A family of approximation algorithms for the maximum duo-preservation string mapping problem

Bartłomiej Dudek¹, Paweł Gawrychowski¹,², and Piotr Ostropolski-Nalewaja¹

¹Institute of Computer Science, University of Wrocław, Poland
²University of Haifa, Israel

Abstract

In the Maximum Duo-Preservation String Mapping Problem we are given two strings and wish to map the letters of the former to the letters of the latter as to maximise the number of duos. A duo is a pair of consecutive letters that is mapped to a pair of consecutive letters in the same order. This is complementary to the well-studied Minimum Common String Partition Problem, where the goal is to partition the former string into blocks that can be permuted and concatenated to obtain the latter string.

Maximum Duo-Preservation String Mapping Problem is APX-hard. After a series of improvements, Brubach [WABI 2016] showed a polynomial-time \(3.25\)-approximation algorithm. Our main contribution is that, for any \(\epsilon > 0\), there exists a polynomial-time \((2 + \epsilon)\)-approximation algorithm. Similarly to a previous solution by Boria et al. [CPM 2016], our algorithm uses the local search technique. However, this is used only after a certain preliminary greedy procedure, which gives us more structure and makes a more general local search possible. We complement this with a specialised version of the algorithm that achieves \(2.67\)-approximation in quadratic time.

1 Introduction

A fundamental question in computational biology and, consequently, stringology, is comparing similarity of two strings. A textbook approach is to compute the edit distance, that is, the smallest number of operations necessary to transform one string into another, where every operation is inserting, removing, or replacing a character. While this can be efficiently computed in quadratic time, a major drawback from the point of view of biological applications is that every operation changes only a single character. Therefore, it makes sense to also allow moving arbitrary substrings as a single operation to obtain edit distance with moves. Unfortunately, such relaxation already makes computing the smallest number of operation NP-hard [15], but Cormode and Muthukrishnan [8] showed an almost linear-time \(O(\log n \cdot \log^* n)\)-approximation algorithm. The problem is already interesting if the only allowed operation is substring move. In such version, it is usually called the Minimum Common String Partition (MCSP). Formally, given two strings \(X\) and \(Y\), where \(Y\) is a permutation of \(X\). The goal is to cut \(X\) into the least number of pieces that can be rearranged (without reversing) and concatenated to obtain \(Y\).

MCSP is known to be APX-hard [11]. Chrobak et al. [7] analysed the performance of the simple greedy approximation algorithm, and Kaplan and Shafrir [14] further improved their bounds. This simple greedy algorithm can be further tweaked to obtain a better practical results [12]. Also, an exact exponential time algorithm [10] and different parameterizations were considered [4,5,9,13].
There was also some interest in the problem complementary to MCSP called the Maximum Duo-Preservation String Mapping (MPSM) introduced by Chen et al. [6]. The goal there is to map letters of $X$ to letters of $Y$ as to maximise the number of preserved duos. A duo is a pair of consecutive letters, and a duo of $X$ is said to be preserved if its pair of consecutive letters is mapped to a pair of consecutive letters of $Y$ (in the same order). These two problems are indeed complementary, as one can think of preserving a duo as not splitting two letters apart to see that the number of preserved duos and the number of pieces add up to $|X|$. Of course, this does not say anything about the relationship between the approximation guarantees for both problems. Chen et al. [6] designed an $k^2$-approximation algorithm based on linear programming for the restricted version of the problem called $k$-MPSM, where each letter occurs at most $k$ times. This was soon followed by an APX-hardness proof of 2-MPSM and a general 4-approximation algorithm provided by Boria et al. [2]. The approximation ratio was then improved to 3.5 [1] using a particularly clean argument based on local search. Finally, Brubach [3] obtained 3.25-approximation.

Our main contribution is a family of polynomial-time approximation algorithms for MPSM: for any $\varepsilon > 0$, we show a polynomial-time $(2+\varepsilon)$-approximation algorithm. We complement this with a specialised (and simplified) version of the algorithm that achieves 2.67-approximation in quadratic time, which already improves on the approximation guarantee and the running time of the previous solutions, as the running time of the 3.5-approximation was $O(n^4)$. At a high level, we also apply local search, that is, we iteratively try to slightly change the current solution as long as such a change leads to an improvement. The intuition is that not being able to find such local improvement should imply an $(2+\varepsilon)$-approximation guarantee. This requires considering larger and larger neighbourhoods of the current solution for smaller and smaller $\varepsilon$ and seems problematic already for $\varepsilon = 1$. To overcome this, we apply local search only after a certain preliminary greedy procedure, which gives us more structure and makes a more general local search possible.

2 Preliminaries

In the Maximum Duo-Preservation String Mapping (MPSM) we are given two strings $X$ and $Y$, where $Y$ is a permutation of $X$. The goal is to map letters of $X$ to letters of $Y$ as to maximise the number of preserved duos. A duo is a pair of consecutive letters, and a duo of $X$ is said to be preserved if its pair of consecutive letters is mapped to a pair of consecutive letters of $Y$ (in the same order). This can be restated by creating a bipartite graph $G = (A \cup B, E)$, where $n = |A| = |B|$ and $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_n\}$. Node $a_i$ corresponds to duo $(X[i], X[i+1])$ and similarly $b_i$ corresponds to $(Y[i], Y[i+1])$. Two nodes are connected with an edge if their corresponding duos are the same, that is, $E = \{(a_i, b_j) : X[i] = Y[j] \text{ and } X[i+1] = Y[j+1]\}$. See Figure 1.

