# 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 [**?**]. 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).



*This manuscript is for review purposes only.*

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.** The section provides some backgraound, definitions, and theoremes required for developed algortithms.

**2.1. Approximate pattern matching.** The approximate pattern matching problem (*AMatch*) defined as follows. Given text $t$, pattern $p$, simularity function $g$, and some threshold $h$ the *approximate pattern matching* problem asks for all substrings from text $t$ that have similarity score with given pattern $p$ at least $h$ according to a function $g$.

There exist different extensions and particular cases of the problem. The most familiar case, *complete approximate pattern matching* (*CompleteAMatch*) that asks for all substrings of text $t$ that are exact clones of pattern $p$. CompleteAMatch can be solved by a number well-knonwn algorithms such as Aho–Korasic, Bouer–Murr, Knuth–Morris–Pratt, and so on. The optimal CompleteAMatch solution running time is $O(|p| + |t|)$ [**?**]. Other special cases of AMatch are *approximate pattern matching with k mismatches* [**?**], search of *pattern with wildcard symbols* [**?**], multidimensional *AMatch* [**?**], AMatch with a length constraint on the resulting substrings [**?**], and many more [**?**].

One of the common approaches to solving AMAtch problem is the utilization of string similarity problem solutions. Latter represents a set of fundamental problems such as *edit distance*, *longest common subsequence*, and *sequence alignment*. TODO: In the paper, we primarily focus on the usage of the latter two when developing algorithms.

Recently there has been developed an algorithm for solving one particular *AMatch* extension — the one with a length constarint on the resulting substrings [**?**]. TODO: Describe why the problem is actual. Despite the algorithm TODO: has poor result in terms of running time complexity, the proposed solution possesses a completeness property, i.e it founds *all* non-intersected clones of a given pattern with specified similarity threshold and length constraint on matching substrings. Precisely due to the completeness property the algorithm is of intereset in this paper. The algorithm is described in section **??** while the developed improved version is presented in section **??**.

**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 $\alpha$ of one of the sequences is not aligned with any other symbol from other sequence it means that $\alpha$ is aligned with *gap*. Thus, this pair contributes $w_0$. The scoring scheme calculates as follows:

$$sa(a,b,w) = w_+k^+ + w_0k^0 + w_-(|a|+|b|-2k^+-2k^0) = \\ k^+(w_+-2w_-) + k^0(w_0-2w_-) + w_-(|a|+|b|) \tag{2.1}$$

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:*

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

*where* $i \in [-|a| : |b|], j \in [0 : |a|+|b|]$ *and* $b^{pad} = ?^{|a|}b?^{|a|}$, ?— *wildcard symbol that matches any other symbol.*

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

$$H_{a,b} = \begin{bmatrix} H^{suf-pre}_{a,b} & H^{sub-str}_{a,b} \\ H^{str-sub}_{a,b} & H^{pre-suf}_{a,b} \end{bmatrix} \tag{2.3}$$

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

$$H[i,j] + H[i^{'},j^{'}](\ge) \le H[i,j^{'}] + H[i^{'},j], \forall i <= i^{'}, j \le j^{'}$$

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:

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

Definition 2.7. *Let $H[0 : m-1, 0 : n-1]$ be a matrix. $H^{\nearrow}[0 : m, 0 : n]$ constructed as sum of element that lies below and left given cell $i, j$ in matrix $H$ called* dominance-sum *matrix of $H$*

The example dominance sum matrix:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{\nearrow} = \begin{bmatrix} 0+0+0 & 1 & 1+1 \\ 0+0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.5}$$

In [**?**] is proved that $H_{a,b}$ is unit anti Monge. Also it is proved that this matrix may be decomposed to permuation matrix i.e into *cross-difference* matrix. It allows to store $H_{a,b}$ implicitly and query any element of $H_{a,b}$ via dominance sum query (orthogonal range queries). Thus, there may be several ways to storing matrix $H_{a,b}$ or one of it quadrant implicitly. A simple storing of two list of permutation gives $O(|a| + |b|)$ space and time complexity with $O(|a| + |b|)$ orthogonal range queries (need to check how many points dominated by given point), whereas more sophisticated approach requires $O(|a| + |b|)$ space with $O((|a| + |b|)\sqrt{\log{(|a| + |b|)}})$ preprocessing time and allows to query any point of $H$ in $O(\frac{\log(|a|+|b|)}{\log\log(|a|+|b|)})$ time.

The one useful proposition of $H$ is following.

Proposition 2.8. *[] Given a permutation matrix $P$ and the value $P^{\nearrow}[i; j]$, the values $P^{\nearrow}[i+-1; j], P^{\nearrow}[i; j+-1]$, where they exist, can be queried in time $O(1)$[].*

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

$$H^{str-sub}_{a,b}[i,j] = lcs(a, b[i,j]), i,j \in [0, |b|] \tag{2.6}$$

There exists several algorithms second on,recursive not described as i see that solve *semi-local lcs*. Both have the optimal running time $O(|a||b|)$ for given dynamic problem[]Impossibility faster then pt.

