

# Engineering Small Space Dictionary Matching

Shoshana Marcus\*

Dina Sokol<sup>†</sup>

## Abstract

The dictionary matching problem is to locate occurrences of any pattern among a set of patterns in a given text. Massive data sets abound and at the same time, there are many settings in which working space is extremely limited. We introduce dictionary matching software for the space-constrained environment whose running time is close to linear. We use the compressed suffix tree as the underlying data structure of our algorithm, thus, the working space of our algorithm is proportional to the optimal compression of the dictionary. We also contribute a succinct tool for performing constant-time lowest marked ancestor queries on a tree that is succinctly encoded as a sequence of balanced parentheses, with linear time preprocessing of the tree. This tool should be useful in many other applications. Our source code is available at <http://www.sci.brooklyn.cuny.edu/~sokol/dictmatch.html>

## 1 Introduction

In recent years, there has been a massive proliferation of digital data. Concurrently, industry has been producing equipment with ever-decreasing hardware availability. Thus, we are faced with scenarios in which this data growth must be accessible to applications running on devices that have reduced storage capacity, such as mobile and satellite devices. Hardware resources are more limited, yet the user's expectations of software capability continue to escalate. This unprecedented rate of digital data accumulation therefore presents a constant challenge to the algorithms and software developers who must work with a shrinking hardware capacity.

The dictionary matching problem is to identify a *set* of patterns, called a dictionary, within a given text. Applications for this problem include searching for specific phrases in a book, scanning a file for virus signatures, and network intrusion detection. The problem also has applications in the biological sciences, such as searching through a DNA sequence for a set of motifs, identifying motifs to characterize protein families, and finding anchors for fast alignment of large genomic sequences.

A series of dictionary matching algorithms that operate in small space have in fact been developed [5, 14, 2, 13]. The latest of these results [13] achieved time and space optimal 1D dictionary matching. That is, their algorithm runs in linear time within space that meets empirical entropy bounds of the dictionary. The empirical entropy of a string ( $H_0$  or  $H_k$ ) describes the minimum number of bits that are needed to encode the string within context.

---

\*Simons Center for Quantitative Biology, Cold Spring Harbor Laboratory, 1 Bungtown Road, Cold Spring Harbor, N.Y., 11724. email: [smarcus@cshl.edu](mailto:smarcus@cshl.edu).

<sup>†</sup>Department of Computer and Information Science, Brooklyn College of the City University of New York, 2900 Bedford Avenue, Brooklyn, N.Y. 11210. email: [sokol@sci.brooklyn.cuny.edu](mailto:sokol@sci.brooklyn.cuny.edu).

Succinct dictionary matching algorithms have remained in the theoretical realm and have not been implemented until now. This work fills the void. We have developed software for dictionary matching in small space that relies on the compressed suffix tree, a popular succinct data structure. Our main challenge lay in combining a dictionary matching algorithm for the generalized suffix tree with compressed suffix tree representations.

Chan et al. developed the first succinct dictionary matching algorithm [5]. Their algorithm uses the compressed suffix tree of Sadakane [21], which they extended so that it can support a dynamically changing dictionary of patterns. Hon et al. presented a more space-efficient dictionary matching algorithm that uses a sampling technique to compress a suffix tree [14]. Their algorithm uses several data structures along with a compressed representation of the suffix tree, among them the string B-tree, a compressed trie of the patterns, and an LCP array of the longest common prefixes between the patterns. Our software uses only the compressed suffix tree, augmented with a succinct framework for lowest marked ancestor queries.

We also contribute a succinct tool for performing lowest marked ancestor queries in constant time after linear time preprocessing of a compressed suffix tree. The compressed suffix tree is augmented by a bit vector and a sequence of balanced parentheses. Lowest marked ancestor queries are answered by a set of constant-time queries to these data structures. The lowest marked ancestor structure that we implemented is appropriate for any succinct representation of an ordered tree that encodes the structure as a sequence of balanced parentheses, which was introduced by Jacobson [15]. Thus, our tool for lowest marked ancestor queries is a contribution that is useful to other applications that use the balanced parentheses representation of a tree.