![Figure 1: An optimal solution for strings $xyzabcb$ and $abbcxyz$, where the maximum number of preserved duos is 3. Note that duos $ab$ and $bc$ cannot be simultaneously matched.](image)

Now, we want to find a maximum matching in $G$ that corresponds to a proper mapping of letters between the strings, that is, such that every two consecutive mapped duos (consisting of three consecutive letters) are mapped to two consecutive duos (in the same order). It is not
necessary that all duos are mapped. Formally, a matching \( M \) is called consecutive if every two neighbouring nodes are either matched to two neighbouring nodes (preserving the order) or at least one of them is unmatched:

\[
\forall_{i,j,j' \in \{1..n\}} \left( \langle a_i, b_j \rangle \in M \land \langle a_{i+1}, b_{j'} \rangle \in M \right) \Rightarrow (j' = j + 1)
\]

and a symmetric condition for the other side of the graph. Even though the graph \( G \) obtained as described above from an instance of MPSM has some additional structure, we focus only on the general Maximum Consecutive Bipartite Matching Problem (MCBM), where the given bipartite graph \( G = (A \cup B, E) \) is arbitrary and we are looking for a consecutive matching of maximum cardinality.

**Definitions.** We say that two edges \( \langle a_i, b_j \rangle \) and \( \langle a_{i'}, b_{j'} \rangle \) are conflicting if \( |i - i'| \leq 1 \) or \( |j - j'| \leq 1 \). Given a consecutive matching \( M \), we define a streak to be a maximal (under inclusion) set of consecutive edges \( e_1, e_2, \ldots, e_k \), such that for some \( p, q \) we have that \( e_i = \langle a_{p+i}, b_{q+i} \rangle \) for all \( i = 1, 2, \ldots, k \). See Figure 2. Note that from the definition, \( e_i \) conflicts with itself, \( e_{i-1} \) and \( e_{i+1} \) (assuming that these edges exist). This notion is extended to sets of edges: \( S_1 \) conflicts with \( S_2 \) if there exist \( e_1 \in S_1, e_2 \in S_2 \) such that \( e_1 \) conflicts with \( e_2 \). Similarly, we define conflicts between an edge and a set of edges. Note that every consecutive matching \( M \) can be uniquely decomposed into a set of streaks such that no two of them are conflicting with each other.

**Figure 2:** Two pairs of conflicting edges (left) and decomposition of a consecutive matching into streaks (right).

### 3 Greedy algorithm

Consider a simple greedy procedure \( \text{Greedy}(k) \) that in every step takes the longest possible streak from \( G \) and, if the streak is big enough, adds it to the solution. By big enough we mean consisting of at least \( k \) edges, where \( k \) is a parameter given to the algorithm.

**Algorithm 1** Choosing the largest possible streak greedily.

1. **function** \( \text{Greedy}(k) \)
2. \( ALG := \emptyset \)
3. while true do
4. \( s := \) the largest streak in \( G \)
5. if \( |s| < k \) then
6. \( \text{break} \)
7. remove \( s \) and all edges conflicting with \( s \) from \( G \)
8. \( ALG := ALG \cup s \)
9. return \( ALG \)

Fix an optimal solution \( OPT \). We would like to compare \( |ALG| \) with \( |OPT| \). Let \( s_i \) be the streak that was removed in \( i \)-th step of the algorithm and \( o_i \) be set of edges from \( OPT \) that are conflicting with \( s_i \), but were not conflicting with \( s_1, s_2, \ldots, s_{i-1} \). In other words, \( o_i \) consists of...
those edges from $OPT$ that after $i - 1$ steps of the algorithm still could have been added to the solution, but are no longer available after the $i$-th step. Note that $o_i$ contains all the edges of $OPT \cap s_i$, because every edge conflicts with itself. Observe that $|o_i| \leq 2|s_i| + 4$ as there can be at most $|s_i| + 2$ edges from $o_i$ conflicting with $s_i$ at each side of $G$. Moreover, even a stronger property holds:

**Lemma 3.1.** $|o_i| \leq 2|s_i| + 2$.

**Proof.** Suppose that the endpoints of $s_i$ at one side of the graph (say $A$) form a sequence of nodes $a_i, a_{i+1}, \ldots, a_{i+|s_i| - 1}$. Define $E = \{a_{i-1}, a_i, \ldots, a_{i+|s_i| - 1}, a_{i+|s_i|}\}$ (assuming that $a_{i-1}$ and $a_{i+|s_i|}$ exist). We will show that at most $|s_i| + 1$ edges from $o_i$ can end in $E$. Then, applying the same reasoning to the other side of the graph finishes the proof. If $|E| < |s_i| + 2$ then the claim holds. Otherwise, if $|E| = |s_i| + 2$ there are three cases to consider:

1. There are two or more streaks from $o_i$ ending in $E$. Then they cannot end in all nodes from $E$, because at least two of them would be conflicting with each other. Thus there is at least one node from $E$ that is not an endpoint of edge from $o_i$, so there are at most $|s_i| + 1$ of them.

