

# APPLICATION OF SEMI-LOCAL LCS TO STRING APPROXIMATE MATCHING\*

DIANNE DOE<sup>†</sup>, PAUL T. FRANK<sup>‡</sup>, AND JANE E. SMITH<sup>‡</sup>

**Abstract.** We present an application of semi-local lcs to approximate string matching by developing a new algorithm and improving the existing one. Our result is based on the utilization of the underlying algebraic structure of semi-local lcs with the usage of the novel data structure for submatrix maximum queries in Monge matrices. This gives two algorithms with the following running time and space complexity. TODO. The improvement of the existing algorithm not only preserves all properties but also outperforms in practice.

In addition, we show that the algorithm for semi-local lcs based on sticky braid multiplication is not perform well with the current complex recursive structure.

**Key words.** semi-local lcs, monge matrix, range queries, approximate matching, near-duplicate detection

**AMS subject classifications.** 68Q25, 68R10, 68U05

**1. Introduction.** Approximate string matching is an important task in many fields such as computational biology, signal processing, text retrieval and etc. It also refers to a duplicate detection subtask.

In general form it formulates as follows: Given some pattern  $p$  and text  $t$  need to find all occurrences of pattern  $p$  in text  $t$  with some degree of similarity.

There are many algorithms that solve the above problem. Nonetheless, the number of algorithms sharply decreases when the algorithm needs to meet some specific requirements imposed by running time, space complexity or specific criterion for the algorithm itself. For example, recently there was developed an approach for interactive duplicate detection for software documentation [2]. The core of this approach is an algorithm that detects approximate clones of a given user pattern with a specified degree of similarity. The main advantage of the algorithm is that it meets a specific requirement of completeness. Nonetheless, it has an unpleasant time complexity.

The algorithm for approximate detection utilizes mainly algorithm for solving the longest commons subsequence ( $LCS$ ) problem. The longest common subsequence is a well-known fundamental problem in computer science that also has many applications of its own. The major drawback of it that it shows only the global similarity for given input strings. For many tasks, it's simply not enough. The approximate matching is an example of it.

There exist generalization for  $LCS$  called *semi-local LCS* [] which overcome this constraint. The effective theoretical solutions for this generalized problem found applications to various algorithmic problems such as bla bla add cited. For example, there has been developed algorithm for approximate matching in the grammar-compresed strings[].

Although the algorithms for *semi-local LCS* have good theoretical properties, there is unclear how they would behave in practice for a specific task and domain.

To show the applicability of semi-local lcs on practice we developed several algorithms based mainly on it and the underlying algebraic structure. As well as devel-

---

\*Submitted to the editors DATE.

**Funding:** This work was funded by the Fog Research Institute under contract no. FRI-454.

<sup>†</sup>Imagination Corp., Chicago, IL (ddoe@imag.com, <http://www.imag.com/~ddoe/>).

<sup>‡</sup>Department of Applied Mathematics, Fictional University, Boise, ID (ptfrank@fictional.edu, jesmith@fictional.edu).

opening new algorithms we improve and significantly outperform the existing one for interactive duplicate detection for software documentation [1]. It should be noted that improvement preserves all properties of this algorithm. **Do we need to state that ant algo is slow for current strucute of algorithm**

The paper is organized as follows. Blablabla ??, our new algorithm is in ??, experimental results are in ??, and the conclusions follow in ??.

## 2. Preliminaries.

**2.1. Approximate matching.** Describe approximate matching formally

**2.2. Semi-local lcs.** Describe semi-local lcs (definition), algorithms that solves (steady and and braid reducing)

**2.3. Monge matrix.** Describe monge property

Say about range queries (about soda12, soda14 and new result that we will be used)

**2.4. Near-duplicate detection algorithm.** Describe luciv algo

## 3. Related work. ?????

could mention about approximation. Need discuss

**4. Algorithm for near duplicate detection.** **TODO: third phase also may be improved/ removed. By filtering in place. Also prserves property.**

We now describe an improved version of Luciv et.al. algorithm [2] by utilizing a *semi-local sa* solution. Then we present proof that improved version preserves completeness property. It is achieved by imitating all phases of the algorithm.

**4.1. Algorithm description.** **NEWALGO.NEED DISCUSS** Algorithm comprises of two phases.

At phase one (Line 1) *semi-local sa* problem is solved for the pattern  $p$  against whole text  $t$ . After that phase we have access to string-substring matrix  $H_{p,t}^{str-sub}$  that implicitly stores scores for any substring of text  $t$  against pattern  $p$ .

At phase two text is processed via sliding window of size  $L_w = \frac{|p|}{k}$ . If alignment score between substring that corresponds to window position and pattern  $p$  is  $\geq k_{di}$  (ref to previous section described luciv algo) then this we proceed as follows.