We begin with an overview of compressed suffix trees in Section 2, since they are the basis of our succinct dictionary matching software. In Section 3 we describe our linear-time dictionary matching software that relies on the uncompressed suffix tree. Then, in Section 4, we describe the techniques employed by our succinct dictionary matching software, and the succinct framework we implemented for lowest marked ancestor queries on a compressed suffix tree. In Section 5, we present experimental results to demonstrate that these techniques are in fact space-efficient. We conclude with a summary and direction for future work in Section 6.

## 2 Compressed Suffix Tree

We begin with a description of the suffix tree and of compressed representations of the suffix tree since our program relies on the compressed suffix tree as its underlying data structure. The suffix tree is a compact trie that represents all suffixes of an input string. The suffix tree for  $S = s_1s_2 \cdots s_n$  is a rooted, directed tree with  $n$  leaves, one for each suffix. Each internal node, except the root, has at least two children. Each edge is labeled with a nonempty substring of  $S$  and no two edges emanating from a node begin with the same character. The path from the root to leaf  $i$  spells out suffix  $S[i \dots n]$ . Suffix links allow an algorithm to move quickly to a distant part of the tree. A suffix link is a pointer from an internal node labeled  $x\alpha$  to another internal node labeled  $\alpha$ , where  $x$  is an arbitrary character and  $\alpha$  is a possibly empty substring. The suffix array is a data structure that indexes a string by storing the lexicographical order of its suffixes.

Recent innovations in succinct full-text indexing provide us with the ability to compress both a suffix array and a suffix tree, using space that is proportional to the optimal compression of the data they are built upon. These self-indexes can replace the original text, as they support retrieval of the original text, in addition to answering queries about the data very quickly. Several compressed suffix array (CSA) representations

Space (bits)	Slowdown	Reference
$O(\ell \log \ell)$	$O(1)$	Uncompressed suffix tree
$O(\ell \log \sigma)$	$O(\text{polylog}(\ell))$	Sadakane [21]
$\ell H_k(T) + o(\ell \log \sigma)$	$O(\log \ell)$	Russo et al. [18]
$(1 + \frac{1}{\epsilon})\ell H_k(T) + o(\ell \log \sigma)$	$O(\log^\epsilon \ell)$ , $0 < \epsilon \leq 1$	Fischer et al. [9]
$ CSA  +  CLCP  + 3\ell$	$O(1)$ for many operations	Ohlebusch et al. [16]

Table 1: Compressed suffix tree representations for an input string  $T$  of length  $\ell$ , where  $|CSA|$  is the number of bits used to store the compressed suffix array and  $|CLCP|$  is the number of bits occupied by the compressed LCP array.

exist, e.g., [20, 11, 7, 8], each with a different time-space trade-off. The most recent results meet  $k$ th order empirical entropy of the input string. Compressed representations of the suffix tree, e.g., [21, 18, 9, 16], use the compressed suffix array as a component. Table 1 summarizes the time-space trade-offs in several compressed suffix tree (CST) representations.

Ohlebusch et al. [16] recognized that the compressed suffix tree generally consists of three separate parts: the lexicographical information in a compressed suffix array (CSA), the information about common substrings in the longest common prefix array (LCP), and the tree topology combined with a navigational structure (NAV). Each of these three components functions independently from the others and is stored separately. Representations of compressed suffix arrays and compressed LCP arrays are interchangeable in many compressed suffix tree representations. Combining the different representations of each component yields a rich variety of compressed suffix trees, although some compressed suffix trees favor certain compressed suffix array or compressed LCP array representations. The Succinct Data Structures Library (SDSL) provides a range of compressed suffix tree implementations, which we used in our dictionary matching software. We experimented with Sadakane’s compressed suffix tree [21] by using an assortment of compressed suffix array and compressed LCP modules to achieve different time and space complexities in our dictionary matching software.

### 3 Linear-Time Dictionary Matching with Suffix Tree

We first developed a linear-time dictionary matching program that uses the uncompressed suffix tree as its primary data structure. Then we modified our approach to use the compressed suffix tree to improve the space complexity. In this section we describe the linear time dictionary matching software that uses an uncompressed suffix tree. Then, in the next section, we delineate the revisions in our techniques so that we perform dictionary matching using compressed suffix tree representations.