2. There is one streak from $o_i$ ending in $E$. Then the streak cannot be larger than $|s_i|$, because then the greedy algorithm would have taken the larger streak (recall that $o_i$ consists of edges that could have been added to the solution in $i$-th step). Thus there are at most $|s_i|$ edges of $o_i$ ending in $E$.

3. There is no streak from $o_i$ ending in $E$. Then the statement holds trivially. 

We still need to specify the algorithm for smaller streaks, but before doing so in the next section we focus on bounding the quality of the solution found by the greedy algorithm.

Let $m$ be the number of steps performed by the greedy algorithm. The algorithm returns $ALG = \bigcup_{i=1}^m o_i$ which should be compared with the set of edges of $OPT$ that can no longer be taken due to the decisions made by the greedy algorithm, that is, $\bigcup_{i=1}^m o_i \subseteq OPT$. Using Lemma 3.1 we can compute the desired ratio as follows:

$$\frac{|\bigcup_{i=1}^m o_i|}{|\bigcup_{i=1}^m s_i|} = \frac{\sum_{i=1}^m |o_i|}{\sum_{i=1}^m |s_i|} \leq \frac{\sum_{i=1}^m (2|s_i| + 2)}{\sum_{i=1}^m |s_i|} = 2 + \frac{m \cdot 2}{\sum_{i=1}^m |s_i|} \leq 2 + \frac{2}{k}$$

where the last inequality holds because all taken streaks consist of at least $k$ edges.

To conclude, the solution $ALG$ found by the greedy algorithm is at most $2 + \frac{2}{k}$ times smaller than the set of edges from $OPT$ that is conflicting with $ALG$. In other words, on average we discarded small amount of $OPT$ edges for every edge from $ALG$. After running that algorithm for $k = 1$, there will be no edges left and that gives us a 4-approximation algorithm. To obtain a better approximation ratio, we will increase $k$ and focus on the subgraph $G'$ of $G$ consisting of edges that are not conflicting with any streak $s_i$ already taken by the algorithm and hence are still available. The crucial insight is that we can analyse the performance of the greedy algorithm on $G' \setminus G'$ and the performance of the small $k$ algorithm on $G'$ separately. We know that the approximation ratio of the greedy algorithm on $G' \setminus G'$ is $2 + \frac{2}{k}$ and size of the optimal solution in $G'$ is at least $|OPT - \bigcup_{i=1}^m o_i|$. Then, due to the definition of $G'$, any solution found for $G'$ can be combined with $ALG$ to obtain a solution for the original instance, so the final approximation ratio is the maximum of $2 + \frac{2}{k}$ and the ratio of the algorithm used for $G'$.
4 Algorithm for small $k$

As stated above, using only the greedy algorithm for $k = 1$ we obtain a 4-approximation algorithm. For larger values of $k$ we need another phase that will find a solution for the remaining edges. For $k = 2$, we present a simple algorithm based on maximum bipartite matching (not consecutive) that can be used to obtain 3-approximation. For larger values of $k$, we apply local optimisations to first design a quadratic-time algorithm for $k = 3$ and then a polynomial-time algorithm for general $k$ presented in the next section.

4.1 3-approximation based on maximum matching for $k = 2$

After running $\text{Greedy}(2)$ there are no streaks of size 2. Recall that $G' = (A \cup B, E')$ is the subgraph of the original graph $G$ consisting of edges that are not conflicting with the already taken edges. Consider the following algorithm:

1. Create a bipartite graph $H = (A' \cup B', F)$ where
   - $A' = \{a_{(1,2)}, a_{(3,4)}, \ldots, a_{(n-1,n)}\}$ and similarly for $B'$.
   - In other words, nodes of $A'$ correspond to merged pairs of neighbouring nodes of $A$ (if $n$ is odd, the last node of $A'$ corresponds to a single node of $A$).
   - $F = \{\{a_{i,i+1}, b_{j,j+1}\} : \{a_i, a_{i+1}\} \times \{b_j, b_{j+1}\} \cap E' \neq \emptyset\}$. In other words, there is an edge between nodes corresponding to two merged pairs of nodes if there was an edge between a node from the first pair and a node from the second pair.
2. Find the maximum matching $M'$ in $H$.
3. For every edge of $M'$, choose an edge of $G'$ connecting nodes from the corresponding pairs (if there are multiple possibilities, choose any of them). Let $M$ be the set of chosen edges.
4. Let $ALG \leftarrow \emptyset$. Process all edges of $M$ in arbitrary order. For an edge $(a_i, b_j) \in M$:
   - remove from $M$ all edges ending in nodes $a_{i-1}, a_{i+1}, b_{j-1}$ and $b_{j+1}$,
   - add $(a_i, b_j)$ to $ALG$.
5. Return $ALG$.

