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 pattern 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 it asks to find all occurrences of pattern p in text t with some degree of similarity.

There exists a lot of algorithms that solve the above problem. Nonetheless, the number of suitable 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 [1]. The core of this approach is an algorithm that detects approximate clones of a given pattern p with a specified degree of similarity and length boundaries for detected clones. The main advantage of the algorithm is that it meets a specific requirement of completeness. Nonetheless, it has an unpleasant time complexity.

The common approach of 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 TODO: 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

*Saint Petersburg State University, Russia (mishinnikitam@gmail.com).
†JetBrains Research, Saint Petersburg, Russia (daniil.berezun@jetbrains.com).

based mainly on it and the underlying algebraic structure. As well as developing new algorithms we improve the existing algorithm for duplicate detection in software documentationfrom [1]. It should be noted that improvement preserves all properties of this algorithm. All presented algorithms also supports length constraints on the resulting substrings. Finally, we provided and proved running time and space complexity for all presented algorithms.

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, *String-searching problem* 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*. During algorithms design, we primarily focused on the usage of the latter two.

In some approximate pattern matching applications, it may be crucial to impose a constraint on length of the matched substrings. In bioinformatics, for example, substrings with small size may be less important in a biological sense then substrings with less similarity score but much larger length [**?**]. Same is also true for the length upper bound. Besides one may use length constraints in clone detection in software code and documentation tools in order to decrease false-positive rate [**?**]. Recently there has been developed an algorithm for solving this particular *AMatch* extension [**?**]. Despite the algorithm has poor 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. Global lcs and sa.

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

Scoring scheme determines how to calculate the alignment score of two aligned sequences. If a pair of character in aligned sequences are matched (equal) then this pair contributes to the final alignment 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 another sequence it means that α is aligned with a *gap*. In this case, this pair contributes w_0. The final scoring scheme calculates as follows:

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

where k^+ stands for the number of matched symbols, k^- — the number of mismatched symbols. Note that LCS is a special case of SA with scoring scheme be $(1,0,0)$. Both described problems can be solved with usual dynamic programming algorithm having running time complexity $O(|a|*|b|)$.

2.3. Semi-local lcs. LCS and SA allow one to find how much whole given strings are similar, i.e. how similar two string in a global sense. Sometimes this is not enough. There exist *fully local* and *semi-local* generalizations of these problems. In the paper, we focus on *semi-local* ones due to natural applicability to approximate pattern matching. The key feature of semi-local problems is that they can be solved within the same time as global ones while collects more information about strings similarity. More precisely, given two strings a and b the semi-local lcs asks about *lcs* scores for following:

- *string-substring*: whole a against every substring of b,
- *substring-string*: whole b against every substring of a,
- *prefix-suffix*: every prefix of a against every suffix of b,
- *suffix-prefix*: every prefix of b against every suffix of a.

Formally, semi-local lcs can be defined via *semi-local lcs matrix* as follows.

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

$$H_{a,b}[i,j] = \textbf{\textit{if}}\ j \le i\ \textbf{\textit{then}}\ j-i\ \textbf{\textit{else}}\ lcs(a, b^{pad}[i,j]) \tag{2.2}$$

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

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

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

Semi-local lcs can be solved within the optimal running time complexity, $O(|a|*|b|)$, by a number of algorithms based, for example, on fast distance multiplication of unit-Monge matrices [**?**] or seaweed combing [**?**].

