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 (sa) algorithms to approximate pattern matching problem. We both developed two new algorithms as well as improved the existing near duplicate search algorithm (Programming and Computer Software'19). The key idea behind the algorithms is a usage of the underlying algebraic structure of semi-local sa (Tiskin, 2007) together with a novel data structure for submatrix maximum queries in Monge matrices (TALG'20). We also show that the improved near duplicate search algorithm not only has a better complexity but also preserves all declared properties. We show that the presented algorithms running time and space complexity are $O(max(|t||p|, \frac{|t| \log^2 |t|}{\log \log |t|}))$ and $O(|t|)$ for the first one and $O(max(|t||p|, |t| \log |t|))$ and $O(|t| \log |t|)$ for the last two, respectively, where t is a text, p — pattern, and $v = O(1)$ is denominator of normalized mismatch score for semi-local sequence alignment.

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 generalazation 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-

*Saint Petersburg State University, Russia (mishinnikitam@gmail.com).
†IntelliJ Labs Co. Ltd., Saint Petersburg, Russia (daniil.berezun@jetbrains.com).

oping new algorithms we improve and significantly outperform the existing one for 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 pattern matching. The approximate pattern matching problem ($AMatch$) defined as follows. Given text t, pattern p and some threshold h the *approximate pattern matching* problem ask for all substrings from text t that have similarity score with given pattern p at least h according to some similarity function g.

There exist different kinds of extensions and particular cases of this problem. For example, *complete approximate pattern matching* (*CompleteMatch*) that ask for substrings of text t that are exact clones of pattern p. The approach for this special case of $AMatch$ is usage of well-know algorithms such as Aho-Korasic, BouerMurr, Knuth-Morris-Pratt, and so on. The latter one have optimal running time complexity of $O(|p| + |t|)$ for *CompleteMatch* problem[]. *Approximate pattern matching with k mismatches* is an another example of special case []. The search of *pattern with wild-card symbols* or search set of patterns in text t[], multidimensional $AMatch$[] , search with lenght constraint of detected duplicates [] are examples of such extension. There exits many more examples of constrainsts, extensions and special cases of $AMatch$ problem[].

The one of the common approach to solve approximate pattern matching is the usage of solution of string similarity problem. Latter represent a set of fundamental problems such as *edit distance*, *longest common subsequence*, *sequence alignment*. In this paper we primarily focuses on the usage of latter two when developing algorithms.

Recently there have been developed algorithm for solving interesting extension of $AMatch$ problem with length constarint[**?**]. Although their algorithm have poor result in terms of running time complexity, the proposed solution possesses a completnessess proprety i.e it founds *all* non-intersected clones of pattern p with specified similar-ity threshold and length constraint on matching substrings. Thus, this algorithm is an subject of interest in this paper. The complete description of algorithm and its improved version may be found in section **??** respectively.

2.2. Semi-local lcs. First of all we give definition of *lcs* and *sa*.

DEFINITION 2.1. *Given two strings a and b the longest common subsequence (*LCS*)* *problem ask for the maximal length of the longest common subsequence of a and b* *(*lcs(a,b)*).*

In other words, LCS problem asks about maximal lcs score of two given string a and b ($lcs(a, b)$).

DEFINITION 2.2. *Given two strings a and b and scoring scheme $w = (w_+, w_0, w_-)$* *the sequence alignment (*SA*) problem ask for the maximal alignment score between a* *and b (sa(a,b)).*

Scoring scheme determines how calulate alignment score of two aligned sequences. If pair of character in aligned sequences are matches (equals) then this pair contributes to final alignemnt score w_+, if their mismatch it contributes w_0. If symbol α of one of the sequences is not aligned with any other symbol from other sequence it means that

α is aligned with *gap*. Thus, this pair contributes w_0. The scoring scheme calculates as follows:

$$(2.1) \quad \begin{aligned} sa(a, b, w) &= w_+ k^+ + w_0 k^0 + w_-(|a| + |b| - 2k^+ - 2k^0) = \\ &\quad k^+(w_+ - 2w_-) + k^0(w_0 - 2w_-) + w_-(|a| + |b|) \end{aligned}$$

The k^+ states for the number of matching symbols, k^- — mismatched symbols.