Consider the optimal solution $OPT$. As $G'$ contains no streaks consisting of 2 or more edges, the endpoints of any two of its edges cannot be neighbours. Therefore, $OPT$ can be translated into a matching in $H$ with the same cardinality, so $|OPT| \leq |M'|$.

We claim that after including an edge $(a_i, b_j) \in M$ in $ALG$ at most 2 other edges are removed from $M$. Assume otherwise, that is there are 3 such edges. Without losing generality, one of them ends in $a_{i-1}$ and one in $a_{i+1}$. Depending on the parity of $i$, edge $(a_i, b_j)$ and the edge ending in either $a_{i-1}$ or $a_{i+1}$ correspond in $H$ to edges ending in the same node of $A'$. This is a contradiction, because all edges of $M'$ have distinct endpoints. Because initially $|M'| = |M|$, we conclude that $|ALG| \geq |M'|/3$.

Combining the inequalities gives us $3 \cdot |ALG| \geq |M'| \geq |OPT|$, so the algorithm for graphs with no streaks of size 2 or more is a 3-approximation. Combining it with $\text{Greedy}(2)$ that guarantees approximation ratio of $2 + \frac{2}{k} = 2 + \frac{2}{2} = 3$ gives us a 3-approximation algorithm for the whole problem.
4.2 2.67-approximation for $k = 3$

For $k = 3$ we use function LOCALIMPROVEMENTS based on local optimisations. See Algorithm 2. Essentially the same method was used to obtain the 3.5-approximation [1]. The algorithm consists of a number of steps in which it tries to either add a single edge or remove one edge so that two other edges can be added. However, the crucial difference is that in our case there are no streaks of size greater than 2 in $G'$. This allows for a better bound on the approximation ratio.

Algorithm 2 Local improvements in $O(m^2n^2)$ time.

1: function LOCALIMPROVEMENTS
2: $ALG := \emptyset$
3: while true do
4:   if $\exists e \notin ALG$ s.t. $ALG \cup \{e\}$ is a valid solution then
5:     $ALG := ALG \cup \{e\}$
6:   if $\exists e_1, e_2 \notin ALG, e' \in ALG$ s.t. $ALG \setminus \{e'\} \cup \{e_1, e_2\}$ is a valid solution then
7:     $ALG := ALG \setminus \{e'\} \cup \{e_1, e_2\}$
8:   if $|ALG|$ was not improved then
9:     break
10: return $ALG$

Fix an optimal solution $OPT$. We want to bound the total number $C$ of conflicts between edges from $ALG$ and $OPT$. First, observe that an edge from $ALG$ can conflict with at most 4 edges from $OPT$, because there are no streaks of size 3 in the graph. Thus:

$$4 \cdot |ALG| \geq C. \quad (1)$$

Second, let $k_1$ be the number of edges from $OPT$ that conflict with exactly one edge from $ALG$. Then all other edges from $OPT$ conflict with at least two edges from $ALG$ (because the algorithm would have taken an edge not conflicting with any already taken edge), so:

$$C \geq k_1 + 2 \cdot (|OPT| - k_1) = 2 \cdot |OPT| - k_1. \quad (2)$$

Lemma 4.1. $k_1 \leq |ALG|$.

Proof. Suppose that $k_1 > |ALG|$. Then there are two edges $e_1, e_2 \in OPT$ that are conflicting with only one and the very same edge $e_{del} \in ALG$. But then the algorithm would have increased size of the solution by removing $e_{del}$ and adding $e_1$ and $e_2$ to the solution, so we obtain a contradiction. \qed

Applying Lemma 4.1 to (2) and combining with (1) we get $4 \cdot |ALG| \geq C \geq 2 \cdot |OPT| - |ALG|$ and thus $2.5 \cdot |ALG| \geq |OPT|$. Recall that the approximation ratio of the first greedy part of the algorithm is $2 + \frac{2}{3} < 2.67$, so the overall ratio of the combined algorithm is also 2.67. The algorithm clearly runs in polynomial time as in every iteration of the main loop the size of $ALG$ increases by one and is bounded by $n$. In [1] the running time was further optimised to $O(n^4)$. In the remaining part of this section we will describe how to decrease the time to $O(n^2)$. We will also show how to implement the greedy algorithm in the same $O(n^2)$ complexity, thus obtaining an 2.67-approximation algorithm in $O(n^2)$ time.
Greedy part in $O(n^2)$ time. We show how to implement Greedy($k$) in $O(n^2)$ time. Recall that in every iteration it greedily takes the longest available streak and removes all edges conflicting with it. The algorithm terminates if the streak contains less than $k$ edges.

We start with creating a list $L$ of edges $(x, y)$ sorted lexicographically first by $x$ and then by $y$. This can be done in $O(n^2)$ time using bucket sort and while sorting we can also retrieve for every edge the edge that would be its predecessor in a streak. Then we iterate over the edges in $L$ and split them into streaks. The edges of every streak are stored in a doubly-linked list and every edge stores a pointer to its streak. Also we group streaks by their size, so that streak of size $s$ is stored in the $s$-th list $D[s]$. This invariant will be maintained during the whole execution of the algorithm.

