An Improved Approximation Algorithm for Scaffold Filling to Maximize the Common Adjacencies

Nan Liu[1,2], Haitao Jiang[1,3], Daming Zhu[1], and Binhai Zhu[4]

[1] School of Computer Science and Technology, Shandong University, Jinan, China
htjiang@mail.sdu.edu.cn, dmzhu@sdu.edu.cn
[2] School of Computer Science and Technology, Shandong Jianzhu University, Jinan, China
liunansdu@gmail.com
[3] School of Mathematics and System Science, Shandong University, Jinan, China
[4] Department of Computer Science, Montana State University, Bozeman, MT 59717-3880, USA
bhz@cs.montana.edu

Abstract. Scaffold filling is a new combinatorial optimization problem in genome sequencing. The one-sided scaffold filling problem can be described as: given an incomplete genome I and a complete (reference) genome G, fill the missing genes into I such that the number of common (string) adjacencies between the resulting genome I' and G is maximized. This problem is NP-complete for genome with duplicated genes and the best known approximation factor is 1.33, which uses a greedy strategy. In this paper, we prove a better lower bound of the optimal solution, and devise a new algorithm by exploiting the maximum matching method and a local improvement technique, which improves the approximation factor to 1.25.

1 Introduction

Motivation. The Next Generation Sequencing technology greatly improves the speed of genome sequencing, and more organisms for genome analysis can be sequenced. However, these sequences are often only a part of the complete genome. The whole genome sequencing problem is, in general, still an intractable problem. Currently, most sequencing results for genomes usually are in the form of scaffolds or contigs. Sometimes, applying these incomplete genomes for genomic analysis will introduce unnecessary errors. So it is natural to fill the missing gene fragments into the incomplete genome in a combinatorial way, and to obtain an 'augmented' genome which is closer to some reference genome.

Related Results. Muñoz *et al.* first investigated the one-sided permutation scaffold filling problem, and proposed an exact algorithm to minimize the genome rearrangement (DCJ) distance [12]. Subsequently, Jiang *et al.* considered the permutation scaffold filling under the breakpoint distance and showed that even the two-sided problem is polynomially solvable.

When genomes contain some duplicated genes, the scenario is completely different. There are three general criteria (or distance) to measure the similarity of genomes: the exemplar genomic distance [13], the minimum common string partition (MCSP) distance [3] and the maximum number of common string adjacencies [1,10,11]. Unfortunately, unless P=NP, there does not exist any polynomial time approximation (regardless of the factor) for computing the exemplar genomic distance even when each gene is allowed to repeat three times [6,4] or even two times [2,8]. The MCSP problem is NP-complete even if each gene repeats at most two times [7] and the best known approximation factor for the general problem is $O(\log n \log^* n)$ [3]. Based on the maximum number of common string adjacencies, Jiang *et al.* proved that the one-sided scaffold filling problem is also NP-complete, and designed a 1.33-approximation algorithm with a greedy strategy [10,11].

Our Contribution. In this paper, we design an approximation algorithm with a factor of 1.25 for the problem of one-sided scaffold filling to maximize the number of string adjacencies, by using a combined maximum matching and local improvement method.

2 Preliminaries

At first, we review some necessary definitions, which are also defined in [11]. Throughout this paper, all genes and genomes are unsigned, and it is straightforward to generalize the result to signed genomes. Given a gene set Σ, a string P is called *permutation* if each element in Σ appears exactly once in P. We use *c(P)* to denote the set of elements in permutation P. A string A is called *sequence* if some genes appear more than once in A, and $c(A)$ denotes genes of A, which is a multi-set of elements in Σ. For example, $\Sigma = \{a, b, c, d\}$, A = *abcdacd*, *c(A)* = $\{a, a, b, c, c, d, d\}$. A *scaffold* is an incomplete *sequence*, typically obtained by some sequencing and assembling process. A substring with m genes (in a sequence) is called an *m-substring*, and a 2-substring is also called a *pair*, as the genes are unsigned, the relative order of the two genes of a pair does not matter, i.e., the pair xy is equal to the pair yx. Given a scaffold A=$a_1a_2a_3 \cdots a_n$, let $P_A = \{a_1a_2, a_2a_3, \ldots, a_{n-1}a_n\}$ be the set of pairs in A.

