English Word Segmentation Problem Comparison of Algorithms

30 November 2011

13509099 **Irvan Jahja** **dolphinigle.mailbox@gmail.com**

Institut Teknologi Bandung

Jalan Ganesha 10,

Bandung 40132, Indonesia

ABSTRACTION

Word Segmentation Problem is the problem of breaking down a long string (presumably a concatenated string consisting of english words) into space-separated English words. We will explain how to solve this problem efficiently using Dynamic Programming problem to achieve O(k*N + M) time O(N + M) space complexity. Then, we will show that by using Extended Suffix Trees, the time complexity can be reduced further to average O(N + M) based on some assumptions by using exact string matching techniques. However, this optimization consumes space and the overhead of using suffix tree is so high that the trie eclipses the suffix tree performance for practical input size.

This paper assumes familiarity with common algorithms, such as Dynamic Programming and Binary Search.

Keywords: bfs, df, dynamic programming, suffix tree, trie

1. INTRODUCTION

1.1 STRING

First, we will formalize the definitions that we will use throughout this paper. A string is a sequence of characters. For the purpose of our paper, a character is defined as either lowercase or uppercase latin alphabet (i.e., 'a'-'z' and 'A'-'Z').

1.2 PROBLEM AND ASSUMPTIONS

Given a string **S** consisting of lower and uppercase letters, and a list of distinct strings **dic** representing the valid English words, return a list of strings such that:

1) The concatenation of the strings in the returned list in the given order is equal to **S**.

2) Each of the strings in the returned list is a member of **dic**.

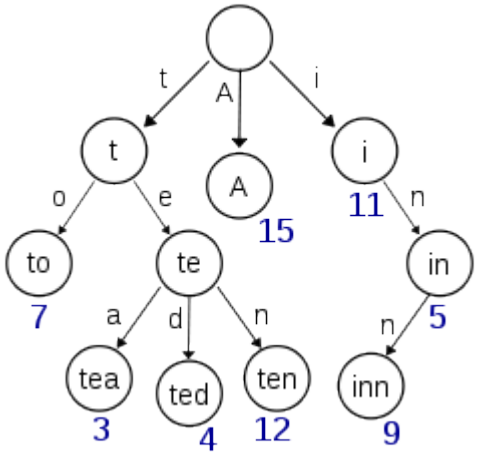
If no such list exists, return an empty list instead. If more than one such list exists, return any of such list.

For example, for a given **S**="catsanddogs", one possible solution is {"cats", "and", "dogs"}. Note that there also exists another solution, namely {"cat", "sand", "dogs"}.

1.3 TRIE

The definition of a trie is as follows:

A trie T for a set of m distinct strings S is a rooted directed tree with at most m leaves numbered 1 to m. Exactly m of the nodes in the tree are marked, all leaves must be marked. Each edge is labeled with a character, no two edges out of a node can have the same edge-label. The key feature of a Trie is that for any marked node i, the concatenation of the edge-labels on the path from the root to node i exactly spells out a member of S.



A trie for : {"A", "to", "tea", "ted", "ten", "i", "in", "inn"} - image courtesy of wikipedia.org

A trie will consume O(k) space, where k is the total number of characters in S. It can be constructed in O(k) space as follows. Start with a tree consisting only a single node: the root. Then, for each T,

member of S, follow the path from the root that spells out the longest possible prefix of T. Then, if it completely spells T, mark the node. Otherwise create several other nodes such that the path from the root to the last of these nodes spells out T. Mark the last node. It is easy to see that with our assumption of constant number of alphabets this works in O(k) time.

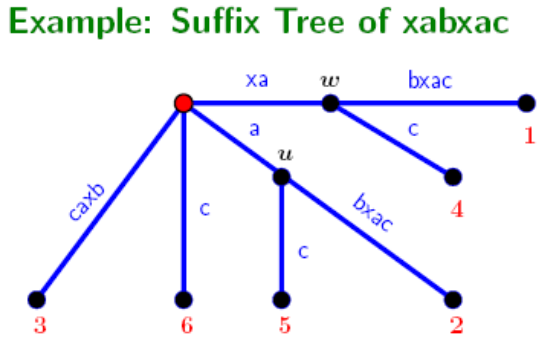
A trie is most useful to find if a particular string T is a member of S. To check that, try to spell as long as possible prefix of T in the trie. If it completely spells T, and the last node is a marked node, then T is one of S. Otherwise it's not. The complexity of such operation is O(m), where m is the number of characters in T. The fact that this complexity does not depend on S is a remarkable property of a trie, but perhaps the most surprising use of a trie is to sort S in O(m) time.