We search of largest text fragment within this window

At phase two the largest text fragment that closest

At phase two the search of largest text fragment with highest score within every sliding window of size  $L_w = \frac{|p|}{k}$  of text  $t$  is performed. More preciesely, we looking for the text fragment that have maximal score with pattern  $p$  across all substrings with sizes  $|p| * k \dots \frac{|p|}{k}$  within each sliding window. It corresponds to looking at the square submatrices within  $H_{p,t}^{str-sub}$  with  $\frac{|p|}{k} - |p| * k$  sizes (see picture). Let's look closely to concrete submatrix (denote it by  $M$ ). The intresting cells lies in upper-right part Let's denote it by  $m$  In other words, we need to find at the submatrix

Also the filtering of duplicate and intersected elements is performed in-place during this phase (this corresponds to third phase of [2]).

In terms of string-substrng matrix sliding window refers to square submatrix of size  $L_w$ . Moreover, sliding window with step 1 refers to sliding square windows that goes diagonally with one step (see picture). Thus, it means that diagonal in string-substring matrix of width  $|p| * k \dots \frac{|p|}{k}$  and length  $|t|$  corresponds to diagonal in string-substring matrix.

88 This means that it is sufficiently to visit only  $O(\frac{|p|}{k} - |p| * k) * O(|t|) = O(|p||t|)$   
 89 cells to gain required text fragments. Now we describe how to visit this cells.

90 **END NEW ALGO**

91 The algorithm comprises three phases as in [2]. At phase one (Line 1) semi-local  
 92 sa problem is solved for the pattern  $p$  against whole text  $t$ . This solution provides  
 93 access to the string-substring matrix which allows performing fast queries of sa score  
 94 for pattern  $p$  against every substring of text  $t$ .

95 At the second phase text  $t$  is scanning with a sliding window of length  $L_w$  with  
 96 step 1. First, it checks that given substring  $w$  that of a maximum possible size of  
 97  $L_w$  have score that is higher or equal to a given threshold (Line 4). If no, then this  
 98 interval will not further be proceeded (Line 5) else this interval will be processed as  
 99 follows. First, for each prefix of text  $t$  it finds suffix that has the highest alignment  
 100 score with the maximal length among all suffixes with that score. It corresponds to  
 101 the searching row position for each column in string-substring matrix with associated  
 102 alignment score. Second, among these suffixes, one is selected with the highest score.  
 103 If several suffixes have the same score the one with maximal length is selected (Line  
 104 8). Then if selected suffix has score higher than the threshold, then it is added to set  
 105  $W_2$ .

106 The third phase is the same as in [2]. More precisely, on the third phrase, set  $W_2$   
 107 is filtered out in a such way that only non-intersected intervals are left. It is simply the  
 108 sorting of set  $W_3$  by starts of intervals with following one way passage with filtering.

---

**Algorithm 4.1** PATTERN BASED NEAR DUPLICATE SEARCH ALGORITHM VIA SEMI-LOCAL SA

---

Input: pattern  $p$ , text  $t$ , similiarity measure  $k \in [\frac{1}{\sqrt{3}}, 1]$

Output: Set of non-intersected clones of pattern  $p$  in text  $t$

$$(4.1) \quad k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2)$$

$$(4.2) \quad L_w = \frac{|p|}{k}$$

Comment:  $w_i, w_j$  — start and end positions of  $w$  in text  $t$

Pseudocode:

```

1:  $W = semilocalsa(p, t)$ 
2:  $W_2 = \emptyset$ 
3: for  $w \in t, |w| = L_w$  do
4:   if  $W.stringSubstring(w_i, w_j) < -k_{di}$  then
5:     continue
6:   end if
7:    $maximums = FindMaxForColumnsBySmawk(w)$ 
8:    $max = FindMaxWithLenghtConstraint(maximums)$ 
9:   if  $max \geq -k_{di}$  then
10:    add substring associated with max to  $W_2$ 
11:   end if
12: end for
13:  $W_3 = UNIQUE(W_2)$  {3rd phase unchanged}
14: for  $w \in W_3$  do
15:   if  $\exists w' \in W_3 : w \subset w'$  then
16:     remove  $w$  from  $W_3$ 
17:   end if
18: end for
19: return  $W_3$ 
```

---

THEOREM 4.1. Algorithm 4.1 could be solved in  $O(|p| * |t|)$  where  $p$  is pattern,  $t$  is text.

*Proof it*

$$D = \text{diag}(d_1, \dots, d_n)$$

THEOREM 4.2. Algorithm 4.1 preserves completeness property of algorithm [2].

