

APPLICATION OF SEMI-LOCAL SA TO APPROXIMATE PATTERN MATCHING

NIKITA MISHIN* AND DANIIL BEREZUN†

Abstract. In the paper we study an application of semi-local sequence alignment algorithms to approximate pattern matching problem. We both developed two new algorithms as well as improved the existing near duplicate search algorithm [?]. The key idea behind the algorithms is a usage of the underlying algebraic structure of semi-local lcs [] together with a novel data structure for submatrix maximum queries in Monge matrices []. We show that the presented algorithms running time and space complexity is ... respectively. We also show that the improved near duplicate search algorithm not only has a better complexity but also preserves all declared properties. Additionally, we show that sticky braid multiplication based semi-local lcs algorithm [] does not perform well in practice because of its 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 common 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-compressed 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 developing new algorithms we improve and significantly outperform the existing one for

*Saint Petersburg State University, Russia (mishinnikitam@gmail.com).

†IntelliJ Labs Co. Ltd., Saint Petersburg, Russia (daniil.berezun@jetbrains.com).

interactive duplicate detection for software documentation []. 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 the 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, transposition 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 comprises several steps (Lines 4-6). First, we want to get for each prefix of the text t the longest suffix that has the highest similarity with the given pattern p with the 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 ways. 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 the following. Note that in M is *Monge matrix* and indices are 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. Because of length constraint we only interesting in elements that lie in the main diagonal and below it (remember, transposition) in these submatrices $w \times w$. Each of these $W := w \times w$ matrix is *Monge matrix* by definition (as a submatrix of *Monge matrix*). This implies that W also totally monotone. If we set to $+\inf$ for elements that lie above the main diagonal that result matrix will remain totally monotone. Thus, we can apply *SMAWK* algorithm to this matrix to find a leftmost element that has a minimum in a given row with a corresponding column position.

88 For our case leftmost means that for each prefix algorithm will detect longest suffix
 89 (remember that M is transposed $H_{p,t}^{str-sub}$).