1.4 Suffix Tree

Suffix Trees is a rooted tree that holds information about all the suffixes of a string. More formally:

*A suffix tree T for an m-character string **S** is a rooted 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 substring 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 starts at position i. That is, it spells out S[i..m]. (Gusfield, 1997)*

For example, a suffix tree for xabxac is as follows.



(image courtesy of http://algorithm.cs.nthu.edu.tw/~abnercyh/blog/archives/2006/04/index.html)

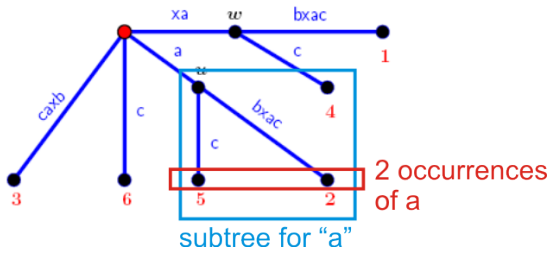
A suffix tree can be constructed in O(m) time, for example, using Ukkonen's algorithm [2]. Interested readers are referred to the excellent explanation by Gusfield in [1].

A particular question must now be answered: how can we represent the suffix tree. If we naively store the label of all the edges explicitly as a string, then the tree will take O(m^2) space (for example, consider the suffix tree for the string: abababab....ab). So, instead of storing them explicitly, we store the labels as two integers i and j, representing the substring of **S** that labels the edge. This way, each edge will take O(1) space.

Another observation is that the tree will have at most O(m) nodes (including internal nodes). This is because it has O(m) leaves, and the out-degree of each internal node is at least 2 (according to the definition of suffix tree). Hence, it can be proven by induction that the number of nodes in the tree is less than 2 * m.

A suffix tree allows many problems to be solved in linear time. For instance, the exact matching problem can be solved in linear time as follows. Construct the suffix tree for the text. Then, follow the path from the root according to the pattern. If a path exists for the entire pattern, then that pattern appears in the text for all the nodes below the end of the path.

Our paper will use the fact that suffix tree can find all the occurrences of a pattern in a text in O(length of pattern + number of occurrences) time (assuming that the text has been preprocessed into a suffix tree). This is achieved by running the algorithm described in the preceeding paragraph, and traversing all the nodes below the end of the path. The key observation is that the number of nodes (including internal nodes) in the subtree below the path is O(K), where K is the number of occurrences of pattern in text. To see this, notice that all the leaves of the subtree corresponds to a starting position for a match between pattern and text. Hence, there are O(K) leaves. It follows that since each of the internal node must have a out-degree greater than or equal to 2 (according to the definition of suffix tree), there are at most O(K) nodes in the subtree. Hence, using either DFS or BFS to traverse the subtree will take at most O(K) time.



Finding all occurrences of "a" in the suffix tree for xabxc

2. Solutions

Both of our proposed solutions uses dynamic programming. However, one will take advantage of trie, while the other will make use of a suffix tree.

2.1 The Dynamic Programming, Naive (O(N^3 log M))

Our dynamic programming will have a state for each integer i between 0 and n, inclusive, where n is the length of **S**. For each state i (we call DP[i]), it will contain the information of whether it is possible to solve the word segmentation problem for the substring S[1..i]. It will also store, if it's possible, another integer j which means that one possible way is by separating S[1..i] into DP[j] + S[j+1..i]. That is, it's for backtracking purpose.

Note that by using this state alone, we already allows a naive O(N^3 log X) (X being the number of words in **dic**) solution, assuming that **dic** is a sorted array of English characters. The pseudocode (we ignore the backtracking for conciseness) is as follows:

```
  Reset DP to all false
  DP[0].is_possible = true
  for i in range(0..n-1):
    if (DP[i].is_possible):
      for j in range(i+1..n):
          if (S[i..j-1] member of
dic):
          DP[j].is_possible = true

  if (DP[n].is_possible):
    Possible
  else:
    Impossible
```

Note that the complexity follows if we use binary search to check the existance of S[i..j-1] in **dic**.

Also, we have to notice that this algorithm makes no use of the assumption that the number of alphabets is small. Hence, this algorithm will still work in O(N^3 log M) time even if the alphabet size grows linearly.

A slight improvement to the above algorithm is immediate. In the second inner loop of the algorithm above, instead of searching in the range (i+1, n), you can search in the range (i+1, min(n, i+k+2)), where k is the length of the longest word in the dictionary. This should work in N * k^2 log M time. We will call this algorithm as Naive+ for the performance results in Section 3.