Note that LCS is a special case of SA when scoring scheme is $(1, 0, 0)$.

Both described problems are solved by classical dynamic programming algorithm and have running time complexity $O(|a||b|)$. LCS and SA allow you to find how much whole given strings are similar i/e how similar two string in a global sense.

In many cases, this is not enogh. There also exist fully local version of these problems and semi-local one. The last one is in sight of this paper due to natural applicability to approximate pattern matching.

2.3. Semi-local lcs. Given two strings a and b the semi-local lcs is asks about lcs scores for following:

1. *string-substring*: whole a against every substring of b
2. *substring-string*: whole b against every substring of a
3. *prefix-suffix*: every prefix of a against every suffix of b
4. *suffix-prefix*: every prefix of b against every suffix of a

The following *semi-local lcs matrix* associated with the defined *semi-local lcs*.

DEFINITION 2.3. *The* semi-local lcs matrix $H_{a,b}$ *for strings a,b defined as follows:*

$$(2.2) \quad H_{a,b}[i, j] = if(j \leq i)j - ielselcs(a, b^{pad}[i, j])$$

where $i \in [-|a| : |b|], j \in [0 : |a| + |b|]$.

The semi-local lcs matrix $H_{a,b}$ comprises from four quadrant associated with described subproblems:

$$(2.3) \quad H_{a,b} = \begin{bmatrix} H_{a,b}^{suf-pre} & H_{a,b}^{sub-str} \\ H_{a,b}^{str-sub} & H_{a,b}^{pre-suf} \end{bmatrix}$$

In [?] is proved that H_a, b is unit anti Monge.

DEFINITION 2.4. *Matrix H called (anti) Monge matrix if*

$$H[i, j] + H[i', j'](\geq) \leq H[i, j'] + H[i', j], \forall i <= i', j \leq j'$$

DEFINITION 2.5. *Let $H[0 : m, 0 : n]$ be a matrix. $H^{\square}[0 : m - 1, 0 : n - 1]$ constructed as a result of taken cross difference between secondary and first diagonal for all adjacent 2 by 2 squares called* cross-difference *matrix of H*

DEFINITION 2.6. *Matrix H called unit anti Monge matrix if H is (anti) Monge matrix and its* cross-difference matrix $(-)H^{\square}$ *is permutation matrix.*

The example of unit anti Monge matrix is following:

$$(2.4) \quad \begin{bmatrix} 0 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}^{\square} = \begin{bmatrix} (2 + 0) - (1 + 0) & (3 + 1) - (2 + 2) \\ (1 + 0) - (1 + 0) & (2 + 1) - (1 + 1) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

DEFINITION 2.7. *Let $H[0 : m - 1, 0 : n - 1]$ be a matrix. $H^{\nearrow}[0 : m - 1, 0 : n - 1]$ constructed as dominance sum matrix difference between secondary and first diagonal for all adjacent 2 by 2 squares called* cross-difference *matrix of H*

129   *We particulary interesting in lower left quadrant that refers to string substring*
130   *problem:*

131   (2.5) $$H_{a,b}^{str-sub}[i,j] = lcs(a, b[i,j]), i, j \in [0, |b|]$$

132   **2.4. Semi-local sa.** *The semi-local sequence alignment (sa) is a generalazation*
133   *of semi-local lcs in same sense as sequence alignemnt is generelaztion of lcs.*
134   *Given two strings a and b and scoring scheme $w = (w_+, w_0, w_-)$ the semi-local*
135   *sa asks about sa scores for following:*
136   1. string-substring*: whole a against every substring of b*
137   2. substring-string*: whole b against every substring of a*
138   3. prefix-suffix*: every prefix of a against every suffix of b*
139   4. suffix-prefix*: every prefix of b against every suffix of a*
140   *The associated matrix for* semi-local sa *is defined analogously as for* semi-local
141   lcs.
142   *blown-up tecnhique.*
143   *theorems of running time complexity.*