*Proof it*

$$D = \text{diag}(d_1, \dots, d_n)$$

**5. CutMax a new approximate mathing algorithm.** We now describe several algorithms that heavily based on semi-local lcs and it's underlying algebraic structure.

The first algorithm 5.1 refers to following constraint. There should be found all non-intersected clones  $\tau_i$  of pattern  $p$  from text  $t$  that has the highest similarity score on the uncovered part of the text  $t$  and  $l \geq |tau_i| \geq r$  i.e algorithm should perform greedy choice at each step with length constraint. This is a more intuitive approach i.e like looking for the most similar one every time that have enough.

124 The algorithm proceeds as follows. First, semi-local sa problem is solved (Line 1).  
 125 Then upon string-substring submatrix of semi-local sa solution is built data structure  
 126 for performing range queries on it (Lines 2-3). *IntervalsToSearch* is structure that  
 127 holds text intervals where search should be performed. Lines 7-21 corresponds to  
 128 searching *interval* with maximal alignment within the current uncovered part of the  
 129 text  $t_{i,j}$ . More precisely, it refers to searching maximum value with corresponding  
 130 position (row and column) in submatrix of string-substring *matrix* within  $t_{i,j}$  (starting  
 131 at  $i$ th position and ending at  $j$ th position of text  $t$ . It is solved via range queries (Line  
 132 9). When detected *interval* has alignment score less then threshold it means that no  
 133 clones of pattern  $p$  are presented in this part of text  $t_{i,j}$ , and further processing should  
 134 be skipped (Line 19). Otherwise, the founded clone is added to final result and the  
 135 current part of the text splits on two smaller parts and processed in the same way  
 136 (Lines 13, 15). Finally, the algorithm outputs a set of the non-intersected intervals of  
 137 clones of pattern  $p$  in text  $t$ .

---

**Algorithm 5.1** GREEDY-PATTERN BASED NEAR DUPLICATE SEARCH ALGORITHM VIA SEMI-LOCAL SA

---

Input: pattern  $p$ , text  $t$ , threshold value  $h$

Output: Set of non-intersected clones of pattern  $p$  in text  $t$

Pseudocode:

```

1:  $sa = semilocalsa(p, t)$ 
2:  $matrix = sa.getStringSubstringMatrix()$ 
3:  $rmQ2D = buildRMQStructure(matrix)$ 
4:  $intervalsToSearch = \emptyset$ 
5:  $intervalsToSearch.add((0, |t|))$ 
6:  $result = \emptyset$ 
7: while  $intervalsToSearch.isNotEmpty()$  do
8:    $i, j = intervalsToSearch.pop()$ 
9:    $interval = rmQ2D.query(i, j, i, j)$ 
10:  if  $interval.score \geq h$  then
11:     $result.add(interval)$ 
12:    if  $interval.startInclusive - i \geq 1$  then
13:       $intervalsToSearch.add((i, interval.startInclusive))$ 
14:    end if
15:    if  $j - interval.endExclusive \geq 1$  then
16:       $intervalsToSearch.add((interval.endExclusive, j))$ 
17:    end if
18:  else
19:     $continue$ 
20:  end if
21: end while
22: return  $result$ 

```

---

138 The second algorithm 5.2 uses a less sophisticated approach and a more light-  
 139 weight one but found fewer duplicates of pattern  $p$  (see example ??). The algorithm  
 140 also follows a greedy approach but instead of looking at the uncovered part of text  $t$   
 141 at each step it looks at the text  $t$  and chooses the first available substring with the  
 142 highest score that doesn't intersect with already taken substrings. More formally, it  
 143 approximates algorithm 5.1.

144 *Algorithm description.* First, the *semi-locsa* problem is solved (Line 1). Then  
 145 we solve *complete approximate matching problem* (Line 3) i.e for each prefix of text  $t$   
 146 we find the shortest suffix that has the highest similarity score with pattern  $p$  (Line  
 147 3):

$$148 \quad (5.1) \quad a[j] = \max_{i \in 0..j} sa(p, t[i, j]), j \in 0..|t|$$

149 Further, we remove suffixes whose similarity is below the given threshold  $h$  (Line  
 150 4). Then remaining suffixes are sorted in descending order (Line 5) and the interval  
 151 tree is built upon them (Lines 7-11). The building process comprises from checking  
 152 that current substring *candidate* not intersected with already added substrings to  
 153 *tree* and adding it to *tree*. Finally, algorithm output set of non-intersected substrings  
 154 (clones) of pattern  $p$  in text  $t$ .