**2.4. Semi-local sa.** The semi-local sequence alignment (sa) is a generalazation of semi-local lcs in same sense as sequence alignemnt is generelaztion of lcs.

Given two strings $a$ and $b$ and scoring scheme $w = (w_+, w_0, w_-)$ the semi-local sa asks about sa 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 associated matrix for *semi-local sa* is defined analogously as for *semi-local lcs*.

The approach for solving *semi-local sa* is as follows. The problem reduced to *semi-local lcs*. First, note that scoring scheme in 2.1 may be simplified by so called normalization[]:

$$\begin{gathered} w = (w_+, w_0, w_-) \to (w_+ + 2x, w_0 + 2x, w_- + x) = \\ (\frac{w_+ + 2x}{w_+ + 2x}, \frac{w_0 + 2x}{w_+ + 2x}, \frac{w_- + x}{w_+ + 2x})_{x=-w_-} = (1, \frac{\mu}{v}, 0) \end{gathered} \tag{2.7}$$

The resulted scoring scheme $w_{normalized} = (1, \frac{\mu}{v}, 0)$ called normalized scoring scheme.

Then to query initial score $sa$ for scoring scheme $w$ knowing $sa_{normalized}$ for $w_{normalized}$ you need to apply reverse regularization:

$$sa(a, b, w) = sa_{normalized}(w_+ - 2w_-) + w_-(|a| + |b|) \tag{2.8}$$

The blown-up tecnhique is applied after reducing scoring scheme which increases both input strings in $v$ times. Nonetheless, only one of the described algorithm time complexity increases in $v^2$ times, the second one only $v$. Bad sentecnce. The space complexity also increses by factor $v$.

For detalied description we refer readers to TISKIN BOOK[].

**2.5. Range maximum/minimum queries.** Range maximum/minimum queries (*rmq*) (submatrix query) refers to search maximum/minimum element in submatrix $[i_1 : i_2] \times [j_1 : j_2]$ of given matrix $M$ of size $n \times n$. The associated data strucutre that can report maximum/minimum element in any submatrix query called *range maximum/minimum data structure*.

For the generic case of Matrix $M$ it is not possible to achieve running time faster then $O(n^2)$ due to fact that storing matrix $M$ requires $O(n^2)$.

Nonetheless, the situation is changed if we consider special cases such as Monge matrices. There have been several researches over several decades about rmq on monge matrices [].

The recent research achives following result[].

THEOREM 2.9. *[] Given an $n \times n$ 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 when random access to Monge matrix is $O(1)$.*

THEOREM 2.10. *[] Given an $n \times n$ staircase[1] 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 when random access to Monge matrix is $O(1)$.*

THEOREM 2.11. *[] Given an $n \times n$ partial Monge matrix[2] $M$, 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 when random access to Monge matrix is $O(1)$.*

The above results applies both to range minimum queries and to monge matrices with non-constant access $O(\beta)$ to queries. The latter one, costs in increased construction time and query time by factor $\beta$.

**2.6. Near-duplicate detection algorithm.** First, we denote several parameters, that is used in algorithm []. $k$— constant in interval $[\frac{1}{\sqrt{3}}, 1]$ that set similarity measure. A window $w$ of size $L_w = |p|/k$ is to process text $t$ with sliding window of one symbol step. $k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2)$ is threshold value for edit distance. $I$ — interval of size $[|p|k, \frac{|p|}{k}]$ that set boundaries for lenght of matching substrings. $d_{di}$ — function that measure similarity between two strings.

The algorithm comprises of three phases.

At the first phrase text $t$ is processed with sliding window of size $L_w$ with one symbol step. Further, substrings that corrsepond to window $w$ compared using edit distance[3] and if $d_{di}(p, t_w) \leq k_{di}$ i.e close enough, then they saved to set $W_1$ to be further proceeded.

[1] Defintion

[2] Definition

[3] Authors of [] used lcs edit distance — where operations substituion,removoval, addition of one symbol costs 2,1,1 respectively

On the second phase each of the detected substrings in $W_1$ are shrunk i.e they lenght could be decreased. More preciesely, within each of the element of $W_1$ the largest one substring with legnth fall in $I$ that most similar to pattern $p$ according to $d_{di}$ is selected. The set $W_2$ is a result of this phase.

At the final third phase set $W_2$ iterated over to remove elements that fully contains in ohter elements of $W_2$ or duplicates.