144   **2.5. Monge matrix.** Describe monge property
145   Say about range queries (about soda12, soda14 and new result that we will be
146   used)

147   **2.6. Near-duplicate detection algorithm.** Describe luciv algo

148   **3. Related work.** ?????
149   could mention about approximation. Need discuss

150   **4. Algorithm for near duplicate detection.** We now describe an improved
151   version of Luciv et.al. algorithm [2] by utilizing a *semi-local sa* solution. Then we
152   present proof that improved version preserves completnesess property. It is achieved
153   by imitating all phases of the algorithm.

154   **4.1. Algorithm description.** The algorithm comprises three phases as in [2].
155   At phase one (Lines 1-3) *semi-local sa* problem is solved for the pattern p against the
156   whole text t. This solution provides access to the string-substring matrix $H_{p,t}^{str-sub}$
157   which allows performing fast queries of *sa* score for pattern p against every substring
158   of text t. We apply implicitly transposition and inverse operation on $H_{p,t}^{str-sub}$:

159   (4.1) $$M[j,i] := -H_{p,t}^{str-sub}[i,j]$$

160   Note that, transposition operation preserves (*anti*) *Monge* property whereas inverse
161   operation make *anti Monge* matrix *Monge* and vice versa. So, matrix M is *Monge*
162   matrix.
163   The second phase comprises several steps (Lines 4-6). First, we want to get for
164   each prefix of the text t the longest suffix that has the highest similarity with the
165   given pattern p with the following constraint. The lengths of obtained suffixes should
166   be in $|p| * k..\frac{|p|}{k}$ interval where $k \in [\frac{1}{\sqrt{3}}, 1]$. It could be done in several ways. For
167   example, direct pass through diagonal with width $w := \frac{|p|}{k} - |p| * k = |p|(\frac{1}{k} - k)$ in
168    $H_{p,t}^{str-sub}$ (see fig) or in M (see fig). The other approach is the following. Note that in
169   M is *Monge matrix* and indices are swapped. It allows us to descry this diagonal as
170   approximately $|t|$ square windows of size $w \times w$ i.e a sliding window of step 1 that goes
171   diagonally. Because of length constraint we only interesting in elements that lie in the
172   main diagonal and below it (remember, transposition) in these submatrices $w \times w$.

173   Each of these $W := wxw$ matrix is *Monge matrix* by definition (as a submatrix
174   of *Monge matrix*). This implies that W also totally monotone. If we set to $+\inf$
175   for elements that lie above the main diagonal that result matrix will remain totally
176   monotone. Thus, we can apply *SMAWK* algorithm to this matrix to find a leftmost
177   element that has a minimum in a given row with a corresponding column position.
178   For our case leftmost means that for each prefix algorithm will detect longest suffix
179   (remember that M is transposed $H_{p,t}^{str-sub}$).
180   The second step, it is one-way pass through these suffixes with a sliding window
181   of size $\frac{|p|}{t}$ to find for each window most similar suffix with the longest length. Then
182   the resulting set is filtered out that remaining suffixes have a score greater or equal
183   to given threshold $-k_{di}$.
184   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, similiarity measure $k \in [\frac{1}{\sqrt{3}}, 1]$
Output: Set of non-intersected clones of pattern p in text t