Now we iterate over the lists $D[n], D[n - 1], \ldots, D[k]$ and retrieve a streak $s$ from the non-empty list with the largest index. We add $s$ to the solution and remove all edges conflicting with $s$ from the graph. Every removed edge either decreases the size of its streak by one or splits it into two smaller streaks. In both cases, the smaller streak(s) is moved to the appropriate list. Removing an edge takes constant time and every edge is removed at most once from the graph. Similarly, moving or splitting of a streak due to a removed edge takes constant time as the size of a smaller streak can be computed in constant time by looking at its first and last edge. Thus, the overall time of the procedure is $O(n^2)$.

Local improvements in $O(n^2)$ time. Recall that to analyse the approximation ratio (in Lemma 4.1), we only need that after termination of the algorithm there are no three edges $e_1, e_2 \notin ALG, e_{del} \in ALG$ such that $ALG \setminus \{e_{del}\} \cup \{e_1, e_2\}$ is a valid solution. At a high level, FASTLOCALIMPROVEMENTS keeps track of edges that can potentially increase size of the solution in a queue $Q$. As long as $Q$ is not empty, we retrieve a candidate edge $e$ from $Q$. First, we verify that $e \notin ALG$ and $e$ conflicts with at most one edge from $ALG$. If $e$ can be added to $ALG$, we do so and continue after adding to $Q$ all edges conflicting with $e$. Otherwise, we check if some other edge $e'$ can be added while removing another edge $e_{del}$ at the same time using function TRYADDINGPAIRWITH($e$), and if so, we add to $Q$ all edges conflicting with one of the modified edges $(e, e'$ and $e_{del})$. See Algorithm 3 and Algorithm 4 below.

Algorithm 3 Local improvements in $O(n^2)$ time.

1: function FASTLOCALIMPROVEMENTS
2: $Q$.ENQUEUE($E$)
3: while $Q$ is not empty do
4: $e := Q$.DEQUEUE()
5: if $e \in ALG$ or $e$ conflicts with more than one edge from $ALG$ then
6: continue
7: if $ALG \cup \{e\}$ is a valid solution then
8: $ALG := ALG \cup \{e\}$
9: $Q$.ENQUEUE(Conf(e))
10: continue
11: TRYADDINGPAIRWITH($e$)

The algorithm uses the following data structures and functions:

- Close($e$) is a set of nodes of $G'$ at distance at most 1 from the endpoints of edge $e$. In other words, Close($e$) is the set of up to 6 nodes where edges conflicting with $e$ can end.

- Conf(e) is the set of edges conflicting with edge $e$. It is computed on the fly, by iterating through edges ending in $v \in$ Close($e$).
• For every node $v \in G'$, we keep a list of all edges from $E$ ending in $v$ and also edges of $ALG$ ending there.

• Queue $Q$ of candidate edges. To maintain $O(m)$ memory, for every edge in $E$ we remember if it is currently in $Q$ in order not to store any duplicates.

• For every node $v \in G'$ we keep a list $L[v]$ of edges from $E \setminus ALG$ that conflict with exactly one edge from $ALG$ and end in $v$. To keep those lists updated, every time an edge $e = \langle x, y \rangle$ is enqueued or added or removed from $ALG$, we count number of edges from $ALG$ it conflicts with. If there is only one of them, we make sure that $e$ is in $L[x]$ and $L[y]$, otherwise we remove $e$ from $L[x]$ and $L[y]$.

Algorithm 4 Adding a pair with edge $e$.

```plaintext
1: function TRYADDINGPAIRWith(e)
2: $e_{del} :=$ the only edge from ALG conflicting with $e$
3: for each $e'$ that can be neighbour to $e$ in a streak do
4: if $ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is a valid solution then
5: $ALG := ALG \setminus \{e_{del}\} \cup \{e, e'\}$
6: $Q$.ENQUEUE($\text{Confl}(e) \cup \text{Confl}(e') \cup \text{Confl}(e_{del})$)
7: return
8: for each node $v \in \text{Close}(e_{del}) \setminus \text{Close}(e)$ do
9: for each edge $e' \in L[v]$ do
10: if $ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is a valid solution then
11: $ALG := ALG \setminus \{e_{del}\} \cup \{e, e'\}$
12: $Q$.ENQUEUE($\text{Confl}(e) \cup \text{Confl}(e') \cup \text{Confl}(e_{del})$)
13: return
```