Definition 1. *Given two scaffolds A=$a_1a_2 \cdots a_n$ and B=$b_1b_2 \cdots b_m$, if $a_ia_{i+1} = b_jb_{j+1}$ (or a_ia_{i+1}=$b_{j+1}b_j$), where $a_ia_{i+1} \in P_A$ and $b_jb_{j+1} \in P_B$, we say that a_ia_{i+1} and b_jb_{j+1} are matched to each other. In a maximum matching of pairs in P_A and P_B, a matched pair is called an* ***adjacency****, and an unmatched pair is called a* ***breakpoint*** *in A and B respectively.*

It follows from the definition that scaffolds A and B contain the same set of adjacencies but distinct breakpoints. The maximum matched pairs in B (or equally, in A) form the *adjacency set* between A and B, denoted as $a(A, B)$. We use $b_A(A, B)$ and $b_B(A, B)$ to denote the set of breakpoints in A and B respectively. A gene is called a *bp-gene*, if it appears in a breakpoint. A maximal substring T of A (or B) is call a *bp-string*, if each pair in it is a breakpoint. The leftmost and rightmost genes of a bp-string T are call the *end-genes* of T, the other genes in T are called the *mid-genes* of T. We illustrate the above definitions with the following example. Given scaffolds $A = \langle a\ b\ c\ e\ d\ a\ b\ a\ \rangle, B = \langle c\ b\ a\ b\ d\ a \rangle$, $P_A = \{ab, bc, ce, ed, da, ab, ba\}$ and $P_B = \{cb, ba, ab, bd, da\}$. The matched pairs are $(ab \leftrightarrow ba)$, $(bc \leftrightarrow cb)$, $(da \leftrightarrow da)$, $(ab \leftrightarrow ab)$. $a(A, B) = \{ab, bc, da, ab\}$, $b_A(A, B) = \{ce, ed, ba\}$, $b_B(A, B) = \{bd\}$. The bp-strings in A are *ced* and *ba*, and the bp-string in B is *bd*.

Given two scaffolds A=$a_1 a_2 \cdots a_n$ and B=$b_1 b_2 \cdots b_m$, as we can see, each gene except the four ending ones is involved in two adjacencies or two breakpoints or one adjacency and one breakpoint. To get rid of this imbalance, we add "#" to both ends of A and B, which fixes a small bug in [10,11]. From now on, we assume that A=$a_0 a_1 \cdots a_n a_{n+1}$ and B=$b_0 b_1 \cdots b_m b_{m+1}$, where a_0=a_{n+1}=b_0=b_{m+1}=$\#$.

For a sequence A and a multi-set of elements X, let $A + X$ be the set of all possible resulting sequences after filling all the elements in X into A. Now, we define the problems we study in this paper formally.

Definition 2. *Scaffold Filling to Maximize the Number of (String) Adjacencies (SF-MNSA).*
Input: *two scaffolds A and B over a gene set Σ and two multi-sets of elements X and Y, where $X = c(B) - c(A)$ and $Y = c(A) - c(B)$.*
Question: *Find $A^* \in A + X$ and $B^* \in B + Y$ such that $|a(A^*, B^*)|$ is maximized.*

The one-sided SF-MNSA problem is a special instance of the SF-MNSA problem where one of X and Y is empty.

Definition 3. *One-sided SF-MNSA.*
Input: *a complete sequence G and an incomplete scaffold I over a gene set Σ, a multi-set $X = c(G) - c(I) \neq \varnothing$ with $c(I) - c(G) = \varnothing$.*
Question: *Find $I^* \in I + X$ such that $|a(I^*, G)|$ is maximized.*

Note that while the two-sided SF-MNSA problem is more general and more difficult, the One-Sided SF-MNSA problem is more practical as a lot of genome analysis are based on some reference genome [12].

We now list a few basic properties of this problem.

Lemma 1. *Let G and I be the input of an instance of the One-sided SF-MNSA problem, and x be any gene which appears the same times in G and I. If x does not constitute breakpoint in G (resp. I), then it also does not constitute any breakpoint in I (resp. G).*

Lemma 2. *Let G and I be the input of an instance of the One-sided SF-MNSA problem, let $bp(I)$ and $bp(G)$ be the multi-set of bp-genes in I and G respectively. Then any gene in $bp(G)$ appears in $bp(I) \cup X$, and $bp(I) \subseteq bp(G)$.*

Proof. Assume to the contrary that there exists a gene x, $x \in bp(G)$, but $x \notin bp(I) \cup X$. Since $x \notin X$, x appears the same number of times in G and I; moreover, $x \notin bp(I)$, then all the pairs in I containing x are adjacencies. From Lemma 1, all the pairs involving x in G are adjacencies, contradicting the assumption that $x \in bp(G)$. So any gene in $bp(G)$ appears in $bp(I) \cup X$. By a similar argument, we can prove $bp(I) \subseteq bp(G)$. □

Each breakpoint contains two genes, from what we discussed in Lemma 2, every breakpoint in the complete sequence G belongs to one of the three multi-sets according to the affiliation of its two bp-genes.

- $BP_1(G)$: breakpoints with one bp-gene in X and the other bp-gene not in X.
- $BP_2(G)$: breakpoints with both of the bp-genes in X.
- $BP_3(G)$: breakpoints with both of the bp-genes not in X.

3 Approximation Algorithm for One-Sided SF-MNSA

In this section, we present a 1.25-Approximation algorithm for the one-sided SF-MNSA problem. The goal of solving this problem is, while inserting the genes of X into the scaffold I, to obtain as many adjacencies as possible. No matter in what order the genes are inserted, they appears in groups in the final $I' \in I + X$, so we can consider that I' is obtained by inserting strings (composed of genes of X) into I.