2.4. Matrix properties.

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

$$\forall i,i',j,j' : \ i \le i', j \le j' \ . \ H[i,j] + H[i',j'](\ge) \le H[i,j'] + H[i',j]$$

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 permutation matrix, i.e into a *cross-difference* matrix:

Theorem 2.8. *[] Let $A[0:m, 0:n]$ be a Monge Matrix. Let $D^{\square}$, $b = A[m::]$ and $c = A[::, 0]$ then following is true $\forall i \in [0:m], j \in [0:n]$:*

$$A[i,j] = D^{\nearrow}[i,j] + b[j] + c[i] - b[0]$$

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 are several ways to store matrix $H_{a,b}$ or one of its quadrants implicitly. A simple storing of two lists of permutations requires $O(|a| + |b|)$ space and time providing $O(|a| + |b|)$ orthogonal range query time complexity (since it is necessary 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. Moreover, while arbitrary query has a non-constant time complexity, the following proposition stands that adjacent matrix elements can be queried within a constant time [**?**].

Proposition 2.9. *Given a permutation matrix P and a value $P^{\nearrow}[i;j]$, values $P^{\nearrow}[i+-1;j], P^{\nearrow}[i;j+-1]$, where they exist, can be queried in time $O(1)$.*

We particularly interested in the left lower quadrant that refers to string-substring problem:

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

2.5. Semi-local sa. The semi-local sequence alignment is a generalization of semi-local lcs in the same manner as sequence alignment is a generalization of lcs. Given two strings a and b and scoring scheme $w = (w_+, w_0, w_-)$[1] semi-local sa asks about sa scores for the following:

- *string-substring*: whole a against every substring of b
- *substring-string*: whole b against every substring of a

[1]Normally assumed that $w_+ \geq 0$, $w_0 < w_+$ and $w_- \leq \frac{w_0}{2}$

- *prefix-suffix*: every prefix of a against every suffix of b
- *suffix-prefix*: every prefix of b against every suffix of a

The *associated matrix* for semi-local sa is defined by analogy as for semi-local lcs.

Semi-local sa problem can be solved by reducing to semi-local lcs. First, note that the scoring scheme from 2.1 may be simplified by so-called normalization [?]:[2]

$$\begin{aligned} w &= (w_+, w_0, w_-) = (w_+ + 2x, w_0 + 2x, w_- + x), \forall x \\ &= (\tfrac{w_+ + 2x}{w_+ + 2x}, \tfrac{w_0 + 2x}{w_+ + 2x}, \tfrac{w_- + x}{w_+ + 2x}) \overset{x=-w_-}{\rightarrow} (1, \tfrac{\mu}{v}, 0) \end{aligned} \tag{2.7}$$

The resulted scoring scheme $w_{normalized} = (1, \frac{\mu}{v}, 0)$ is called the *normalized scoring scheme*. Then to query initial score sa for scoring scheme w knowing $sa_{normalized}$ for $w_{normalized}$ one need to apply reverse regularization:

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

Finally, the blown-up technique[3] is applied after reducing the scoring scheme. At this stage, we have transitioned to semi-local lcs problem which can be solved in $O(|a| * |b| * v^2)$ time by algorithm from [?]. Finally, unit-Monge matrix multiplication algorithm reduces time complexity in v timesTODO: [?].

2.6. Range maximum/minimum queries. Range maximum (minimum) queries (*rmq*, submatrix query) refer to a search of the maximum (minimum) element in submatrix $[i_1 : i_2] \times [j_1 : j_2]$ of a given matrix M of size $n \times n$. The associated data structure that can report maximum (minimum) element in any submatrix query is called *range maximum (minimum) data structure*. In general, the optimal space complexity of this data structure is $O(n^2)$ since storing the matrix requires at least $O(n^2)$ space. Nonetheless, it is not the case if considering Monge matrices which can be stored implicitly. *rmq* on Monge matrices has been a source of research over the past decades []. The recent research provides the following results [?].

THEOREM 2.10. *[?] 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 $O(1)$.*

LEMMA 2.11. *The blank entries of $m \times n$ partial[4] Monge matrix M can be imlicitily replaced in time $O(m + n)$ that M becomes Monge and $\forall\ i \in [0 : m], j \in [0 : n]$ $M[i, j]$ can be queried in $O(1)$*

THEOREM 2.12. *[?] Given an $n \times n$ partial Monge matrix[5] 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 $O(1)$.*

If the random access to Monge matrix is non-constant, say $O(\beta)$, then the results above are satisfied with time construction and query times increased by factor β.