Clearly, after termination of the algorithm there is no triple of edges $e_1, e_2$ and $e_{del}$ that can be used to increase the solution, because every time an edge is added to or removed from the solution, all its conflicting edges are enqueued. It remains to prove that Algorithm 3 indeed runs in $O(n^2)$ time. First, observe that $|\text{Close}(e)| \leq 6$, so from the definition of conflicts between edges $|\text{Confl}(e)| \leq |\text{Close}(e)| \cdot n \in O(n)$ as there are at most $n$ edges ending in a node. So, every time the algorithm enqueues a set of edges, there are at most $O(n)$ of them. As this happens only after increasing the size of $ALG$, which can happen at most $n$ times, in total there are $O(n^2)$ enqueued edges. So it suffices to prove that every time an edge $e$ is dequeued, it takes $O(1)$ time to check if it can be used to increase the solution. Here we disregard the time for enqueuing edges due to increasing the size of $ALG$, as it adds up to $O(n^2)$ as mentioned before. Note that both counting number of edges conflicting with $e$ and finding the unique edge from $ALG$ conflicting with $e$ takes $O(1)$ time, as we just need to check edges from $ALG$ ending in $\text{Close}(e)$. Similarly, as $ALG$ is always a valid solution, each validity check takes $O(1)$ time, as we always try to modify a constant number of edges. By the same argument, loops in lines 3 and 8 take constant number of iterations, and also:

**Lemma 4.2.** There are $O(1)$ iterations of the loop in line 9 of TRYADDINGPAIRWith(e).

**Proof.** Consider an edge $e' \in L[v]$ such that $ALG' := ALG \setminus \{e_{del}\} \cup \{e, e'\}$ is not a valid solution. From the definition and the invariant of $L[]$, $e'$ was conflicting only with $e_{del} \in ALG$, so both $ALG \setminus \{e_{del}\} \cup \{e\}$ and $ALG \setminus \{e_{del}\} \cup \{e'\}$ are valid solutions. Thus, the only reason for $ALG'$ not being valid is that $e'$ conflicts with $e$. But $v$ is at distance 2 or more from the endpoint of $e$, so $e$ and $e'$ can be conflicting only at the other side of the graph. There are only
at most 3 possible endpoints of such $e'$ at the other side, so after checking 4 edges from $L[v]$ we will surely find one that can used to increase $|ALG|$.

**Greedy(3)** with **FastLocalImprovements** yield 2.67-approximation in $O(n^2)$ time.

## 5 (2 + $\varepsilon$)-approximation

Given $\varepsilon > 0$ we would like to create a polynomial time (2 + $\varepsilon$)-approximation algorithm. We set $k = \lceil \frac{2}{\varepsilon} \rceil$ and run the greedy algorithm to remove all streaks of size at least $k$ from the graph $G$. From now on we focus on the subgraph $G'$ remaining after the first greedy phase and let $OPT$ denote an optimal solution in $G'$.

Let $t = \lceil \frac{4}{\varepsilon} \rceil + 1$ and $ALG$ be the solution found by $BoundedSizeImprovements(t)$, see Algorithm 5. Similarly to the case $k = 3$, the algorithm tries to improve the current solution using local optimisations, however now the number of edges that we try to add or remove in every step is bounded by $t$ (which depends on $\varepsilon$). We want to prove that $(2 + \varepsilon) \cdot |ALG| \geq |OPT|$. To this end, we assign $(2 + \varepsilon)$ units of credit to every edge of $ALG$. Then the goal is to distribute the credits from the edges of $ALG$ to the edges of $OPT$, so that every edge of $OPT$ receives at least one credit. Alternatively, we can think of transferring credits to the streaks from $OPT$, so that a streak consisting of $s$ edges receives at least $s$ credits. This will clearly demonstrate the required inequality.

### Algorithm 5 Improvements of bounded size.

1: function $BoundedSizeImprovements(t)$
2: $ALG := \emptyset$
3: while true do
4:   for each $E'_{\text{remove}}, E'_{\text{add}} \subseteq E$ such that $|E'_{\text{remove}}| < |E'_{\text{add}}| \leq t$ do
5:     $ALG' := ALG \setminus E'_{\text{remove}} \cup E'_{\text{add}}$
6:     if $ALG'$ is a valid solution then
7:         $ALG := ALG'$
8:         break
9:   if $ALG$ was not improved then
10:      break
11: return $ALG$

#### Credit distribution scheme. Every edge from $ALG$ distributes $(1 + \frac{\varepsilon}{2})$ credits for each of its two endpoints independently. Consider an endpoint $v_i$ of an edge from $ALG$. Recall that nodes at one side of the graph $G$ are numbered $\ldots, v_{i-1}, v_i, v_{i+1}, \ldots$. If there is an edge $e \in OPT$ ending in $v_i$, then $e$ receives 1 credit. Now consider case when no edge of $OPT$ ends in $v_i$. If exactly one edge from $OPT$ ends in $v_{i+1}$ or $v_{i-1}$ then the credit is transferred to that edge. If there are no edges ending there then the credit is not transferred anywhere. Finally, if there is an edge $e \in OPT$ ending at $v_{i-1}$ and another edge $e' \in OPT$ ending at $v_{i+1}$, then for the time being neither $e$ nor $e'$ receives the credit. In such a situation we say that the node $v_i$ is between the streak containing $e$ and the streak containing $e'$, call the credit uncertain and defer deciding whether it should be transferred to $e$ or $e'$. Observe that the only one case when an edge $e \in ALG$ conflicting with a streak $s$ does not transfer the credit to $s$ is when the endpoint of $e$ is between two streaks $s$ and $s'$. See Figure 3. Note that two credits can be transferred from $e$ to $s$ if both endpoints of $e$ transfer its credits to $s$. The remaining $\frac{\varepsilon}{2}$ credits
are not transferred to any specific edge yet. We will aggregate and redistribute them using a more global argument, but first need some definitions.