Obviously, inserting a string of length one (i.e., a single gene) will generate at most two adjacencies, and inserting a string of length m will generate at most m+1 adjacencies. Therefore, we will have two types of inserted strings.

1. Type-1: a string of k missing genes $x_1, x_2, \ldots, x_k$ are inserted in between $y_i y_{i+1}$ in the scaffold I to obtain k+1 adjacencies (i.e., $y_i x_1, x_1 x_2, \ldots, x_{k-1} x_k, x_k y_{i+1}$), where $y_i y_{i+1}$ is a breakpoint.
   In this case, $x_1 x_2 \ldots x_k$ is called a k-Type-1 string, $y_i y_{i+1}$ is called a *dock*, and we also say that $y_i y_{i+1}$ *docks* the corresponding k-Type-1 string $x_1 x_2 \ldots x_k$.
2. Type-2: a sequence of l missing genes $z_1, z_2, \ldots, z_l$ are inserted in between $y_j y_{j+1}$ in the scaffold I to obtain l adjacencies (i.e., $y_j z_1$ or $z_l y_{j+1}, z_1 z_2, \ldots, z_{l-1} z_l$), where $y_j y_{j+1}$ is a breakpoint; or a sequence of l missing genes $z_1, z_2, \ldots, z_l$ are inserted in between $y_j y_{j+1}$ in the scaffold I to obtain l+1 adjacencies (i.e., $y_j z_1$, $z_1 z_2$, …, $z_{l-1} z_l$, $z_l y_{j+1}$), where $y_j y_{j+1}$ is an adjacency.

This is the basic observation for devising our algorithm. Most of our work is devoted to searching the Type-1 strings.

3.1 Searching the 1-Type-1 Strings

To identify the 1-Type-1 strings, we construct a bipartite graph $\mathcal{G}_1 = (X, b_I(I, G), E)$, where for each gene x_i of X and each breakpoint $y_j y_{j+1}$ of $b_I(I, G)$, if we can obtain two adjacencies by inserting x_i in between $y_j y_{j+1}$, then there is an edge connecting x_i to $y_j y_{j+1}$. Therefore, a matching of $\mathcal{G}_1$ gives us the 1-Type-1 strings and their corresponding docks. In our algorithm, we compute a maximum matching of $\mathcal{G}_1$, then the number of 1-Type-1 strings obtained by our algorithm will not be smaller than that of the optimal solution.

```
Algorithm 1: Max-Matching(G, I)
1  Construct a bipartite graph $\mathcal{G}_1 = (X, b_I(I, G), E)$,
   $(x_i, y_j y_{j+1}) \in E$ iff $y_j y_{j+1}$ docks $x_i$, for all $x_i \in X$ and $y_j y_{j+1} \in b_I(I, G)$.
2  Compute a maximum matching $M$ of $\mathcal{G}_1$.
3  Return $M$.
```

3.2 Searching the 2-Type-1 Strings

To identify the 2-Type-1 strings, we first find a set Q of 2-Type-1 strings greedily. For Q, let $c(Q)$ be the multi-set of genes in Q, and let $D(Q)$ be the set of corresponding docks. Then, we improve Q to find more 2-Type-1 strings using a local search method. There are two ways to improve Q:

1. Delete a 2-Type-1 string x_ix_j from Q, release its corresponding dock y_py_{p+1}, construct two new 2-Type-1 strings x_ix_k and x_jx_l, where $x_k, x_l \notin c(Q)$.
2. Delete a 2-Type-1 string x_ix_j from Q, release its corresponding dock y_py_{p+1}, construct two new 2-Type-1 strings x_ix_k and x_rx_l, where x_rx_l docks y_py_{p+1}, and $x_k, x_l, x_r \notin c(Q)$.

```
Algorithm 2: Local-Optimize($G, I$)
1   Identify the breakpoints and adjacencies in $G$ and $I$, $Q \leftarrow \emptyset$, $D(Q) \leftarrow \emptyset$.
2   WHILE ($x_ix_j$ is docked at $y_py_{p+1}$, where $x_i, x_j \in X$ and $y_py_{p+1} \in b_I(I,G)$)
3       { $Q \leftarrow Q \cup \{x_ix_j\}$, $D(x_ix_j) \leftarrow \{y_py_{p+1}\}$,
        $X \leftarrow X - \{x_i, x_j\}$, $b_I(I,G) \leftarrow b_I(I,G) - \{y_py_{p+1}\}$ }.
4   For each 2-Type-1 string $x_ix_j \in Q$ docked at $y_py_{p+1} \in b_I(I,G)$
    4.1   if ($x_ix_k$ is docked at $y_uy_{u+1}$, $x_jx_l$ is docked at $y_vy_{v+1}$, where $x_k, x_l \in X$,
              $y_uy_{u+1}, y_vy_{v+1} \in b_I(I,G)$)
          then    { $Q \leftarrow Q \cup \{x_ix_k, x_jx_l\} - \{x_ix_j\}$, $D(x_ix_k) \leftarrow \{y_uy_{u+1}\}$,
            $D(x_jx_l) \leftarrow \{y_vy_{v+1}\}$, $X \leftarrow X - \{x_k, x_l\}$,
            $b_I(I,G) \leftarrow b_I(I,G) \cup \{y_py_{p+1}\} - \{y_uy_{u+1}, y_vy_{v+1}\}$ }.
          goto step 4.
    4.2   if ($x_ix_k$ is docked at $y_uy_{u+1}$, $x_lx_r$ is docked at $y_py_{p+1}$, where $x_k, x_l, x_r \in X$,
              $y_uy_{u+1} \in b_I(I,G)$)
          then    { $Q \leftarrow Q \cup \{x_ix_k, x_lx_r\} - \{x_ix_j\}$, $D(x_ix_k) \leftarrow \{y_uy_{u+1}\}$,
            $D(x_lx_r) \leftarrow \{y_py_{p+1}\}$, $X \leftarrow X - \{x_k, x_l, x_r\} \cup \{x_j\}$,
            $b_I(I,G) \leftarrow b_I(I,G) - \{y_uy_{u+1}\}$ }.
          goto step 4.
5   Return $Q$ and $D(Q)$.
```