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

94 The third phase is the 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:  $W = semilocalsa(p, t)$  {1st phase}
2:  $H_{p,t}^{str-sub} = semilocalsa(p, t).stringSubstringMatrix$ 
3:  $M[j, i] = -H_{p,t}^{str-sub}[i, j]$ 
4:  $sufixes = processDiagonal(M, L)$  {2d phase}
5:  $W_2 = SuffixMaxForEachWindow(sufixes, L_w)$ 
6:  $filter(W_2, k_{di})$ 
7:  $W_3 = UNIQUE(W_2)$  {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  $W_3$ 
```

95 **THEOREM 4.1.** *Algorithm 4.1 could be solved in $\max(O(|t| * |p|), O(|t| * \log |t|))$*
 96 *time with $O(|t| \log |t|)$ additional space where p is pattern, t is text when $|p| \leq |t|$,*
 97 *$v = O(1)$ where v is denominator of normalized mismatch score for semi-local sa*
 98 *$w_{normalized} = (1, \frac{\mu}{v}, 0)$.*

99 For each phases of algorithm we provide it's time and space bounds.

100 *First phase.* . We will store solution H of *semi-local sa* by decomposing it to
 101 permutation matrix P of size $O(v * |t| \times v * |t|)$ (Lines 1-3) (ref add). The permutation
 102 matrix can be stored via two permutations of size $v * |t|$ for column and rows. It is
 103 simply two lists of size $v * |t|$. Then, to random access query in specific position i, j of
 104 matrix H we need to check how many points dominated by i, j . It is just pass through
 105 permutations that requires $O(v * |t|)$. Thus, the total time and space complexity of

1st phase is $O(v * |p| * |t|)$ (time complexity for solving *semi-local sa*) and $O(v * |t|)$. Given $v = O(1)$ we have $O(|p| * |t|)$ and $O(|t|)$ respectively. Also random access query for our case is $O(|t|)$.

Second phase. We omit k factor in analysis because when $k \in [\frac{1}{\sqrt{3}}, 1]$ $O(k) = 1$

We will use the first approach described in the algorithm description for this phase. First, although the random access query to a matrix element requires $O(|t|)$. We only need one such query to step on the diagonal. Precisely, to the cell that represents substring $t_{0, |p| * k}$, starting at zero position and ending in $|p| * k$ position. Further we use **Theorem about adjacent cell query** that allows us to perform $O(1)$ access to adjacent elements for given i, j cell in matrix M . Thus, we can visit each cell in the desired diagonal of size at most $O(|t|) * O(|p|)$ in $O(|t| * |p|)$ time in the following way. Process row i' with starting j' (recall it cell by $M[i', j']$) position (go right i.e increment j') until $i' - j' \geq |p| * k$. Then shift by one i' down and j' to right by one if needed (see picture **This about the top left corner**).

When we pass through a slice of the specific column, we also will find the longest suffix with the highest similarity simply by checking elements twice. First for detect maximum score, second for detect the longest suffix among those who have this score. Thus, for storing for each prefix its longest suffix we need additionally $O(|t|)$ space. Also for each substring of length $\frac{p}{k}$ we store similarity score by querying them during diagonal passage because they lie also on this diagonal. Let's denote it by C . At the end of *processDiagonal* we will have $O(t)$ suffixes that require $O(t)$ space for storing them. Then, *processDiagonal* requires $O(|t| + |t| * |p|) = O(|t| * |p|)$ time for processing diagonal with $O(|t| + |p|) = O(|t|)$ additional space.

Further (Line 5), we need to find longest suffix within $O(|p|)$ window with step one in list of size $|t|$ with additional condition that within each window of size $O(\frac{|p|}{k} - |p| * k) = O(|p|)$ the suffix with length $\frac{p}{k}$ have similarity score at least $-k_{di}$. It is simply a one-way pass-through list of suffixes where the processing of each window requires at most $O(|p| + 1) = O(|p|)$. More precisely, first, we check that for current window of size $O(|p|)$ associated suffix has similarity not less then given threshold k_{di} . It is simply lookup for a specific element in C with $O(1)$. If that true, then we need $O(p)$ lookups within *suffixes* to query the most similar and longest one. The total number of such windows at most $O(|t|)$. Thus, *SuffixMaxForEachWindow* requires $O(|t|) * O(|p|) = O(|t| * |p|)$ time with $O(|t|)$ space for storing suffix for each window.

The filtering process (Line 6) is a one-way pass through a list of suffixes W_2 . It requires at most $O(t)$ time.

As we see, the total running time and space complexity of the second phase is $O(|t| * |p|)$ and $O(|t|)$ respectively.

Third phase. The third phase remains unchanged, thus have the same time and space-bound. Note that it possible to perform this phase in-place during a second phase which make the algorithm even faster i.e decrease space and time complexity to $O(|t|)$ and $O(|t| * |p|)$. The third phase is $O(|t| \log |t|)$ at most both for space and running time complexity.

Thus, the total running time is $\max(O(tp), O(t \log t))$ and space complexity $t \log t$. **It be good if we also improve third phase))**

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

First we show, equivalence between similarity functions, then we show that set W_2 from algorithm ?? equals to set W_2 from algorithm 4.1. Let be A_1 a set W_2 from algorithm ?. Let be A_2 a set W_2 from algorithm 4.1. We will show that $A_2 = A_1$.

155 *First part. Take from dimpla*

156 *Second part.* At first algorithm ?? pass through text t with sliding window to
 157 detect those fragments which has similarity above given threshold k_{di} with size $\frac{p}{k}$.
 158 Then within these fragments algorithm detects longest suffixes most similar to pattern
 159 p with size within $pk \dots \frac{p}{k}$ interval. That how A_1 constructed.

160 The second algorithm 4.1 proceed in similar way but it first detects longest suffixes
 161 with size in $pk \dots \frac{p}{k}$ interval for each prefix of text t . Then it proceeds in a such way
 162 that for each window the longest suffix it detected that have similarity above given
 163 threshold h for current window of size $\frac{p}{k}$. That how A_1 constructed.

164 Thus, $A_1 = A_2$ by resulting equivalence of construction.

165 Note that set A_1 contains only those fragments of size $\frac{p}{k}$ from text t that close
 166 enough to pattern p i.e

167 The fragment from W_1 then shrunked. It means that after second phase set W_2
 168 will have size of W_1 .

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

172 The first algorithm 5.2 refers to following constraint. There should be found all
 173 non-intersected clones τ_i of pattern p from text t that has the highest similarity score
 174 on the uncovered part of the text t i.e algorithm should perform greedy choice at each
 175 step. This is a more intuitive approach i.e like looking for the most similar one every
 176 time. *Formally:*

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

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

181 Second, algorithm make recursive call to subroutine *greedy*. The *greedy* routine
 182 perfoms greedy choice of τ_i with maximal alignment within the current uncovered
 183 part of the text $t_{i,j}$. More precisely, it refers to searching maximum value with
 184 corresponding position (row and column) in matrix M within $t_{i,j}$ (starting at i th
 185 position and ending at j th position of text t . It is solved via range queries. When
 186 detected *interval* has alignment score less then threshold it means that no clones
 187 of pattern p are presented in this part of text $t_{i,j}$, and further processing should be
 188 skipped. Otherwise, the founded clone is added to final result and the current part
 189 of the text splits on two smaller parts and processed in the same way. Finally, the
 190 algorithm outputs a set of the non-intersected intervals of clones of pattern p in text
 191 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:

$GreedyMatching(M, h, t)$

