Non-overlapping Common Substrings Allowing Mutations

H.L. Chan, T.W. Lam, W.K. Sung, Prudence Wong and S.M. Yiu

Abstract. This paper studies several combinatorial problems arising from finding the conserved genes of two genomes (i.e., the entire DNA of two species). The input is a collection of n maximal common substrings of the two genomes. The problem is to find, based on different criteria, a subset of such common substrings with maximum total length. The most basic criterion requires that the common substrings selected have the same ordering in the two genomes and they do not overlap among themselves in either genome. To capture mutations (transpositions and reversals) between the genomes, we do not insist the substrings selected to have the same ordering. Conceptually, we allow one ordering to go through some mutations to become the other ordering. If arbitrary mutations are allowed, the problem of finding a maximum-length, non-overlapping subset of substrings is found to be NP-hard. However, arbitrary mutations probably overmodel the problem and are likely to find more noise than conserved genes. We consider two criteria that attempt to model sparse and non-overlapping mutations. We show that both can be solved in polynomial time using dynamic programming.

Mathematics Subject Classification (2000). Primary 92-08; Secondary 68W05.

Keywords. Whole genome alignment, conserved genes, mutations, algorithms.

1. Introduction

In the past decade, the genomes of many species (including the human genome) have been decoded. One important work for the biologists is to uncover and align the conserved genes (i.e., genes sharing the same functions) of different species [1, 8, 9, 12, 15, 10]. Consider any two genomes, a conserved gene rarely comprises the same entire sequence in the two genomes, yet there are usually a lot of sort common substrings and many of these substrings are indeed unique to this conserved gene. Thus, the first step to align two genomes would be to identify maximal common substrings that appear uniquely in each genome. Of course, not every such

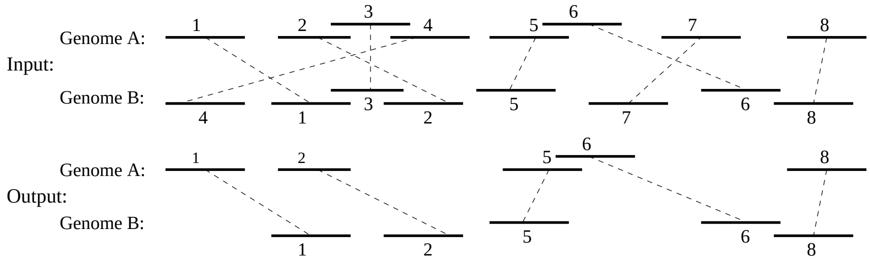


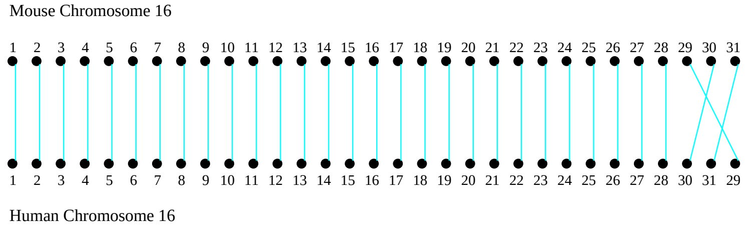
FIGURE 1. The LCS algorithm is used to identify pairs of matched substrings on two sequences. Such pairs are considered as the possible locations of conserved genes

pair of common substrings corresponds to a conserved gene; many common substrings actually originate from intergenic regions and some are overlapping. The key problem is how to extract the right common substrings. In this paper, we analyze the combinatorial problems arising from using different criteria to select the common substrings.

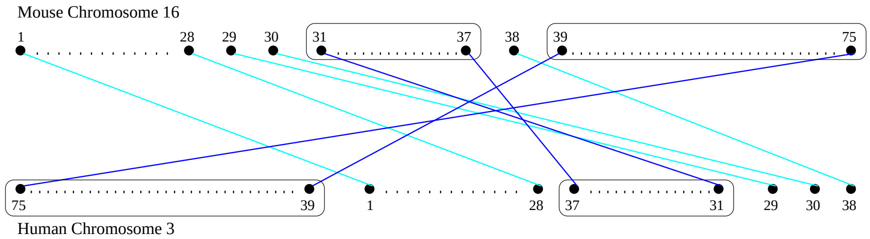
The simpliest way to select the common substrings is to find as many as possible the common substrings that have the same ordering in both genomes. This is based on the assumption that two related species should preserve the ordering of most conserved genes. In this case, one can make use of the longest common subsequence (LCS) algorithm (for unique characters). See Figure 1 for an example. MUMmer-1, a once widely used software for finding conserved genes, is based on such an approach [7, 13]. The LCS approach provides a good starting point for biologists, but experiments with real data reveal that many conserved genes are often missed.

