An Efficient Algorithm for Identifying the Most Contributory Substring

Ben Stephenson

Department of Computer Science, Room 355 Middlesex College, University of Western Ontario, 1151 Richmond Street, London, Ontario, Canada N6A 5B7,
ben@csd.uwo.ca

Abstract. Detecting repeated portions of strings has important applications to many areas of study including data compression and computational biology. This paper defines and presents a solution for the Most Contributory Substring Problem, which identifies the single substring that represents the largest proportion of the characters within a set of strings. We show that a solution to the problem can be achieved with an $O(n)$ running time (where n is the total number of characters in all of the input strings) when overlapping occurrences of the most contributory substring are permitted. Furthermore, we present an extended algorithm that does not permit occurrences of the most contributory substring to overlap. The expected running time of the extended algorithm is $O(n \log n)$ while its worst case performance is $O(n^2)$.

1 Introduction

This paper considers the problem of determining the most contributory substring of a set of strings. We define the most contributory substring to be the string of characters w such that the number of occurrences of w times its length, $|w|$, is maximal. Put another way, the most contributory substring is the string w such that removing all occurrences of w from the set of strings reduces the size of the set by the greatest number of characters. Being able to identify the most contributory substring is useful because it can aid in pattern matching tasks such as analyzing profile data and identifying strings that should be replaced with short code words in compression algorithms.

The remainder of this paper is organized in the following manner. Sections 2 and 3 outline related problems and define the terms used in the remainder of the paper. Our initial algorithm is presented in 4. It is followed by an extended version of the algorithm which ignores overlapping substrings in Section 5. Section 6 discusses applications for this algorithm. We briefly identify areas of future work and summarize in Section 7.

2 Related Work

The most contributory substring problem is related to two other well known problems: the Longest (or Greatest) Common Substring Problem and the All Maximal Repeats Problem (AMRP). However it is distinct from each of these.

While the Longest Common Substring Problem identifies the longest substring common across all strings in a set, the Most Contributory Substring Problem identifies the string that contributes the greatest number of characters. One important distinction between these problems is that there is no guarantee that the most contributory substring will occur in every string in the set while the longest common substring will. The longest common substring problem also ignores multiple occurrences of the same substring within each string in the set while the Most Contributory Substring Problem counts every occurrence.

A variation on the Longest Common Substring Problem has also been presented [4] that identifies the longest common substring to k strings in a set. This problem is also distinct from the Most Contributory Substring Problem because it fails to consider the value of multiple occurrences of a substring within one string in the input set.

The All Maximal Repeats Problem identifies occurrences of identical substrings α and β in a string s such that the characters to the immediate left and right of α are district from the characters to the immediate left and right of β. In contrast, the Most Contributory Substring Problem is concerned with identifying strings that occur many times. Each occurrence of the string may or may not have distinct characters to its immediate left and right.

3 Definitions

A substring w of a string s is defined to be a string for which there exist strings (possibly empty) p and q such that $s = pwq$. A suffix, w of a string s is defined to be a non empty string meeting the constraint $s = pw$ for some (possibly empty) string p.

Let $\mathcal{L}$ represent the set of m strings under consideration. We will denote a specific string within the set as $\mathcal{L}_i$ such that $0 \leq i < m$. Let $\Sigma_0 ... \Sigma_{m-1}$ represent the alphabet employed by each of the m strings in $\mathcal{L}$. The alphabet used across all strings is constructed as $\Sigma_0 \cup ... \cup \Sigma_{m-1}$ and will be denoted by Σ.

A generalized suffix tree is a data structure constructed from a collection of strings. An equivalent tree can be constructed for a single string that is the concatenation of all of the strings in the set provided that a unique *sentinel* character is placed after each string. The set of sentinel characters is denoted by $\mathcal{S}$. In our examples we will represent sentinel characters with the punctuation marks # and \$. We will use the symbol s to represent the string generated by concatenating the strings in $\mathcal{L}$ with a distinct sentinel character between each string. Because a sentinel character is introduced after each input string during the construction of s, $|\mathcal{L}|$ is equal to $|\mathcal{S}|$.