2.7. Near-duplicate detection algorithm. First, we denote several parameters are used in algorithm [?]. Let k, a set similarity measure, be a constant in interval $[\frac{1}{\sqrt{3}}, 1]$. A window w of size $L_w = |p|/k$ is to process text t with sliding window of

[2] Assumed that $w_+ - 2w_- > 0$

[3] Each of the input length increases by factor v.

[4] Partial matrix is the matrix where some entries undefined, but defined entries in each coloumn and row are contigous

[5] A partial totally Monge matrix is a partial matrix whose defined entries satisfy the Monge property.

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:
 1: $W_1 = \emptyset$ {1st phase}
 2: for $\forall w_1 : w_1 \in t \wedge |w_1| = L_w$ do
 3:     if $d_{di} \leq k_{di}$ then
 4:         add $w_1$ to $W_1$
 5:     end if
 6: end for
 7: $W_2 = \emptyset$ {2d phase}
 8: for $w \in W_1$ do
 9:     $w_2^{'} = w$
10:     for $l \in I$ do
11:         for $\forall w_2 : w_2 \subseteq w \wedge |w_2| = l$ do
12:             if $Compare(w_2, w_2^{'}, p)$ then
13:                 $w_2^{'} = w_2$
14:             end if
15:         end for
16:     end for
17:     add $w_2^{'}$ to $W_2$
18: end for
19: $W_3 = UNIQUE(W_2)$
    {3rd phase }
20: for $w \in W_3$ do
21:     if $\exists w^{'} \in W_3 : w \subset w^{'}$ then
22:         $remove\ w\ from\ W_3$
23:     end if
24: end for
25: return $W_3$
```

one symbol step. Let $k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2)$ be a threshold value for edit distance, I — interval of size $[|p|k, \frac{|p|}{k}]$ that sets boundaries for lenght of matching substrings, d_{di} — function measures similarity between given strings.

The algorithm (see Algorithm 2.1) 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 are compared using edit distance[6] and if $d_{di}(p, t_w) \leq k_{di}$, i.e. close enough, then they are added to set W_1 to be further proceeded. On the second phase each of the detected substrings in W_1 shrunk, i.e their length can be decreased. More precisely, each element $w \in W_1$ is iterated over to find its longest substring such that its length falls in interval I and it is the most similar to pattern p. Set W_2 stands for the result of this phase. At the final third phase, set W_2 is filtered by removing elements that fully contains or duplicates any other element of set W_2.

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

Running time analysis. At first phase at most $O(|t| - \frac{|p|}{k}) = O(|t|)$ windows of size L_w will be processed. Since edit distance running time complexity is $O(|p|^2)$, the overall time complexity of the first phase is $O(|t| * |p|^2)$. Note, cardinality of set W_1 could be meaused as $O(|t|)$ at worst case. Thus, the the first loop of the second phase will be iterated over $O(|t|)$ times (line 8). The second loop (line 10) refers to iteration over set I with cardinality $O(\frac{|p|}{k} - |p|k) = O(|p|)$. The third loop (line 11) requires at most $O(|p|)$ iterations since again a sliding window is used. The running time complexity of *Compare* operation (line 12) is at most $O(|p|^2)$ since both strings are of size $O(|p|)$. Thus, the total running time complexity of the second phase can be estimated as $O(|p|^4|t|)$ at worst case. Note, cardinality of set W_2 is equal to W_1 and thus at worst case is $O(|t|)$. Finally, the third phase requires $O(|t| * \log |t|)$ time. Thus, the total time complexity of the algirthm can be estimated as $O(max(|t| * |p|^4, |t| * \log |t|))$.