From a technical viewpoint, a natural way to improve the LCS approach is to take into account the lengths of the common substrings and to select only common substrings that are disjoint in the two corresponding genomes. For example, in Figure 1, the common substring labeled with 7 should be reported instead of the one labeled with 6; in general, overlapped substrings are likely to be noise or give no additional information about the location of genes.

To further uncover more conserved genes, we realize that no matter how close two species are, some mutations [4] would have occurred during the evolution and changed the ordering of some genes. A typical mutation would move a block of contiguous genes from one place to another and these genes may also be reversed during the move. The LCS approach can be successful in aligning the genes only if very few mutations have occurred. Figure 2 depicts all the known conserved genes on the mouse chromosome 16 and the human chromosomes 16 and 3; note that the alignment in the first case is very close to a LCS, yet for the second case, the LCS approach at the best can uncover 31 gene pairs out of 75 conserved gene pairs. In this paper we study more general criteria in selecting the common substrings that can capture mutations.



(a) There are 31 conserved gene pairs found in Mouse chromosome 16 and Human chromosome 16. In both chromosomes, the genes are labeled according to their positions in the chromosomes from the 5' end.



(b) There are 75 conserved gene pairs found in Mouse chromosome 16 and Human chromosome 3. The genes are labeled as in (a). Note that Genes 31 to 37 in the mouse chromosome are reversed in the human chromosome. The same situation occurs in Genes 39 to 75.

FIGURE 2. Mapping of Conserved Genes between Mouse and Human Chromosomes

Problem definitions and summary of results: To ease our discussion, it is useful to denote the input common substrings as a set Σ of special symbols. Consider two genomes G_1 and G_2, which are strings over the alphabet $\{A, C, G, T\}$. Let Σ be a set of n special symbols such that each symbol c in Σ represents a maximal common substring of two genomes G_1 and G_2, which is also unique in G_1 and in G_2. Note that c is characterized by two intervals (enclosing the substring) in G_1 and G_2, and the weight of c is length of this interval. The intervals associated with two different symbols may overlap in a genome, but one cannot be contained by another (due to the maximal property).

Let $A[1..n]$ and $B[1..n]$ be two permutations of Σ. The order the symbols appear in A follows the order of the corresponding intervals appearing in G_1, and similarly for B and G_2. Consider any subset X of Σ. Let $A|_X$ denote the subsequence of A composing of symbols in X only. X is said to be non-overlapping if in G_1 the intervals associated with the symbols in X do not overlap, and the same condition also holds for X and G_2.

Problem 1. Given Σ, A and B, find a maximum weight subset X of Σ such that X is non-overlapping and $A|_X = B|_X$ (i.e., the order X appears in A is the same as the order X appears in B).

Problem 1 can be solved in $O(n^2)$ time using dynamic programming. We can improve the time complexity to $O(n \log n)$. One possible way is by extending a sparsification technique used in [2]; the key idea is to exploit an implicit representation of the intermediate values computed by the dynamic programming which then allows us to use a persistent data structure to efficiently update these values. The sparsification technique is also needed to solve other more complicated problem in this paper, thus we will present this $O(n \log n)$ time solution as a warm-up for other sections. Note that Problem 1 can be reduced to the problem of finding a subset of disjoint trapezoids of maximum weight in a trapezoid graph, and an $O(n \log n)$ time algorithm for solving the latter problem also exists [5].

As mentioned before, Problem 1 often fails to capture the pairing of all conserved genes (regions) between two genomes, as the ordering of genes might have changed due to mutations. A mutation can move a block of genes from one place to another (transposition) and/or reversing the order of a block of genes (reversal). To capture the effect of mutations, we modify Problem 1 as follows:

Problem 2*. Given Σ, A and B, find a maximum weight subset X of Σ such that X is non-overlapping and $A|_X$ can be transformed to $B|_X$ via some sequence of mutations.

Theoretically speaking, any subsequence of A can be rearranged into any order using some sequence of mutations. If we take that to extreme, we can ignore the ordering issue as there always exists some sequence of mutations to explain the difference in ordering. Thus, Problem 2* is equivalent to the following problem.

Problem 2. Given Σ, A and B, find a maximum weight subset X of Σ such that X is non-overlapping.