2.2 Using Trie: O(N*K + M)

We note that the bottleneck of the solution above is in the second loop. That is, the bottleneck is when we check whether a substring originating at i+1 is part of S. But we notice that this is exactly what a trie is best at. Hence, the second solution tries to improve this situation by using a trie. The algorithm looks as follows:

```
  DP[0].is_possible = true, the
others false.
  for i in range(0..n-1):
    if (DP[i].is_possible):
      node = trie.root
      for j in range(1..n-1):
                                    if
!node.has_edge_label(S[j]):
          break
        node = node.edge_label(S[j])
        if node.marked:
          DP[j].is_possible = true

  if (DP[n].is_possible):
    Possible
  else:
    Impossible
```

Now, let's try to calculate its complexity. First, the outer for loop works in O(N) time. Then, the inner loop tries to traverse the trie without ever going back. Hence, there must be at most k steps in the second loop, where k is the length of the longest path from the trie's root to one of its leaves. In other word, k is the maximum length of an English word in **dic**. The length of the longest word found in any major published dictionary is 45 ([4]), hence, this algorithm works in a much better time than the naive algorithm. Furthermore one can observe that although this is the worst case complexity, the average case complexity should be much smaller because there exists only an extremely small number of words that's extremely long.

We have to note that trie implicitly assumes that the alphabet size is small constant. Should it not, traversing a node in a trie takes O(log N) time as opposed to O(1) time. Hence, this algorithm will work in O(N * k * log(N) + M * log(M)) time instead of O(N*k + M).

2.3 USING SUFFIX TREE: AVERAGE O(N + M)

The suffix tree approach imagines the states of the dynamic programming as nodes. A directed edge connecting node i to node j means that S[i+1..j] is contained in **dic.**

Theorem 1: *There is a one-to-one correspondence between a word segmentation problem's solution for a given string S and a given dictionary dic with a path from node 0 to node n in the graph given above.*

Proof:

Consider a path in the graph we constructed from node 0 to node n {0, x1, x2, ..., xm, n}. The corresponding solution is: {1, x1} {x1+1, x2}, ..., {xm+1, n}. From the way we construct the graph, all of these are words in **dic** so this is a solution. Furthermore, no solution will be mapped by two distinct paths for otherwise, let xj be the first node that differs in the path. Then, the {x(j-1) + 1, xj} in both solutions will be different words.

Now, consider a solution: {w1, w2, ... wk}. Let len(w) be the length of the word. Then, construct the path as follows: {0, len(w1), len(w1) + len(w2), ..., len(w1) + ... + len(wk) = n}. Since each of wi is part of **dic**, there will be edges between the nodes in our path. Hence, it's a path in our graph and our proof is complete.

Our solution works by first constructing the graph, then using memoized graph traversal for searching a path from node 0 to node n. We construct the graph by using suffix tree to achieve great performance. First, we construct a suffix tree for **S**. Then, for each word W in **dic**, we look for all occurrences of that pattern in **S**. Then, for each occurrence i, we add an edge from node i-1 to node {i + len(W) - 1}. The pseudocode is as follows.

```
suffix_tree_root = Create(S)
for word in dic:
  int[] occurrences =

suffix_tree_root.FindAll(word)
  for i in occurrences:
    node[i - 1].AddEdge(
      i + word.length() - 1)
```

The complexity of this part of the code is equal to sum of the out-degrees of all the nodes plus the total length of all the patterns. We can bound this by bounding the degree of each node. The maximum possible out-degree of a node is the maximum number of distinct words in **dic** such that for each pair of words in this set, one is a prefix of another. This is, however, an obvious over-estimation over the average case. However this alone already yields a linear time bound for the pseudocode above since that number turns out to be a small constant for the case of English words.

To complete our algorithm, we present a pseudocode for searching a path in the graph. For simplicity, we do not include the backtracking in our pseudocode:

```
boolean FindAPath(int node):
  if node is visited:
    return failed
  else:
    if (node == n):
      return success
    for i adjacent to node:
      if FindAPath(i):
        return success
  return failure
```

The complexity of this part of code is O(n). Hence, the entire code works in O(n + m)

A little note should be made about this complexity. This linear complexity is dependant on the content of **dic**, hence, it's semantic-sensitive. Therefore, the author has opted to use average O(n) instead of regular O(n) for declaring our complexity. Note that there exists a detrimental dictionary that causes this algorithm to use both O(n^2) time and space complexity. For example, a dictionary consisting {"a", "aa", ..., "aaa...aaa"}.