Several linear time and space suffix tree construction algorithms have been developed [6, 7, 8, 9] for alphabets that are a small, constant size compared to the length of the input string. Another algorithm was developed subsequently that provided the ability to construct a suffix tree in linear time and space over integer alphabets [3]. Assuming that the string ends in a sentinel character, the following properties hold for a suffix tree independent of the algorithm used to construct it:

1. The tree contains n leaves where each leaf corresponds to one of the n suffixes of a string of length n.
2. Each branch in the tree is labeled with one or more characters from the string. The label is represented by two integers which specify the starting and ending position as indices in the concatenated string. Collecting the characters as one traverses from the root of the tree to a leaf will give one of the suffixes of the string.
3. The first character on each branch leaving a node is distinct. As a result, no node has more than $|\Sigma| + |\mathcal{S}|$ children.
4. Each non-leaf node has a minimum of two children.
5. The string depth of a leaf node is the length of the suffix of s represented by that node. The string depth of an interior node is the length of the prefix that is common to all of the suffixes represented by leaf nodes below it.

Figure 1 shows a suffix tree constructed for the strings *ababab* and *abab* after they have been merged into a single string and the sentinel characters have been added. The suffix generated by traversing the tree is shown in each leaf node.

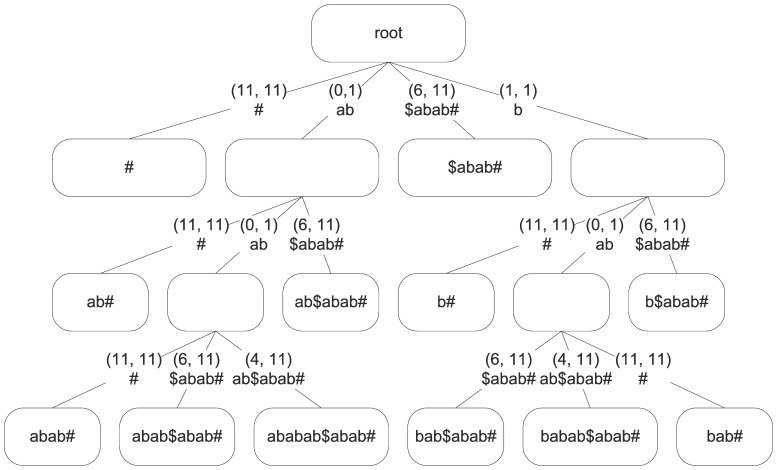

Fig. 1. Suffix Tree for the string *ababab*$*abab*#

4 Initial Algorithm

This section presents an initial algorithm for solving the most contributory substring problem which permits overlapping occurrences of the identified string. It is subsequently extended to consider only non-overlapping occurrences of the most contributory substring in Section 5.

Our algorithm for determining the most contributory substring begins by creating a suffix tree for s using one of the linear time construction algorithms published previously. Once the suffix tree is constructed it is transformed so that all of the strings in $\mathcal{L}$ are represented by an interior node, and so that all strings containing a sentinel character are represented by a leaf node. Any string containing a sentinel character occurs due to the construction of s. Consequently,

the identification process can disregard any string represented by a leaf node while ensuring that every substring of a string in $\mathcal{L}$ is considered. The following steps are taken to transform the tree to meet these constraints:

- Any leaf node that is reached by a branch label that begins with a sentinel character is left untouched.
- Any leaf node that is reached by a branch labeled with a string that begins with a letter in Σ is split into two nodes. A new interior node with only one child is inserted between the leaf and its parent. The branch to the new node is labeled with all characters before the first sentinel character in the original branch label. The branch from the new node to the leaf node is labeled with the remaining characters.

Once this transformation has been performed the number of nodes in the tree has increased by at most n, retaining a total number of nodes that is $O(n)$. The algorithm employed is shown in Figure 2. The effect of this transformation can be seen in the bottom-most nodes in Figure 3.