The algorithm also possesses completeness property. To detailed description refer to [**?**]. We just state the following theorem:

THEOREM 2.13. *[**?**] 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 with respect to the output of phase 2.*

3. Algorithm for near duplicate detection. In the section the improved version of Luciv et.al. algorithm [1] is presented. The utilization of semi-local sequence alignment algorithms in algorithm phases improves overall time complexity. It is also proved that the improved algorithm preserves all advantages of the initial one stated at [1] such as search completeness.

3.1. Algorithm description. As the base one (see section 2.7 and **??**), the presented algorithm consists of three sequential phases. Aldorithm pseudocode is presented at Algorithm 3.1.

At the first phase (lines 1-3) semi-local sa problem is solved for 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. Then, we construct matrix M by applying transposition and inverse operation implicitly on $H_{p,t}^{str-sub}$:

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

Note, transposition operation preserves (anti) Monge property whereas inverse operation transforms anti-Monge matrix into the Monge one and vice versa. Thus, M is a Monge matrix.

The second phase comprises several steps (lines 4–6). First, we want to get for each substring with length exactly L_w in t its alignment score against pattern p where $k \in (\frac{1}{\sqrt{3}}, 1]$. Second, for each substring w of size L_w in text t that meets criteria on similarity with given threshold $-k_{di}$ the following should be done. The longest substring most similar to pattern p should be taken. it could be done by processing the diagonal matrix M with width $c := \frac{|p|}{k} - |p| * k = |p|(\frac{1}{k} - k)$ (Fig. 1).

The third phase (lines 7–11) is the same as in the base algorithm.

THEOREM 3.1. *Algorithm 3.1 runs in $max(O(|t| * |p|), O(|t| * \log |t|))$ time with $O(|t| \log |t|)$ additional space where p is pattern, t is text, $|p| \leq |t|$, and $v = O(1)$ where v is denominator of normalized mismatch score for semi-local sa $w_{normalized} = (1, \frac{\mu}{v}, 0)$.*

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

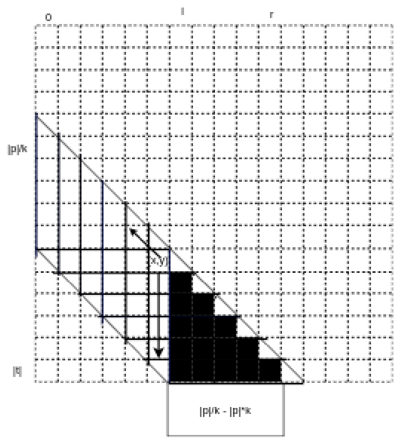

FIG. 1. *LOL*

Algorithm 3.1 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

$$k_{di} = |p| * (\frac{1}{k} + 1)(1 - k^2) \tag{3.1}$$

$$L_w = \frac{|p|}{k} \tag{3.2}$$

$$w = |p|(\frac{1}{k} - k) \tag{3.3}$$

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: $W_3 = UNIQUE(W_2)$ {3rd phase unchanged}
7: **for** $w \in W_3$ **do**
8:    **if** $\exists w' \in W_3 : w \subset w'$ **then**
9:       $remove\ w\ from\ W_3$
10:   **end if**
11: **end for**
12: **return** $W_3$
```

First phase. We 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, Theorem 2.8). The permutation matrix can be stored via two permutations of size $v * |t|$ for columns and rows. It is simply two lists of size $v * |t|$. Then, for random access query in specific position (i, j)

of matrix H one need to check how many points are dominated by $H[i,j]$. It can be done by checking all points of permutation matrix and requires $O(v*|t|)$ steps. Thus, the total time and space complexity of the first phase are $O(v*|p|*|t|)$ (time needed to solve semi-local sa) and $O(v*|t|)$ respectively. Given $v=O(1)$ we have $O(|p|*|t|)$ and $O(|t|)$ respectively.

