missed the matches found were inversions which STS doesn't handle, so STS could only return single word matches. For instance the two matches for 'human cloning ban' that STS missed were: "ban the cloning of human beings" and "ban on cloning human beings." Which would be fixed by handling repositioning in the fuzzy match algorithm.

Analyzing the logged results also provides an opportunity to examine the effectiveness of the fuzzy match algorithm. For some queries there were not many variations that matched, while others returned many similar sub-string matches. The results for "NASA launches mars global surveyor" in Figure 4.1, and "four canisters of radioactive plutonium" in Figure 4.2, each include matches containing several different numbers of matched words and requiring up to 9 extra words to make the matches. Almost all of the queries in the test set exhibit at least one of these features: matching with various sub-strings or matching with many different amounts of extra words interspersed between the matched words; and many of the queries do fairly well on both. Charts of all the query results can be found in Appendix A.

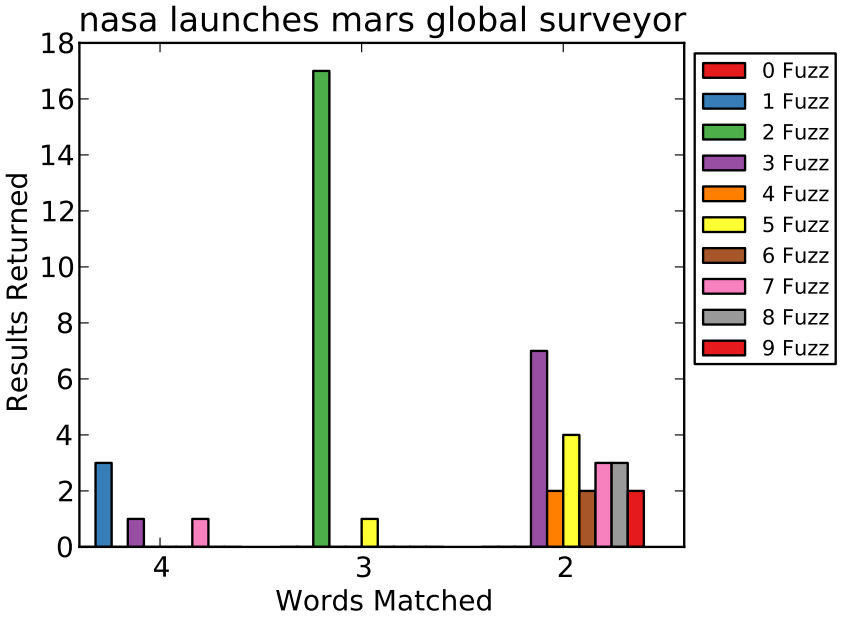


Figure 4.1: Categorized Results for Query 5

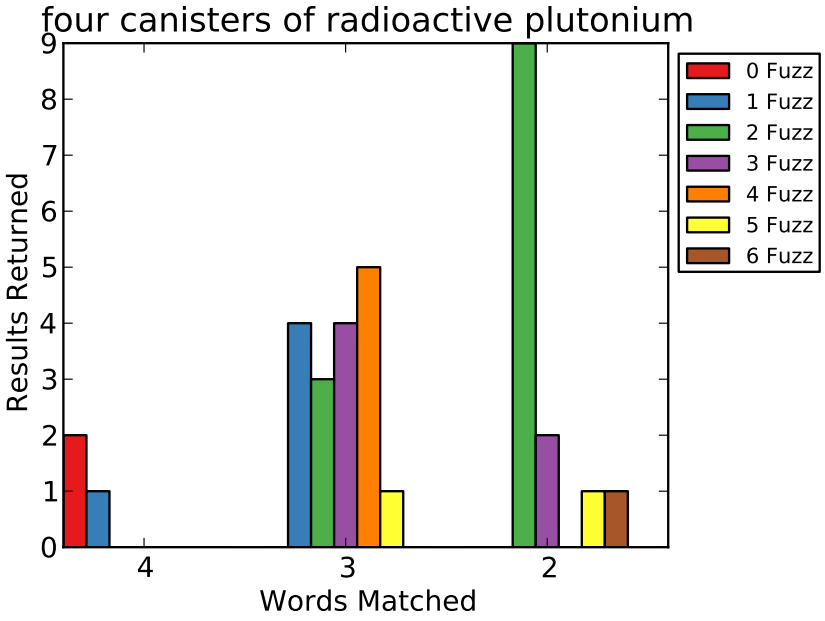


Figure 4.2: Categorized Results for Query 16

Chapter 5

Conclusion

The proof-of-concept implementation proved successful in that the suffix tree indexes were just as capable of returning accurate search results as the standard flat reverse indexes. The main differences between them center around a few main areas: the difference in proof-of-concept and production ready systems, the preserved information in the suffix tree index, and users' familiarity with flat indexes.

Suffix trees can be an alternative to flat reverse indexes, but the advantages are fairly constrained. A suffix tree index handles phrase queries very efficiently, but our limited user evaluation has suggested that users do not use phrase queries very often. Additionally a suffix tree index will always be larger than a flat index, due to the extra information it contains, so performance or memory use might become a problem with larger indexes. Creating a robust suffix tree index implementation would be a lot of work and commercial quality open source implementations of standard flat reverse indexes are readily available.

5.1 Advantages

The primary advantage of a suffix tree based index is the ability for fast efficient phrase searches. In a standard flat reverse index matched terms must be reconstructed into phrases to find a match, but a suffix tree based index can find phrasal matches much more efficiently.

A suffix tree based index contains all the information a flat index does so is at least as capable as a flat index. The added data makes some uses more efficient since that data doesn't have to be reconstructed later, such as when determining word sequence, or searching for phrases.

5.2 Disadvantages

Since a suffix tree based index stores more information than a flat index it will occupy more space.

Limited user testing suggests that phrase queries are not generally used when searching for information. At least when users are looking for unknown information in unfamiliar data sets, they seem to be used to standard term matching search methods. Though under different circumstances than those of the test it remains possible that users could benefit from full phrase matching.

Lastly flat reverse indexes have a long history and are already widely used. It would take quite a bit of effort to get a suffix tree based reverse index to be competitive with these existing projects.

5.3 Possible Applications

Given the trade-offs suffix tree search indexes could be justified for certain applications. Anything where exact or inexact phrase matching is useful, for example an application to check documents for plagiarism, or a search tool to find near-exact quotes from movies, books, or scholarly papers.

I think a plagiarism detection tool would work really well. By adding each term's new papers to the index suspiciously similar papers are easily detectable. Additionally if a user reads a familiar sounding line in a new paper, they can find any previous papers with a similar line quickly and easily. The near matching phrase abilities and common sub-string matching are what really benefits this application.

A near match quote index would be effective as well. A suffix tree index could quickly find near matches for quotes or lines from movies. Also by indexing individual lines, the index would remain shallower and consume less space than one with longer documents. The near matching ability of a suffix tree index really benefits this sort of use since it will find the right quote when you are off by a few words, as long as you have some of them in the right places.

In general a suffix tree search index would be more useful when looking for something similar to the search text, not just something containing the same words. This feature is why the suffix tree indexes showed no advantage over Lucene in the user evaluation: the users were not searching for specific strings but just general term queries. Situations like finding quotes or lyrics, or examining papers for signs of plagiarism, or anything where you are searching for something that closely matches your search text, are the situations when a suffix tree search index would be useful.