A suffix tree can be used to index several strings, in a *generalized suffix tree*. The dictionary can be merged to form a single string by concatenating the patterns with a unique delimiter separating them. Because it is *online*, Ukkonen’s suffix tree construction algorithm can insert one string at a time and index only the actual suffixes of a set of strings in a suffix tree [12]. The dictionary of patterns is indexed by a generalized suffix tree to preprocess it for dictionary matching queries. Then, the text is searched for pattern occurrences in linear time, in a manner similar to Ukkonen’s insertion of a new string to the suffix tree. We briefly summarize Ukkonen’s suffix tree construction algorithm in the following paragraph, and depict its steps in Algorithm 1.

The elegance of Ukkonen’s algorithm is evident in its key property. The algorithm admits the arrival of

---

**Algorithm 1** Ukkonen's suffix tree construction algorithm

---

```
j = -1;
{j is last suffix inserted}
for i = 0 to n - 1 do
    {phase i: i is current end of string}
    while j < i do
        {let j catch up to i}
        if singleExtensionAlgorithm(i, j) then
            break {implicit suffix so proceed to next phase}
        end if
        if lastNodeInserted ≠ root then
            lastNodeInserted.SuffixLink ← root
        end if
        lastNodeInserted ← root
    end while
end for
```

---

the string during construction. Yet, each suffix is inserted exactly once, and a leaf is never updated after its creation. As a new character is appended to the input string, Ukkonen's algorithm ensures that all suffixes of the string are indexed by the tree. As soon as a suffix is implicitly found in the tree, modification of the tree ceases until the next new character is examined. The next phase begins by extending the implicit suffix with the new character. Using suffix links and a pointer to the last suffix inserted, each suffix is inserted in the tree in amortized constant time. The combination of one-time insertion of each suffix and rapid suffix insertion results in an overall linear-time suffix tree construction algorithm.

Our dictionary matching algorithm over a generalized suffix tree of patterns was inspired by Ukkonen's process for inserting a new string into a generalized suffix tree (as shown in Algorithm 1), pretending to index the text, without modifying the index. The text is processed in an online fashion, traversing the suffix tree of patterns as each successive character of text is processed. A pattern occurrence is announced when a labeled leaf is encountered, i.e., a leaf that represents the first suffix of a pattern. At a position of mismatch and at a pattern occurrence, suffix links are used to navigate to successively smaller suffixes of the matching string. When a suffix link is used within the label of a node, the corresponding number of characters can be skipped, obviating redundant character comparisons. In the spirit of Ukkonen's skip-count trick, this ensures that the text is scanned in linear time.

The skip-count trick is based on Lemma 1. A suffix link is a directed edge from the internal node at the end of the path labeled  $x\alpha$  to another internal node at the end of the path labeled  $\alpha$ , where  $x$  is an arbitrary character and  $\alpha$  is a possibly empty substring. We can similarly define suffix links for leaves in the tree. The suffix link of the leaf representing suffix  $i$  points to the leaf representing suffix  $i + 1$ .

**Lemma 1** [12] *In a suffix tree, the number of nodes along the path labeled  $\alpha$  is at least as many as the number of nodes along the path labeled  $x\alpha$ .*

**Proof:** Suppose not. That is, there is some  $\alpha$  for which the path labeled  $\alpha$  has fewer nodes than the path labeled  $x\alpha$ . This means that some suffix of  $x\alpha$  is not indexed by the suffix tree. This implies that the suffix tree is not fully constructed. Hence, a contradiction and the premise must be valid. ■

**Corollary 1** *If the suffix link of the root points to itself, every node of the suffix tree has a suffix link.*

Ukkonen uses suffix links to navigate across a suffix tree and then skip over the appropriate number of characters labeling the beginning of an edge. In dictionary matching, we navigate a fully constructed suffix tree, and every node must have a suffix link established for it. To avoid redundant comparisons, we follow a suffix link across a suffix tree and then jump up to the position at which the mismatch occurred. It is more efficient to navigate up a suffix tree than down. That is, every node has a single parent but when navigating to a child, several branches can be considered. When a suffix link is traversed, we know the number of characters to skip going up the edge as this is the number of characters that remain along the edge after the position of mismatch. Yet, we do not know at which node traversal will halt. The shorter label may be split over more edges than the longer label spans, by Lemma 1.