Problem 2 is found to be NP-hard. It can be considered as a special case of finding a subset of 2-intervals of maxium weight from a given set of 2-intervals [14]. The NP-hardness proof of the latter problem also applies to this special case. Approximation algorithms for solving the latter problem and its variations have been known [3]. Nevertheless, Problem 2 probably admits more noise than conserved genes in practice; solving Problem 2 exactly or approximately is unlikely to provide a satisfactory result for uncovering conserved genes. Technically speaking, this is because we assume the existence of arbitrarily frequent and complicated mutations to take care of the difference of the ordering. Yet in reality, mutations do not occur in such an arbitrary fashion. Very often, especially between related species (say, human and mouse), mutations are relatively sparse, and there are many blocks of genes whose order is preserved. In this paper, we propose two simple problems that model the case when mutations are sparse and do not overlap.

Recall that a mutation moves a block of characters $A[i..j]$ to another place, say, between $A[k]$ and $A[k+1]$ where $k \geq j$ or $k < i$, and the block may get reversed during the move. If $k \geq j$, then all characters at the positions from i to k are changed; if $k < i$, then all characters at the positions from $k+1$ to j are changed. The range of the mutation is defined to be $[i, k]$ and $[j, k]$, respectively. We say that a sequence of mutations is *simple* if for each mutation, the block to be moved is entirely outside the range of all previous mutations, and the destination is also not inside the range of a previous mutation. Intuitively, the same characters will not be involved in more than one mutations.

Problem 3. Given Σ, A and B, find a maximum weight subset X of Σ such that X is non-overlapping, and $A|_X$ can be transformed to $B|_X$ by a sequence of simple mutations.

Furthermore, a sequence of mutations is said to be *local* if the ranges of any two mutations are disjoint.

Problem 4. Given Σ, A and B, find a maximum weight subset X of Σ such that X is non-overlapping, and $A|_X$ can be transformed to $B|_X$ by a sequence of local mutations.

In Figure 2, the first case can be modeled by a single local mutation while the second case can be modeled by two simple mutations (first, reverse genes 31 to 37 and move them before gene 29; second, reverse genes 39 to 75 and move them before gene 1). In this paper, we show that Problem 3 can be solved in $O(n^6)$ time, and Problem 4 can be solved in $O(n^3 \log n)$ time.

Remark: A similar subsequence selection problem with mutations has been studied before [2]. However, the setting of the problems are not the same. In [2], they simplify the problem by allowing the symbols selected to associate with possibly overlapping intervals (substrings) in either genome. Since the common symbols selected are supposed to represent a set of conserved genes, allowing the intervals to overlap implies overlapping genes. Although overlapping genes do exist, for prokaryotes (such as bacteria), the overlapping regions are only a very small portion (for example, 0.5%) of the total genic regions and only exist at the ending regions of the genes [6]. Allowing intervals to overlap may not be appropriate in some applications. So, it is also important to study the non-overlapping version of the problem. On the other hand, [2] only proposed a polynomial-time approximation algorithm to tackle the problem.

Notations: For any $A[i]$, we denote the index of the symbol in B that matches $A[i]$ as p_i, i.e., $A[i] = B[p_i]$. Similarly, we denote that index of the symbol in A that matches $B[j]$ as q_j. Furthermore, we use $prev_A(i)$ (resp. $next_A(i)$) to denote the index of the last symbol before $A[i]$ (resp. the first symbol after $A[i]$) whose interval does not overlap with $A[i]$'s interval. Define $prev_B(i)$ and $next_B(i)$ for B in a similar way. For any string S, we define $\overline{S[i..j]}$ to be the reverse of $S[i..j]$.

2. Problem 1: a warm-up

In this section we show how to solve Problem 1 in $O(n \log n)$ time. Given two permutations A and B of n distinct symbols, Problem 1 asks for a maximum-weight, non-overlapping subset of symbols that have the same ordering in A and B. Denote the weight of such a subset as $\mathrm{NCS}(A, B)$. Below we give an algorithm for computing $\mathrm{NCS}(A, B)$ in $O(n \log n)$ time, which will also be used in subsequent sections.

Note that this algorithm is indeed a simple extension of a similar algorithm in [2], which solves the same problem but without the overlapping constraint. We provide the details below for the sake of completeness.

