

APPLICATION OF SEMI-LOCAL LCS TO STRING APPROXIMATE MATCHING*

DIANNE DOE[†], PAUL T. FRANK[‡], AND JANE E. SMITH[‡]

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.

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

[‡]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. 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. The algorithm comprises three phases as in [2]. At phase one (Lines 1-3) semi-local sa problem is solved for the pattern p against whole text t . This solution provides access to the string-substring matrix $H_{p,t}^{str-sub}$ which allows performing fast queries of sa score for pattern p against every substring of text t . We apply implicitly transposition and inverse operation on $H_{p,t}^{str-sub}$:

$$(4.1) \quad M[j, i] := -H_{p,t}^{str-sub}[i, j]$$

Note that, inverse operation preserves (*anti*) *Monge* property whereas inverse operation make *anti Monge* matrix *Monge* and vice versa. So, matrix M is *Monge* matrix.

The second phase consist of several steps (Lines 4-6). First, we want to obtain for each prefix of the text t a longest suffix that have a highest similarity with given pattern p with following constraint. The lengths of obtained suffixes should be in $|p| * k \dots \frac{|p|}{k}$ interval where $k \in [\frac{1}{\sqrt{3}}, 1]$. It could be done in several way. For example, direct pass through diagonal with width $w := \frac{|p|}{k} - |p| * k = |p|(\frac{1}{k} - k)$ in $H_{p,t}^{str-sub}$ (see fig) or in M (see fig). The other approach is following. Note that in M is *Monge matrix* and indices is swapped. It allows us to descry this diagonal as approximately $|t|$ square windows of size $w \times w$ i.e a sliding window of step 1 that goes diagonally. Due to length constraint we only interesting in elements that lies in main diagonal and below it. Each of this $W := w \times w$ matrix is *Monge matrix* by definition. This implies that W also totally monotone. If we set to $+\infty$ that lies above diagonal that matrix will remain totally monotone. Thus, we can apply *SMAWK* algorithm to this matrix to find leftmost element that has minimum in a given row with corresponding column position. For our case leftmost means that for each prefix algorithm will detect longest suffix (remember that M is transposed $H_{p,t}^{str-sub}$).

87 Second step, it is simply one way pass through these suffixes with sliding window
 88 of size $\frac{|p|}{t}$ to find for each window most similar suffix with the longest length. Then
 89 resulting set is filtered out that remaining suffixes have score greater or equal to given
 90 threshold $-k_{di}$.

91 The third phase is same as in [2] (Lines 8-12).

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

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

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

$$(4.2) \quad k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2)$$

$$(4.3) \quad L_w = \frac{|p|}{k}$$

$$(4.4) \quad w = |p|(\frac{1}{k} - k)$$

Pseudocode:

```

1:  $W = \text{semilocalsa}(p, t)$  {1st phase}
2:  $H_{p,t}^{str-sub} = \text{semilocalsa}(p, t).stringSubstringMatrix$ 
3:  $M[j, i] = -H_{p,t}^{str-sub}[i, j]$ 
4:  $\text{suffixes} = \text{processDiagonal}(M, L)$  {2d phase}
5:  $W_2 = \text{SuffixMaxForEachWindow}(\text{suffixes}, L_w)$ 
6:  $\text{filter}(W_2, k_{di})$ 
7:  $W_3 = \text{UNIQUE}(W_2)$  {3rd phase unchanged}
8: for  $w \in W_3$  do
9:   if  $\exists w' \in W_3 : w \subset w'$  then
10:     remove  $w$  from  $W_3$ 
11:   end if
12: end for
13: return  $W_3$ 
```

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

94 *Proof it*

$$95 \quad D = \text{diag}(d_1, \dots, d_n)$$

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

97 *Proof it*

$$98 \quad D = \text{diag}(d_1, \dots, d_n)$$

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

102 The first algorithm 5.2 refers to following constraint. There should be found all
 103 non-intersected clones τ_i of pattern p from text t that has the highest similarity score
 104 on the uncovered part of the text t i.e algorithm should perform greedy choice at each

105 step. This is a more intuitive approach i.e like looking for the most similar one every
 106 time. **Formally:**

$$107 \quad (5.1) \quad \tau_i = \arg \max_{l, r \in (t \cap (\cup_{j=1}^{i-1} \tau_j), l < r, t_{l,r} \cap (\cup_{j=1}^{i-1} \tau_j) = \emptyset)} sa(t_{l,r}, p)$$

108 The algorithm proceeds as follows. First, upon string-substring Monge matrix
 109 M of semi-local solution is built data structure for performing range queries on it
 110 denoted by $rmq2D$ (Lines 1).

111 Second, algorithm make recursive call to subroutine *greedy*. The *greedy* routine
 112 perfoms greedy choice of τ_i with maximal alignment within the current uncovered
 113 part of the text $t_{i,j}$. More precisely, it refers to searching maximum value with
 114 corresponding position (row and column) in matrix M within $t_{i,j}$ (starting at i th
 115 position and ending at j th position of text t . It is solved via range queries. When
 116 detected *interval* has alignment score less then threshold it means that no clones
 117 of pattern p are presented in this part of text $t_{i,j}$, and further processing should be
 118 skipped. Otherwise, the founded clone is added to final result and the current part
 119 of the text splits on two smaller parts and processed in the same way. Finally, the
 120 algorithm outputs a set of the non-intersected intervals of clones of pattern p in text
 121 t .