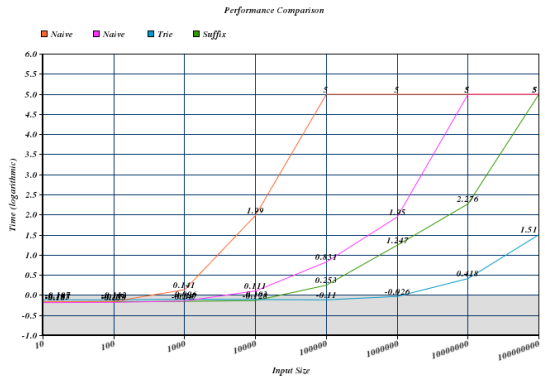
As a final note, suffix tree performance depends on the alphabet size. If the alphabet size is unbounded, constructing a suffix tree takes O(N log N) time and traversing an edge takes O(log N) time. Hence, the algorithm would work in O(n log n + m log n) time in this situation.

3. PERFORMANCE COMPARISON

We compared performance of the three algorithms. For simplicity, we do not implement backtracking because the backtracking extension for all three algorithms, although obvious, is cumbersome and irrelevant to our performance measure. The author is aware that different implementations of the algorithm may yield to different performance and hence the following measurements are not absolute. The author used the list of English words provided for free in [3]. Again for simplicity we only use lowercase letters.

We pick random words from the dictionary until the length exceeds a given threshold. We retry this for all algorithms. The dictionary remains constant. We obtained the following result.

input size	naive (in second)	naive+ (in second)	trie (in second)	suffix (in second)
10	0.696	0.661	0.780	0.686
100	0.708	0.670	0.772	0.692
1,000	1.384	0.724	0.800	0.712
10,000	97.982	1.292	0.788	0.744
100,000	> 600	6.784	0.776	1.792
1,000,000	> 600	90.683	0.940	17.661
10,000,000	> 600	> 600	2.620	189.024
100,000,000	> 600	> 600	32.360	> 600



Performance Comparison of 4 algorithms. Data with value 5 means the time required is > 600 seconds.

We only tried the naive implementation up to 10,000 (We tried 100,000, but it didn't terminate after 10 minutes). Also, we tried the suffix tree only until 10,000,000 because it requires a larger memory than the trie solution.

From the experiment, we observe that all of the implementations takes at least 0.6 seconds. We hypotesized that this is because of the overhead incurred when reading the large dictionary file. Furthermore, the trie implementation has a somewhat higher lowerbound, probably because it requires preprocessing the dictionary's content for any size of text.

We observe that the naive algorithm slows much faster than the other two algorithm, which is in line with the large asymptotic complexity of that solution. However, we notice that even though it's complexity is O(N^3 log N), it still works sufficiently fast for N=10,000. We think this is because most of the dynamic programming state will be unreachable. For example, "cat", both DP[1] and DP[2] will not be reachable because "c" and "ca" are not proper words. This will also help the performance of the two other algorithms.

The suffix tree algorithm is slower than the trie algorithm. For the larger datasets, it's slower by a magnitude. We conclude that this is inline with what we expect since suffix tree construction and suffix tree representation incurs large overhead. Furthermore it requires N * alphabet_size memory (where N is the size of text) as opposed to trie which requires M * alphabet_size memory (where M is the total size of the dictionary). However, we note that the time required for the suffix tree algorithm to increase almost linearly with the size of the text for larger tests.

Surprisingly, trie also achieves this almost linear increase. We hypotesize that this is because the upperbound for trie is rarely achieved.

4. Conclusion

Word separation problem can be solved with all three algorithms: naive, suffix tree, and trie. Trie achieves best performance if the string consists of random English words. The overhead incurred when using suffix tree is high and limits its usefulness for this problem.

The naive algorithm has a special property that the other algorithms don't, its complexity does not depends on the assumption that the alphabet is finite. However, even though the other algorithms depends on this assumption, when this assumption is removed their complexity are only increased by a factor of log(N) or log(M). Hence, they are still faster than the O(N^3 log N) complexity of the naive solution.

The suffix tree algorithm, although theoretically interesting, is slow because suffix tree is slow. Furthermore it requires a large memory which depends directly on the size of the input string. This is contrast with the trie solution which requires large memory which depends directly on the size of the dictionary.

Although the trie solution has a somewhat higher complexity than the suffix tree algorithm, if the length of the longest word in a dictionary is constant, it's actually a linear time algorithm. Hence, if the overhead in the suffix tree algorithm proves too large, the trie algorithm will still work great even for large data.

5. References

[1] Gusfield, Dan. Algorithms on Strings, Trees, and Sequences. 1997

[2] Ukkonen E. On-line Construction of Suffix Trees. Algorithmica - 1995. Springer
[3] http://www.mieliestronk.com/wordlist.html
[4] http://en.wikipedia.org/wiki/Longest_word_in_English

The undersigned hereby declares that this work is a true work of Irvan Jahja and is not a copy or translated version of the work of another party. All parties whose work was referenced are mentioned in the writing.

Irvan Jahja