3.3 Searching the 3-Type-1 Strings

To search the 3-Type-1 strings, we use a greedy method.

```
Algorithm 3: Greedy-Search($G, I$)
1   Identify the breakpoints and adjacencies in $G$ and $I$, $F \leftarrow \emptyset$, $D(F) \leftarrow \emptyset$.
2   WHILE ($x_ix_jx_k$ is docked at $y_wy_{w+1}$, where $x_i, x_j, x_k \in X$ and $y_wy_{w+1} \in b_I(I,G)$)
3       {$F \leftarrow F \cup \{x_ix_jx_k\}$, $D(x_ix_jx_k) \leftarrow \{y_wy_{w+1}\}$,
        $X \leftarrow X - \{x_i, x_j, x_k\}$, $b_I(I,G) \leftarrow b_I(I,G) - \{y_py_{p+1}\}$}.
4   Return $F$ and $D(F)$.
```

3.4 Inserting the Remaining Genes

In this subsection, we present a polynomial time algorithm guaranteeing that the number of adjacencies increases by the same number of the genes inserted. A general idea of this algorithm was mentioned in [11], with many details missing, and we will present the details here.

Given the complete sequence G and the scaffold I, as we discussed in Section 2, the breakpoints in G can be divided into three sets: $BP_1(G)$, $BP_2(G)$, and $BP_3(G)$. In any case, the breakpoints in $BP_3(G)$ cannot be converted into adjacencies; so we try to convert the breakpoints in $BP_1(G)$ and $BP_2(G)$ into adjacencies.

Lemma 3. *If $BP_1(G) \neq \emptyset$, then there exists a breakpoint in I where after some gene of X is inserted, the number of adjacencies increases by one.*

Proof. Let $t_i t_{i+1}$ be a breakpoint in G, satisfying that $t_i t_{i+1} \in BP_1(G)$, $t_i \in X$, and, from Lemma 2, $t_{i+1} \in bp(I)$. Then, there exists a breakpoint $t_{i+1} s_j$ or $s_k t_{i+1}$ in I. Hence, if we insert t_i in between that breakpoint, we will obtain a new adjacency $t_i t_{i+1}$ without affecting any other adjacency. □

Thus, it is trivial to obtain one more adjacency whenever $BP_1(G) \neq \emptyset$.

Lemma 4. *For any $x \in X \cap c(I)$, if there is an "xx" breakpoint in G then after inserting x in between some "xy" pair in I, the number of adjacencies increases by one.*

Proof. If "xy" is a breakpoint, then after inserting an 'x' in between it, we obtain a new adjacencies "xx". If "xy" is an adjacency, then after inserting an 'x' in between it, we have "xxy". The adjacency "xy" still exists, and we obtain a new adjacencies "xx". □

Lemma 5. *If there is a breakpoint "xy" in $BP_2(G)$ and a breakpoint "xz" (resp. "yz") in I, then after inserting y (resp. x) in between "xz" (resp. "yz") in I, the number of adjacencies increases by one.*

Proof. From the definition of $BP_2(G)$, we know that $x, y \in X$. Since "xy" is a breakpoint in G and "xz" is a breakpoint in I, we obtain a new adjacency "xy" by inserting y in between "xz", without affecting any other adjacency. A similar argument for inserting x in between "yz" also holds. □

Next, we show that the following case is polynomially solvable. This case satisfies the following conditions.

