CS5238 Combinatorial Methods in Bioinformatics 2004/2005 Semester 1

Lecture 4: Whole Genome Alignment - September 9, 2005

Lecturer: Wing-Kin Sung *Scribe: Tan Huiyi, Max*

4.1 Introduction

People believe that many widely divergent organisms are decended from a common ancestor through a process called evolution. Evolution is a process that results in inheritable changes in the properties of populations of organisms that spread over many generations. Since the changes are inheritable, scientists conclude that there must be some changes in the genome of these organisms that correspond to each evolutionary change. Knowing this fact, biologists start comparing the genomes of two organisms in order to understand the genetic relationship and evolutionary linkage between the two organisms.

It is found that two closely related species share a lot of gene pairs. One example of two closely related species is mouse and human. Please refer to Table 4.1 [10] for the number of gene pairs they share.

Mouse Chr No.	Human Chr No.	# of Published Gene Pairs	# of MUMs
2	15	51	96,473
7	19	192	52,394
14	3	23	58,708
14	8	38	38,818
15	12	80	88,305
15	22	72	71,613
16	16	31	66,536
16	21	64	51,009
16	22	30	61,200
17	6	150	94,095
17	16	46	29,001
17	19	30	56,536
18	5	64	131,850
19	9	22	62,296
19	11	93	29,814

Table 4.1: Mouse and Human Share a lot of Gene Pairs

4.1.1 Why whole genome alignment?

For the reason mentioned above, now our objective is to find a method that can help us to compare the whole genomes of two species and to extract all the conserved gene pairs they share. One possible way to accomplish the task is to perform *whole genome alignment*. Beside finding all the conserved gene pairs, whole genome alignment is also useful to help us performing the following functions:

- to find orthologous regions between two genomes,
- to do strain-to-strain or evolutionary comparison,
- to study genomic duplications,
- to analyze syntenic chromosomal regions.

As we shall see later, not all common subsequences of the two genomes found are published gene pairs. It opens an opportunity for biologists to study these excessive findings and possibly to discover new genes.

	Coverage	Preciseness
MUM	$\approx$ 100%	Many false positives
LCS (MUMmer1)	Very less	Not many false positives
Clustering (MUMmer3)	76.6%	26.5%
Mutation-Sensitive Alignment	91.3%	29.3%

Table 4.2: Coverage and Preciseness of MUM, MUMmer1, MUMmer3, and MSA

4.1.2 Why not use standard pairwise methods?

The standard sequence alignment methods are not used because of the following reasons:

1. Different features that we are looking at
   - Standard alignment: only point mutation, insertion and deletion can be observed.
   - Genome alignment: we can also learn about transposition, large insertion/deletion, syntenic blocks, etc.
2. Resource limitation
   - Given all the available resources today, it is infeasible to align whole genomes using standard methods. Their time and space complexity are too high, that it takes us too much time and space if we employ those methods.

4.2 The MUM System

4.2.1 Definition of MUM

Although a pair of conserved genes rarely contain the same entire sequence, they share a lot of short common substrings and some of them are indeed unique to this pair of genes. For example, consider the following two sequences, S and T:

$$S = \underline{\text{ac}}\ \text{ga}\ \underline{\text{ctc}}\ \text{a}\ \underline{\text{gctac}}\ \text{t}\ \underline{\text{ggtcagctatt}}\ \underline{\text{acttaccgc}}\ \#$$
$$T = \underline{\text{ac}}\ \text{tt}\ \underline{\text{ctc}}\ \text{t}\ \underline{\text{gctac}}\ \underline{\text{ggtcagctatt}}\ \text{c}\ \underline{\text{acttaccgc}}\ \$$$

It is clear that sequences S and T have many commmon substrings, they are ac, ctc, gctac, ggtcagctatt, and acttaccgc. Among those five common substrings, ac is the only substring that is not unique. It occurs more than once in both sequences. You can also observe that actually a, c, t, and g are common substrings of S and T. However, they are not maximal, i.e. they are contained in at least one longer common substrings. We are only interested in those that are of maximal length.