![Figure 3: Dashed lines denote edges from ALG. According to the scheme, $e_1$ and $e_2$ transfer a credit to an edge from $s$, but $e_3$ does not because its endpoint is between $s$ and $s'$.](image)

**Gaps and balance.** Define the balance of a streak $s$ from OPT as the number of credits obtained in the described scheme (ignoring the uncertain credits) minus the number of edges in $s$. A gap is an edge of OPT without any credit. Then the balance of a streak $s$ is at least minus number of gaps in $s$. We prove that after running the greedy algorithm and Bounded-SizeImprovements($t$), even a stronger property holds:

**Lemma 5.1.** The balance of every streak is at least $-2$.

**Proof.** Consider a streak $s$. If there are less than 2 gaps in $s$ then the claim holds. Otherwise, let $g_1$ and $g_2$ be the first and the last gap in $s$, so that we can write $s = Ag_1Mg_2B$, see Figure 4. Note that the balance of both $A$ and $B$ is non-negative, as from the definition there are no gaps inside, so every edge receives at least one credit. However, there might be multiple gaps in $M$. Suppose that the balance of $M$ is negative. But the size of $M$ is smaller than $k < t$, so BoundedSizeImprovements($t$) would have replaced a subset of edges from ALG with $M$ to increase size of the solution. Therefore, the balance of $M$ is nonnegative. Finally, observe that the balance of $s$ is equal to the sum of balances of $A, M$ and $B$ minus 2 (for the gaps $g_1$ and $g_2$), so it is at least $-2$ in total.

The following corollary from the analysis in the lemma will be useful later:

**Corollary 5.2.** Every streak $s$ with balance $-2$ can be represented as $s = Ag_1Mg_2B$ where $g_1$ and $g_2$ are the first and last gap of $s$, respectively. The balance of $Ag_1$ and $g_2B$ is $-1$ while the balance of $M$ is 0.

![Figure 4: Let black dots denote endpoints of edges from ALG, then $g_1$ and $g_2$ is the first and the last gap, respectively.](image)

**Analysis of scheme.** We construct an auxiliary multi-graph $H$, where the vertices are streaks of OPT of balance at least $-1$. Streaks with balance $-2$ are split into two smaller streaks with balance $-1$ as explained in Corollary 5.2. We create an edge between two streaks in $H$ when they both conflict with the endpoint of an edge from ALG. In other words, when edge $e$ from ALG has an endpoint conflicting with two streaks there is edge in $H$ between the vertices corresponding to these streaks, see Figure 5. Observe that then there is no edge of OPT
ending in \( x \) and there can be at most two edges between any pair of streaks \( s_1 \) and \( s_2 \). Now we will show that for every connected component of \( H \) there are enough credits to distribute at least one credit to every edge from \( OPT \) in the component.

Consider one connected component \( C \) on \( w \) vertices. We want to prove that there are at least \( w \) credits transferred to all edges of \( C \) in total. From the construction we have that every vertex of \( C \) has balance at least \(-1\). Moreover, as the component is connected, there are at least \( w - 1 \) edges, each of which adds one uncertain credit. Thus the total balance of the whole component (including the uncertain credits) is at least \(-1\). Observe that the only case when the total balance of the component is \(-1\) is a tree (with exactly \( w - 1 \) edges) where every node has balance \(-1\). In all other cases the balance is non-negative already.

We denote by \( K_C \) the set of edges of \( OPT \) from all vertices of \( C \) (recall that they correspond to substreaks of \( OPT \)). We also define an auxiliary set \( M_C \) that consists of the middle parts \( M \) of the original streaks. More precisely, for every streak \( s \) of balance \(-2\), if it was a part of \( C \) (say the substreak \( Ag_1 \) or \( g_2B \) in \( s = Ag_1Mg_2B \) representation), we add to \( M_C \) all edges from \( M \). From Corollary 5.2, the balance of every such \( M \) is \( 0 \). Now consider the following set of edges \( X_C = K_C \cup M_C \). There are two cases to consider depending on how many credits were transferred to \( X_C \):

1. If there are at least \( c \geq \frac{4}{\varepsilon} \) credits transferred to the edges of \( X_C \) (each credit from an endpoint of an edge from \( ALG \)), then we can use half of the remaining \( \frac{\varepsilon}{2} \) credit of each endpoint and transfer it to the component. Note that for each credit from those \( c \) that were previously assigned to \( X_C \) there is one endpoint still having \( \frac{\varepsilon}{4} \) to spend on \( X_C \). We can only use half of \( \frac{\varepsilon}{4} \) because some edges (from the middle parts of original streaks) can belong to both \( X_C \) and \( X_C' \) for two different components \( C \) and \( C' \), see Figure 6, so to be able to add credits to both of them, we can transfer only \( \frac{\varepsilon}{8} \) credits. Thus, as there are at least \( \frac{4}{\varepsilon} \) credits assigned, in total there will be at least one full credit transferred from the quarters of \( \varepsilon \)-parts, which is enough to cover the deficit of the component.

