Space-Efficient Approximate String Matching Allowing Inversions in Fast Average Time*,**

Hwee Kim and Yo-Sub Han

Department of Computer Science, Yonsei University
50, Yonsei-Ro, Seodaemun-Gu, Seoul 120-749, Korea
{kimhwee,emmous}@cs.yonsei.ac.kr

Abstract. An inversion is one of the important operations in bio sequence analysis and the sequence alignment problem is well-studied for efficient bio sequence comparisons. We investigate the string matching problem allowing inversions: Given a pattern P and a text T, find all indices of matching substrings of T when non-overlapping inversions are allowed. We design an $O(nm)$ algorithm using $O(m)$ space, where n is the size of T and m is the size of P. The proposed algorithm improves the space complexity of the best-known algorithm, which runs in $O(nm)$ time with $O(m^2)$ space. We, furthermore, improve the algorithm and achieve $O(\max\{n, \min\{nm, nm^{\frac{5-t}{2}}\}\})$ average runtime for an alphabet of size t, which is faster than $O(nm)$ when $t \geq 4$.

1 Introduction

In modern biology, it is important to determine exact orders of DNA sequences, retrieve relevant information of DNA sequences and align these sequences [1, 7, 10, 11]. For a DNA sequence, a *chromosomal translocation* is to relocate a piece of the DNA sequence from one place to another and, thus, rearrange the sequence [8]. A *chromosomal inversion* is a reversal and complement of a DNA sequence. Chromosomal inversion occurs when a single chromosome undergoes breakage and rearrangement within itself [9]. For instance, consider a string $w = w_1 w_2 \cdots w_{i-1} w_i w_{i+1} \cdots w_k$.

$$w' = w_1 \cdots \overbrace{\underline{w_{i+2} w_{i+3}}\ \underline{w_{i-1} w_i w_{i+1}}}^{\text{translocation}} \cdots w_k, \qquad w'' = w_1 \cdots \overbrace{w_{i+2} w_{i+1} w w_{i-1}}^{\text{inversion}} \cdots w_k.$$

Here w' is a translocation of w and w'' is an inversion of w.

Based on these biological events, there is a well-defined string matching problem: given two strings P and T with two operations—translocation or inversion—the string matching problem is to find all matching substrings of T that match P under the operations.

* This research was supported by the Basic Science Research Program through NRF funded by MEST (2012R1A1A2044562).

** Kim was supported by NRF (National Research Foundation of Korea) Grant funded by the Korean Government (NRF-2013-Global Ph.D. Fellowship Program).

J. Chen, J.E. Hopcroft, and J. Wang (Eds.): FAW 2014, LNCS 8497, pp. 141–150, 2014.

Many researchers investigated efficient algorithms for this problem [1–6, 11, 12]. Note that inversions in strings are not automatically detected by the traditional alignment algorithms [12]. Schöniger and Waterman [11] introduced the alignment problem with non-overlapping inversions and showed a dynamic programming algorithm that computes local alignments with inversions between two strings of length n and m in $O(n^6)$, where $n \geq m$. Vellozo et al. [12] presented an improved $O(n^2m)$ algorithm. They built a matrix for one string and partially inverted the other string using table filling method in the extended edit graph. Recently, Cantone et al. [1] introduced an $O(nm)$ algorithm using $O(m^2)$ space for the string matching problem, which is to find all locations of a pattern P of length m with respect to a text T of length n based on non-overlapping inversions. Cho et al. [4] introduced an $O(n^3)$ algorithm using $O(n^2)$ space that solves the alignment problem allowing inversions to two strings of length n.