Our aim is to search for all these short common substrings. Given genomes A and B, we need to find all common substrings which are unique and of maximal length. Each of such common substrings is known as Maximum Unique Match (MUM). For almost every conserved gene pairs, there exist at least one MUMs which are unique to them.

So, it is clear that Maximal Unique Match (MUM) have the following two properties:

- It occurs exactly once in both genomes A and B (unique to both A and B)
- It is not contained in any longer MUMs (it is of maximal length)

Definition 4.1 *Maximal Unique Match(MUM) substring is a common substring of the two genomes that is longer than a specific minimum length d such that it is maximal, that is, it cannot be extended on either end without incurring a mismatch and it is unique in both sequences. (By default, d = 20.)*

For example, assuming d = 3, sequences S and T in the previous example has four MUMs: ctc, gctac, ggtcagctatt, acttaccgc. Substring ac is not an MUM because its length is smaller than the value of d and it is not unique to both sequences.

S = acga <u>ctc</u> a <u>gctac</u> t <u>ggtcagctatt</u> <u>acttaccgc</u> #
T = actt <u>ctc</u> t <u>gctac</u> <u>ggtcagctatt</u> c <u>acttaccgc</u> $

Consider another example, for S = acgat# and T = cgta$, assuming d = 1, there are two MUMs: cg and t. However, a is not an MUM because it occurs twice in S.

S = a <u>cg</u> a <u>t</u> #
T = <u>cg</u> <u>t</u> a$

The concept of MUM is important in whole genome alignment because a significantly long MUM is very likely to be part of the global alignment.

4.2.2 How to find MUMs

In this section, we present two methods to find MUMs of a pair of sequences.

Brute-force approach:

The idea of this method is to first find all common substrings of the two sequences, then for each substring, we check whether it is longer than d and unique in both sequences. The algorithm is as follows:

```
Input: Two genome sequences S[1..m1] and T[1..m2]
For every position i in S
    For every position j in T
        Find the longest common prefix P of S[i..m1] and T[j..m2]
        Check whether |P| ≥ d and whether P is unique in both genomes.
        If yes, report it as a MUM.
```

This solution requires at least O($m_1 m_2$)time. It is too slow!

Finding MUMs by suffix tree

The key idea in this method is to build a generalized suffix tree for genomes S and T. The algorithm consists of three steps, they are as follows:

1. Build a generalized suffix tree for S and T.
2. Mark all the internal nodes that have exactly two leaf children, which represent both suffixes of S and T.

3. For each marked node, suppose it represents the i-th suffix S_i of S and the j-th suffix T_j of T. We check whether S[i-1] = T[j-1]. If not, the path label of this marked node is an MUM.

In step 2, we only mark internal nodes that represent exactly one suffix of S and exactly one suffix of T in order to ensure that the path labels of those nodes are shared by both sequences and at the same time unique to both sequences.

In step 3, we check if S[i-1] = T[j-1] in order to make sure that the path labels of those nodes are of maximal length. If S[i-1] = T[j-1], it means that the substring is not maximal as we can extend it at least by one character by adding S[i-1] before its first character.

Example:

- Let S=acgat#, T=cgta$
- Assume d = 1
- Step 1: Build the generalized suffix tree for S and T. It is shown in Figure 4.1.