The SplitLeaves transformation can be performed in $O(n \log |\mathcal{S}|)$ time if the positions of the sentinel characters are recorded when the concatenated string is formed. Then finding the position of the first sentinel character can be implemented as a binary search for the first value in the sentinel character position list that is greater than the starting index of branch `b`. This strategy requires $O(s)$ additional space to store the list of sentinel characters.

An alternative strategy is to build a table which maps each position in s to the position of the next sentinel character. This table can be constructed in linear time during the construction of s and requires additional space that is $O(n)$. With this table, finding the position of the first sentinel character in the label be can be accomplished in constant time. This results in overall time complexity for SplitLeaves which is $O(n)$. Which of these strategies is superior depends on the size of $\mathcal{S}$ and the trade-off between space and time costs for the specific context in which the algorithm is employed.

A depth first traversal of the tree is performed once the leaf nodes have been split. This traversal assigns a score to each node, determined by computing the product of the string depth of the node and the number of leaf nodes below it. Depending on the application, this score can be recorded in the node and utilized later or two variables can be used to record the best string and score encountered so far.

Figure 3 shows Figure 1 after scoring has been completed. It shows the values used to compute the score in addition to the score awarded for each node. From this diagram we can see that the node corresponding to the string *abab* received the highest score with the value 12. As a result, we would conclude that the string *abab* is the most contributory substring for the input strings *ababab* and *abab*. The occurrences of the string are indicated below using underlining and over-lining.

$$\underline{ab\overline{ab}}\overline{ab}\$\underline{\underline{ab}ab}\#$$

```
Algorithm SplitLeaves(node n)

For each branch, b, leaving n
  If the node, m, at the end of b is a leaf and the label on b starts with a
  character in Sigma
    Create a new node, p
    Create a branch, c, from p to m
    Change the target of branch b to node p

    Find the position, x, of the first sentinel character in the label on b

    Label c with all characters from x to the end of the string
    Change the label on b so that it only includes those characters before x

  Else
    SplitLeaves(b.Target)
End For
```

Fig. 2. Algorithm SplitLeaves

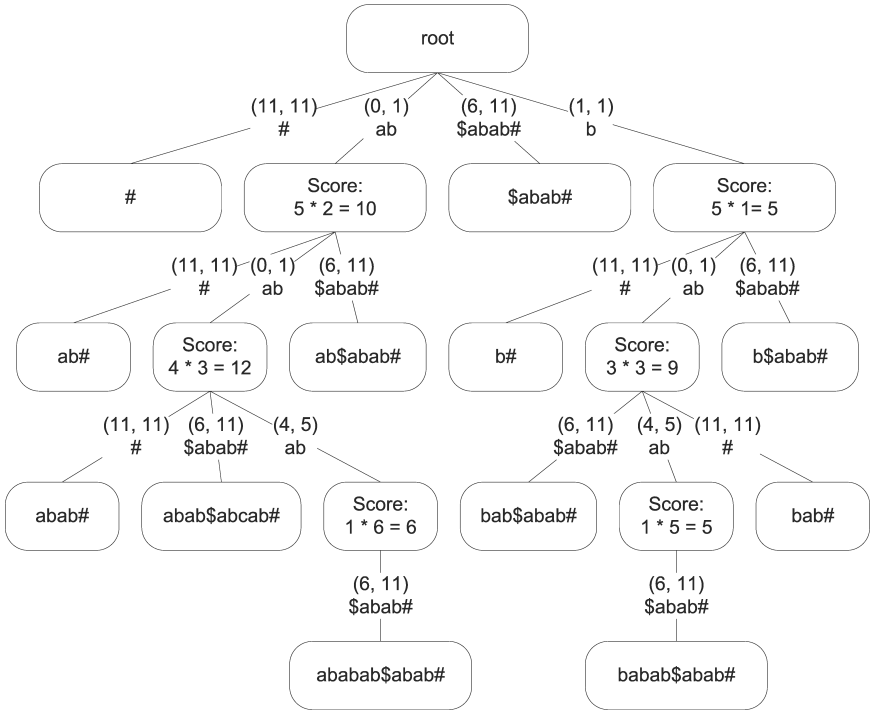

Fig. 3. Suffix Tree for the string *ababab*$*abab*# Including Scores

5 Handling Overlapping Occurrences

It is important to observe that the single underlined and single over-lined occurrences of *abab* are overlapping. While this may not be a problem in some situations, any application that 'uses up' the string once it is identified will destroy the subsequent overlapping occurrence(s). Consequently the current identification technique will over estimate the value of some strings if overlapping should be prohibited. This section extends the preceding algorithm so that it detects and ignores overlapping occurrences of the string.