We extended Algorithm 1 to perform dictionary matching. Pseudocode of our program is delineated in Algorithm 2, with its submodules extracted to Algorithms 3 and 4. Our program reports the longest pattern occurrence that ends at each text position. When a pattern is a suffix of a longer pattern, and the longer pattern occurs in the text, we do not spend time reporting an occurrence of the shorter pattern.

A key challenge in implementing dictionary matching on the suffix tree is the scenario in which one pattern is a proper substring of another pattern [1]. Traversing the suffix tree using suffix links (as in Algorithm 2), these pattern occurrences can be passed unnoticed in the text. This limitation is addressed by augmenting each node of the suffix tree with a pointer to the longest prefix of the label along its path from the root that is a complete pattern. The nodes are marked with this information in linear time by a depth-first traversal of the suffix tree.

The suffix tree is a versatile tool in string algorithms, and is already needed in many applications to facilitate other queries. Thus, in practice, our linear-time dictionary matching program with the uncompressed suffix tree requires very little additional space. This tool is itself a contribution, allowing efficient dictionary matching in small space, however, we improved this application by using a compressed suffix tree as the underlying data structure.

## 4 Dictionary Matching with Compressed Suffix Tree

In this section we describe how we redesigned our dictionary matching code to run over a compressed suffix tree in linear time, overlooking the slowdown of queries on the compressed suffix tree. Since the existing compressed suffix tree construction algorithms are not online algorithms, it is not possible to build the compressed suffix tree incrementally, inserting one pattern at a time. Instead, the dictionary is merged into a single string by concatenating the patterns with a unique delimiter between them. We used the Succinct Data Structures Library (SDSL)<sup>1</sup> since it provides a C++ implementation of a variety of compressed suffix tree representations and it was proven to be more efficient than previous compressed suffix tree implementations [10].

Although the ultimate capability of the compressed suffix tree is modeled after the functionality of its uncompressed counterpart, many operations that are straightforward in the uncompressed suffix tree require creativity in the compressed data structures. Understanding how the suffix tree components are represented in the compressed variation is a necessary prerequisite to implementing seemingly straightforward navigational tasks. Furthermore, the compressed suffix tree is a self-index and allows us to discard the original set of patterns. Thus, we had to figure out which component data structure to query in order to randomly access a single pattern character. For instance, announcing a pattern occurrence (Algorithm 2, line 25) is

---

<sup>1</sup> <http://simongog.github.com/sdsl/>

---

**Algorithm 2** Dictionary matching over the generalized suffix tree

---

```
1: curNode  $\leftarrow$  root
2: textIndex  $\leftarrow$  0
3: curNodeIndex  $\leftarrow$  0
4: skipcount  $\leftarrow$  0
5: usedSkipcount  $\leftarrow$  false
6: repeat
7:   lastNode  $\leftarrow$  curNode
8:   if usedSkipCount  $\neq$  true then
9:     textIndex  $+=$  curNodeIndex
10:    curNodeIndex  $\leftarrow$  0
11:    curNode  $\leftarrow$  curNode.child(text[textIndex])
12:    if curNode.length  $>$  0 then
13:      curNodeIndex  $++$  {already compared the first character on the edge}
14:    end if
15:  else
16:    usedSkipCount  $\leftarrow$  false
17:  end if
18:  {compare text}
19:  while curNodeIndex  $<$  curNode.length AND curNodeIndex  $+$  textIndex  $<$  text.length do
20:    if text[textIndex + curNodeIndex]  $\neq$  pat[curNode.stringNum][curNode.beg + curNodeIndex]
21:      then
22:        break {mismatch}
23:      end if
24:    curNodeIndex  $++$ 
25:  end while
26:  if curNodeIndex  $=$  curNode.length AND curNode.firstLeaf() then
27:    announce pattern occurrence
28:  end if
29:  if curNodeIndex  $=$  curNode.length AND curNode.length  $>$  0 AND text[textIndex + curNodeIndex - 1]
30:     $=$  pat[curNode.stringNum][curNode.beg + curNodeIndex - 1] then
31:    continue {branch and continue comparing text to patterns}
32:  end if
33:  handleMismatch
34: until textIndex + curNodeIndex  $\geq$  text.length {scan entire text}
```