1. $BP_1(G) = \emptyset$;
2. It does not contain a breakpoint like "xx" in G unless $x \notin X \cap c(I)$;
3. For any breakpoint of the form "xy" in $BP_2(G)$, all the pairs in I involving x or y are adjacencies.

Let $BS_2(G)$ be the set of bp-strings in G with all breakpoints belonging to $BP_2(G)$.

Lemma 6. *In the case satisfying (1), (2) and (3), the number of times a gene appears as an end-gene of some bp-string of $BS_2(G)$ is even.*

From Lemma 6, if we denote each bp-string of $BS_2(G)$ by a vertex, and there is an edge between two vertices iff their corresponding bp-strings have a common end-gene, the resulting graph contains a cycle of distinct vertices. Traveling this cycle, concatenating the bp-strings corresponding to the vertices, and deleting one copy of the common end-gene, eventually we can obtain a string composed of genes of X. The following lemma and corollary shows that this string can be inserted into I entirely, generating no breakpoint at all.

Lemma 7. *In the case satisfying (1),(2) and (3), for a gene x, let q_1 be the number that it appears as an end-gene, let q_2 be the number that it appears in some bp-string of $BS_2(G)$ as a mid-gene, and let r be the number that it appears in X. Then, we have $r = q_1/2 + q_2$.*

We can summarize the above ideas as the algorithm Insert-Whole-Strings(•), which ensures us to obtain as many adjacencies as the number of missing genes inserted. The details will be given in the full version.

Theorem 1. *The algorithm Insert-Whole-Strings(•) guarantees that the number of adjacencies increased is not smaller than the number of genes inserted.*

4 Analysis of the Approximation Algorithm

4.1 A Lower Bound

Given an instance of One-sided SF-MNSA, let $I^* \in I+X$ be the final scaffold in the optimal solution after inserting all genes of X into I. Compared to I, all genes belonging to X appear as substrings in I^*. Let $x_1x_2\ldots x_l$ be a string inserted in between y_iy_{i+1} in I^*, then either y_ix_1 or x_ly_{i+1} or both are adjacencies. Since otherwise, we could delete this string from I^* (number of adjacencies decreases by at most l-1), re-insert it following the algorithm *Insert-Whole-Strings*(•) (number of adjacencies increases by at least l), and obtain one more adjacency. Thus, we have the following corollary of Theorem 1,

Corollary 1. *Each substring in I^* composed of genes of X is either Type-1 or Type-2.*

Now, we present a new lower bound for the optimal number of adjacencies.

Lemma 8. *Let OPT be the number of adjacencies between G and I^*, k_0 be the number of adjacencies between G and I, and k_1=$|X|$. Let b_i be the number of i-Type-1 substrings and q be the maximum length of Type-1 substrings in the optimal solution between G and I^*. Then*

$$OPT - k_0 = k_1 + b_1 + b_2 + \ldots + b_q \le \frac{5}{4}(k_1 + \frac{3}{5}b_1 + \frac{2}{5}b_2 + \frac{1}{5}b_3) \qquad (1)$$

Following Lemma 8, if the number of Type-1 substrings computed in the approximation algorithm is not smaller than $(3b_1 + 2b_2 + b_3)/5$, then the approximation factor is 5/4.

4.2 Description of the Main Algorithm

There are four steps in our algorithm. Firstly, we try to search the 1-Type-1 strings; secondly, we try to search the 2-Type-1 strings; thirdly, we try to search the 3-Type-1 strings; finally, we insert the remaining genes in X, guaranteeing that on average we will obtain at least one adjacency for each inserted missing gene.

```
Main Algorithm
Input: Complete sequence $G$ and incomplete scaffold $I$, $X$=$c(G) - c(I)$.
Output: $I' \in I + X$
 1   Call Max-Matching($G, I$), which returns a maximum matching $M$.
        { Let $M_1$ be the set of 1-Type-1 strings and $D(M_1)$ be their corresponding docks.
        Insert the 1-Type-1 strings into their corresponding docks.
        The resulting incomplete scaffold is denoted as $I_1$. }
 2   Call Local-Optimize($G, I_1$), which returns $Q$ and $D(Q)$.
        { Insert the 2-Type-1 strings of $Q$ into their corresponding docks.
        The resulting incomplete scaffold is denoted as $I_2$. }
 3   Call Greedy-Search($G, I_2$), which returns $F$ and $D(F)$.
        { Insert the 3-Type-1 strings of $F$ into their corresponding docks.
        The resulting incomplete scaffold is denoted as $I_3$. }
 4   Call Insert-Whole-Strings($G, I_3$).
        { Let the resulting complete scaffold be $I'$. }
 5   Return $I'$.
```

4.3 Proof of the Approximation Factor

In our algorithm, we make effort to insert Type-1 substrings as much as possible. But a Type-1 substring (say I_s) inserted by our algorithm may make other Type-1 substrings in some optimal solution infeasible, we say I_s *destroys* them. The following lemma shows the number of Type-1 substrings that could be destroyed by a given Type-1 substring.