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

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

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

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 ∈ W_3 do
 9:    if ∃w' ∈ W_3 : w ⊂ w' then
10:       remove w from W_3
11:    end if
12: end for
13: return W_3
```

---

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

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

190   *First phase.* . We will store solution H of *semi-local sa* by decomposing it to
191   permutation matrix P of size $O(v*|t| \times v*|t|)$ (Lines 1-3) (ref add). The permutation
192   matrix can be stored via two permuations of size $v*|t|$ for column and rows. It is
193   simply two lists of size $v*|t|$. Then, to random access query in specific position i,j of
194   matrix H we need to check how many points dominated by i,j. It is just pass through
195   permutations that requires $O(v*|t|)$. Thus, the total time and space complexity of
196   1st phase is $O(v*|p|*|t|)$ (time complexity for solving *semi-local sa*) and $O(v*|t|)$.
197   Given $v = O(1)$ we have $O(|p|*|t|)$ and $O(|t|)$ respectively. Also random access query
198   for our case is $O(|t|)$.
199   *Second phase.* We omit k factor in analysis because when $k \in [\frac{1}{\sqrt{3}}, 1]$ $O(k) = 1$
200   We will use the first approach described in the algorithm description for this
201   phase. First, although the random access query to a matrix element requires $O(|t|)$.
202   We only need one such query to step on the diagonal. Precisely, to the cell that
203   represents substring $t_{0,|p|*k}$, starting at zero position and ending in $|p|*k$ position.
204   Further we use Theorem about adjacent cell query that allows us to perform $O(1)$
205   access to adjacent elements for given i,j cell in matrix M. Thus, we can visit each
206   cell in the desired diagonal of size at most $O(|t|) * O(|p|)$ in $O(|t|*|p|)$ time in the
207   following way. Process row i' with starting j' (recall it cell by $M[i',j']$) position (go
208   right i.e increment j') until $i' - j \geq |p|*k$. Then shift by one i' down and j' to right
209   by one if needed (see picture This about the top left corner).
210   When we pass through a slice of the specific column, we also will find the longest
211   suffix with the highest similarity simply by checking elements twice. First for detect
212   maximum score, second for detect the longest suffix among those who have this score.
213   Thus, for storing for each prefix its longest suffix we need additionally $O(|t|)$ space.
214   Also for each substring of length $\frac{p}{k}$ we store similarity score by querying them during
215   diagonal passage because they lie also on this diagonal. Let's denote it by C. At
216   the end of *processDiagonal* we will have $O(t)$ suffixes that require $O(t)$ space for
217   storing them. Then, *processDiagonal* requires $O(|t| + |t|*|p|) = O(|t|*|p|)$ time for
218   processing diagonal with $O(|t| + |p|) = O(|t|)$ additional space.
219   Further (Line 5), we need to find longest suffix within $O(|p|)$ window with step
220   one in list of size $|t|$ with additional condition that within each window of size $O(\frac{|p|}{k} -$
221    $|p|*k) = O(|p|)$ the suffix with length $\frac{p}{k}$ have similarity score at least $-k_{di}$. It is
222   simply a one-way pass-through list of suffixes where the processing of each window
223   requires at most $O(|p|+1) = O(|p|)$. More precisely, first, we check that for current
224   window of size $O(|p|)$ associated suffix has similarity not less then given threshold
225   k_{di}. It is simply lookup for a specific element in C with $O(1)$. If that true, then we
226   need $O(p)$ lookups within *suffixes* to query the most similar and longest one. The
227   total number of such windows at most $O(|t|)$. Thus, $SuffixMaxForEachWindow$
228   requires $O(|t|) * O(|p|) = O(|t|*|p|)$ time with $O(|t|)$ space for storing suffix for each
229   window.
230   The filtering process (Line 6) is a one-way pass through a list of suffixes W_2. It
231   requires at most $O(t)$ time.
232   As we see, the total running time and space complexity of the second phase is
233    $O(|t|*|p|)$ and $O(|t|)$ respectively.
234   *Third phase.* The third phase remains unchanged, thus have the same time and
235   space-bound. Note that it possible to perform this phase in-place during a second
236   phase which make the algorithm even faster i.e decrease space and time complexity
237   to $O(|t|)$ and $O(|t|*|p|)$. The third phase is $O(|t|log|t|)$ at most both for space and
238   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 completnesses 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 $A_2 = A_1$.

First part. Take from dimpla

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