We consider the same problem that Cantone et al. [1] examined and tackle the problem by taking a different approach based on the relationship between substrings generated by inversions. We start from designing an algorithm for two strings of the same length. Based on the algorithm, we design an $O(m^2)$ algorithm using $O(m^2)$ space that checks the existence of the matching at a given index of T with respect to P. We, furthermore, modify the algorithm to process m indices simultaneously and present a new algorithm that finds all matching substrings of T in $O(nm)$ time using $O(m)$ space, which uses less amount of space compared with the best-known algorithm—$O(nm)$ time and $O(m^2)$ space [1]. Finally, based on the frequency of character appearances in a random string, we prove that our algorithm can achieve $O(\max\{n, \min\{nm, nm^{\frac{5-t}{2}}\}\})$ average runtime for an alphabet of size t by adding a permutation match filter suggested by Grabowski et al. [5]. For a DNA pattern, our algorithm runs in $O(n\sqrt{(}m))$, which outperforms the previous algorithm. Moreover, for $t \geq 5$, our algorithm runs in $O(n)$ time.

2 Preliminaries

Let $A[a_1][a_2]\cdots[a_n]$ be an n-dimensional array, where the size of each dimension is a_i for $1 \leq i \leq n$. Let $A[i_1][i_2]\cdots[i_n]$ be the element of A at an index $(i_1, i_2, \ldots, i_n)$. Given a finite set Σ of characters and a string w over Σ, let $|w|$ be the length of w and $w[i]$ be the symbol of w at position i, for $1 \leq i \leq |w|$. We use $w[i:j]$ to denote a substring $w[i]w[i+1]\cdots w[j]$, where $1 \leq i \leq j \leq |w|$.

Let the symbol θ denote inversion[1]. Then, $\theta(w)$ is the inversion of w. Given a range (i, j), we define the inversion operation $\theta_{(i,j)}(w)$ to be $\theta(w[i:j])$, the inversion of the substring $w[i:j]$. We say that $\theta_{(i,j)}$ *yields* the string $\theta_{(i,j)}(w)$ from w. The *size* of $\theta_{(i,j)}$ is $|\theta_{(i,j)}| = j - i + 1$. When the context is clear, we denote $\theta_{(i,j)}$ as (i, j).

[1] In biology, inversion is a composition of a reversal operation and a complement operation. However, inversion is often regarded as reversal in the string matching and alignment literature for the simplicity of analysis. We also follow this convention.

We define a sequence $\Theta = ((p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k))$ of inversions for a string w to be non-overlapping if it satisfies the following conditions: For $1 \leq i \leq k$, $p_1 = 1$, $q_k = |w|$, $p_i \leq q_i$ and $p_{i+1} > q_i$ for $1 \leq i \leq k-1$. Since $\theta(w) = w$ if $|w| = 1$, we can assume that all positions of w are within one of the ranges in a sequence of non-overlapping inversions; For example, given a string of size 6, two sequences of non-overlapping inversions $((1,2),(5,6))$ and $((1,2),(3,3),(4,4),(5,6))$ result in the same string. Therefore, we further assume that $p_{i+1} = q_i + 1$ for $1 \leq i \leq k-1$. When the context is clear, we call a sequence of non-overlapping inversions just a sequence of inversions.

Now, in summary, given a sequence $\Theta = ((p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k))$ of inversions and a string w,

$$\Theta(w) = \Theta[1](w)\Theta[2](w) \cdots \Theta[k](w).$$

Definition 1 (Problem). *Given a text T and a pattern P, we define the approximate string matching problem allowing inversions to be finding all indices i of T that has a sequence Θ_i of inversion satisfying $\Theta_i(P) = T[i : i+m-1]$.*

3 The Algorithm

We tackle the approximate string matching problem allowing inversions using $\theta(P)$—the inversion of P. We start with a special case when $|P| = |T|$ and extend the idea to the general case when $|P| \leq |T|$. For example, suppose that $P = AGTCTAG$ and $T = TGACATG$. A sequence $\Theta = ((1,3),(4,4),(5,6),(7,7))$ of inversions makes $\Theta(P) = T$. The sequence of $\Theta[i](P)$ can be represented on $\theta(P)$ in the reverse order; the sequence (TGA, C, AT, G) of strings yielded by Θ appears as the reversed sequence (G, AT, C, TGA) in $\theta(P)$.