**Algorithm 2.1** PATTERN BASED NEAR DUPLICATE SEARCH ALGORITHM

[] Input: pattern $p$, text $t$, $k$ —similiarity measure
Output: Set of non-intersected clones of pattern $p$ in text $t$
Pseudocode:

$W_1 = \emptyset$ {1st phase}
**for** $\forall w_1 : w_1 \in t \wedge |w_1| = L_w$ **do**
    **if** $d_{di} \leq k_{di}$ **then**
        add $w_1$ to $W_1$
    **end if**
**end for**
$W_2 = \emptyset$ {2d phase}
**for** $w \in W_1$ **do**
    $w_2^{'} = w$
    **for** $l \in I$ **do**
        **for** $\forall w_2 : w_2 \subseteq w \wedge |w_2| = l$ **do**
            **if** $Compare(w_2, w_2^{'}, p)$ **then**
                $w_2^{'} = w_2$
            **end if**
        **end for**
    **end for**
    add $w_2^{'}$ to $W_2$
**end for**
$W_3 = UNIQUE(W_2)$
{3rd phase }
**for** $w \in W_3$ **do**
    **if** $\exists w^{'} \in W_3 : w \subset w^{'}$ **then**
        $remove\ w\ from\ W_3$
    **end if**
**end for**
**return** $W_3$

*Running time analysis. 1st phase.*The first phase requires at most $O(|t||p|^2)$ due to fact that computing cost of edit distance is $|p|^2$ for strings of size $O(|p|)$ and we need to process $O(|t| - \frac{|p|}{k}) = O(|t|)$ windows. *2nd phase.* The cardinality of set $W_1$ at worst case be $O(|t|)$. Thus, first loop will be iterated over $O(|t|)$ times. The second loop refers to iteration over $I$ set with cardinality $O(\frac{|p|}{k} - |p|k) = O(|p|)$. The third loop requires at most $O(|p|)$ since we again goes with sliding window. The compare operation requires at most $O(|p|^2)$ since both stirng of size $O(|p|)$. Thus, the total running time compleixty of second phase is $O(|p|^4|t|)$ at worst case.

*3rd phase.* Note that cardinality of set $W_2 == W_1$ and thus at worst case is $O(|t|)$. Then, this phase requires $O(|t| \log |t|)$ time.

Thus,the total time complexity of algirthms estimates as $O(max(|t||p|^4, |t| \log |t|))$.

Algorithm also possesses completnesses property defined in their paper. To detailed description refer to []. We just state their following theorem:

THEOREM 2.12. *[**?**] For any $p \in t$ , $k \in (\frac{1}{\sqrt{3}}, 1]$ and near duplicate group $G$ of fragment $p$ with similarity $k$ the criterion of* completeness *is satisfied in respect to the output of phase 2.*

Theorem states that if pattern $p$ is presented in text $t$ then all duplicates of it would be found in text $t$.

**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 [**?**] by utilizing a *semi-local sa* solution. Then we present proof that improved version preserves completnesess property. It is achieved by imitating all phases of the algorithm.

**4.1. Algorithm description.** The algorithm comprises three phases as in [**?**]. 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 $H_{p,t}^{str-sub}$:

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

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 $|p| * k..\frac{|p|}{k}$ 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 $W := wxw$ 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. For our case leftmost means that for each prefix algorithm will detect longest suffix (remember that $M$ is transposed $H_{p,t}^{str-sub}$ ).

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

The third phase is the same as in [**?**] (Lines 8-12).

*This manuscript is for review purposes only.*

**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 $t$

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

$$L_w = \frac{|p|}{k} \tag{4.3}$$

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

Pseudocode:

1: $W = semilocalsa(p, t)$ {1st phase}
2: $H^{str-sub}_{p,t} = semilocalsa(p, t).stringSubstringMatrix$
3: $M[j, i] = -H^{str-sub}_{p,t}[i, j]$
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** $W_3$

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

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

*First phase.* . We will store solution $H$ of *semi-local sa* by decomposing it to permutation matrix $P$ of size $O(v*|t| \times v*|t|)$ (Lines 1-3) (ref add). The permutation matrix can be stored via two permuations of size $v * |t|$ for column and rows. It is simply two lists of size $v*|t|$. Then, to random access query in specific position $i, j$ of matrix $H$ we need to check how many points dominated by $i, j$. It is just pass through 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 completnesses property of algorithm [?].*

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 $\tau_i$ 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:

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

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, algorthm make recursive call to subroutine *greedy*. The *greedy* routine perfoms greedy choice of $\tau_i$ 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 $i$th position and ending at $j$th 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:

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

$interval = rmq2D.query(i, j, i, j)$<br>
$result = \emptyset$<br>
**if** $interval.score < h$ **then**<br>
    **return** $result$<br>
**end if**<br>
**if** $interval.i - i \geq 1$ **then**<br>
    $cl = greedy(rmq2D, h, t_{i,interval.i})$<br>
    $result.add(cl)$<br>
**end if**<br>
**if** $j - interval.j \geq 1$ **then**<br>
    $cl = greedy(rmq2D, h, t_{j,interval.j})$<br>
    $result.add(cl)$<br>
**end if**<br>
**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:

$GreedyMathing(M, h, t)$

$rmq2D = buildRMQStructure(M)$<br>
$result = greedy(rmq2D, 0, |t|, t)$<br>
**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):

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

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 $t$, 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 ∈ 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* $semi-localsa$ *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 intialization of interval tree is perfomed 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 perfomed 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 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 × n partial Monge matrix M, a data structure of size O(n) can be constructed in O(n logn) time to answer submatrix maximum queries in O(log logn) time.*

*Proof it*

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

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 [4]. 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.

[4] 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[**?**]

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