Second phase. For the sake of clarity, we omit k factor in algorithm analysis since k is just a constants within interval $(\frac{1}{\sqrt{3}},1]$.

First, we query elements that lie in the diagonal that represent substrings of size $L_w=\frac{|p|}{k}$ (to step onto $(0,|p|/k)$ cell we need to perform one orthogonal range query). We denote them by $lstW$. Since we can use proposition 2.9 to access adjacent elements the total complexity of this step is $O(|t|)$. The total amount of querying cells is $O(|t|)$. Second, we again process matrix M. More precisely, we process the diagonal of width $O(\frac{|p|}{k}-|p|*k)=O(|p|)$ that corresponds to all substrings with size in $I=[|p|*k,\frac{|p|}{k}]$ interval (on Fig. 1 it is trapezoid). Note, that the bold triangle in Fig 1 corresponds to the set of substrings with sizes in the interval I that contains in the last substring with size L_w of text t. During processing each triangle we additionally store list $longestForRow$ of size $\frac{|p|}{k}-|p|k$ that contains for each row (represent prefix) the suffix that is most similar to pattern p for given window (triangle on Fig. 1). This list updated every time when we shift triangle by one symbol step $((x,y)\to(x-1,y-1))$. Thus, to get the longest substring that most similar to pattern p for the given window we simply need to check $longestForRow$. Thereby, we proceed as follows. We check that the current window meets criteria on similarity (it is just lookup to list $lstW$) If so, then we find maximal substring among all associated with the current window (triangle) by checking $O(|p|)$ elements of $longestForRow$ and save it to set W_2 as described above.

Thus, the processing of each column of trapezia requires at most $O(|p|)$ time. The total amount of columns is $(|t|)$ Total time complexity of this phase is $O(|t||p|)$. The space complexity of this phase is $O(|p|+|t|)=O(|t|)$ at most.

Third phase. The third phase remains unchanged, thus have the same time and space bounds as in the base algorithm case. Note, it possible to perform this phase in-place during the second phase which speedups the algorithm, i.e decreases space and time complexity to $O(|t|)$ and $O(|t|*|p|)$ respectively. The third phase can be approximated as $O(|t|*log|t|)$ for both space and running time complexity.

Thus, the total alforithm running time is $max(O(t*p),\ O(t*\log t))$ while space complexity is $O(t*\log t)$. ∎

THEOREM 3.2. *Algorithm 3.1 with scoring scheme $w=(0,-2,-1)$ preserves completnesses property of algorithm [1] and has running time and space complexity $max(O(t*p),\ O(t*\log t))$ and $O(t*\log t)$ respectively.*

Proof. Edit distance in the base algorithm [?] may be expressed as sequence alignment with following scoring scheme:

$$w_{sa}=(w_+,w_0,w_-)=(0,-2,-1).$$

First, to get intial edit score we need to apply inverse operation:

$$editscore(a,b)=-sa(a,b,w_{sa}).$$

Next, w_{sa} may be normalized using normalization 2.7:

$$(0,-2,-1)\to(1,\frac{\mu=0}{v=1},0).$$

Thus, $d_{di} \leq k_{di}$ is the same as $sa \geq -k_{di}$.

Second, let's carefull review phases 1 and 2 of given algorithms. The base algorithm passes through the text with a sliding window to detect those fragments of size L_w which have edit score above given threshold k_{di}. Then within these fragments algorithm detects longest suffixes that are most similar to pattern p with size within interval $I = [pk,\ L_w]$. The presented improved algorithm proceeds in a very similar way but, informally, phases are swapped. First, it detects the longest suffixes with size in interval I for each prefix of the text. Then, it proceeds in such way that for each window of size L_w that has alignment score with pattern p below given threshold $-k_{di}$ the longest suffix most similar to p is detected. Due to TODO: formula **??** results of the second phases of the algorithms are equal. The third phase remains unchaned in the presented algorithm. Thus, the presented algorithm is complete. For $w = (0, -2, -1)$, $v = 1$ and algorithm running time and space complexity are as claimed. □

.