Observation 1 (Relationship between $\Theta(P)$ and $\theta(P)$). *Given a sequence Θ of inversions and a pattern P, $\theta(P) = \Theta[k](P)\Theta[k-1](P) \cdots \Theta[1](P)$, where $k = |\Theta|$.*

Definition 2 (Successfully comparable substring and paired string). *Based on Observation 1, we say that $T[i : i+l-1]$ is a* successfully comparable substring *(SCS in short) for the given index i, given length l, strings T and $\theta(P)$ of the same length m if $\theta(P)_{(m+2-i-l,m+1-i)} = T[i : i+l-1]$. Moreover, we call the left-handed side of the equality a* paired string *of a successfully comparable substring (PSCS in short).*

Definition 3 (Sequence of successfully comparable substrings). *We define a sequence $\mathcal{S}_{(a,b)}$ to be a* sequence of successfully comparable substrings *for the given range (a, b), strings T and $\theta(P)$, where $\mathcal{S}_{(a,b)}[j] = T[i_j : i_j + l_j - 1]$ is an SCS for T and $\theta(P)$, $i_1 = a$, $i_{|\mathcal{S}_{(a,b)}|} + t_{|\mathcal{S}_{(a,b)}|} - 1 = b$ and $i_{j+1} = i_j + l_j$ for $1 \leq j < |\mathcal{S}_{(a,b)}|$.*

It is clear that if $\mathcal{S}_{(1,m)}$ exists for T and $\theta(P)$ such that $|T| = |P| = m$, then there exists Θ satisfying $\Theta(P) = T$. We design a simple algorithm that retrieves $\mathcal{S}_{(1,m)}$ from T and $\theta(P)$ as described in Algorithm 1. Fig. 1 is an example of the algorithm.

Algorithm 1.

Input: Strings T and $\theta(P)$ of length m
Output: Sequence $\mathcal{S}_{(1,m)}$ of SCSs

```
1 i ← 1, l ← 1
2 while i ≤ m do
3     if T[i : i+l−1] is an SCS then
4         add T[i : i+l−1] to S_(1,m)
5         if i+l−1 = m then return S_(1,m) i ← i + l, l ← 1
6     else l ← l + 1
7 return "no S_(1,m) exists"
```

Lemma 1. *It takes $O(m^2)$ time and $O(m)$ space to retrieve $\mathcal{S}_{(1,m)}$ from T and $\theta(P)$, where $|T| = |P| = m$.*

Note that the algorithm always compares substrings with increasing order of length and chooses the shortest SCS. It may seem possible that, at some index, the algorithm chooses a shorter SCS where there exists a longer SCS that should be chosen for $\mathcal{S}_{(1,m)}$. We prove that such a case is impossible.

Theorem 2. *For given strings T and $\theta(P)$ of length m, given an index i and given lengths l_1 and l_2 where $l_1 < l_2$, if $T[i : i+l_1-1]$ and $T[i : i+l_2-1]$ are SCSs, there always exists a sequence $\mathcal{S}_{(i,i+l_2-1)}$ of SCSs such that $\mathcal{S}_{(i,i+l_2-1)}[1] = T[i : i+l_1-1]$. Namely, if there exist two SCSs from an index i, then the longer string can always be expressed as a sequence of SCSs that starts with the shorter string.*

For all cases, at some index, if there exist both a shorter and a longer SCSs, then the longer string is always decomposed to a sequence of SCSs starting with the shorter string. Now we can adapt the algorithm for the approximate pattern matching problem.

As a running example for describing the algorithm for the approximate pattern matching, we use $P = GTTAG$ and $T = TGTGATTG$. For each index i, we start from building a table $R^i_P[m][m]$, where

$$R^i_P[j][k] = \begin{cases} R^i_P[j-1][k-1] + 1 & \text{if } T[i+k-1] = P[j], \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The role of R^i_P is as follows: we regard each diagonal in R^i_P as $\theta(P)$, and calculate the length of the longest SCS that ends at each index of $\theta(P)$.

The process of finding $\mathcal{S}_{(1,m)}$ for each index of T is described in Algorithm 2.