Lemma 9. *A i-Type-1 substring can destroy at most i+1 Type-1 substrings in some optimal solution.*

Proof. Assume that an i-Type-1 substring I_s is inserted in between some breakpoint $y_j y_{j+1}$ in I. Then each of the genes in I_s, if not use by I_s, could form a distinct Type-1 substring in some optimal solution. Also, there may exist another Type-1 substring that could be inserted in between the breakpoint $y_j y_{j+1}$ in the optimal solution. Totally, at most i+1 Type-1 substrings in the optimal solution could be destroyed by I_s. □

Next, we will compare the number of i-Type-1 substrings between our algorithm and some optimal solution. Above all, we focus on the 1-Type-1 substrings, which are computed by the algorithm *Max-Matching(•)*.

Lemma 10. *The 1-Type-1 substrings computed in our algorithm is not smaller than those in any optimal solution.*

For the simplicity of comparison, given an edge $e \in M$ and an edge $\tilde{e} \in W$, we say that e *destroys* $\tilde{e}$ if e and $\tilde{e}$ have exactly one common endpoint. Actually, e *destroys* $\tilde{e}$ implies that the 1-Type-1 substring corresponding to e makes the 1-Type-1 substring corresponding to $\tilde{e}$ *infeasible*.

Since M is a maximum matching, each connect component of $M \cup W$ is either a path or an even cycle, and at least one end-edge of a path belongs to M. So, for each edge in $W - M$, we can assign a distinct edge in $M - W$ that destroys it. Under this assignment, each edge in $M - W$ destroys at most one edge in $W - M$, i.e., a 1-Type-1 substring of our algorithm can destroy at most one 1-Type-1 substring of the optimal solution.

Cases of such substrings which are destroyed by a 1-Type-1 substring of our algorithm (except the 1-Type-1 substrings of $M \cap W$), are described in the following Table 1, where b_{1j} denotes the number of 1-Type-1 substrings of the jth case and "other" means an i-Type-1 substring (i >3).

Table 1. Cases of substrings destroyed by a 1-Type-1 substring

number of 1-Type-1 substrings	one substring destroyed	another substring destroyed
b'_{11}	1-Type-1	2-Type-1
b'_{12}	1-Type-1	3-Type-1
b'_{13}	1-Type-1	other or none
b'_{14}	2-Type-1	2-Type-1
b'_{15}	2-Type-1	3-Type-1
b'_{16}	2-Type-1	other or none
b'_{17}	3-Type-1	3-Type-1
b'_{18}	3-Type-1	other or none
b'_{19}	other	other or none

Let b'_1 be the number of 1-Type-1 substrings computed by our algorithm, i.e., $b'_1 = |M|$. Then we have,

$$b'_1 = b'_{11} + b'_{12} + b'_{13} + \cdots + b'_{19} + |M \cap W| \tag{2}$$

Now, we will analyze the number of 2-Type-1 substrings, which are computed by the algorithm *Local-Optimize(•)*.

Lemma 11. *Let b_2 be the number of 2-Type-1 substrings of some optimal solution, b'_2 be the number of 2-Type-1 substrings obtained by our algorithm. Then*

$$b'_2 \geq \frac{1}{2} \times (b_2 - b'_{11} - 2b'_{14} - b'_{15} - b'_{16}). \tag{3}$$

Proof. Let Q be the set of 2-Type-1 substrings obtained at step 2 (by the *Local-Optimize(•)* algorithm), and P be the set of 2-Type-1 substrings in some optimal solution which does not include those 2-Type-1 substrings destroyed at step 1 (by the *Max-Matching(•)* algorithm), i.e., $|Q| = b'_2$, $|P| = b_2 - b'_{11} - 2b'_{14} - b'_{15} - b'_{16}$. Let $R = Q \cap P$ be the set of 2-Type-1 substrings which are contained in both Q and P. We just proceed to compare the number of 2-Type-1 substrings in $P - R$ and $Q - R$, even though we cannot compute $P - R$ explicitly.

Let $Q-R = \{x_1x_2, x_3x_4, \ldots, x_{2q-1}x_{2q}\}$ and $P-R = \{y_1y_2, y_3y_4, \ldots, y_{2p-1}y_{2p}\}$. For simplicity, we mark these gene appearing more than one times in $P-R$ (and $Q-R$) with distinct labels. Consider an imaginary bipartite graph $\mathcal{G}' = (Q - R, P - R, E)$, where there exists an edge between two vertices iff their corresponding 2-Type-1 substrings share a common gene (same gene and same label). Then, each vertex in $\mathcal{G}'$ has degree at most two.