5.4 Scoring Algorithm

The scoring as presented in this paper is geared for recall over precision. When comparing it to Lucene phrase searches, Lucene returns fewer results since it is only matching the full phrase and won't return partial phrase matches, whereas the suffix tree based index also returns any partial match it can find. It is left to future research to experiment with the scoring algorithm and achieve a better harmony among precision, recall, and search performance.

5.5 Future Research

Continued research into using suffix trees as search indexes could take several directions. Research into effectively searching a suffix tree from disk would serve to overcome the practical limitations of index size. There is a lot of research into this already, though most schemes exploit the small alphabet size of their strings which doesn't apply here like it does with DNA sequences. Though enumerating all words in the index could have additional advantages as well.

Enumerating the words in the index could help save space by removing duplicate occurrences of a word from the index and replacing it with a unique id. It may be possible to combine this approach with something like WordNet [10] and be able to do synonym based similarity scoring at the same time. But WordNet enumerates synsets and not unique words, so it cannot be used for the word enumeration itself, unless exact matching was not important.

Implementing repositioned fuzzy matching would also improve the search algorithm by accurately finding swapped words from the search string. It might be a fairly simple addition to the implemented algorithm by passing the omitted

and substituted words along in the match, but further research is still necessary.

Research into where to divide documents could provide more space-efficient suffix tree indexes. Currently documents are not subdivided at all, treating the entire document as one long string of words. This allows any sequence of words in the document to be found but consumes a lot of space. An index built on paragraphs, chapters, or even sentences, could provide sufficient search fidelity in many use cases and save enormous amounts of space. I really feel that few user entered query strings will usefully cross a sentence divide, but that's just my intuition. Some uses, such as logs of research papers to detect plagiarism would necessarily use larger divisions of the documents.

Bibliography

[1] H. Bosch and F. J. Kurfess. Information storage capacity of incompletely connected associative memories. *Neural Netw.*, 11(5):869–876, July 1998.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, Apr. 1998.

[3] H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 121–130, New York, NY, USA, 2007. ACM.

[4] S. M. Z. Eissen, B. Stein, and M. Potthast. The suffix tree document model revisited. In *In Proceedings of the 5th International Conference on Knowledge Management*, pages 596–603, 2005.

[5] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, January 1997.

[6] E. Hatcher and G. O. *Lucene in Action (In Action series).* Manning Publications, December 2004.

[7] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[8] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, 2004.

[9] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[10] G. A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[11] A. Singhal. Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4)::35–43, 2001.

[12] F. T. Sommer and P. Dayan. Bayesian Retrieval in Associative Memories with Storage Errors. *IEEE Transactions on Neural Networks*, 9:705–713, 1998.

[13] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 484–492, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

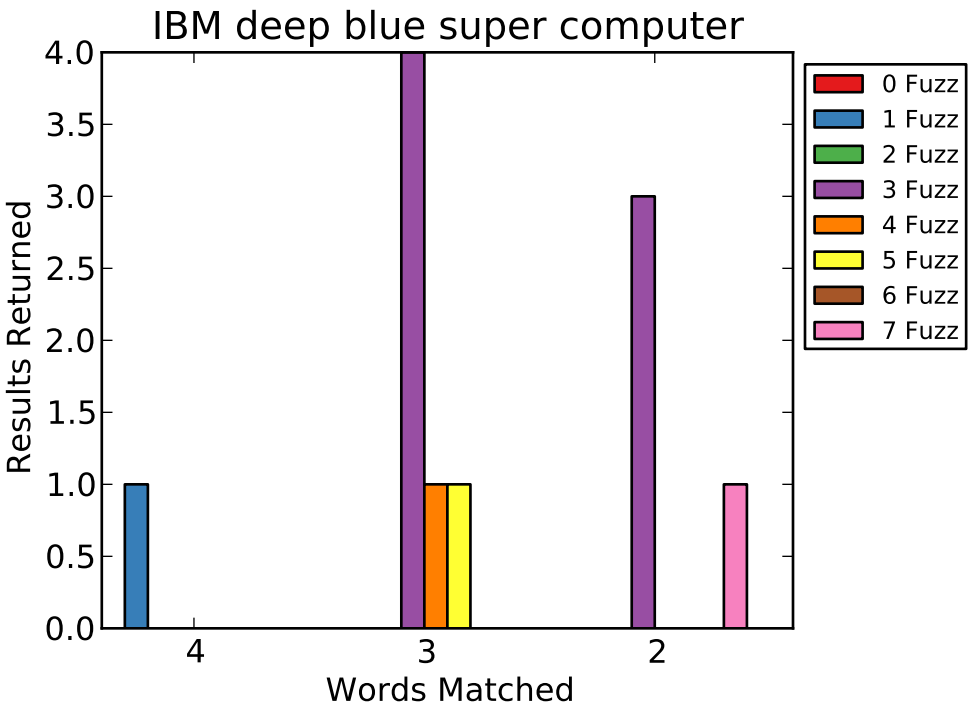
[14] P. Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

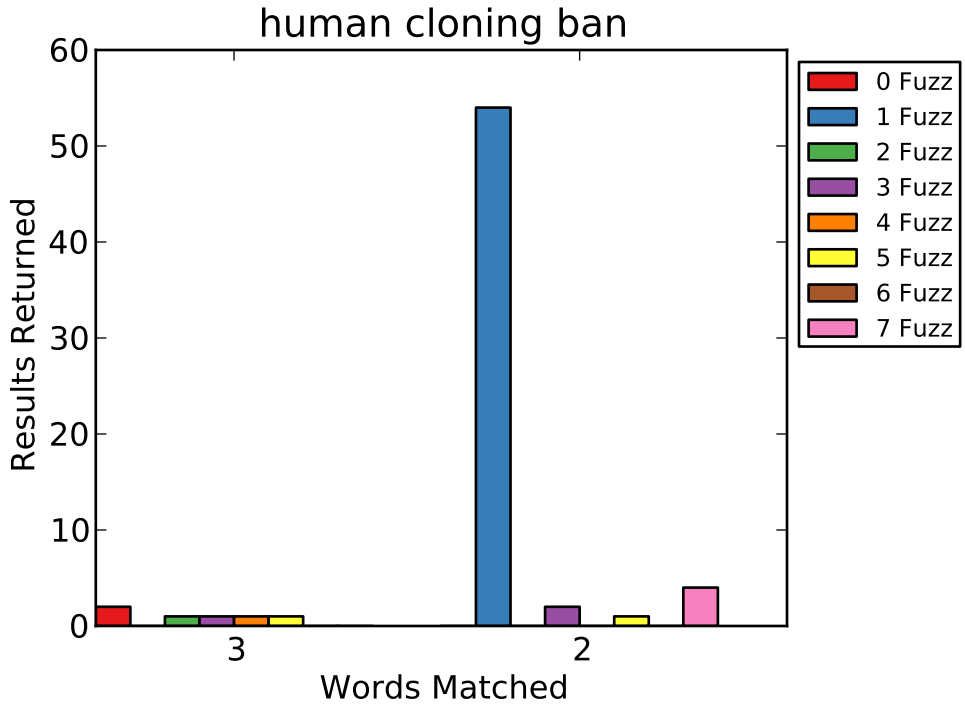
[15] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.