We claim that Algorithm 2 returns a correct answer for the approximate string matching problem for an index i and establish the following result:

Lemma 2. *Algorithm 2 returns true if and only if there exists a sequence $\mathcal{S}_{(1,m)}$ of SCSs for strings P and $T[i : i+m-1]$.*

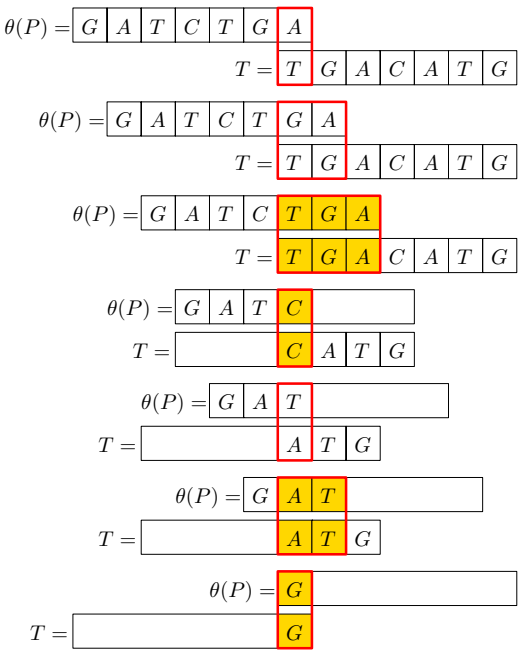

Fig. 1. Comparing $\theta(P)$ and T. Red boxes represent comparing lengths. SCSs (and their PSCSs in $\theta(P)$) are erased for better readability.

Algorithm 2 shows the existence of a sequence of SCSs for an index i and requires $O(m^2)$ space. Next, we show how to reduce the space requirement by relying on the properties of R_P^i.

Observation 3 (Properties of R_P^i). *For an index i, R_P^i has the following properties:*

1. $R_P^i[j][k] = R_P^{i+1}[j][k-1]$ *for all* i, j, k.
2. *The value of* $R_P^i[j][k]$ *is computed from* $R_P^i[j-1][k-1]$ *and* $T[i+k-1]$ *(from Equation (1)).*

The first property of Observation 3 ensures that there exists a table $R_P[m][n]$, where $R_P^i[j][k] = R_P[j][i+k-1]$ for all i, j, k. In other words, there exists a common table from which we can retrieve every $R_P^i[j][k]$ value. The second property of Observation 3 ensures that we need only the $(i+k-2)$th column of R_P and $T[i+k-1]$ to construct the $(i+k-1)$th column of R_P. Based on these properties, we design an improved algorithm that requires $O(m)$ space instead of $O(m^2)$.

In Algorithm 3, $R[m][2]$ is the table that keeps the $(i-1)$th and ith columns of R_P, and p_i is a pointer indicating the corresponding investigating cell for an index i. Note that line 8 to 9 of Algorithm 3 is same as running Algorithm 2

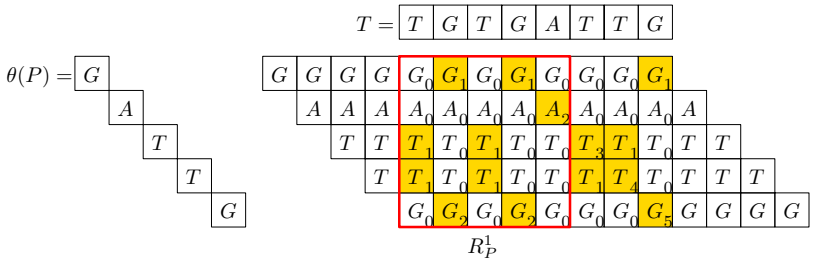

Fig. 2. An example of building R^1_P. The symbol in each cell represents the character in $\theta(P)$ which is compared to the character in T. The number at the bottom right of each cell represents the length of the longest SCS, which is the element in R^1_P.

Algorithm 2.

