APPLICATION OF SEMI-LOCAL LCS TO STRING APPROXIMATE MATCHING*

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

Abstract. We present an application of semi-local lcs to approximate string matching by developing a new algorithm and improving the existing one. Our result is based on the utilization of the underlying algebraic structure of semi-local lcs with the usage of the novel data structure for submatrix maximum queries in Monge matrices. This gives two algorithms with the following running time and space complexity. TODO. The improvement of the existing algorithm not only preserves all properties but also outperforms in practice.

In addition, we show that the algorithm for semi-local lcs based on sticky braid multiplication is not perform well with the current complex recursive structure.

Key words. semi-local lcs, monge matrix, range queries, approximate matching, near-duplicate detection

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

Approximate string matching is an important task in many fields such as computational biology, signal processing, text retrieval and etc. It also refers to a duplicate detection subtask.

In general form it formulates as follows: Given some pattern p and text t need to find all occurrences of pattern p in text t with some degree of similarity.

There are many algorithms that solve the above problem. Nonetheless, the number of algorithms sharply decreases when the algorithm needs to meet some specific requirements imposed by running time, space complexity or specific criterion for the algorithm itself. For example, recently there was developed an approach for interactive duplicate detection for software documentation [2]. The core of this approach is an algorithm that detects approximate clones of a given user pattern with a specified degree of similarity. The main advantage of the algorithm is that it meets a specific requirement of completeness. Nonetheless, it has an unpleasant time complexity.

The algorithm for approximate detection utilizes mainly algorithm for solving the longest commons subsequence (LCS) problem. The longest common subsequence is a well-known fundamental problem in computer science that also has many applications of its own. The major drawback of it that it shows only the global similarity for given input strings. For many tasks, it's simply not enough. The approximate matching is an example of it.

There exist 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-

*Submitted to the editors DATE.

Funding: This work was funded by the Fog Research Institute under contract no. FRI-454.

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

‡Department of Applied Mathematics, Fictional University, Boise, ID (ptfrank@fictional.edu, jesmith@fictional.edu).

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 matching.

Describe approximate matching formally

2.2. Semi-local lcs.

Describe semi-local lcs (definition), algorithms that solves (steady and and braid reducing)

2.3. Monge matrix.

Describe monge property

Say about range queries (about soda12, soda14 and new result that we will be used)

2.4. Near-duplicate detection algorithm.

Describe luciv algo

3. Related work.

?????

could mention about approximation. Need discuss

4. Algorithm for near duplicate detection.

We now describe an improved version of Luciv et.al. algorithm [2] by utilizing a *semi-local sa* solution. Then we present proof that improved version preserves completnesess property. It is achieved by imitating all phases of the algorithm.

4.1. Algorithm description.

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

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

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

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

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

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

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

Input: pattern p, text t, 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* $max(O(tp), O(t \log t))$ *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. .

For storing *semi-local lcs* solution, specifically decomposed kernel P we will use simply two list (two permuations) of size at most $v * |t|$ each because kernel P has at most $v|t|$ non zeros (after we use $blow - up$ tecnique). Due to fact that $v = O(1)$ $O(v * |t|)$ becomes $O(|t|)$. Note that for such simple data structure to make random access to P we need to calculate amount of poinst that dominated by given point. It require to check at most $O(|t|)$ points. Thus, random access query is $O(|t|)$.

The solution of $semi - localsa$ when $v = O(1)$ is just $O(|t| * p)$.

The total bounds for time and space complexity for this phase is at most $O(|t| * p)$

and $O(|t|)$ correspondingly.

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

We will use first approach described in algorithm description for this phase. First, although the random access query to matrix element require $O(t)$. We only need one such query to step on the diagonal. 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, the visit of each cell in diagonal of size at most $O(t) \times O(p)$ require at most 1 random access and $O(t * p - 1) = O(t * p)$ adjacent accesses. When we pass through slice of specific column we also will find the longest suffix with highest similarity. It requires at most store $O(p)$ cells for each column but we only process one column at the time, thus, we store only additional $O(p)$ for that whole cell processing. Also we store $O(t)$ cells that corresponds to suffixes of length $\frac{p}{k}$ At the end of $processDiagonal$ we will have t suffixes that requires $O(t)$ space for storing them. Then, $processDiagonal$ requires $O(t+tp) = O(t)$ time for processing diagonal with $O(t+p) = O(t)$ additional space. Further, 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 the suffix with length $\frac{p}{k}$ have similarity score at least $-k_{di}$. It is simply one way pass through list of suffixes where processing of each window requires at most $O(p+1) = O(1)$. The total number of such windows at most $O(t)$. Thus, $SuffixMaxForEachWindow$ requires $O(t) * O(p) = O(tp)$ time with at $O(t)$ space for storing suffix for each window.

The filtering process is one way pass throgugh list of suffixes W_2. It requires at most $O(t)$ time.

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

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

Thus, the total runing 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].*

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 $A_2 = A_1$.

At first algorithm **??** pass through text t with sliding window to detect those fragmetns 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 in $pk ... \frac{p}{k}$ interval.

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 filtering is perfomed in that way that only for those windows of size $\frac{p}{k}$ the longest suffix is left.

Thus $A_1 = A_2$

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:

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

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 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:

$GreedyMathing(M, h, t)$

```
rmq2D = buildRMQStructure(M)
result = greedy(rmq2D, 0, |t|, t)
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

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

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

6. Evaluation.

Semi-local algorithms. Show perfomance between lcs and semi-local lcs??? and poor perfomance of recursive algorithm based on steady ant?

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

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

Show that sparse table bad when large?

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]

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

REFERENCES

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 4th ed., 2013.

[2] D. V. Luciv, D. V. Koznov, A. Shelikhovskii, K. Y. Romanovsky, G. A. Chernishev, A. N. Terekhov, D. A. Grigoriev, A. N. Smirnova, D. Borovkov, and A. Vasenina, *Interactive near duplicate search in software documentation*, Programming and Computer Software, 45 (2019), pp. 346–355.