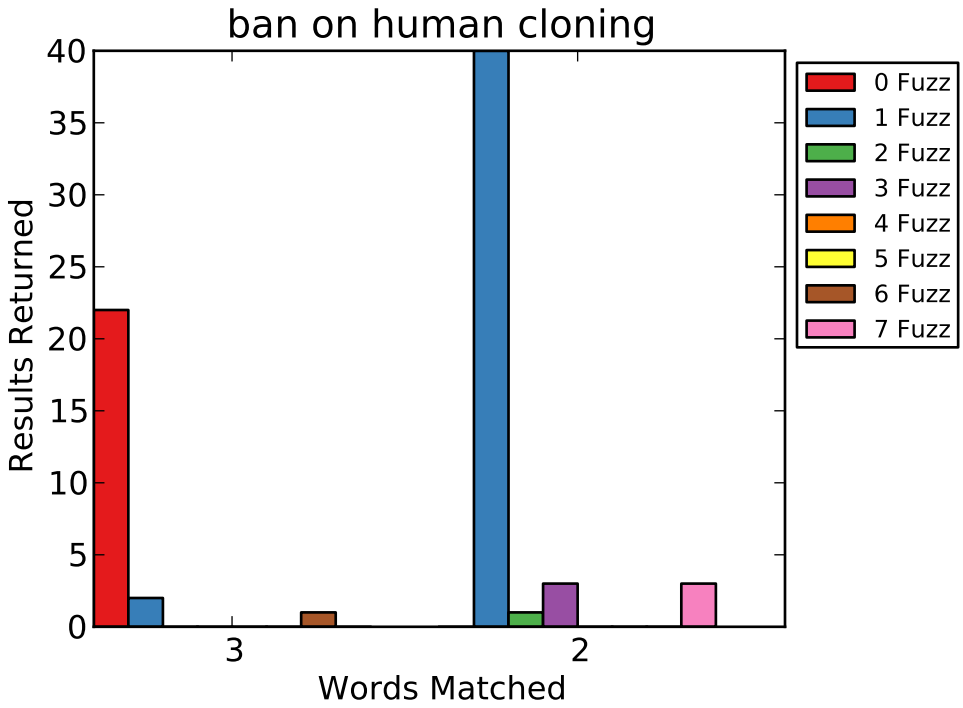
Appendix A

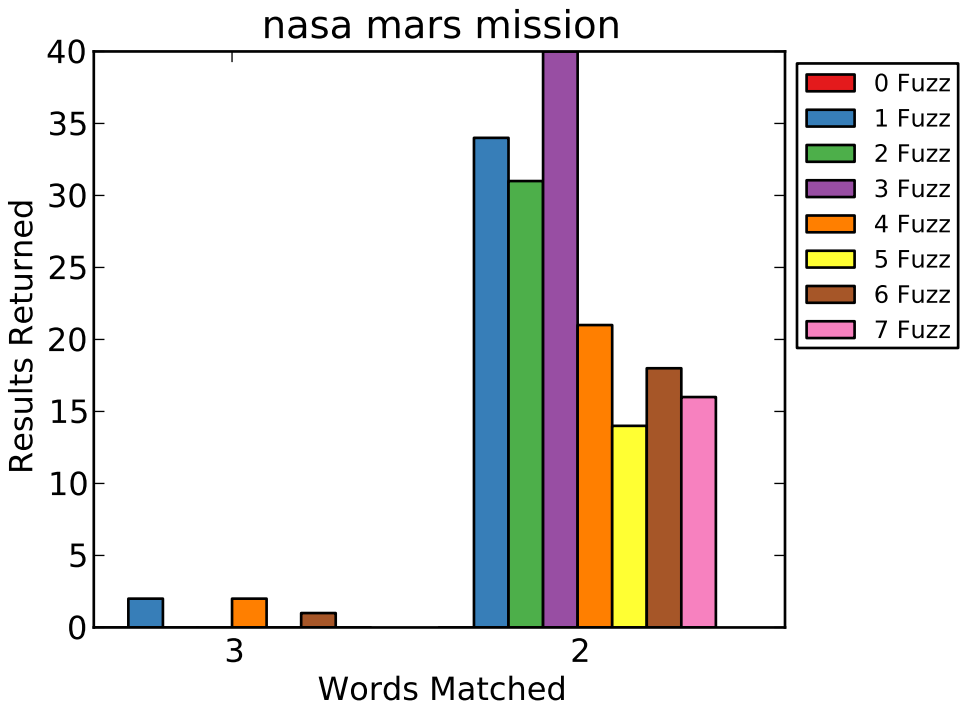
Charts of Categorized Query Results

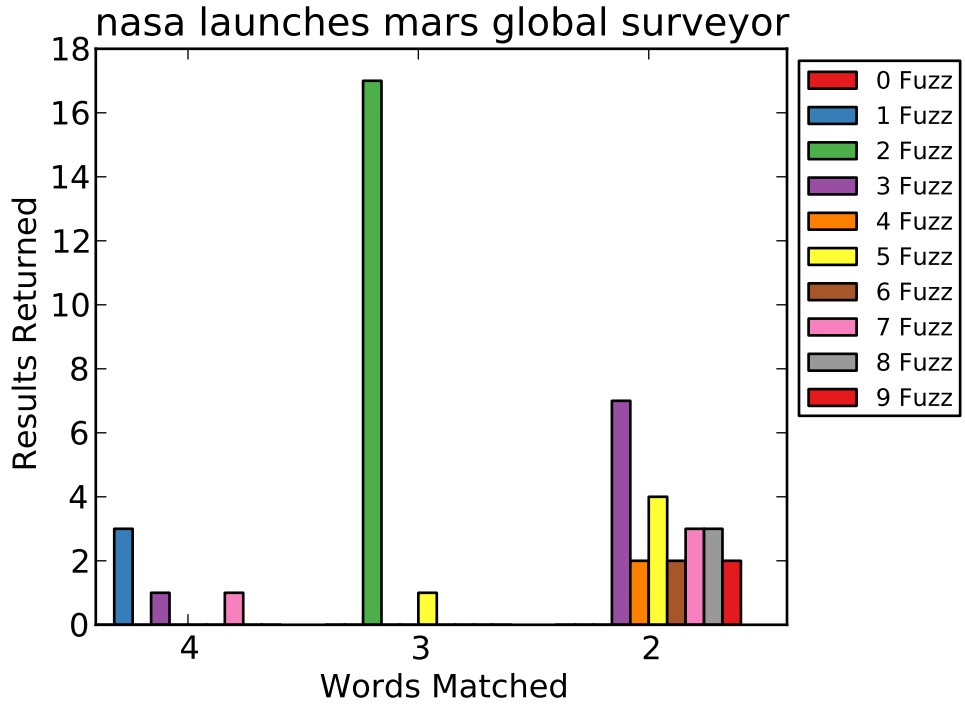
These charts show the number of returned results for each query separated on the x axis by the number of words matched and color coded by the Levenshtein distance between the matched sub-string and the query. In general they show how varying the allowed edit distance can increase the number of returned results.