To ease our discussion, we further define $C_\ell[k]$ to be NCS$(A[1..\ell], B[1..k])$. Note that

$$C_\ell[k] = \max \begin{cases} C_{\ell-1}[k] \\ w(A[\ell]) + C_{prev_A(\ell)}[prev_B(P_\ell)] & \text{if } k \geq P_\ell \end{cases} \tag{1}$$

Note that we can compute in advance $prev_A(i), prev_B(i), next_A(i)$, and $next_B(i)$ for all $i = 1, 2, \ldots, n$, using $O(n \log n)$ time. By computing the function C_ℓ for $\ell = 1, 2, \ldots, n$ incrementally, we can eventually compute NCS(A, B), which equals $C_n[n]$. This would take $O(n^2)$ time.

We observe that the values in $C_\ell[1..n]$ are increasing. That is, $C_\ell[1] \leq C_\ell[2] \leq \ldots \leq C_\ell[n]$. Instead of storing the values in C_ℓ explicitly, we store only the boundaries at which the values change. Precisely, $C_\ell[1..n]$ can be represented by storing the pairs $(C_\ell[i], i)$ where $C_\ell[i] > C_\ell[i-1]$. To allow efficient retrieval, we store these pairs in a binary search tree, denoted T_ℓ.

Given $T_1, \ldots, T_{\ell-1}$, we can make use of Equation 1 to compute $C_\ell[P_\ell]$ in $O(\log n)$ time. Then we can build T_ℓ from $T_{\ell-1}$ as follows: If $C_\ell[P_\ell] \leq C_{\ell-1}[P_\ell]$, then it can be verified that the arrays C_ℓ and $C_{\ell-1}$ have exactly the same entries. In this case, T_ℓ is the same as $T_{\ell-1}$. The non-trivial case is when $C_\ell[P_\ell] > C_{\ell-1}[P_\ell]$. Among all integers bigger than P_ℓ, let k_o be the smallest one such that $C_\ell[P_\ell] < C_{\ell-1}[k_o]$. We have the following lemma about C_ℓ.

Lemma 1. *If $C_\ell[P_\ell] > C_{\ell-1}[P_\ell]$, then* (i) $C_\ell[k] = C_{\ell-1}[k]$ *for all $k < P_\ell$ and $k \geq k_o$; and* (ii) $C_\ell[k] = C_\ell[P_\ell]$ *for $P_\ell \leq k < k_o$*

Proof. If $k < P_\ell$, $A[\ell]$ will not be included in the common subsequence since $B[P_\ell]$ is not in $B[1..k]$, so $C_\ell[k] = C_{\ell-1}[k]$. For $k \geq k_o$, if $A[\ell]$ is included in the common subsequence, all $B[r]$ for $r > P_\ell$ will not be included in the subsequence as $A[\ell]$ (that is, $B[P_\ell]$) must be the last element in the subsequence. And the weight of such a common subsequence is at most $C_\ell[P_\ell]$. However, $C_\ell[P_\ell] < C_{\ell-1}[k_o]$, so any common subsequence containing $A[\ell]$ is not the subsequence with maximum weight. So, $C_\ell[k] = C_{\ell-1}[k]$. For $P_\ell \leq k < k_o$, since k_o is the smallest integer greater than P_ℓ such that $C_\ell[P_\ell] < C_{\ell-1}[k_o]$, so $C_{\ell-1}[k] \leq C_\ell[P_\ell]$. Therefore, $C_\ell[k] = C_\ell[P_\ell]$. □

By Lemma 1, we can construct T_ℓ from $T_{\ell-1}$ by deleting all pairs $(C_{\ell-1}[i], i)$ where $i \geq P_\ell$ and $C_{\ell-1}[i] < C_\ell[P_\ell]$, then followed by inserting the pair $(C_\ell[P_\ell], P_\ell)$. Denote α_ℓ as the number of pairs being deleted. Then the time for computing T_ℓ is $O((\alpha_\ell + 1) \log n)$.

Apparently, the above method implies that $T_{\ell-1}$ is erased once T_ℓ is obtained. Nonetheless, by exploiting a persistent data-structure [11], both T_ℓ and $T_{\ell-1}$ can coexist after the insert and delete operations, while retaining the same time complexity for constructing and accessing. In summary, the total time for constructing $T_1, \ldots, T_n$ is $O(\sum_{\ell=1}^{n}(\log n + \alpha_\ell \log n))$. As we insert at most n pairs into these trees, we can delete at most n pairs, and $\sum_{\ell=1}^{n} \alpha_\ell \leq n$. Hence, $T_1, \ldots, T_n$, as well as NCS(A, B), can all be computed in $O(n \log n)$ time. And for any i, j, NCS$(A[1..i], B[1..j]$ can be retrived in $O(\log n)$ time using T_i. Thus Lemma 2 follows. This result will be used in the subsequent sections.