2. In the second case, the edges from \( X_C \) received less than \( \frac{4}{\varepsilon} \) credits, so there are less than \( \frac{4}{\varepsilon} + 1 \) edges from \( OPT \) (recall that overall balance of the component is \(-1\)). Note that if we add all edges from \( X_C \) and remove all edges from \( ALG \) that transferred credits to the edges from \( X_C \), the size of the solution will increase as earlier the overall balance was negative. It will still be a valid solution, because we have removed all edges from \( ALG \) conflicting with the edges of \( X_C \). Also for the split streaks, we took edges up to (but not including) a gap which from the definition does not share an endpoint with an edge from \( ALG \). Furthermore, as the size of \( X_C \) is at most \( \frac{4}{\varepsilon} + 1 \leq t \), it would have been considered as the set \( E_{add} \) of edges to be checked by our algorithm. Thus, this case is impossible, as we would have been able to improve the current solution.

To conclude, every connected component containing \( w \) edges receives at least \( w \) credits, so \((2 + \varepsilon) \cdot |ALG| \geq |OPT|\). As the approximation ratio of the first greedy part is also \((2 + \varepsilon)\),
Figure 6: As there is an uncertain credit between streaks $x$ and $Ag_1$, there will be an edge between them in $H$, so they will be in a connected component $C$ of $H$. Similarly for $g_2B$ and $y$ in $C'$. Observe that the middle part $M$ of the split streak $s$ is accounted for in both $M_C$ and $M_{C'}$.

as explained before the overall algorithm is $(2 + \varepsilon)$-approximation for MCBM. It remains to analyse its time complexity. Let $m$ denote the number of edges of $G'$. There are at most $n$ steps of the algorithm, as in each of them size of the solution increases by at least one and is bounded by $n$. There are $\binom{m}{t} \in O(m^t)$ candidates for $E_{\text{add}}$ and $E_{\text{remove}}$ and we can check in $O(m)$ time if a given solution is valid. In total, as $t = \lceil \frac{2}{\varepsilon} \rceil + 1$ the total time complexity is $O(m^{2t+1}) = O(n^{4t+2}) = O(n^{16/\varepsilon+6}) = n^{O(1/\varepsilon)}$.

Theorem 5.3. Combining the greedy algorithm with local improvements yields an $(2 + \varepsilon)$-approximation for MCBM in $n^{O(1/\varepsilon)}$ time, for any $\varepsilon > 0$.

Corollary 5.4. There exists a $(2 + \varepsilon)$-approximation algorithm for MPSM running in $n^{O(1/\varepsilon)}$ time, for any $\varepsilon > 0$.

References

[1] Nicolas Boria, Gianpiero Cabodi, Paolo Camurati, Marco Palena, Paolo Pasini, and Stefano Quer. A 7/2-approximation algorithm for the maximum duo-preservation string mapping problem. In CPM, volume 54 of LIPIcs, pages 11:1–11:8. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[2] Nicolas Boria, Adam Kurpisz, Samuli Leppänen, and Monaldo Mastrolilli. Improved approximation for the maximum duo-preservation string mapping problem. In WABI, volume 8701 of Lecture Notes in Computer Science, pages 14–25. Springer, 2014.

[3] Brian Brubach. Further improvement in approximating the maximum duo-preservation string mapping problem. In WABI, volume 9838 of Lecture Notes in Computer Science, pages 52–64. Springer, 2016.

[4] Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In WABI, volume 8126 of Lecture Notes in Computer Science, pages 244–258. Springer, 2013.

[5] Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In SODA, pages 102–121. SIAM, 2014.

[6] Wenbin Chen, Zheng Zhang Chen, Nagiza F. Samatova, Lingxi Peng, Jianxiong Wang, and Maobin Tang. Solving the maximum duo-preservation string mapping problem with linear programming. Theoretical Computer Science, 530(Complete):1–11, 2014.
[7] Marek Chrobak, Petr Kolman, and Jiří Sgall. The greedy algorithm for the minimum common string partition problem. *ACM Trans. Algorithms*, 1(2):350–366, October 2005.

[8] Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.

[9] Peter Damaschke. Minimum common string partition parameterized. In *WABI*, volume 5251 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2008.

[10] Bin Fu, Haitao Jiang, Boting Yang, and Binhai Zhu. Exponential and polynomial time algorithms for the minimum common string partition problem. In *COCOA*, volume 6831 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2011.

[11] Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. In *Journal of Combinatorics*, pages 484–495. Springer, 2004.

[12] Dan He. A novel greedy algorithm for the minimum common string partition problem. In *ISBRA*, volume 4463 of *Lecture Notes in Computer Science*, pages 441–452. Springer, 2007.

[13] Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23(4):519–527, 2012.

[14] Haim Kaplan and Nira Shafrir. The greedy algorithm for edit distance with moves. *Information Processing Letters*, 97(1):23 – 27, 2006.

[15] Dana Shapira and James A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007.