Input: Pattern P of length m, text T of length n, index i
Output: $\mathcal{S}_{(1,m)}$ for an index i

```
1 j ← m
2 construct R^i_P[m][m]
3 for k ← 1 to m do
4     if R^i_P[j][k] ≥ j + k − m then
5         add T[i+m−j : i+k−1] to S_(1,m)
6         j ← m − k
7 if j = 0 then return S_(1,m) else return "no S_(1,m) exists"
```

on m indices simultaneously to retrieve indices that match P. We show that Algorithm 3 runs in $O(nm)$ time using $O(m)$ space.

Lemma 3. *Algorithm 3 runs in $O(nm)$ time using $O(m)$ space, where $m = |P|$ and $n = |T|$.*

From Lemmas 2 and 3, we know that Algorithm 3 returns a correct answer for our problem in Definition 1. We establish the following statement for our problem:

Theorem 4. *We can solve an approximate string matching problem for inversions in Definition 1 in $O(nm)$ time using $O(m)$ space, where m is the size of the pattern and n is the size of the text.*

Note that Algorithm 3 improves the space complexity compared with the previous algorithm by Cantone et al. [1] while keeping the same runtime.

4 Average Runtime Analysis

Now we consider the frequency of character appearances in T and improve the average runtime when the alphabet size $|\Sigma| = t$ is larger than 3. The improved

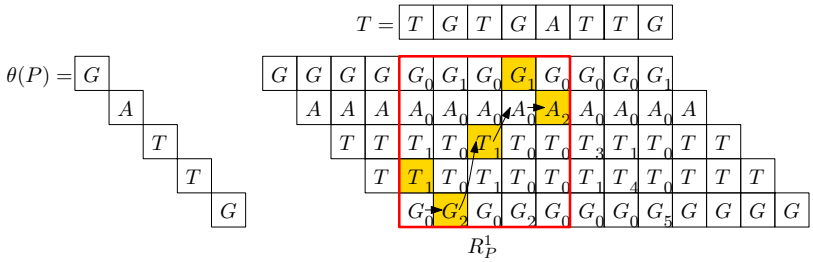

Fig. 3. Searching the existence of $\mathcal{S}_{(1,m)}$ for the first index. $\mathcal{S}_{(1,m)} = \{TG, T, GA\}$. The algorithm actually finds the ending cells of all PSCSs. Arrows in R_P^1 shows the track of inspection cells.

Algorithm 3.

Input: Pattern P of length m, text T of length n
Output: every index i where $\mathcal{S}_{(1,m)}$ exists

```
1  build R[m][2].
2  initialize R[j][1] as 0 for all j.
3  for i ← 1 to n do
4      for j ← 1 to m do
5          if T[i] = P[j] then R[j][2] ← R[j−1][1] + 1 else R[j][2] ← 0
6      p_i = m
7      for j ← 1 to m do
8          if R[p_{i−m+j}][2] ≥ p_{i−m+j} − j + 1 then p_{i−m+j} ← j − 1
9      if p_{i−m+1} = 0 then return i − m + 1 R[j][1] = R[j][2] for all j.
```

algorithm runs in $O(n\sqrt{m})$ when $t = 4$, which is the case of a DNA or RNA pattern. Moreover, the algorithm runs in linear time when $t \geq 5$.

Observation 5. *If there exists a sequence Θ_i of inversions at an index i such that $\Theta_i(P) = T[i : i+m-1]$, then $T[i : i+m-1]$ is a permutation of P.*

Based on Observation 5, we add a filter module that checks whether or not $T[i : i+m-1]$ is a permutation of P at an index i before line 4 of Algorithm 3.