Algorithm 5.1 Greedy subroutine

Input: $rmq2D$ — range maximum query data structure for performing range queries
 on monge matrix M , h — threshold value, i, j — start and end positions of current
 text $t_{i,j}$

Output: Set of non-intersected intervals from $t_{i,j}$

Pseudocode:

$greedy(rmq2D, h, i, j, t_{i,j}) :$

```

1:  $interval = rmq2D.query(i, j, i, j)$ 
2:  $result = \emptyset$ 
3: if  $interval.score < h$  then
4:   return  $result$ 
5: end if
6: if  $interval.i - i \geq 1$  then
7:    $cl = greedy(rmq2D, h, t_{i,interval.i})$ 
8:    $result.add(cl)$ 
9: end if
10: if  $j - interval.j \geq 1$  then
11:    $cl = greedy(rmq2D, h, t_{j,interval.j})$ 
12:    $result.add(cl)$ 
13: end if
14: return  $result$ 

```

Algorithm 5.2 GREEDY-PATTERN BASED NEAR DUPLICATE SEARCH ALGORITHM

Input: monge matrix M that correspond to string-substring matrix for pattern p and text t , threshold value h

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

Pseudocode:

GreedyMathing(M, h, t)

- 1: $rmq2D = buildRMQStructure(M)$
- 2: $result = greedy(rmq2D, 0, |t|, t)$
- 3: **return** $result$

The second algorithm 5.3 uses a less sophisticated approach and a more lightweight one but found fewer duplicates of pattern p (see example ??). The algorithm also follows a greedy approach but instead of looking at the uncovered part of text t at each step it looks at the text t and chooses the first available substring with the highest score that doesn't intersect with already taken substrings. More formally, it approximates algorithm 5.2.

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

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

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

Algorithm 5.3 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** $result$

THEOREM 5.1. Algorithm 5.3 could be solved in $\max(O(|p| * |t| * |v|), O(|t| * \log^2 |t|v))$ time with $O(|t| * v * \log |t| * v)$ space when $|p| < |t|$ where p is pattern, t is text and v is denominator of normalized mismatch score for semi-local sequence alignment $w_{\text{normalized}} = (1, \frac{p}{v}, 0)$ assuming we are storing solution matrix implicitly.

First phase. As shown in section 2 the time complexity of solving semi-local is $O(|p| * |t| * |v|)$. The space complexity of storing monge matrix of semi-local solution is $O(|t| * v * \log |t| * v)$ at most due to fact that v -substochastic matrix has at most v non-zeros in each row and upon these $v * |t|$ points we build two dimensional range tree data structure with $|t| * v * \log |t| * v$ nodes that have report range sum queries in $O(\log^2 |t|v)$ time.

Second phase. SMAWK algorithm requires $O(|t| * q)$ time where q stands for time complexity of random access of monge matrix. Thus, the total time complexity of line 3 is $O(|t| * \log^2 |t|v)$. Filtering and sorting have at most $O(|t|)$ and $O(|t| * \log |t|)$ time complexity. In Line 6 simple initialization of interval tree is performed that requires $O(1)$.

Third phase colmax array has as worst case $O(|t|)$ elements when filtering does not eliminate any substrings. Thus, adding to interval tree (both operation at most require $O(\log |t|)$ time) as well as intersection in (Lines 8-9) will be performed at most $O(|t|)$. Thus, the total complexity of last phase is $O(|t| * \log |t|)$.

As we see, the third phase is dominated by the second phase in terms of running time and second phase is dominated by the space complexity of third phase. Thereby, the total time and space complexity is $\max(O(|p| * |t| * |v|), O(|t| * \log^2 |t|v))$ and $O(|t| * v * \log |t| * v)$ respectively.

COROLLARY 5.2. Algorithm 5.3 could be solved in $\max(O(|p| * |t|), O(|t| * \log |t|))$ when $v = O(1)$.

When $v = O(1)$ we will use simple range tree for orthogonal range queries with $O(\log |t|)$ query time.

COROLLARY 5.3. Algorithm 5.3 could be solved in $O(|p| * |t|)$.

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

THEOREM 5.4. Algorithm 5.2 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 $w_{\text{normalized}} = (1, \frac{p}{v}, 0)$.

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

6. Evaluation.

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

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

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

192 Show that sparse table bad when large?

193 **7. Conclusion.** Say may be successfully be applied on practice (showed by algo-
 194 rithm luciv updated)

195 Open problem.— >

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

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

199 df[1]

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

202 REFERENCES

- 203 [1] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press,
 204 Baltimore, 4th ed., 2013.
- 205 [2] D. V. LUCIV, D. V. KOZNOV, A. SHELIKHOVSKII, K. Y. ROMANOVSKY, G. A. CHERNISHEV,
 206 A. N. TEREKHOV, D. A. GRIGORIEV, A. N. SMIRNOVA, D. BOROVKOV, AND A. VASENINA,
 207 *Interactive near duplicate search in software documentation*, Programming and Computer
 208 Software, 45 (2019), pp. 346–355.