For a vertex $x_i x_{i+1}$ of $Q - R$ and an isolated vertex $y_j y_{j+1}$ in $P - R$, if both $x_i x_{i+1}$ and $y_j y_{j+1}$ dock $s_k s_{k+1}$, then mark both $x_i x_{i+1}$ and $y_j y_{j+1}$ with a same color. Each such pair are marked with a distinct color. Since in the *Local-Optimize(•)* algorithm the original 2-Type-1 substrings in Q are formed by a greedy search, any isolated vertex of $P - R$ in $\mathcal{G}'$ must be colored. So $\mathcal{G}'$ is composed of disjoint paths, even cycles, colored isolated vertices of $P - R$ and isolated vertices of $Q - R$.

In an even cycle or even path (i.e., with an even number of vertices), the number of vertices in $Q - R$ and $P - R$ is equal, and in the worst case all the vertices on even cycles/paths in $Q - R$ are colored.

For an odd path (i.e., with an odd number of vertices), as $\mathcal{G}'$ is bipartite, the two endpoints are either both in $P - R$ or both in $Q - R$. In an odd path with both endpoints in $Q - R$, the number of vertices in $Q - R$ is one more than that in $P - R$.

Let $p_1, q_1, p_2, q_2, \ldots, p_r, q_r, p_{r+1}$ be an odd path in $\mathcal{G}'$, where $p_i \in (P - R)$ ($1 \leq i \leq r + 1$), and $q_j \in (Q - R)$ ($1 \leq j \leq r$). We claim that q_1 cannot be colored. Since otherwise, let p_t be the isolated vertex that has the same color with q_1, according to step 4.2 of the *Local-Optimize(•)* algorithm, q_1 should be deleted from Q, p_1 and p_t should be added to Q. This contradicts the local optimality of Q. A similar claim holds for q_t. So each odd path in $\mathcal{G}'$ ends with some vertices in $P - R$ whose neighbors are uncolored.

Let l be the number of odd paths in $\mathcal{G}'$ which end with some vertices in $P - R$. Putting all together, we can conclude that: (1) the number of isolated colored vertices in $P - R$ is at most $|Q - R| - l$; (2) the vertices in $P - R$ appearing on the cycles or paths in $\mathcal{G}'$ is at most $|Q - R| + l$. Then $|P - R| \leq 2|Q - R|$. So $|P| \leq 2|Q|$. Since $|Q| = b'_2$ and $|P| = b_2 - b'_{11} - 2b'_{14} - b'_{15} - b'_{16}$, we have $b'_2 \geq (b_2 - b'_{11} - 2b'_{14} - b'_{15} - b'_{16})/2$, hence the lemma follows. □

When computing the 2-Type-1 substrings at Step 2, except the 2-Type-1 substrings of $R = Q \cap P$, these 2-Type-1 substrings computed by our algorithm could destroy some other Type-1 substrings of the optimal solution. It follows from Lemma 9 that a 2-Type-1 substring can destroy at most three Type-1 substrings in some optimal solution. Cases of such substrings, which are destroyed by a 2-Type-1 substring of our algorithm, are described in the following Table 2, where b'_{2j}s denote the number of 2-Type-1 substring in the jth case and "other" means an i-Type-1 substring ($i > 3$).

Let b'_2 denote the number of 2-Type-1 substrings computed by our algorithm, then we have $b'_2 = b'_{20} + b'_{21} + b'_{22} + b'_{23} + \cdots + b'_{29} + |R|$, where $R = Q \cap P$.

At Step 3 (i.e., the *Greedy-Search(•)* algorithm), let Z be the set of 3-Type-1 substrings both in our algorithm and in the optimal solution. Except those 3-Type-1 substrings of Z, a 3-Type-1 substring computed by our algorithm can destroy at most four Type-1 substrings in some optimal solution. Cases of such substrings which are destroyed by a 3-Type-1 substring are described in the following Table 3, where b'_{3j}s denote the number of 3-Type-1 substrings of each case j and "other" means an i-Type-1 substring ($i > 3$).

Table 2. Cases of substrings destroyed by a 2-Type-1 substring

number of 2-Type-1 substrings	the first substring destroyed	the second substring destroyed	the third substring destroyed
b'_{20}	2-Type-1	2-Type-1	2-Type-1
b'_{21}	2-Type-1	2-Type-1	3-Type-1
b'_{22}	2-Type-1	2-Type-1	other or none
b'_{23}	2-Type-1	3-Type-1	3-Type-1
b'_{24}	2-Type-1	3-Type-1	other or none
b'_{25}	2-Type-1	other or none	other or none
b'_{26}	3-Type-1	3-Type-1	3-Type-1
b'_{27}	3-Type-1	3-Type-1	other or none
b'_{28}	3-Type-1	other or none	other or none
b'_{29}	other	other or none	other or none

Let b'_3 denote the number of 3-Type-1 substrings of our algorithm, then we have $b'_3 = b'_{31} + b'_{32} + b'_{33} + b'_{34} + b'_{35} + |Z|$, where and Z is the set of 3-Type-1 substrings both in our algorithm and in the optimal solution.