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

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

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

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

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

$$(5.1) \qquad \tau_i = \operatorname*{arg\,max}_{l,r \in (t \cap (\cup_{j=1}^{i-1} \tau_j)), l < r, t_{l,r} \cap (\cup_{j=1}^{i-1} \tau_j) = \emptyset} sa(t_{l,r}, p)$$

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

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

Algorithm 5.1 Greedy subroutine

Input: $rmq2D$— range maximum query data structure for perfoming range queries on monge matrix M, h — theshold 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, theshold 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):

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

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

295 tree is built upon them (Lines 7-11). The building process comprises from checking
296 that current substring *candidate* not intersected with already added substrings to
297 *tree* and adding it to *tree*. Finally, algorithm output set of non-intersected substrings
298 (clones) of pattern p in text t.

Algorithm 5.3 Greedy approximate

Input: pattern p, text t, theshold 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** $candidate \in colmax$ **do**
8:     **if** $candidate \cap tree = \emptyset$ **then**
9:        $tree.add(candidate)$
10:    **end if**
11: **end for**
12: $result = tree.toList()$
13: **return** $result$

---

299   THEOREM 5.1. *Algorithm 5.3 could be solved in* $\max(O(|p|*|t|*|v|), O(|t|*\log^2|t|v))$ ■
300 *time with* $O(|t| * v * \log|t| * v)$ *space when* $|p| < |t|$ *where p is pattern, t is text and*
301 *v is denominator of normalized mismatch score for semi-local sequence alignment*
302 *$w_{normalized} = (1, \frac{\mu}{v}, 0)$ assuming we are storing solution matrix implicitly.*
303   First phase. *As shown in section 2 the time complexity of solving $semi-localsa$*
304 *is $O(|p|*|t|*|v|)$. The space complexity of storing monge matrix of semi-local solution*
305 *is $O(|t|*v*\log|t|*v)$ at most due to fact that $v-subbistochasticmatrix$ has at most*
306 *v non-zeros in each row and upon these $v*|t|$ points we build two dimensional range*
307 *tree data structure with $|t|*v*\log|t|*v$ nodes that have report range sum queries in*
308 *$O(\log^2|t|v)$ time.*
309   Second phase. *$SMAWK$ algorithm requires $O(|t|*q)$ time where q stands for time*
310 *complexity of random access of monge matrix. Thus, the total time complexity of line*
311 *3 is $O(|t|*\log^2|t|v)$. Filtering and sorting have at most $O(|t|)$ and $O(|t|*log|t|)$ time*
312 *complexity. In Line 6 simple intialization of interval tree is perfomed that requires*
313 *$O(1)$.*
314   Third phase *colmax array has as worst case $O(|t|)$ elements when filtering does*
315 *not eliminate any substrings. Thus, adding to interval tree (both operation at most*
316 *require $O(\log|t|)$ time) as well as intersection in (Lines 8-9) will be perfomed at most*
317 *$O(|t|)$. Thus, the total complexity of last phase is $O(|t| * \log t)$.*
318   *As we see, the third phase is dominated by the second phase in terms of run-*
319 *ning time and second phase is dominated by the space complexity of third phase.*
320 *Thereby,the total time and space complexity is $\max(O(|p| * |t| * |v|), O(|t| * \log^2|t|v))$*
321 *and $O(|t| * v * \log|t| * v)$ respectively.*

322   COROLLARY 5.2. *Algorithm 5.3 could be solved in* $\max(O(|p| * |t|), O(|t| * \log|t|))$
323 *when $v = O(1)$.*
324   *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 perfomed 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* $w_{normalized} = (1, \frac{\mu}{v}, 0)$.

On the first phase of alg

The first phase of algorithm requires $O(|p| * |t| * v)$ *with* $O(|t| * v)$ *additional space for stroring 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

$$D = \operatorname{diag}(d_1, \ldots, d_n)$$

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

6. Evaluation.

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

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

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

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

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

---

[1]add link to github

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

RQ2. On fig **??** the comparision between computing *prefix lcs* and *semi-local lcs* is presented. More preciesely, the comparison among computing prefix lcs via dynamic programing 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 succesfully 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]

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.