Two occurrences of a substring are known to overlap if the absolute value of the difference in their starting positions within s is less than the length of the substring. We annotate each interior node in the tree with the score based on the number of non-overlapping occurrences of the substring represented by the node using the depth first traversal algorithm shown in Figure 4.

During this traversal, a balanced binary search tree is constructed for each node in the suffix tree. It contains the starting indexes of the substring represented by the node in the suffix tree. For leaf nodes, the binary search tree consists of a single node. At each interior node, the balanced binary search trees from its children are merged to form a new larger tree. Each tree from a child is merged in sequence, with the smaller trees being merged into the largest tree.

In the pathological case where the height of the suffix tree is n, the amount of time required to merge the binary search trees is $O(\log n)$ for each level in the tree because only one new node is merged, giving an overall complexity of $O(n \log n)$. A similar situation results in the pathological case where $n = |\Sigma| + |\mathcal{L}|$. In this case, the height of the suffix tree is 1, but n leaf nodes must be merged into the tree at a cost that is $O(\log n)$ for each merge. As a result, this case also has a complexity which is $O(n \log n)$.

A myriad of suffix trees exist with heights between the pathological cases described in the previous paragraph. However, the amount of time required to construct the balanced binary search trees is also $O(n \log n)$ for these cases. The number of merges is minimized by merging the smaller trees into the largest tree at each merge point, resulting in each merge still having a cost that is $O(\log n)$. Due to the construction of a suffix tree, it is known that the size of the largest binary search tree will grow by at least one at each level in the tree (since each interior node in the suffix tree has at least two children except for the split leaf nodes). When the binary search tree grows by only one element at each level, the tree has height n and the overall time required to build the binary search trees is $O(n \log n)$ as described previously. If the size of the largest binary search tree grows by more than 1 at any point, then the height of the suffix tree is known to be less than n. In fact, if the increase in size of the largest binary search tree is denoted by Δ_j at level j then the maximum height of the suffix tree is $n - \sum(\Delta_j - 1)$, where $\sum$ indicates summation in this instance. While any level that has a value of Δ_i that is greater than 1 requires multiple merges that each cost $O(\log n)$, the additional merges performed at that level are offset by the decreased height of the tree, resulting in an overall running time that remains $O(n \log n)$.

```
Algorithm NonOverlappingScore(node n)

ChildTrees: an array of pointers to binary search trees
Retval: a pointer to a balanced binary search tree

If n.NumChildren == 0  // it's a leaf node
  Retval = new binary search tree with one node
  Set the value in Retval's root node to |s| - n.StringDepth
  Return Retval
End If

Allocate n.NumChildren pointers for ChildTrees  // Proceed depth first
For i = 0 to n.NumChildren - 1                  // through the suffix tree
  ChildTrees[i] = NonOverlappingScore(n.Child[i])
End For

For i = 1 to n.NumChildren - 1 // Ensure ChildTrees[0] points to largest tree
  If ChildTrees[0].num_nodes < ChildTrees[i].num_nodes
    swap(ChildTrees[0], ChildTrees[n])
  End If
End For

Retval = ChildTrees[0]
For i = 1 to n.NumChildren - 1 Retval = BSTreeMerge(Retval, ChildTrees[i])
n.Score = CountUniqueOccurrences(Retval,n.StringDepth) * n.StringDepth

Deallocate the binary tree for each child and the ChildTrees array
return Retval
```