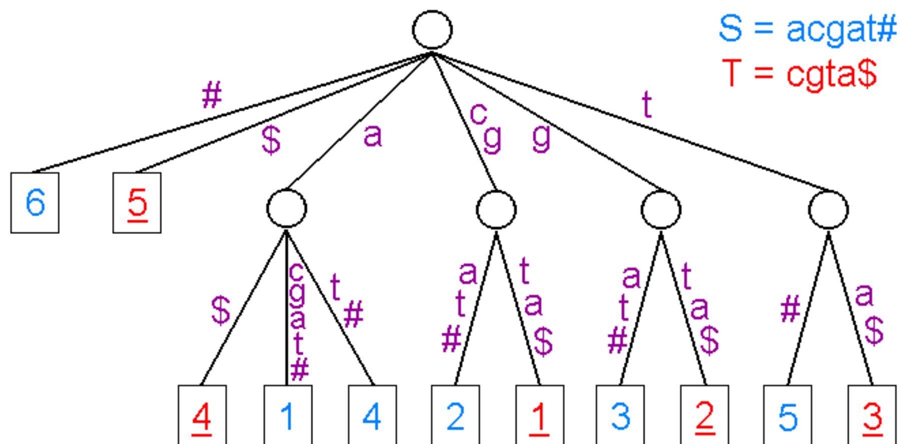


Figure 4.1: Step 1

- Step 2: Mark internal nodes with path labels cg, g, and t since they have exactly two leaf children, which represent both suffixes of S and T. Please refer to Figure 4.2.
- Step 3: It can be seen in Figure 4.3, the node with the path label g represents the 3rd suffix gat# of S and the 2nd suffix gta$ of T. Since both S[3-1] and T[2-1] are c, so the path label g of this marked node is not an

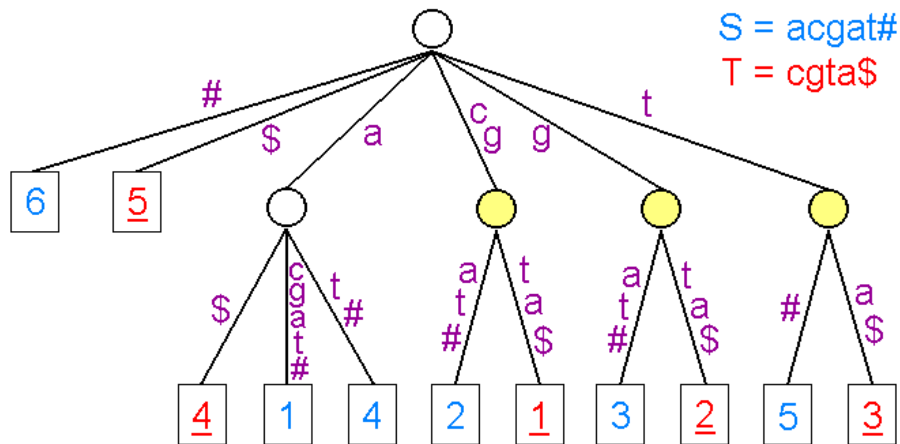


Figure 4.2: Step 2

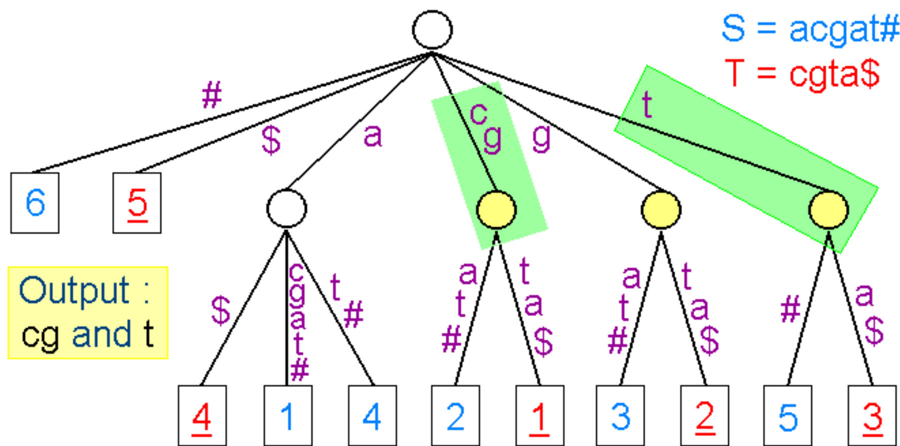


Figure 4.3: Step 3

MUM. So the output is cg and t.