---

---

**Algorithm 3** Handling a Mismatch

---

```
if  $curNode.depth \neq 0$  OR  $lastNode.depth \neq 0$  then
  if  $curNode.suffixLink = root$  AND  $lastNode.suffixLink \neq root$  then
     $curNode \leftarrow lastNode$ 
     $curNodeIndex \leftarrow curNode.length$  {mismatched when trying to branch}
     $textIndex - = curNode.length$ 
  end if
  if  $curNode.parent = root$  AND  $curNodeIndex = 1$  then
     $textIndex++$ 
     $curNodeIndex = 0$ 
     $curNode = curNode.parent$ 
    continue {when traverse suffix link: will be at mismatch, so skip 1 char}
  end if
  useSkipcountTrick(skipcount, curNode)
else
  {mismatch at root}
   $textIndex++$ 
end if
```

---

---

**Algorithm 4** Skip-Count trick

---

```
repeat
   $curNode \leftarrow curNode.suffixLink$ 
   $usedSkipCount \leftarrow true$ 
   $textPos = curNodeIndex + textIndex$ 
   $skipcount \leftarrow curNode.length - curNodeIndex$ 
  if  $skipcount \geq curNode.length$  then
    if  $curNode.length = 0$  then
       $usedSkipCount \leftarrow false$  {branch at next iteration of outer loop, look for next text char}
       $curNodeIndex \leftarrow 0$ 
       $skipcount \leftarrow 0$ 
    else
      if  $skipcount = curNode.length$  then
         $curNodeIndex--$ 
         $usedSkipCount \leftarrow false$  {branch at next iteration of outer loop}
      end if
       $skipcount - = curNode.length$ 
       $curNode \leftarrow curNode.parent$ 
    end if
  else
     $curNodeIndex \leftarrow curNode.length - skipcount$ 
     $skipcount \leftarrow 0$ 
  end if
until  $skipcount \leq 0$ 
 $textIndex = textPos - curNodeIndex$ 
```

---

---

**Algorithm 5** Announcing Pattern Occurrence in CST

---

```
if  $\text{getCharAtNodePos}(\text{curNode}, \text{curNodeIndex}) = \text{END\_OF\_STRING\_MARKER}$  then
   $\text{pos} \leftarrow \text{csa}[\text{lb}(\text{curNode})] - 1$ 
   $\{\text{lb}(v)$  returns the left bound of node  $v$  in the suffix array $\}$ 
   $\{\text{pos}$  is dictionary index immediately preceding this leaf's ancestor emanating from  $\text{root}\}$ 
  if  $\text{pos} < 0$  then
     $\text{occ} \leftarrow \text{true}$  {beginning of first pattern}
  else
     $c \leftarrow \text{getCharAtPatternPos}(\text{pos})$ 
    if  $c = \text{END\_OF\_STRING\_MARKER}$  then
       $\text{occ} \leftarrow \text{true}$  {beginning of some pattern after first}
    end if
  end if
end if
```

---

not simply a question of checking whether traversal has reached the end of a leaf representing the first suffix of a pattern. A simple *if* statement is replaced by the segment of pseudocode delineated in Algorithm 5 and described in the following paragraph.

Instead of an *if* statement that checks properties of a leaf, we perform the following computation, involving several function calls, to determine if a pattern occurrence has been located in the text. When traversing the compressed suffix tree according to the text, a mismatch along an edge leading into a leaf may in fact be a pattern occurrence. Thus, we first check if the mismatch is a string delimiter, which mismatches every text character. Then, we determine if this leaf represents the first suffix of some pattern. This is done by finding out which character precedes the beginning of this leaf's path from the root. If the path begins at the beginning of the dictionary, this leaf represents the first suffix of the first pattern, and a pattern occurrence is announced. Similarly, if the character at that position is a pattern delimiter, the suffix is a complete pattern, and a pattern occurrence is announced.