Lemma 2. *In $O(n \log n)$-time, we can construct a data structure on A and B that allows us to retrieve* NCS$(A[1..i], B[1..j])$ *for any $i, j = 1, 2, \ldots, n$ in $O(\log n)$ time.*

3. Local mutations

Given two permutations A and B of n distinct symbols, we want to find a maximum-weight, non-overlapping subset S of symbols $A|_S$ can be transformed to $B|_S$ using a sequence of local mutations. Below such a subset S is referred to as NCS-LM(A, B).

In this section, we first show a simple dynamic programming to compute the total weight of NCS-LM(A, B) (and hence NCS-LM(A, B)) in $O(n^4)$ time, then we show how to speed up the computation to $O(n^3 \log n)$ time.

Define $LM[i, j]$ to be the weight of NCS-LM$(A[1..i], B[1..j])$. We can compute $LM[i, j]$ based on the recurrence equation in the following lemma.

Lemma 3. $LM[i, j] = \max\{LM[i-1, j], LM[i, j-1], \Phi\}$, *where $\Phi = -\infty$ if $P_i > j$ or $Q_j > i$, otherwise $\Phi =$*

$$\max \begin{cases} \Phi_1 = LM[prev_A(Q_j), prev_B(P_i)] + \text{NCS}(A[Q_j..i], \overline{B[P_i..j]}) \\ \Phi_2 = \max_{1 \leq r \leq Q_j; next_B(P_i) \leq P_r \leq j} \left\{ \begin{array}{l} LM[prev_A(r), prev_B(P_i)] + \\ \text{NCS}(A[r..Q_j], B[P_r..j]) + \\ \text{NCS}(A[next_A(Q_j)..i], \overline{B[P_i..prev_B(P_r)]}) \end{array} \right\} \\ \Phi_3 = \max_{1 \leq s \leq P_i; next_A(Q_j) \leq Q_s \leq i} \left\{ \begin{array}{l} LM[prev_A(Q_j), prev_B(s)] + \\ \text{NCS}(A[Q_s..i], B[s..P_i]) + \\ \text{NCS}(A[Q_j..prev_A(Q_s)], \overline{B[next_B(P_i)..j]}) \end{array} \right\} \\ \Phi_4 = \max_{\substack{1 \leq s \leq P_i; next_A(Q_j) \leq Q_s \leq i; \\ 1 \leq r \leq Q_j; next_B(P_i) \leq P_r \leq j}} \left\{ \begin{array}{l} LM[prev_A(r), prev_B(s)] + \\ \text{NCS}(A[Q_s..i], B[s..P_i]) + \\ \text{NCS}(A[r..Q_j], B[P_r..j]) \end{array} \right\} \end{cases}$$

Proof. Let $S =$ NCS-LM$(A[1..i], B[1..j])$. The lemma follows from the following case analysis.

1. $A[i]$ is not contained in S. So, $LM[i, j] = LM[i-1, j]$.
2. $A[Q_j]$ (that is, $B[j]$) is not contained in S. So, $LM[i, j] = LM[i, j-1]$.
3. Both $A[i]$ and $A[Q_j]$ are contained in S. Case (i) $A[i] = A[Q_j]$. Case (ii) They are involved in the same mutation. This mutation must involve a reversal. For cases (i) and (ii), $LM[i, j] = \Phi_1$. If $A[i]$ and $A[Q-j]$ are not involved in the same mutation, then only one of them can be involved in a mutation based on the definition of local mutations. Case (iii) $A[i]$ is involved in a mutation (this mutation must move the block somewhere before $A[Q_j]$, say

before $A[r]$). (a) If this mutation is also a reversal, then $LM[i,j] = \Phi_2$. (b) If this mutation does not reverse the block, then $LM[i,j] = \Phi_4$. Note that s marks the beginning of the block to be moved by the mutation. Case (iv) $A[Q_i]$ is involved in a mutation. (a) If this mutation is also a reversal, we consider all possible cases of forming the block to be moved by the mutation. $LM[i,j]$ can be computed by Φ_3. (b) If this mutation does not reverse the block, this case is the same as Case (iii)(b). So, $LM[i,j] = \Phi_4$.

□

In the following lemma, we further show that in the course of applying Lemma 3 to compute all $LM[i,j]$, all the NCS() values involved can be computed in advance using $O(n^3 \log n)$ time. Then, based on Lemma 3, each entry $LM[i,j]$ can be filled in $O(n^2)$ time, and hence $LM[n,n]$ can be computed in $O(n^4)$ time.