4. New approximate pattern mathing algorithms. In the section we present two algorithms for approximate pattern mathing that heavily based on semi-local sequence alignment and its underlying algebraic structure. The presented algorithms also support length constraints on the result (on detected clones).

4.1. First algorithm. The first algorithm 4.2 implements an idea of the greedy choice of the most suitable clone on each step, i.e. it looks for the most similar substring with length in interval $[l,\ r]$ each time for some predefined constants l and r. In other words, it constructs a non-intersected set $\mathcal{T}$ of clones of pattern p in text t such that on each step it adds τ_k to $\mathcal{T}$ iff τ_k has the highest similarity score with p in the rest of the text and its length satisies the length constraint, i.e. $l \leq |\tau_k| \leq r$.

The algorithm proceeds as follows.

First, the semi-local sa problem is solved for given pattern p and text t (line 1). Then, the solution for the string-substring subproblem is queried from it (lines 2–3). Further, the diagonal slice of width $r - l$ which corresponds to scores of substrings of size in $[l,\ r]$ is cropped to get a partial Monge matrix (line 4) (see Fig. 2). Next, using Theorem 2.12 a *rmq2D* data structure is constructed (lines 5–6) which make it possible to perform range minimum queries.

Then, algorithm calls recursive subroutine *greedy* (algorithm 4.1). The *greedy* routine performs greedy choice of τ_k with maximal alignment within a continious uncovered part of the text $t_{i,j}$ with length boundaries for τ_k in $[l,\ r]$ where $t_{i,j}$ denotes a substing of text t starting at i-th element and ending at j-th element. More precisely, it refers to searching maximum value with corresponding position (row and column) in matrix M within $t_{i,j}$. The search is performed 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 τ_k is added to the final result and the current part of the text splits into two smaller parts and processed recursively in the same way.

Finally, the algorithm outputs a set of the non-intersected intervals of clones of pattern p in text t.

THEOREM 4.1. *Algorithm 4.2 runs in $O(max(|t||p|, \frac{|t|\log^2|t|}{\log\log|t|}))$ time and $O(|t|)$ space when $|p| < |t|$ where p is pattern, t is text, and $v = O(1)$ is denominator of normalized mismatch score for semi-local sequence alignment $w_{normalized} = (1, \frac{\mu}{v}, 0)$.*

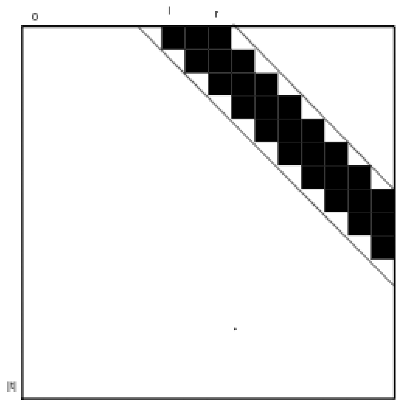

FIG. 2. *TODO: .*

Algorithm 4.1 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 $t_{i,j}$, l, r— length boundaries for detected intervals
Output: Set of non-intersected intervals from $t_{i,j}$
Pseudocode:
$greedy(rmq2D, h, i, j, t_{i,j}, l, r)$:

```
 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 $interval.i - i \geq l$ then
 7:     $cl = greedy(rmq2D, h, i, interval.i, t_{i,interval.i}, l, r)$
 8:     $result.add(cl)$
 9: end if
10: if $j - interval.j \geq l$ then
11:     $cl = greedy(rmq2D, h, j, interval.j, t_{j,interval.j}, l, r)$
12:     $result.add(cl)$
13: end if
14: return $result$
```

Proof. Note that $|p| < |t|$ and $v = O(1)$. For simplicity let $v = 1$ (same true for other $v = O(1)$).