The skip-count trick we described in the previous section enables us to navigate the compressed suffix tree while processing the text in linear time. When we use this technique and traverse suffix links to find pattern occurrences in the text, some pattern occurrences can pass unnoticed. This concern is limited to a dictionary in which one pattern is a proper substring of another. Consider the suffix tree in Figure 1 for the dictionary of patterns  $D = \{a, \text{ate}, \text{bath}, \text{later}\}$ . Two of the patterns in the dictionary are substrings of other patterns. If the text contains the word *lately*, an occurrence of the pattern *ate* should be identified within this word. However, using suffix links, we navigate from the node labeled *later* to the node labeled *ater* to the node labeled *ter*, without recognizing an occurrence of *ate*. This is because we are looking for the longer pattern *later*.

In the uncompressed suffix tree, we mark nodes that are pattern occurrences and preprocess the suffix tree with a depth-first traversal so that lowest marked ancestor (LMA) queries can be answered in constant time. Then, an LMA query at each traversal of a suffix link ensures that no pattern occurrence is skipped over by the skip-count trick. In a compressed suffix tree, this is not as straightforward since the nodes are not stored as independent entities. Thus, we implemented a framework for answering lowest marked ancestor queries in constant time that consists of bit arrays and sequences of balanced parentheses. We coded this framework with the compressed suffix tree in mind. Yet, it is suitable for any compressed representation of an ordered tree that represents the nodes as a sequence of balanced parentheses. This is a more general





---

**Algorithm 6** Lowest Marked Ancestor Query on node in CST

---

{returns root if node has no marked ancestor}  
{rank and select queries assume that the bit-array is 0-based}

```
if M[node]=1 then
    return node {node is marked}
else
     $pre\_y \leftarrow M.rank(node+1)$ 
    if  $pre\_y = 0$  then
        return root
    else
         $y \leftarrow M.select(pre\_y) - 1$ 
        if B[y]=1 then
            return y { y is the LMA since B[y]='(' }
        else
             $y1 \leftarrow M.rank(y)$  {coresponding index in D}
             $y2 \leftarrow D.find\_open(y1)$ 
             $y3 \leftarrow D.enclose(y2)$ 
            if  $y3 = NULL$  then
                return root {no enclosing parentheses }
            else
                 $y4 \leftarrow M.select(y3 + 1) - 1$  {map from D to M}
                return y4
            end if
        end if
    end if
end if
end if
```

---

position  $i$ . We used efficient implementations of these data structures that are included in the Succinct Data Structures Library.

## 5 Experimental Results

We implemented the algorithms in C++ and ran experiments on computers that feature an Intel(R) Xeon(R) processor at 2.93 GHz, with 5 GB of RAM, running Linux kernel version 2.6.32. For one set of experiments, we searched a 5 MB English text for common English words in a 2.5 MB dictionary that comes from ClueWeb09<sup>2</sup> and was used in [3]. We performed another set of experiments on biological data. We searched 5 MB of the human genome for patterns in a 3 MB dictionary of promoter sequences in the human genome<sup>3</sup>. The texts are 5 MB of DNA and 5 MB of English text from the Pizza&Chili corpus<sup>4</sup>.

We used the framework of Sadakane’s compressed suffix tree [21], `cst_sada`, in our experiments since it stores nodes as a sequence of balanced parentheses and we were able to augment it for constant-time lowest marked ancestor queries. Configurations of `cst_sada` with newer representations of its components beat the runtime of configurations of the other types of compressed suffix trees on almost all operations and its space savings is of comparable significance [10]. In particular, the navigational operations are very fast. Sadakane’s CST consists of a compressed suffix array, a compressed LCP array, and a navigational structure. We ran experiments on four different variations of Sadakane’s CST using two different types of compressed suffix arrays, `csa_sada` and `csa_wt`, and two different types of compressed LCP arrays, `lcp_dac` and `lcp_support_tree2`. We also ran our experiments on an uncompressed suffix tree. The `csa_sada` class is a very clean reimplement of Sadakane’s compressed suffix array [19] and the `csa_wt` class is based on a wavelet tree. The `lcp_dac` class uses the direct accessible code solution of Brisaboa et al. [4], which represents the LCP array in suffix array order, and `lcp_support_tree2` uses a tree compressed representation of the LCP array, which is based on the topology of the compressed suffix tree.