Lemma 4. *For all* $i, u, v = 1, 2, \ldots, n$, $\mathrm{NCS}(A[i..u], B[P_i..v])$, $\mathrm{NCS}(A[u..i], B[v..P_i])$, $\mathrm{NCS}(A[u..i], \overline{B[P_i..v]})$, *and* $\mathrm{NCS}(A[i..u], \overline{B[v..P_i]})$ *can be computed in* $O(n^3 \log n)$ *time.*

Proof. We only show how to compute $\mathrm{NCS}(A[i..u], B[P_i..v])$, for $i, u, v = 1, 2, \ldots, n$. For a particular $i \in \{1, 2, \ldots, n\}$, by Lemma 2, we can build the NCS data structure for $A[i..n]$ and $B[P_i..n]$ in $O(n \log n)$ time. So, for every u, v, $\mathrm{NCS}(A[i..u], B[P_i..v])$ can be retrieved in $O(\log n)$ time. That is, $\mathrm{NCS}(A[i..u], B[P_i..v])$ for a particular i and for all u, v can be computed in $O(n^2 \log n)$ time. In total, $\mathrm{NCS}(A[i..u], B[P_i..v])$, for $i, u, v = 1, 2, \ldots, n$ can be computed in $O(n^3 \log n)$ time. □

3.1. Speed up

Regarding the computation of $LM[i,j]$, we note that Φ_4 is the only value in Lemma 3 that requires $O(n^2)$ time; the others can be computed in $O(n)$ time. Improving the time for computing Φ_4 is not trivial; nevertheless, we found that we do not need to compute the exact value for Φ_4, but an upper bound of it (see the lemma below), which can be computed in a more efficient way.

Lemma 5. *Given* i *and* j*, we define* Ψ *as follows.*
$\Psi = \max_{1 \le s \le P_i; next_A(Q_j) \le Q_s \le i} \{\mathrm{NCS}(A[Q_s..i], B[s..P_i]) + K_{i,s}[prev_A(Q_s), j]\}$,
where

$$K_{i,s}[u,v] = \max \left\{ \begin{array}{l} LM[u, prev_B(s)] \\ \max_{1 \le r \le u; next_B(P_i) \le P_r \le v} \{LM[prev_A(r), prev_B(s)] + \\ \qquad\qquad\qquad\qquad \mathrm{NCS}(A[r..u], B[P_r..v])\} \end{array} \right\}.$$

Then, $\Phi_4 \le \Psi \le LM(i,j)$.

Proof. For any fixed s such that $Q_s \ge next_A(Q_j)$, $\mathrm{NCS}(A[r..prev_A(Q_s)], B[P_r..j])$ $> \mathrm{NCS}(A[r..Q_j], B[P_r..j])$ for any r as $prev_A(Q_s) \ge Q_j$. By definition of Φ_4 and $K_{i,s}$, $\Psi > \Phi_4$. On the other hand, Ψ is the weight of a valid mutated common subsequence between A and B using local mutations. So, $\Psi \le LM(i,j)$. □

Lemma 5 implies that if we redefine $\Phi = \max\{\Phi_1, \Phi_2, \Phi_3, \Psi\}$, Lemma 3 is still correct. Then we can fill in the table LM row by row for $i = 1, 2, \ldots, n$ incrementally. Below, we show that for any fixed row i, we can compute $K_{i,s}[prev_A(Q_s), j]$ for all s and for all j in $O(n^2 \log n)$ time. This implies that the row i of the table LM can be filled in using $O(n^2 \log n)$ time. $LM[n, n]$ can be computed by filling n rows of the table LM using $O(n^3 \log n)$ time.

It remains to show the details of the computation of $K_{i,s}$. $K_{i,s}$ satisfies a nice recursive formula as stated in the following lemma.