First phase. Since $v = 1$ we need to solve semi-local lcs problem. It could be solved implicitly via algorithm from [?] in $O(|t| * |p|)$ with $O(|t|)$ additional space when $|p| < |t|$. Note, we are only interested in string-substring submatrix $H_{p,t}^{str-sub}$ of size $|t| \times |t|$. By Theorem 2.8 we build data structure of size $O(|t|)$ in $O(|t|\sqrt{\log(|t|)})$ time from associated permutation matrix with $H_{p,t}^{str-sub}$ anti-Monge matrix. The data structure performs orthogonal range queries in $O(\frac{\log(|t|)}{\log\log(|t|)})$ time. Thus, the overall time and space complexity of first phase is $max(O(|t| * \sqrt{\log(|t|)}),\ O(|p| * |t|))$ and

Algorithm 4.2 GREEDY-PATTERN BASED NEAR DUPLICATE SEARCH ALGORITHM

Input: pattern p and text t, threshold value h
Output: Set of non-intersected clones of pattern p in text t
Pseudocode:
$GreedyMathing(M, h, t)$

```
1: $sa = semilocalsa(a, b)$ {1st phase}
2: $H = sa.getAssociatedMatrix()$
3: $H^{str-sub} = H.stringSubstringMatrix()$
4: $M_{partial} = -getPartialMatrix(H^{str-sub}, l, r)$ {2nd phase}
5: $M = builtMongeMatrix(M_{partial})$
6: $rmq2D = buildRMQStructure(M)$
7: $result = greedy(rmq2D, h, 0, |t|, t, l, r)$ {3rd phase}
8: return $result$
```

$O(|t|)$ respectively.

Second phase. In string-substring matrix $H_{p,t}^{str-sub}$ we are only interested in diagonal of length $r - l$ that refers to all substrings of text t with length in $[l : r]$ interval. Hence, we inverce matrix $H_{p,t}^{str-sub}$ and pick out the diagonal of interest, resulting in partial Monge matrix $M_{partial}$. Then we apply theorem 2.12 to build *rmq2D* data structure to perform minimum range queries. Note, access to the element in $M_{partial}$ can be performed in non-constant and it returns not the only element itself but also the associated indices. Hence, the data structure of size $O(|t|)$ can be built in $O(|t| \log |t|) * O(\frac{\log(|t|)}{\log\log(|t|)}) = O(\frac{|t| \log^2 |t|}{\log\log(|t|)})$ time to perform range minimum queries in $O(\log\log(|t|)) * O(\frac{\log(|t|)}{\log\log(|t|)}) = O(\log |t|)$ time. Thus, overall time and space complexity of second phase are $O(|t| \log^2 |t|)$ and $O(|t|)$ respectively.

Third phase. To analyze the third phase we need to look at the recursive algorithm 4.1. Note, at the worst case we will have $O(|t|)$ nodes while proceeded recursion since at the worst case on each node we will detect interval of size 1. There will be at most t such non-intersecting intervals. Thus, the total amount of calls to *query* operation will be at most $O(|t|)$. The query operation requires time $O(\log |t|)$ as shown in the previous phase. Hence, the total running time and space complexity of third phase are $O(|t|| \log t|)$ and $O(|t|)$

Thus, overall algorithm running time and space complexity are as claimed when $v = O(1)$ and $|p| < |t|$. ∎

4.2. Second algorithm. The second algorithm 4.3 uses a less sophisticated approach and finds fewer duplicates of pattern p. It is similar to the previous one but uses simplier approach instead of the *greedy* subrutine.