We compare the time-space trade-off of dictionary matching using different variations of 1D dictionary matching software. For a baseline, we use uncompressed components, which consume the most space but perform operations in constant time. The remaining runs use different underlying representations of compressed suffix arrays and compressed LCP arrays as components.

Compressed suffix trees conserve a considerable amount of space while the sacrifice is a negligible slowdown in running time. This is illustrated in Figure 2 and in Tables 2 and 3.

## 6 Conclusion

We have introduced dictionary matching software that runs in small space. Its underlying data structure is the compressed suffix tree. This program runs in linear time, disregarding the slowdown of querying the compressed self-index. We have shown that our implementation conserves considerable space in practice. Our software includes a space-efficient technique for performing lowest marked ancestor queries on compressed suffix trees, a contribution that is useful for many other applications.

We would like to extend our small-space dictionary matching software to accommodate a dynamically

---

<sup>2</sup><http://lemurproject.org/clueweb09/>

<sup>3</sup><http://epd.vital-it.ch>

<sup>4</sup><http://pizzachili.dcc.uchile.cl>

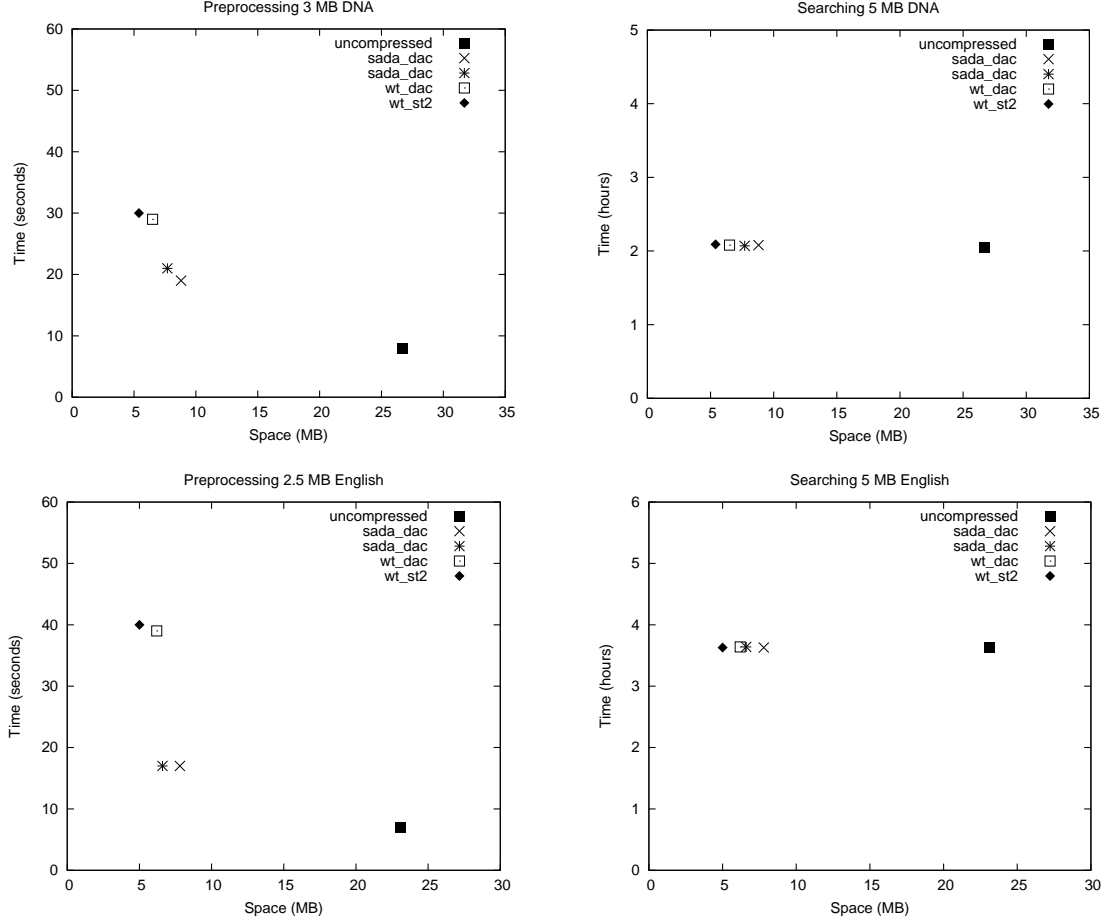


Figure 2: Time-space trade-offs of different representations of the compressed suffix array and the compressed LCP array in Sadakane's compressed suffix tree for dictionary matching. The uncompressed is the outlier in space consumption. The four compressed versions have similar time and space complexities.

changing set of patterns in the dictionary. Several dynamic compressed suffix tree representations have been presented [5, 17] but they lack implementations. Extending this work to the dynamic setting would begin by implementing the dynamic compressed suffix tree to accommodate insertion, deletion, and modification of dictionary patterns, without rebuilding the index of the entire dictionary.

## Acknowledgements

The authors would like to thank Simon Gog for his help installing and working with the Succinct Data Structures Library.

<b>Experiments on DNA: 3 MB dictionary and 5 MB text</b>			
<b>CST components</b>	<b>Space</b>	<b>Preprocessing Time</b>	<b>Searching Time</b>
uncompressed	26.7 MB	8 sec	2.05 hours
sada_dac	8.8 MB	19 sec	2.08 hours
sada_st2	7.7 MB	21 sec	2.07 hours
wt_dac	6.5 MB	29 sec	2.08 hours
wt_st2	5.4 MB	30 sec	2.09 hours

Table 2: Time-space trade-offs of using different representations of the compressed suffix array and the compressed LCP array in Sadakane’s compressed suffix tree for searching 5 MB of the human genome for promoter sequences that comprise a 3 MB dictionary. 13 pattern occurrences were found in the text.