```

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):

$$(5.2) \quad a[j] = \max_{i \in 0..j} sa(p, t[i, j]), j \in 0..|t|$$

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 ≥ h)
5: colmax = colmax.sortedByDescending(it.score)
6: tree = buildIntervalTree()
7: for candidate ∈ colmax do
8:   if candidate ∩ tree = ∅ 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 $w_{normalized} = (1, \frac{\mu}{v}, 0)$ 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* – subbistochasticmatrix 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

235 $O(\log|t|)$ query time.

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

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

240 THEOREM 5.4. Algorithm 5.2 could be solved in $\max(O(|p| * |t| * v), O(|t| * \log |t|))$
 241 time with $O(|t| \log |t|)$ space when $|p| < |t|$ where p is pattern, t is text and v is denom-
 242 inator of normalized mismatch score for semi-local sequence alignment $w_{normalized} =$
 243 $(1, \frac{p}{v}, 0)$.

244 On the first phase of alg

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

248 Theorema 3.4 First, note that

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

252 *Proof it*

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

254 COROLLARY 5.5. Algorithm 5.2 could be solved in $\max(O(|p| * |t|), O(|t| * \log |t|))$
 255 when $v = O(1)$.

256 6. Evaluation.

257 *Research questions.* To present evaluation of algorithms we need to investigate
 258 the performance of algorithms that computes solution for *semi-local* problem first. It is
 259 justified by fact that all described algorithms heavily based on it. Thus, the following
 260 research questions have been settled by evaluation in this paper:

- 261 1. RQ1. Does both theoretical algorithms for solving *semi-local* problem appli-
 262 cable in practice? (perform well on practice)
- 263 2. RQ2. How differ in terms of running time computation of *semi-local lcs* and
 264 *prefix lcs*?

265 We had implemented algorithms and required data structures to answer RQ1
 266 and RQ2 ¹. Evaluation have been done in laptop machine with operation system
 267 *Ubuntu18.04* that have processor *Intel-Core i5* with *16GB* RAM.

268 *RQ1.* On fig. ?? the comparison between two algorithm for computing *semi-*
 269 *local lcs* is presented. The plot marked as *recursive* refers to the algorithm based
 270 on steady ant multiplication of associated sticky braids. The second one *reducing*
 271 refers to algorithm that based on reducing the associated unreduced sticky braid to
 272 reducing one.

273 Although both algorithms have the same theoretical running time, the figure com-
 274 pletely shows that there are significant differences in practice. The complex recursive
 275 structure of the algorithm by fast multiplication of sticky braids makes it inapplicable
 276 in practice for long input. Nonetheless, such complex stucture with combination of
 277 steady ant multiplication indeed allows to get rid of one v when computing *semi-*
 278 *local sa* (see fix ??). The recursive structure of multiplication itself is also a subject
 279 of required optimizations due to fact that it used in several theoretical algorithms.

¹add link to github

For example, in solution for *Window substring* problem or *Bounded Length Smith-Waterman alignment* (implicitly).

RQ2. On fig ?? the comparison between computing *prefix lcs* and *semi-local lcs* is presented. More precisely, the comparison among computing prefix lcs via dynamic programming with explicit (denoted by *prefix-lcs*) and implicit (denoted by *light-prefix*) construction of 2D matrix and *semi-local lcs* via reducing approach is presented. The fig ?? show that computation of *semi-local lcs* not only applicable to large input but also comparable with computing of simple *prefix lcs*. The difference between speed computation is relatively subtle.

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.