Lemma 6. $K_{i,s}[u, v] =$

$$\max \begin{cases} K_{i,s}[u-1, v] & \\ w(A[u]) + K_{i,s}[prev_A(u), prev_B(P_u)] & \text{if } v \geq P_u \geq next_B(P_i) \\ LM[u, prev_B(s)] & \text{if } P_u \leq prev_B(s), v \geq P_u \end{cases}$$

Proof. Let S be a non-overlapping subsequence of $A[1..u]$ and $B[1..v]$ which optimizes $K_{i,s}[u, v]$. There are three cases:

1. $A[u]$ is not contained in S. We have $K_{i,s}[u, v] = K_{i,s}[u-1, v]$.
2. $A[u]$ is in S and $v \geq P_u \geq next_B(P_i)$. Let S' be a subsequence of B induced by the symbols of S. Note that $A[u]$ and $B[P_u]$ should be the last symbols of S and S', respectively. The remaining of S and S' must be non-overlapping subsequences of $A[1..prev_A(u)]$ and $B[1..prev_B(P_u)]$, respectively. Thus, we have $K_{i,s}[u, v] = K_{i,s}[prev_A(u), prev_B(P_u)] + w(A[u])$.
3. $A[u]$ is in S and $P_u \leq prev_B(s)$. S is then a maximum weight NCS-LM($A[1..u]$, $B[1..prev_B(s)]$). Thus, we have $K_{i,s}[u, v] = LM[u, prev_B(s)]$.

□

We also observe that $K_{i,s}[u, v] \leq K_{i,s}[u, v+1]$ for $v = P_i, P_i + 1, \ldots, n$. Thus, $K_{i,s}[u, P_i..n]$ can be represented using the boundaries at which the values change. Then, using the technique similar to Lemma 2, we have the following lemma.

Lemma 7. *For a fixed i, s, a data-structure for $K_{i,s}$ can be constructed in $O(n \log n)$ time so that for any u, v, the value of $K_{i,s}[u, v]$ can be retrieved in $O(\log n)$ time.*

By the above lemma, for a fixed i and s, the values $K_{i,s}[prev_A(Q_s), j]$ for all j can be computed in $O(n \log n)$ time. Thus, for a fixed i, we can compute $K_{i,s}[prev_A(Q_s), j]$ for all s and for all j in $O(n^2 \log n)$ time.

4. Simple mutations

Given two permutations A and B of n distinct symbols, we want to find a maximum-weight, non-overlapping subset S of symbols such that $A|_S$ can be transformed to $B|_S$ using a sequence of simple mutations. Below such a subset S is referred to as NCS-SM(A, B). This section shows that an $O(n^6)$ algorithm to solve this problem.

Let $SM[s, t, u, v]$ be the weight of the NCS-SM($A[s..t], B[u..v]$). We have the following lemma.

Lemma 8. *For $s > t$ or $u > v$, $SM[s,t,u,v] = 0$. Otherwise, $SM[s,t,u,v] =$*

$$\max \begin{cases} 0 \\ \max\{SM[s,t-1,u,v], SM[s+1,t,u,v]\} \\ \max\{SM[s,t,u,v-1], SM[s,t,u+1,v]\} \\ SM[next_A(Q_v),t,u,prev_B(P_s)] + \mathrm{NCS}(A[s..Q_v],B[P_s..v]) \\ \qquad\qquad \text{if } s \le Q_v \le t \text{ and } u \le P_s \le v \\ SM[s,prev_A(Q_u),next_B(P_t),v] + \mathrm{NCS}(A[Q_u..t],B[u..P_t]) \\ \qquad\qquad \text{if } s \le Q_u \le t \text{ and } u \le P_t \le v \\ \max_{s<r\le t,u<b\le v}\{SM[s,prev_A(r),u,prev_B(b)] + SM[r,t,b,v]\} \end{cases}$$

Proof. Note that $SM[s,t,u,v] = 0$ if we cannot find any NCS-SM between $A[s..t]$ and $B[u..v]$. Otherwise, suppose that S is a NCS-SMbetween $A[s..t]$ and $B[u..v]$. There are five cases:

1. $A[s]$ or $A[t]$ is not contained in S. $SM[s,t,u,v] = \max\{SM[s,t-1,u,v], SM[s+1,t,u,v]\}$.
2. $A[Q_u]$ or $A[Q_v]$ is not contained in S. $SM[s,t,u,v] = \max\{SM[s,t,u,v-1], SM[s,t,u+1,v]\}$.
3. Both $A[s]$ and $A[Q_v]$ are involved in the same mutation. We can compute $SM[s,t,u,v]$ as $SM[next_A(Q_v),t,u,prev_B(P_s)] + \mathrm{NCS}(A[s..Q_v],B[P_s..v])$.
4. Both $A[Q_u]$ and $A[t]$ are in the same mutation. We can compute $SM[s,t,u,v]$ as