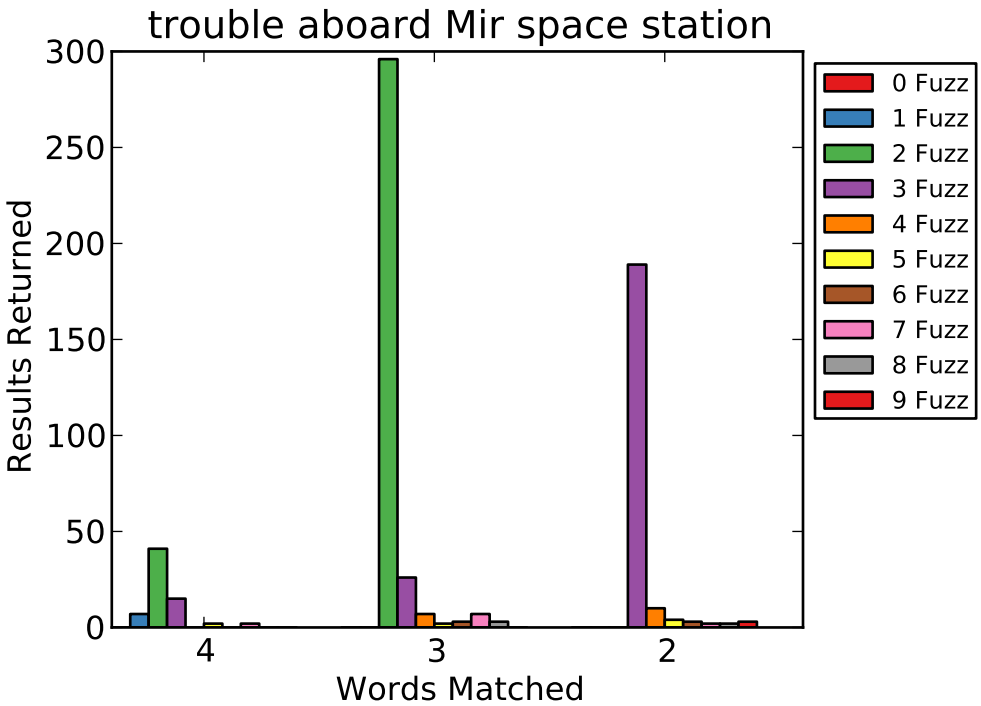

IBM deep blue super computer
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