The following probability analysis is from Grabowski et al. [5]. Suppose $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_t\}$ and $|\Sigma| = t$. Without loss of generality, assume that m is divisible by t and $k = \frac{m}{t}$. It requires $O(n)$ time and $O(t)$ space to return all i such that $T[i : i + m - 1]$ is a permutation of P [5]. Assume that P and T are random strings in which each character has $\frac{1}{t}$ occurrence probability for each position. Let $Pr\{\pi\}$ be the probability that the substring $T[i : i + m - 1]$ is a permutation of P. Then we have

$$\Pr\{\pi\} \leq \frac{t^{\frac{t}{2}}}{m^{\frac{t-1}{2}}}. \tag{2}$$

Based on Observation 5, instead of examining all substrings of size m of T, we only need to consider the substrings of size m of T that are permutations

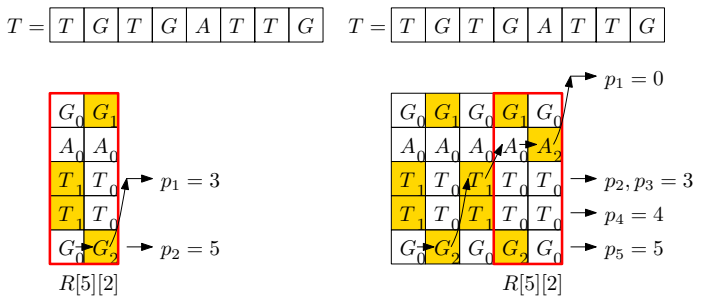

Fig. 4. An example of Algorithm 3 for $i = 2$ and $i = 5$. Since $p_1 = 0$ for $i = 5$, there exists a matching for $i = 1$.

of P. Suppose line 4 to 9 of Algorithm 3 takes cm number of calculations. Then, with $\Pr\{\pi\}$—the probability that $T[i : i+m-1]$ is a permutation of P—in Equation (2) and $(n-m+1)$ substrings of T, we have an upper bound of average runtime

$$AUB_1 = \frac{t^{\frac{t}{2}}}{m^{\frac{t-1}{2}}} \times (n - m + 1) \times cm^2 = O(nm^{\frac{5-t}{2}}).$$

AUB_1 is calculated under the assumption that we run the non-overlapping inversion matching algorithm to each permutation-matched index independently, similar to the average runtime analysis in [5]. Now we compute a smaller upper bound of the average runtime by tightly analyzing Algorithm 3.

In Algorithm 3, line 4 to 9, which takes cm, is repeated from index i to $i+m-1$ to check the non-overlapping inversion matching for i. Therefore, if we apply the permutation filter to Algorithm 3, cm is required for an index i if one of the indices from $i - m + 1$ to i has a permutation matching. The probability that one of the indices from $i - m + 1$ to i has such a matching is

$$1 - \left(1 - \frac{t^{\frac{t}{2}}}{m^{\frac{t-1}{2}}}\right)^m.$$

Therefore, we have another upper bound of average runtime

$$AUB_2 = \left(1 - \left(1 - \frac{t^{\frac{t}{2}}}{m^{\frac{t-1}{2}}}\right)^m\right) \times cm \times (n - m + 1) = O(nm^{\frac{5-t}{2}}). \quad (3)$$

Fig. 5 compares the upper bound calculations for AUB_1 and AUB_2. From the figure, we know that $AUB_2 \leq AUB_1$. Though the order of AUB_1 and AUB_2 are the same, we retrieved an upper bound of average runtime smaller than AUB_1.

Note that our average runtime in Equation (3) becomes faster than $O(nm)$ when $t \geq 4$ and becomes sublinear when $t \geq 6$. Combined with the runtime of the permutation filter [5], we establish the following statement for our problem:

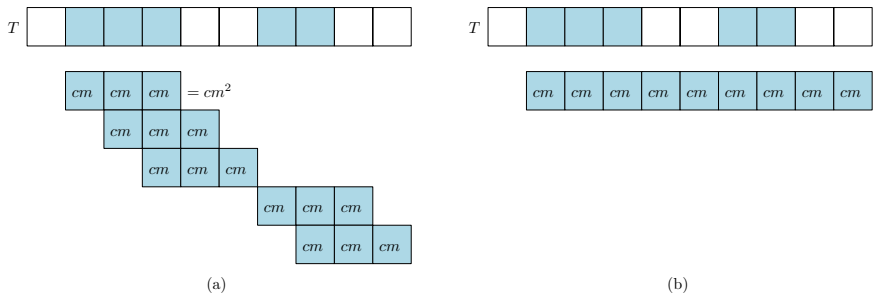

Fig. 5. Upper bound analysis of number of calculations for (a) AUB_1 and (b) AUB_2 for $m = 3$. Colored boxes in T represent the indices where permutation matching occurs.