Algorithm description. The first phase as in algorithm 4.2. On the second phase we use Lemma 2.11 to implicitly fill elements of partial Monge matrix to get the Monge matrix M. Then we solve the following problem (Line 6). For each prefix of text t we find the suffix that has the highest similarity score with pattern p:

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

Further, we remove suffixes whose similarity score is below the given threshold h (line 4). Then remaining suffixes are sorted in descending order (line 5) and an interval tree [?] is built upon them (lines 9–14). The building process consists of checking that

Algorithm 4.3 Greedy approximate

Input: pattern p, text t, threshold value h
Output: Set of non-intersected clones of pattern p in text t
Pseudocode:

```
1:  $sa = semilocalsa(p, t)$ {1st phase}
2:  $H = sa.getAssociatedMatrix()$
3:  $H^{str-sub} = H.stringSubstringMatrix()$
4:  $M_{partial} = -getPartialMatrix(H^{str-sub}, l, r)$
5:  $M = fillToMongeMatrix(M_{partial})$ {2nd phase}
6:  $colmax = smawk(M)$
7:  $colmax.filter(it.score >= h)$
8:  $colmax.sortByDescending(it.score)$
9:  $tree = buildIntervalTree()$
10: for $candidate \in colmax$ do
11:   if $candidate \cap tree = \emptyset$ then
12:     $tree.add(candidate)$
13:   end if
14: end for
15: $result = tree.toList()$
16: return $result$
```

current substring *candidate* not intersected with already added substrings to the *tree* and adding it in case of success. Finally, algorithm outputs a set of non-intersected substrings (clones) of pattern p in text t.

THEOREM 4.2. *Algorithm 4.3 runs in time* $\max(O(|p| * |t|),\ O(|t| * log|t|))$ *with* $O(|t| * log|t|)$ *space when* $|p| < |t|$ *where* p *is pattern,* t *is text, and* $v = O(1)$ *is denominator of normalized mismatch score for semi-local sequence alignment* $w_{normalized} = (1, \frac{\mu}{v}, 0)$.

Proof. First phase. The total running time and space complexity of first phase are $max(O(|t||p|, |t| \log |t|)$ and $O(|t|)$ as in Algorithm 4.2.

At the *second phase* we use Lemma 2.11 with a matrix element access cost $O(\frac{\log |t|}{\log \log |t|})$. Then, running time complexity of implicitly filling blank entries in $M_{partial}$ to get Monge matrix M is $O(\frac{|t| \log |t|}{\log \log |t|})$. Next, we use the *SMAWK* algorithm to find for each prefix a suffix that is most similar to pattern p with length in $[l : r]$ (line 6). Then, we filter out resulting suffixes by given threshold h and sort the remaining suffices in descending order (lines 7–8). There will be at most $O(r - l) = O(|t| - 0) = O(|t|)$ suffices. Thus, sorting operation complexity is $O(|t| * \log |t|)$. The result of the phase is an array denoted as *colmax*.

Third phase. colmax has as worst-case $O(|t|)$ elements when filtering doesn't eliminate any substring. Thus, there are at most $O(|t|)$ additions and intersections to the interval tree (lines 11–12). Since both operations has time complexity $O(\log |t|)$ the overall running time complexity is $O(|t| * \log |t|)$.

Thereby, the total time and space complexity of algorithm are $\max(O(|p| * |t|), O(|t| * \log |t|))$ and $O(|t|)$ respectively. ∎

COROLLARY 4.3. *Algorithm 4.3 runs in time* $O(|p| * |t|)$.

When the amount of clones is relatively small and the threshold value is high, then after filtering out t intervals (line 4) sorting is performed on a small set of elements.

Thus, this part is dominated by calculating the semi-local sa solution.

5. Conclusion. In the paper, several new algorithms solving the approximate pattern matching problem are presented. The presented algorithms also support length constraints on the resulting substrings. Although presented algorithms have a good running time and space complexity, there is unclear how would they behave in practice. Algorithms are heavily based on *semi-local sa*, thus there are also need to provide an empirical evaluation of algorithms that solve *semi-local sa*. Also, the research question arises about hidden constants of used data structures and algorithms itself.

REFERENCES

[1] 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.