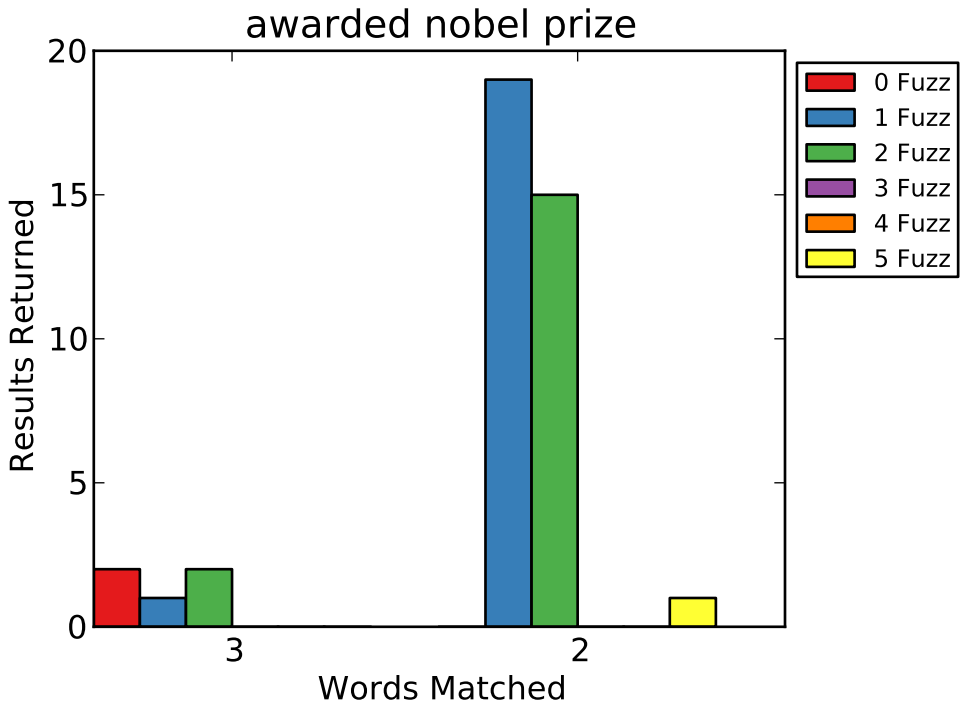

human cloning ban
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

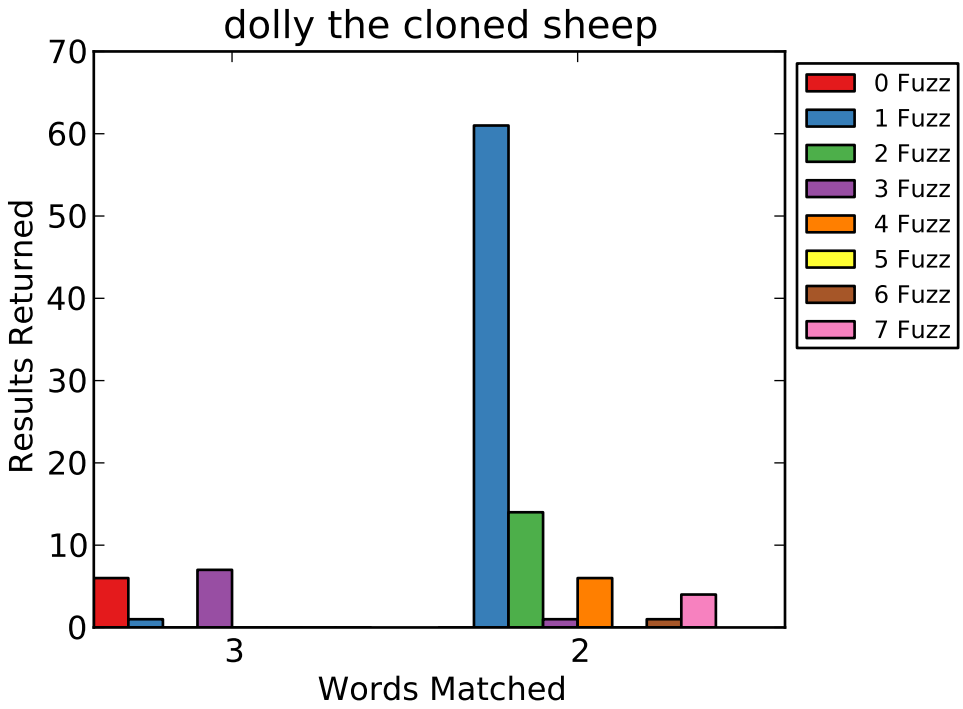

ban on human cloning
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

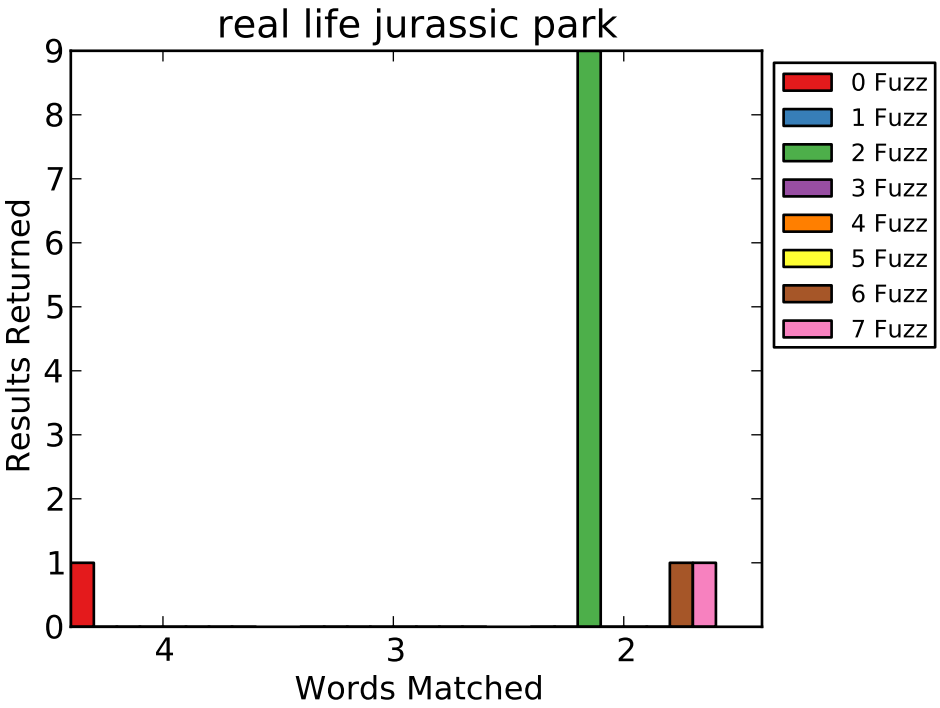

nasa mars mission
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

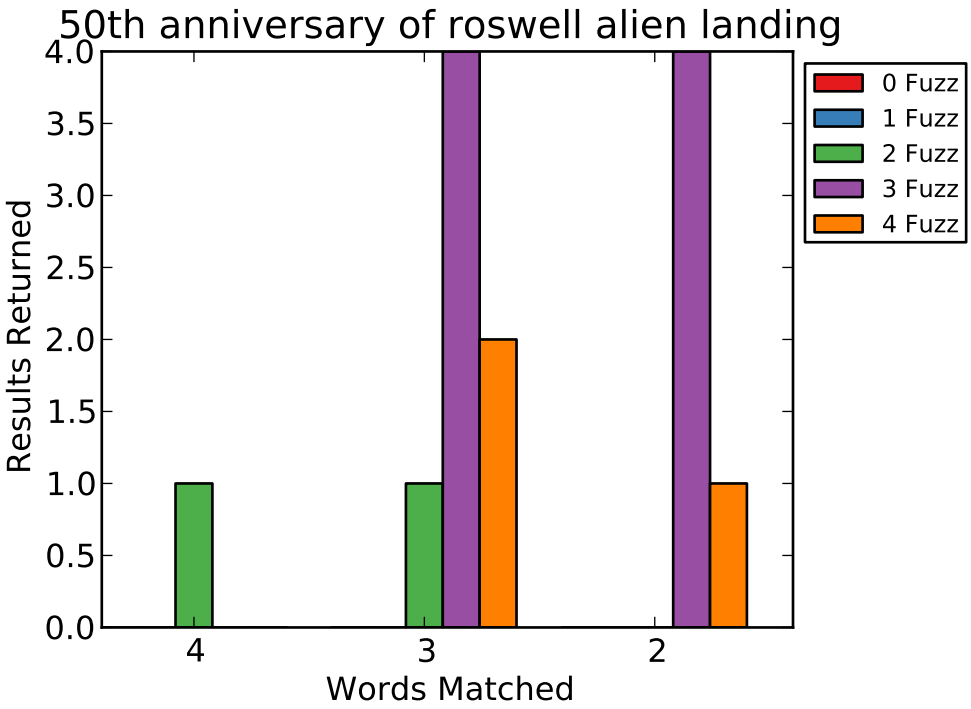

nasa launches mars global surveyor
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz
9 Fuzz

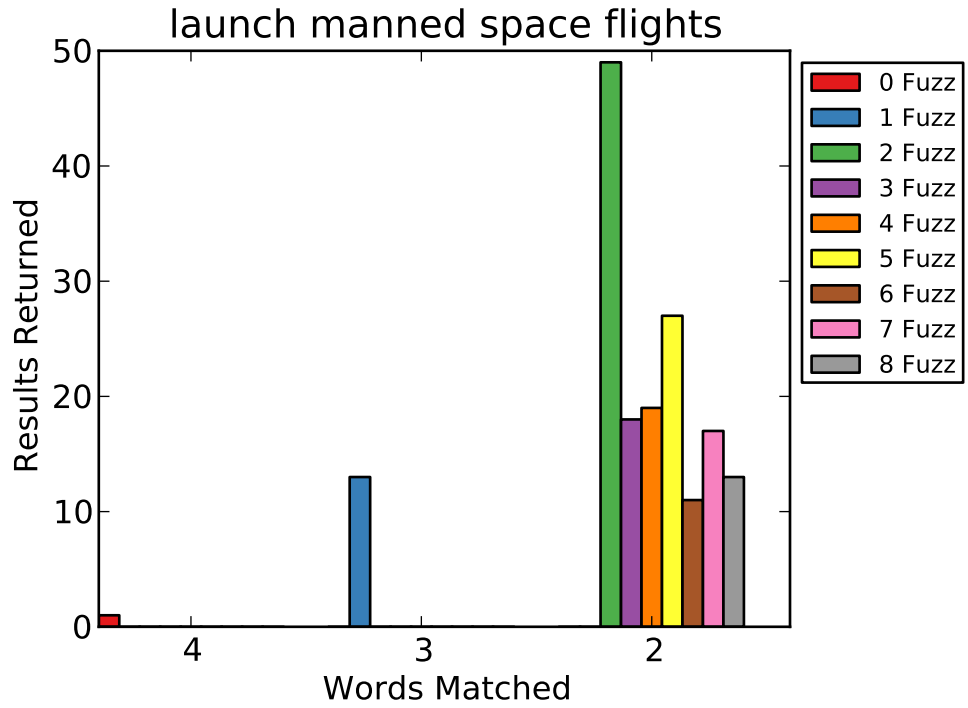

trouble aboard Mir space station
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz
9 Fuzz

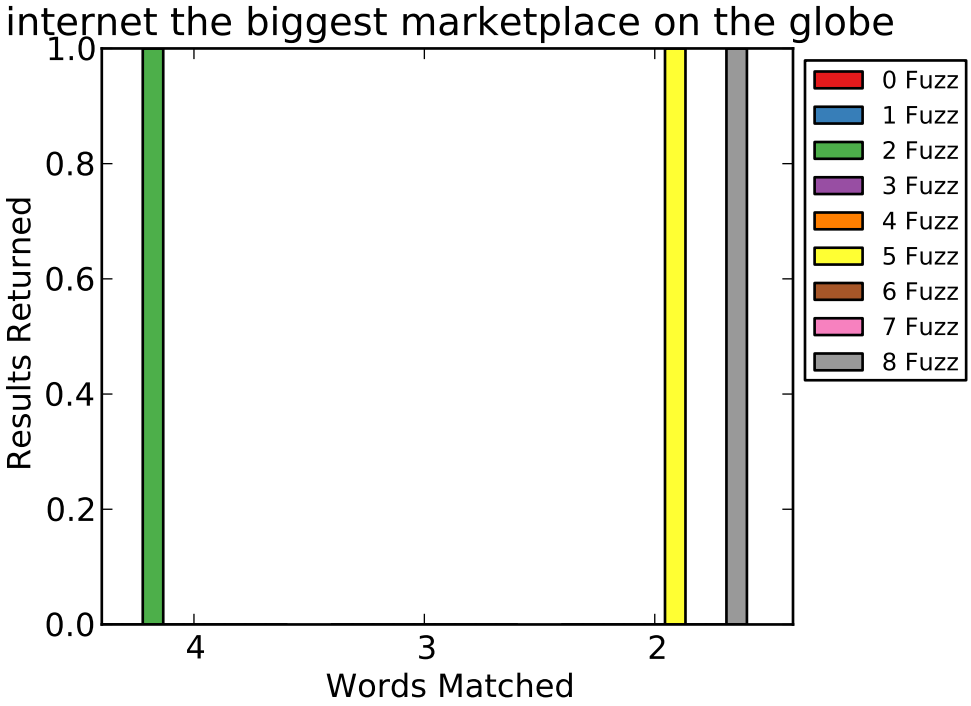

awarded nobel prize
Results Returned
Words Matched
0
5
10
15
20
3
2
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz

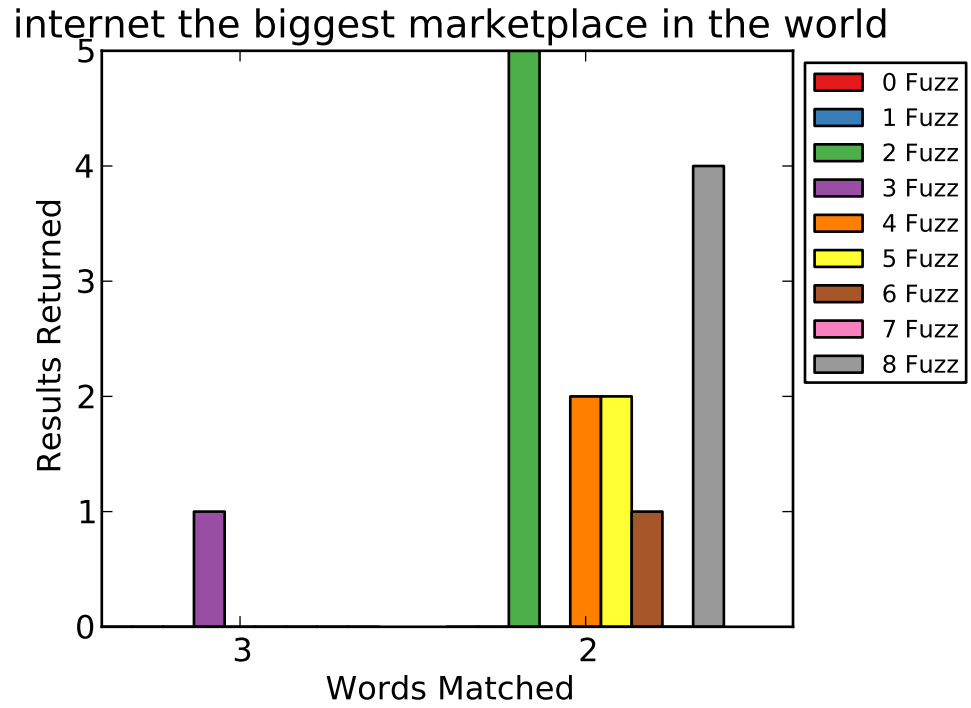

dolly the cloned sheep
Results Returned
Words Matched
0
10
20
30
40
50
60
70
3
2
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

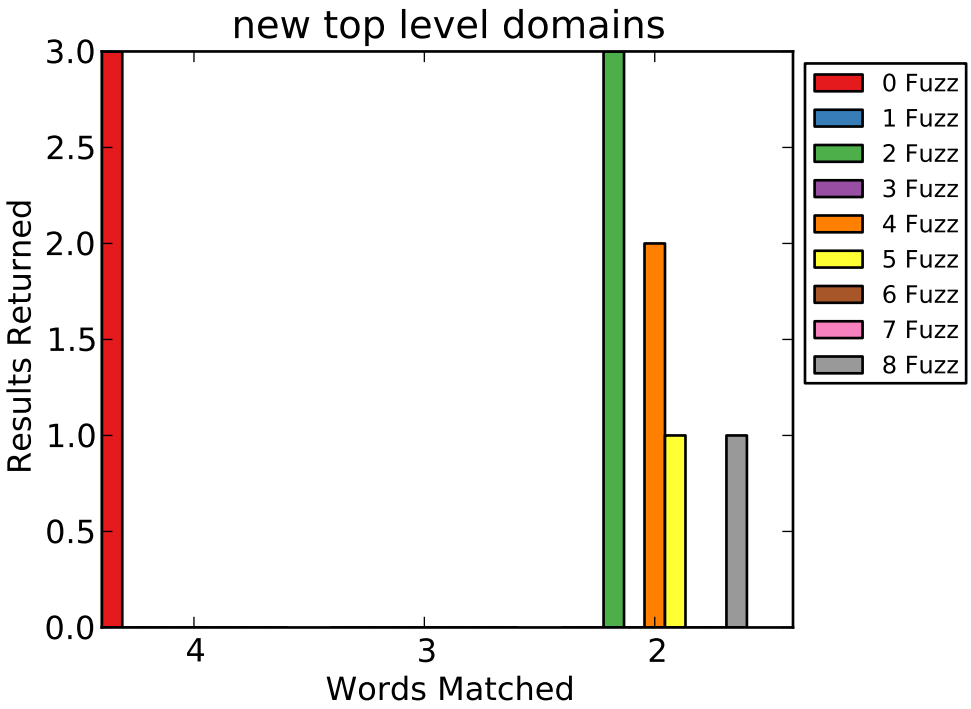

real life jurassic park
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz

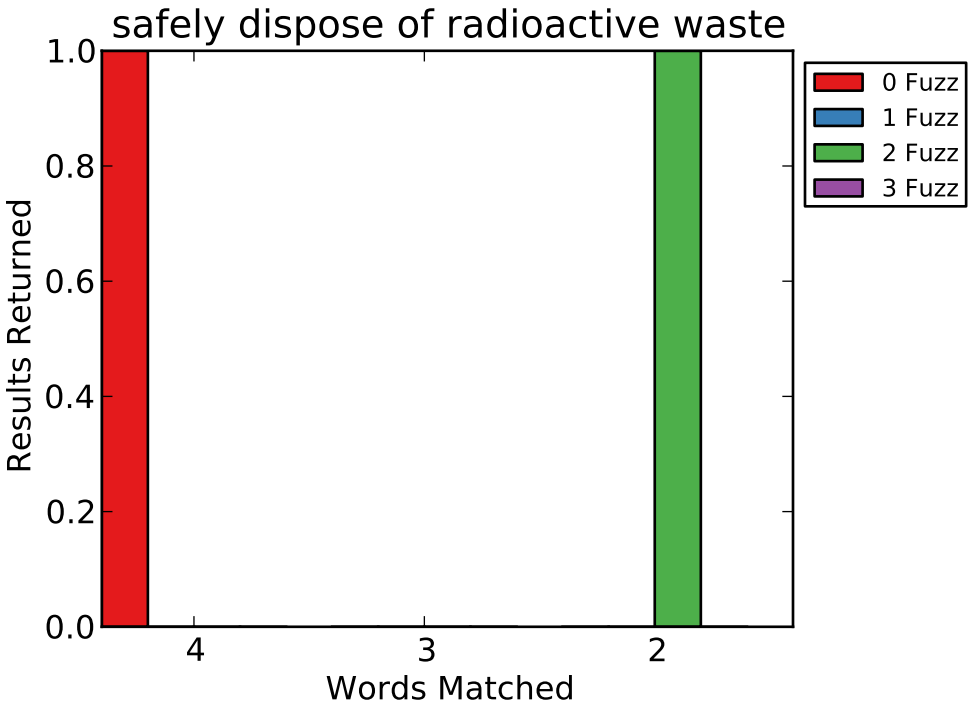

50th anniversary of roswell alien landing
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz

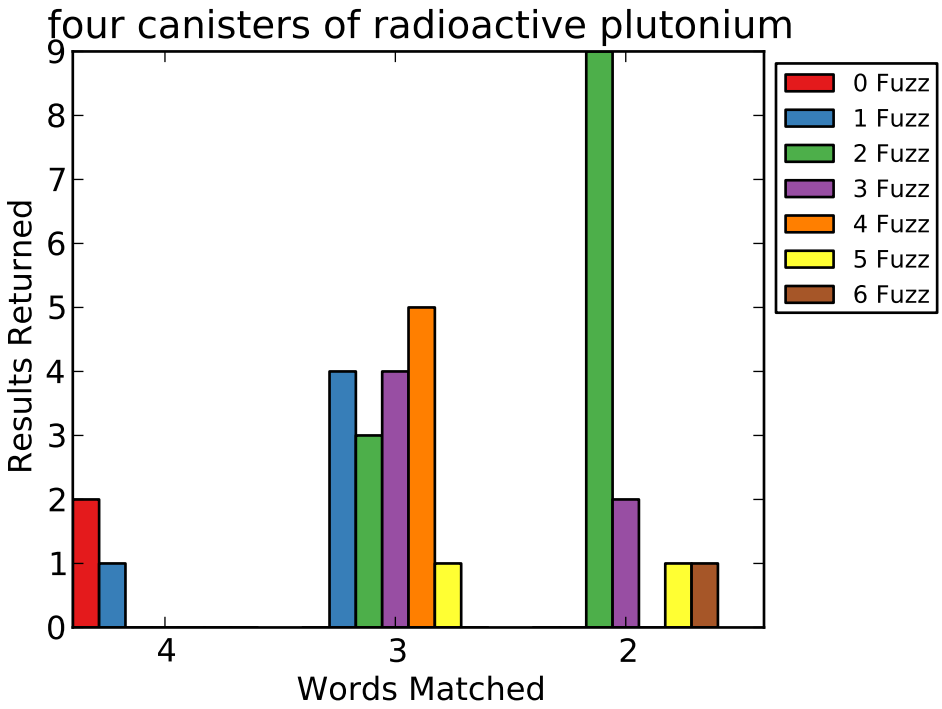

launch manned space flights
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz

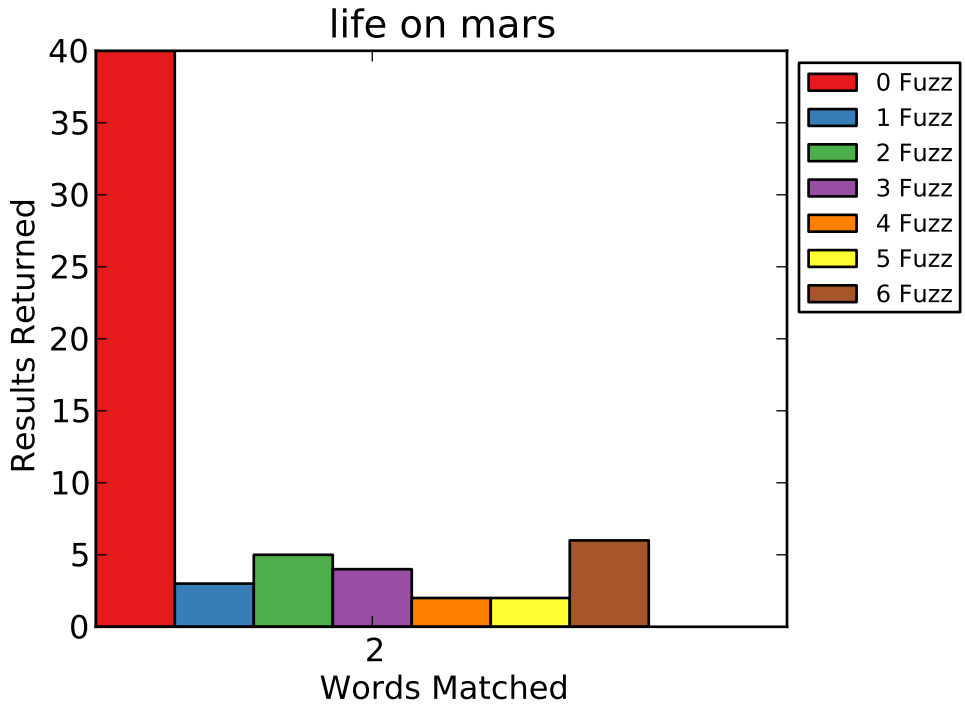

internet the biggest marketplace on the globe
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz

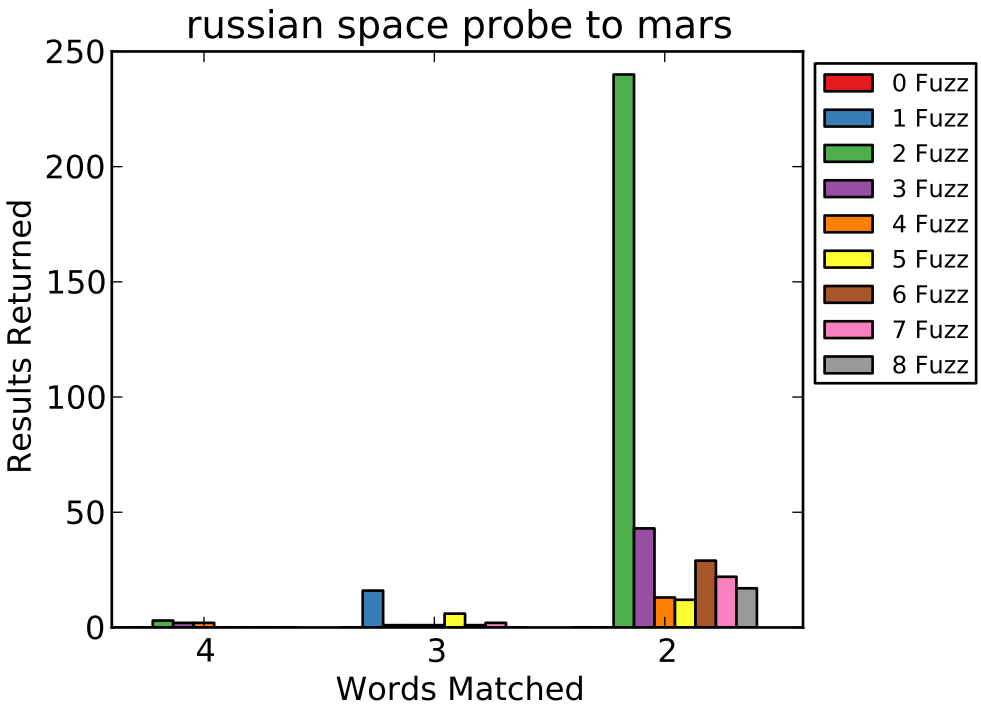

internet the biggest marketplace in the world
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz

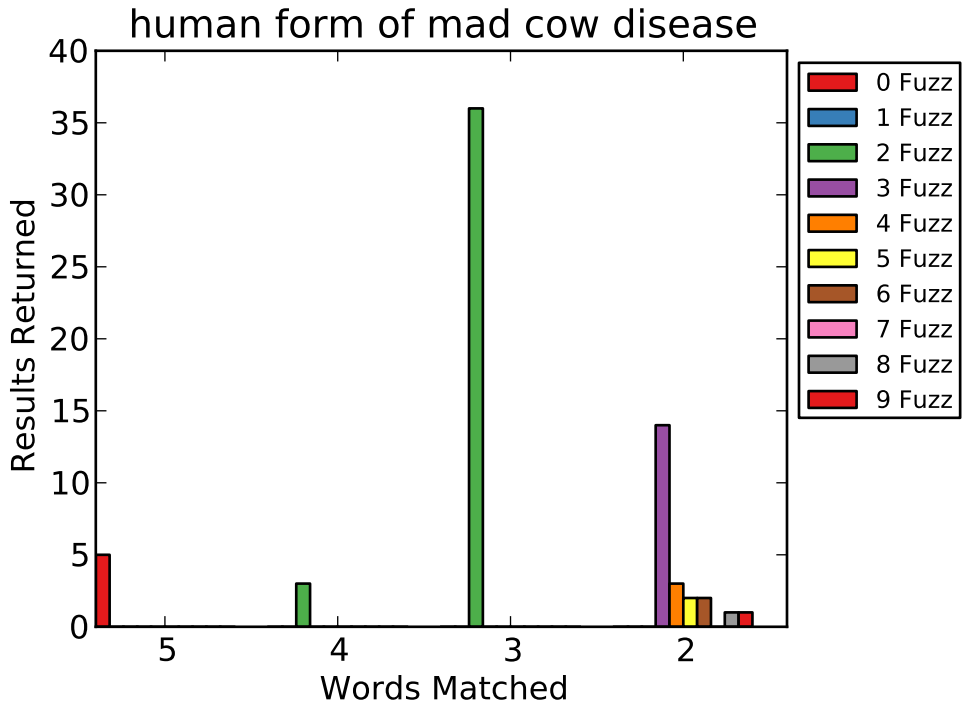

new top level domains
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz

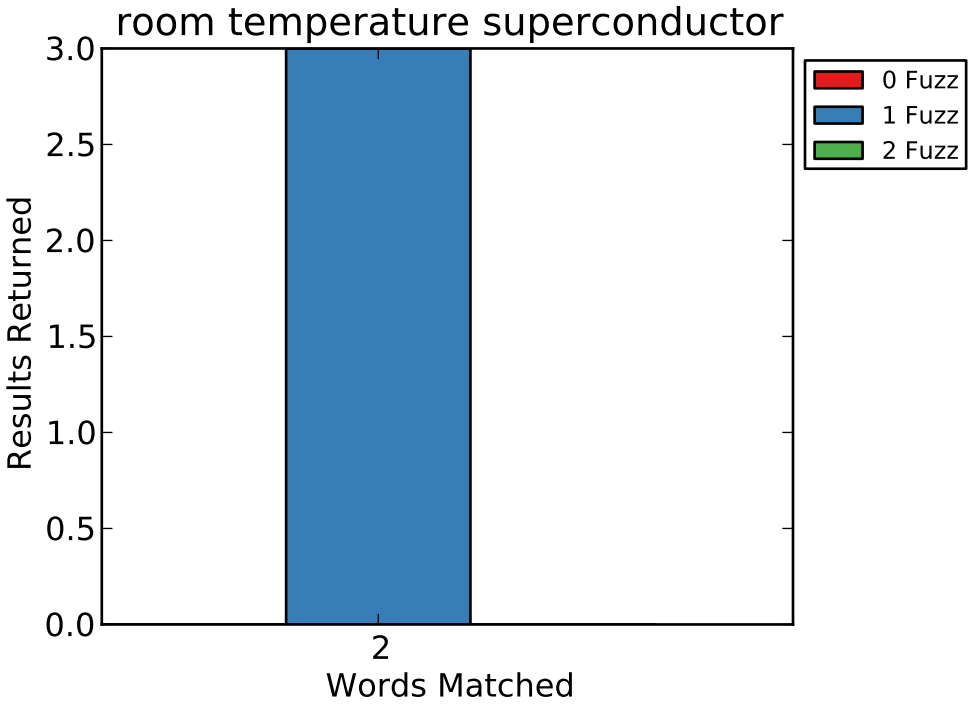

safely dispose of radioactive waste
Results Returned
1.0
0.8
0.6
0.4
0.2
0.0
4
3
2
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz


four canisters of radioactive plutonium
Results Returned
9
8
7
6
5
4
3
2
1
0
4
3
2
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz


life on mars
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz


russian space probe to mars
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz


human form of mad cow disease
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz
3 Fuzz
4 Fuzz
5 Fuzz
6 Fuzz
7 Fuzz
8 Fuzz
9 Fuzz


room temperature superconductor
Results Returned
Words Matched
0 Fuzz
1 Fuzz
2 Fuzz

Appendix B

Supplemental Python Code

Extra code needed to complete the code given throughout Chapter 4. Defines a few missing but straightforward methods and objects. The only remotely interesting pieces of code are the *betterThan* method for keeping the better of any pair of FuzzyMatchLocations, and the *extend* method which extends a match forward since the recursive calls end up building matches from the end forwards.

Algorithm B.1: Supplemental Python Code

```
1  def getLength(loc):
2    start, end = loc
3    return end - start + 1
4
5  def wrapFuzzy(results, node):
6    '''
7    Needs node passed in because rSearch handles the current node
         differently than fuzzyRSearch
8    '''
9    fuzzed = dict()
10   for k,(start,end) in results.items():
11     start += len(node.words)    # undo what rSearch does differently
12     fuzzed[k] = FuzzyMatchLocation(start, end, 0, 0, getLength(
           (start,end) ))
13   return fuzzed
14
15 UNRESTRICTED=0
16 NO_INSERT=1
17 NO_OMIT=2
18
```

```
19 class TreeNode():
20   def __init__(self):
21     self.words=list()
22     self.children=dict()
23     self.doc_positions=dict()
24
25 class FuzzyMatch():
26   def __init__(self, fuzz, length, matchCount, preFuzz,
       wordsConsumed, restrict):
27     self.fuzz = fuzz
28     self.length = length
29     self.matchCount = matchCount
30     self.preFuzz = preFuzz
31     self.wordsConsumed = wordsConsumed
32     self.restrict = restrict
33
34 class FuzzyMatchLocation():
35   def __init__(self, start, end, fuzz, offset, matchCount):
36     self.start = start
37     self.end = end
38     self.fuzz = fuzz
39     self.offset = offset
40     self.matchCount = matchCount
41
42   def extend(self, preLength, match):
43     if match.preFuzz == match.length:
44       newPreFuzz = self.offset + match.preFuzz
45       newFuzz = self.fuzz
46       start = self.start
47     else:
48       newPreFuzz = self.offset + match.preFuzz
49       newFuzz = self.fuzz + self.offset + match.fuzz - match.preFuzz
50       start = self.start - self.offset - match.length +
           match.preFuzz
51     return FuzzyMatchLocation(start, self.end, newFuzz, newPreFuzz,
         self.matchCount + match.matchCount)
52
53   def betterThan(self, other):
54     if self.matchCount != other.matchCount:
55       return self.matchCount > other.matchCount
56     else:
57       return self.fuzz + self.offset < other.fuzz + other.offset
```