Fig. 4. Algorithm NonOverlappingScore

```
Algorithm CountUniqueOccurrences(BinarySearchTree t, Integer str_len)

count = 0                // number of non-overlapping occurrences
last_position = MIN_INT  // last_position is initialized to the
                         // negative value of largest magnitude
InOrderTraversal(t.root, count, last_position)
return count


Algorithm InOrderTraversal(BSTNode b, Ref Integer count,
                           Ref Integer last_position)

If (b.left != NULL) InOrderTraversal(b.left, count, last_position)
If ((last_position + str_len) <= b.value)
  count++
  last_position = b.value
End If
If (b.right != NULL) InOrderTraversal(b.right, count, last_position)
```

Fig. 5. Algorithm CountUniqueOccurrences

Once the binary search tree for a node is constructed, it is traversed using algorithm CountUniqueOccurrences, shown in Figure 5. This algorithm is responsible for determining if two or more of the occurrences of the substring represented by suffix tree node `n` overlap. This is accomplished by traversing the balanced binary search tree using an in-order traversal, counting only those nodes whose value differs by at least `length`. This value is returned to the NonOverlappingScore algorithm, which uses it to compute the score for the node.

Since each starting position can only exist in one binary search tree at each level in the suffix tree, the total time required to traverse all of the binary search trees at a level is $O(n)$. Unfortunately, this leads to worst case running time for CountUniqueOccurrences which is $O(n^2)$ in those rare cases when the height of the suffix tree approaches n. Studies have been conducted that show, contrary to the worst case height, the expected height of a suffix tree is $O(\log n)$ [1, 2, 5]. Consequently, while it is possible that CountUniqueOccurrences may cause the overall running time of NonOverlappingScore to reach $O(n^2)$, its expected performance should not exceed the $O(n \log n)$ performance of CountUniqueOccurrences. This expected performance exceeds the cost associated with constructing the suffix tree initially ($O(n)$), and splitting its leaf nodes ($O(n \log |\mathcal{S}|)$). Thus we conclude that the overall cost of determining the Most Contributory Substring when overlapping occurrences are prohibited is $O(n^2)$ in the worst case, with an expected performance of $O(n \log n)$.

The NonOverlappingScore algorithm is $O(n)$ in space. In a degenerate suffix tree with height n, it is necessary to allocate the ChildTrees array for every level in the tree before any call to NonOverlappingScore returns. Initially, this may appear to require $O((|\Sigma| + |\mathcal{L}|)n)$ space. However, it is impossible for every node in the tree to to have $|\Sigma| + |\mathcal{L}|$ children unless $|\Sigma| + |\mathcal{L}|$ is three or less. This constraint exists because the total number of nodes in the modified suffix tree is bounded by $3n$. Thus, even if we must allocate space to hold a pointer to every child's tree before any calls to NonOverlappingScore return, the total amount of space allocated with still be $3n$ pointers or less.

The amount of space required to store the binary trees used to identify the unique occurrences is also $O(n)$. The total number of nodes in the binary trees at any level in the suffix tree is bounded by n. Furthermore, binary trees only exist at two levels in the suffix tree at any time. Once the binary tree for the current node in the suffix tree is constructed, the binary trees for all of its children are deallocated. This means that a total of no more than $2n$ nodes will be present in the binary trees before the children's binary trees are deallocated, giving a space requirement that is $O(n)$.

After algorithm NonOverlappingScore executes the number of disjoint occurrences of each substring has been determined and the resulting score has been computed. Figure 6 shows the tree after scoring has been performed using the NonOverlappingOccurrences algorithm. It shows that the best substring is *ab* with a score of 10 – a different result than the *abab* achieved when overlapping strings were permitted.