Theorem 6. *We can solve an approximate string matching problem for inversions in Definition 1 in* $O(\max\{n, \min\{nm, nm^{\frac{5-t}{2}}\}\})$ *average runtime using* $O(m)$ *space, where* m *is the size of* P*,* n *is the size of* T *and* t *is the size of* Σ*.*

Theorem 6 guarantees a faster runtime—$O(n\sqrt{m})$—for a DNA string over $\Sigma = \{C, G, T, A\}$ and $t = 4$. Furthermore, for $t \geq 5$, the algorithm shows a linear runtime.

5 Conclusions

An inversion is an important operation for bio sequences such as DNA or RNA and is closely related to mutations. We have examined the string matching problem allowing inversions. Given a text T, a pattern P where $|P| = m \leq n = |T|$, our algorithm finds all indices i in which there is a matching alignment between P and $T[i : i+m-1]$ in $O(nm)$ time using $O(m)$ space. Moreover, we improve the algorithm and achieve $O(\max\{n, \min\{nm, nm^{\frac{5-t}{2}}\}\})$ average runtime using $O(m)$ space for an alphabet of size t. Compared with the previous algorithm, the new algorithm improves the space complexity, and shows a better average runtime for $t \geq 4$. A possible future direction is to consider multiple patterns instead of a single pattern for the pattern matching problem.

References

1. Cantone, D., Cristofaro, S., Faro, S.: Efficient string-matching allowing for non-overlapping inversions. Theoretical Computer Science 483, 85–95 (2013)
2. Cantone, D., Faro, S., Giaquinta, E.: Approximate string matching allowing for inversions and translocations. In: Proceedings of the Prague Stringology Conference 2010, pp. 37–51 (2010)
3. Chen, Z.-Z., Gao, Y., Lin, G., Niewiadomski, R., Wang, Y., Wu, J.: A space-efficient algorithm for sequence alignment with inversions and reversals. Theoretical Computer Science 325(3), 361–372 (2004)

4. Cho, D.-J., Han, Y.-S., Kim, H.: Alignment with non-overlapping inversions on two strings. In: Pal, S.P., Sadakane, K. (eds.) WALCOM 2014. LNCS, vol. 8344, pp. 261–272. Springer, Heidelberg (2014)
5. Grabowski, S., Faro, S., Giaquinta, E.: String matching with inversions and translocations in linear average time (most of the time). Information Processing Letters 111(11), 516–520 (2011)
6. Kececioglu, J.D., Sankoff, D.: Exact and approximation algorithms for the inversion distance between two chromosomes. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) CPM 1993. LNCS, vol. 684, pp. 87–105. Springer, Heidelberg (1993)
7. Li, S.C., Ng, Y.K.: On protein structure alignment under distance constraint. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 65–76. Springer, Heidelberg (2009)
8. Ogilvie, C.M., Scriven, P.N.: Meiotic outcomes in reciprocal translocation carriers ascertained in 3-day human embryos. European Journal of Human Genetics 10(12), 801–806 (2009)
9. Painter, T.S.: A New Method for the Study of Chromosome Rearrangements and the Plotting of Chromosome Maps. Science 78, 585–586 (1933)
10. Sakai, Y.: A new algorithm for the characteristic string problem under loose similarity criteria. In: Asano, T., Nakano, S.-i., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 663–672. Springer, Heidelberg (2011)
11. Schniger, M., Waterman, M.S.: A local algorithm for DNA sequence alignment with inversions. Bulletin of Mathematical Biology 54(4), 521–536 (1992)
12. Vellozo, A.F., Alves, C.E.R., do Lago, A.P.: Alignment with non-overlapping inversions in $O(n^3)$-time. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 186–196. Springer, Heidelberg (2006)