<b>Experiments on English text: 2.5 MB dictionary and 5 MB text</b>			
<b>CST components</b>	<b>Space</b>	<b>Preprocessing Time</b>	<b>Searching Time</b>
uncompressed	23.1 MB	7 sec	3.63 hours
sada_dac	7.8 MB	17 sec	3.63 hours
sada_st2	6.6 MB	17 sec	3.64 hours
wt_dac	6.2 MB	39 sec	3.64 hours
wt_st2	5.0 MB	40 sec	3.63 hours

Table 3: Time-space trade-offs of using different representations of the compressed suffix array and the compressed LCP array in Sadakane’s compressed suffix tree for searching 5 MB of English text for common English words in a 2.5 MB dictionary. 12,717 pattern occurrences were located in the text.

## References

- [1] A. Amir and M. Farach. Adaptive dictionary matching. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 760–766, 1991.
- [2] D. Belazzougui. Succinct dictionary matching with no slowdown. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 88–100, 2010.
- [3] L. Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1), 2011.
- [4] N. R. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013.
- [5] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007. Article 21.
- [6] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.
- [7] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
- [8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.
- [9] J. Fischer. Wee LCP. *Information Processing Letters*, 110(8-9):317–320, 2010.
- [10] S. Gog. Compressed suffix trees: Design, construction, and applications. 2011. PhD thesis, Universität Ulm.
- [11] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

- [12] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [13] W.-K. Hon, T.-H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster compressed dictionary matching. In *Symposium on String Processing and Information Retrieval (SPIRE)*, pages 191–200, 2010.
- [14] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Compressed index for dictionary matching. In *Data Compression Conference (DCC)*, pages 23–32, 2008.
- [15] G. Jacobson. Space-efficient static trees and graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
- [16] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Symposium on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.
- [17] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully-compressed suffix trees. In *Latin American Theoretical Informatics Symposium (LATIN)*, pages 362–373, 2008.
- [18] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53:1–53:34, 2011.
- [19] K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.
- [20] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [21] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- [22] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.
- [23] S. Vigna. Broadword implementation of rank/select queries. In *Workshop on Experimental Algorithms (WEA)*, pages 154–168, 2008.