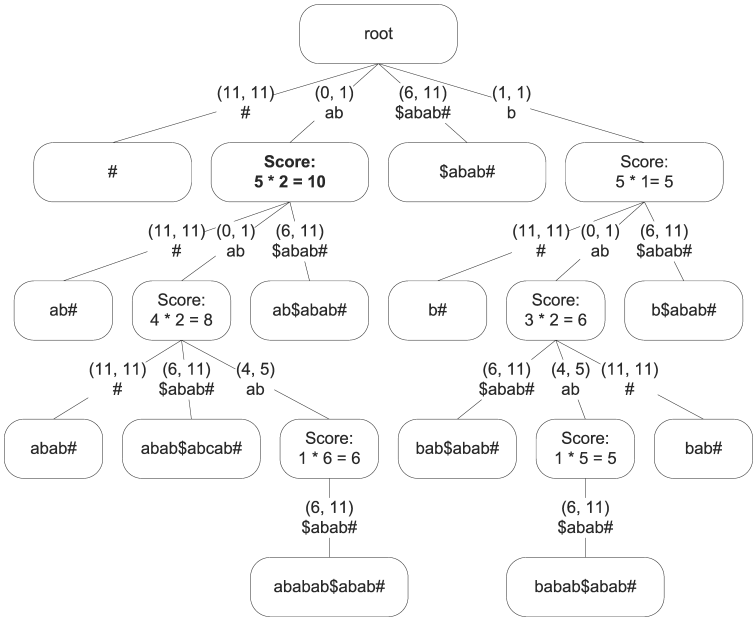

Fig. 6. Suffix tree for the string *ababab*$*abab*# Including Score Computed Using Only Non-Overlapping Occurrences

6 Applications

This algorithm was originally developed to analyze profile data collected during the execution of an application. The profile data consisted of approximately 200 distinct events which occurred millions of times. While determining which *distinct* event occurred with greatest frequency was straightforward, determining which *sequence* of events contributed the most to the execution of the application was more challenging. Analyzing this data using an inefficient algorithm took a prohibitively large amount time. As a result, the most contributory substring problem was identified and solved. Using the solution to the most contributory substring problem made it possible to quickly identify what sequence or sequences of events contributed the most to the execution of the application.

We also believe that using this algorithm can improve the level of compression achieved by Huffman coding. In its simplest form, Huffman coding uses variable length bit sequences to represent each symbol. Shorter sequences of bits are used to represent those symbols that occur with greater frequency while longer sequences are used to represent symbols that occur less frequently.

A variation on Huffman coding has also been developed which considers fixed length groups such as 2 symbols at a time. The frequency of each sequence of 2 symbols is determined and bit sequences are assigned based on the frequencies of the sequences. Using the most contributory substring algorithm can extend this idea further by using variable length sequences. The most contributory substring can be identified repeatedly in order to determine what sequence (possibly consisting of only one symbol) represents the largest portion of the string be-

ing compressed. This sequence would then be encoded using the shortest code word. Using this strategy degenerates to standard Huffman encoding when each most contributory substring identified has length one. We have yet to implement this variation of Huffman coding, so we are presently unsure how much of an improvement is achieved for real data sets.

Other applications for this algorithm may also exist. In particular, we believe that it may have utility in the realm of computational biology.

7 Conclusion

An algorithm is presented which solves the Most Contributory Substring Problem. This problem identifies a substring of its input that represents the largest proportion of the characters in the input string. In its first presentation, the algorithm identified potentially overlapping occurrences of the most contributory substring in a running time that was $O(n)$. An extended version of the algorithm was also presented that discounted overlapping occurrences of the most contributory substring. While the extended algorithm may require a running time as large as $O(n^2)$ for pathological input cases, previous studies have shown the expected height of a suffix tree is $O(\log n)$ rather than $O(n)$, reducing the complexity of the Most Contributory Substring Algorithm to $O(n \log n)$ in the expected case.

While our algorithm gives an optimal result when one substring is identified, it does not necessarily give an optimal result when applied iteratively to identify the k most contributory substrings of s. We hope to extend our algorithm to efficiently solve this problem in the future.

References

[1] A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. *J. Algorithms*, 13(3):446–467, 1992.

[2] L. Devroye, W. Szpankowski, and B. Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992.

[3] M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 137, Washington, DC, USA, 1997. IEEE Computer Society.

[4] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[5] P. Jacquet and W. Szpankowski. Autocorrelation on words and its applications: Analysis of suffix trees by string-ruler approach. *JCTA: Journal of Combinatorial Theory, Series A*, 66, 1994.

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

[7] 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. North-Holland, 1992.

[8] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[9] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symp on Switching and Automata Theory*, pages 1–11, 1973.