---

**Algorithm 5.2** Greedy approximate

---

Input: pattern  $p$ , text  $t$ , threshold value  $h$

Output: Set of non-intersected clones of pattern  $p$  in text  $t$

Pseudocode:

```

1:  $sa = semilocalsa(p, t)$ 
2:  $matrix = sa.getStringSubstringMatrix()$ 
3:  $colmax = smawk(matrix)$ 
4:  $colmax = colmax.filter(it.score \geq h)$ 
5:  $colmax = colmax.sortedByDescending(it.score)$ 
6:  $tree = buildIntervalTree()$ 
7: for  $candidate \in colmax$  do
8:   if  $candidate \cap tree = \emptyset$  then
9:      $tree.add(candidate)$ 
10:  end if
11: end for
12:  $result = tree.toList()$ 
13: return  $result$ 

```

---

155 **THEOREM 5.1.** *Algorithm 5.2 could be solved in  $\max(O(|p| * |t| * v), O(|t| * \log |t|))$*   
 156 *time with  $O(|t| \log |t|)$  space when  $|p| < |t|$  where  $p$  is pattern,  $t$  is text and  $v$  is denom-*  
 157 *inator of normalized mismatch score for semi-local sequence alignment  $w_{normalized} =$*   
 158  *$(1, \frac{\mu}{v}, 0)$  assuming we are storing solution matrix implicitly.*

159 *As shown in section 2 the time complexity of solving semi-localsa is  $O(|p| * |t| * |v|)$*   
 160 *respectively. The space complexity of storing monge matrix of semi-local solution is*  
 161  *$O(|t| * v * \log |t| * v)$  at most due to fact that  $v$  - subbistochasticmatrix has at most  $v$*   
 162 *non-zeros in each row and upon  $v * |t|$  points we build range tree data structure that .*

163 **COROLLARY 5.2.** *Algorithm 5.2 could be solved in  $\max(O(|p| * |t|), O(|t| * \log |t|))$*   
 164 *when  $v = O(1)$ .*

165 **COROLLARY 5.3.** *Algorithm 5.2 could be solved in  $O(|p| * |t| * v)$  when and amount*  
 166 *of clones is small.*

167 *When amount of clones is relatively small and threshold value is set high then*  
 168 *after filtering out  $t$  intervals (Line 4) sorting is performed on  $s$  small set of elements.*  
 169 *Thus this part is dominated by calculating semi-local sa solution.*

THEOREM 5.4. *Algorithm 5.1 could be solved in  $\max(O(|p| * |t| * v), O(|t| * \log |t|))$  time with  $O(|t| \log |t|)$  space when  $|p| < |t|$  where  $p$  is pattern,  $t$  is text and  $v$  is denominator of normalized mismatch score for semi-local sequence alignment  $w_{normalized} = (1, \frac{\mu}{v}, 0)$ .*

*On the first phase of alg*

*The first phase of algorithm requires  $O(|p| * |t| * v)$  with  $O(|t| * v)$  additional space for storing monge matrix implicitly. We denote this matrix, specifically it's lower-left quadrant that refers to string-substring solution as  $M$  with size  $|t| \times |t|$ .*

*Theorema 3.4 First, note that*

*Building structure for rmq queries for staircase matrix requires Theorem 5.8. Given an  $n \times n$  partial Monge matrix  $M$ , a data structure of size  $O(n)$  can be constructed in  $O(n \log n)$  time to answer submatrix maximum queries in  $O(\log \log n)$  time.*

*Proof it*

$$D = \text{diag}(d_1, \dots, d_n)$$

COROLLARY 5.5. *Algorithm 5.1 could be solved in  $\max(O(|p| * |t|), O(|t| * \log |t|))$  when  $v = O(1)$ .*

## 6. Evaluation.

**Semi-local algorithms.** Show performance between lcs and semi-local lcs??? and poor performance of recursive algorithm based on steady ant?

**Approximate matching algorithms.** Show outperforming for different cases between luciv and our algorithm.

Show quality between our new algo and luciv algo (our should be better)

Show that sparse table bad when large?

**7. Conclusion.** Say may be successfully be applied on practice (showed by algorithm luciv updated)

Open problem. — >

Say that need to implement with monge2020 (what we not finished)

Improve algo based on recursive steady ant. Because it's critical for algos based on it.

df[1]

**Acknowledgments.** We would like to acknowledge the assistance of volunteers in putting together this example manuscript and supplement.

## REFERENCES

- [1] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 4th ed., 2013.
- [2] D. V. LUCIV, D. V. KOZNOV, A. SHELIKHOVSKII, K. Y. ROMANOVSKY, G. A. CHERNISHEV, A. N. TEREKHOV, D. A. GRIGORIEV, A. N. SMIRNOVA, D. BOROVKOV, AND A. VASENINA, *Interactive near duplicate search in software documentation*, Programming and Computer Software, 45 (2019), pp. 346–355.