Table 3. Cases of substrings destroyed by a 3-Type-1 substring

number of 3-Type-1 substrings	the 1st substring destroyed	the 2nd substring destroyed	the 3rd substring destroyed	the 4th substring destroyed
b'_{31}	3-Type-1	3-Type-1	3-Type-1	3-Type-1
b'_{32}	3-Type-1	3-Type-1	3-Type-1	other or none
b'_{33}	3-Type-1	3-Type-1	other or none	other or none
b'_{34}	3-Type-1	other or none	other or none	other or none
b'_{35}	other	other or none	other or none	other or none

Lemma 12. *Let b'_1, b'_2, b'_3 be the number of new 1-Type-1, 2-Type-1 and 3-Type-1 substrings inserted at step1, step 2 and step 3 in our main algorithm respectively. Then*

$$b'_1 + b'_2 + b'_3 \geq \frac{1}{5} \times (3b_1 + 2b_2 + b_3).$$

Theorem 2. *The One-sided SF-MNSA problem admits a polynomial time factor-1.25 approximation.*

Proof. Following the approximation algorithm, Theorem1, Lemma 8, and Lemma 12, we have the approximation solution value APP, which satisfies the following inequalities.

$$APP - k_0 = k_1 + b'_1 + b'_2 + b'_3 \geq k_1 + \frac{3}{5}b_1 + \frac{2}{5}b_2 + \frac{1}{5}b_3 \geq \frac{4}{5}(OPT - k_0).$$

So, we have $APP \geq \frac{4}{5}OPT + \frac{1}{5}k_0 \geq \frac{4}{5}OPT$. Hence $\frac{OPT}{APP} \leq 1.25$, and the theorem is proven. □

5 Concluding Remarks

In this paper, we used a mixture of maximum matching and local improvement methods to obtain a factor-1.25 approximation for One-sided MNSA. It would be interesting to know whether the 1.25 factor can be further improved.

Acknowledgments. This research is partially supported by NSF of China projects 60070019 and 60928006, by the Doctoral Fund of Ministry of Education of China under grant 20090131110009, by China Postdoctoral Science Foundation under grant 2011M501133, by NSF under grant DMS-0918034, and by the Open Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

References

1. Angibaud, S., Fertin, G., Rusu, I., Thevenin, A., Vialette, S.: On the approximability of comparing genomes with duplicates. J. Graph Algorithms and Applications 13(1), 19–53 (2009)
2. Blin, G., Fertin, G., Sikora, F., Vialette, S.: The EXEMPLAR BREAKPOINT DISTANCE for non-trivial genomes cannot be approximated. In: Das, S., Uehara, R. (eds.) WALCOM 2009. LNCS, vol. 5431, pp. 357–368. Springer, Heidelberg (2009)
3. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. In: Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA 2002), pp. 667–676 (2002)
4. Chen, Z., Fowler, R., Fu, B., Zhu, B.: On the inapproximability of the exemplar conserved interval distance problem of genomes. J. Combinatorial Optimization 15(2), 201–221 (2008)
5. Chen, Z., Fu, B., Xu, J., Yang, B., Zhao, Z., Zhu, B.: Non-breaking similarity of genomes with gene repetitions. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 119–130. Springer, Heidelberg (2007)
6. Chen, Z., Fu, B., Zhu, B.: The approximability of the exemplar breakpoint distance problem. In: Cheng, S.-W., Poon, C.K. (eds.) AAIM 2006. LNCS, vol. 4041, pp. 291–302. Springer, Heidelberg (2006)
7. Goldstein, A., Kolman, P., Zheng, J.: Minimum common string partition problem: Hardness and approximations. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 484–495. Springer, Heidelberg (2004); also in: The Electronic Journal of Combinatorics 12, paper R50 (2005)
8. Jiang, M.: The zero exemplar distance problem. In: Tannier, E. (ed.) RECOMB-CG 2010. LNCS, vol. 6398, pp. 74–82. Springer, Heidelberg (2010)
9. Jiang, H., Zheng, C., Sankoff, D., Zhu, B.: Scaffold filling under the breakpoint distance. In: Tannier, E. (ed.) RECOMB-CG 2010. LNCS, vol. 6398, pp. 83–92. Springer, Heidelberg (2010)
10. Jiang, H., Zhong, F., Zhu, B.: Filling scaffolds with gene repetitions: Maximizing the number of adjacencies. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 55–64. Springer, Heidelberg (2011)
11. Jiang, H., Zheng, C., Sankoff, D., Zhu, B.: Scaffold filling under the breakpoint and related distances. IEEE/ACM Trans. Bioinformatics and Comput. Biology 9(4), 1220–1229 (2012)
12. Muñoz, A., Zheng, C., Zhu, Q., Albert, V., Rounsley, S., Sankoff, D.: Scaffold filling, contig fusion and gene order comparison. BMC Bioinformatics 11, 304 (2010)
13. Sankoff, D.: Genome rearrangement with gene families. Bioinformatics 15(11), 909–917 (1999)