$$SM[s,prev_A(Q_u),next_B(P_t),v] + \mathrm{NCS}(A[Q_u..t],B[u..P_t]).$$

5. Otherwise, due to the fact that the mutations are nested, we can partition S into S_1 and S_2 and $(A[s..t],B[u..v])$ into two region pairs $(A[s..prev_A(r)], B[u..prev_B(b)])$ and $(A[r..t],B[b..v])$ so that S_1 and S_2 are a NCS-SM between $(A[s..prev_A(r)],B[u..prev_B(b)])$ and $(A[r..t],B[b..v])$, respectively. Thus, we have compute $SM[s,t,u,v]$ as

$$\max_{s<r\le t,u<b\le v}\{SM[s,prev_A(r),u,prev_B(b)] + SM[r,t,b,v]\}.$$

□

By Lemma 4, all NCS() values can be computed in $O(n^3 \log n)$ time. For the $O(n^4)$ entries in table SM, each entry can be filled in using $O(n^2)$ time. Hence, the NCS-SM between A and B can be computed in $O(n^6)$ time.

5. Conclusion

In this paper, we have studied several variations of the problem of selecting non-overlapping common substrings. From the application point of view, the new algorithms can improve tremendously the quality of the alignment than the LCS approach which does not consider mutations. From our empirical studies, we found that the simple mutation model is able to capture almost all known conserved genes between the human and mouse genomes. However, the running time is too slow when the input is large. We hope our work could trigger further studies on mutation models and more efficient algorithms.

Acknowledgement

This work was supported in part by Hong Kong RGC Grant HKU 7139/04E. We also thank the reviewers for the useful comments and pointing out the relationship between our work and [5, 14].

References

[1] David L. Baillie and Ann M. Rose. WABA success: A tool for sequence comparison between large genomes. *Genome Research*, 10(8):1071–1073, 2000.

[2] H.L. Chan, T.W. Lam, W.K. Sung, Prudence W.H. Wong, S.M. Yiu, and X. Fan. The mutated subsequence problem and locating conserved genes. *Bioinformatics*, 21(5):2271–2278, 2005.

[3] Maxime Crochemore, Danny Hermelin, Gad M. Landau, and Stéphane Vialette. Approximating the 2-interval pattern problem. In *Proceedings of 13th Annual European Symposium on Algorithms (ESA 2005)*, pages 426–437, 2005.

[4] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[5] S. Felsner, R. Müller, and L. Wernisch. Trapezoid graphs and generalizations, geometry, and algorithms. *Discrete Applied Mathematics*, 74(1):13–32, 1997.

[6] Yoko Fukuda, Takanori Washio, and Masaru Tomita. Comparative study of overlapping genes in the genomes of *mycoplasma genitalium* and *mycoplasma pneumoniae*. *Nucleic Acids Research*, 27(8):1847–1853, 1999.

[7] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[8] B. Morgenstern. Dialign 2: Improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15:211–218, 1999.

[9] B. Morgenstern, K. Frech, D. Dress, and T. Werner. Dialign: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14:290–294, 1998.

[10] E. Ohlebusch and M.I. Abouelhoda. Chaining algorithms and applications in comparative genomics. In Srinivas Aluru, editor, *Handbook of Computational Molecular Biology*, Computer and Information Science Series, chapter 15. Chapman & Hall/CRC, 2006.

[11] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[12] Scott Schwartz, Zheng Zhang, Kelly A. Frazer, Arian Smit, Cathy Riemer, John Bouck, Richard Gibbs, Ross Hardison, and Webb Miller. Pipmaker - a web server for aligning two genomic dna sequences. *Genome Research*, 10(4):577–586, 2000.

[13] T.G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[14] S. Vialette. On the computational complexity of 2-interval pattern matching. *Theoretical Computer Science*, 312(2-3):223–249, 2004.

[15] P. Vincens, L. Buffat, C. Andre, J.P. Chevrolat, J.F. Boisvieux, and S. Hazout. A strategy for finding regions of similarity in complete genome sequences. *Bioinformatics*, 14:715–725, 1998.

H.L. Chan
Department of Computer Science
University of Pittsburgh
Pittsburgh, USA
e-mail: `hlchan@cs.pitt.edu`

T.W. Lam
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
e-mail: `twlam@cs.hku.hk`

W.K. Sung
Department of Computer Science
National University of Singapore
Singapore
e-mail: `ksung@comp.nus.edu.sg`

Prudence Wong
Department of Computer Science
The University of Liverpool
Liverpool, UK
e-mail: `pwong@csc.liv.ac.uk`

S.M. Yiu
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
e-mail: `smyiu@cs.hku.hk`