Complexity Analysis

- Step 1: Building generalized suffix tree can be done in $O(m_1 + m_2)$ time using Weiner's, McGreight's, or Farach's algorithm.
- Step 2: Marking internal nodes takes $O(m_1+m_2)$ time. Since each internal node has at least 2 children, this generalized suffix tree has at most ($2m_1$-1)+($2m_2$-1) nodes. Knowing there are exactly m_1+m_2 leaf nodes, we are left with at most (m_1-1)+(m_2-1)=m_1+m_2-2 internal nodes to traverse.
- Step 3: Comparing $S[i-1]$ and $T[j-1]$ for each marked nodes takes $O(m_1+m_2)$ time as the number of marked nodes is at most m_1+m_2-2. By

the same reasoning, traversing all internal nodes to extracting MUMs also takes $O(m_1 + m_2)$ time.

- In total, this algorithm takes $O(m_1 + m_2)$ time to find all MUMs of the input sequences.
- The space complexity of this method is $O((m_1 + m_2) \log (m_1 + m_2))$ bits as we need to store the generalized suffix tree of the input sequences.

Based on some experiments, it is found that MUMs can cover 100% of the known conserved gene pairs. Moreover, finding all MUMs can be done in linear time. However, it does not mean that the problem is solved. From Table 4.1, we can see that the number of MUMs is much larger than the number of gene pairs. Due to noise, the amount of irrelevant information much larger compared to that of relevant information. We shall attempt to select the right MUMs in the sections described below.

4.3 MUMmer1 : LCS

4.3.1 Introduction

Definition 4.2 *Let $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$ be the order of the n MUMs in two genome sequences S and T, respectively. A sequence $C = (c_1, c_2, \cdots, c_m)$ is a* ***subsequence*** *of A if there exists indices $(i_1, i_2, \cdots, i_m)$ such that $i_1 < i_2 < \cdots < i_m$ and $c_j = a_{i_j}$ for all j. C is a* ***common subsequence*** *of S and T if C is a subsequence of both A and B.*

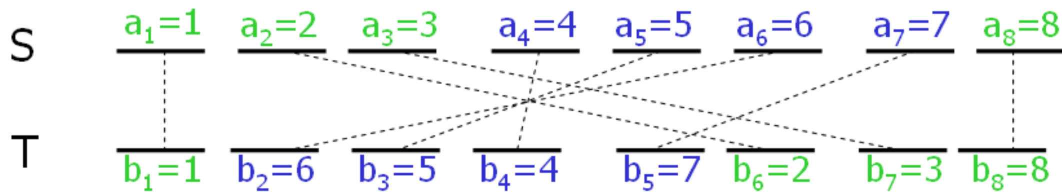
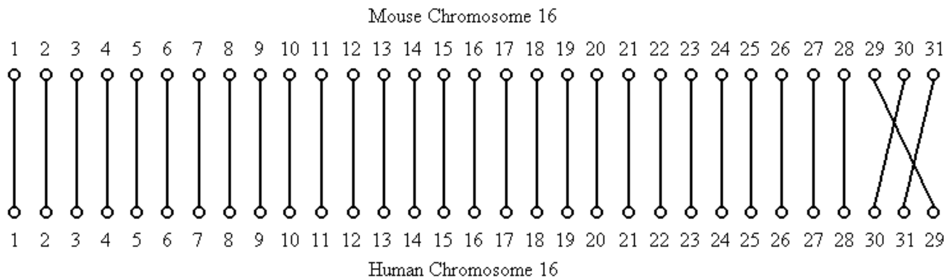


Figure 4.4: Example of Common Subsequence

For example, (1,2,3,8) is a common subsequence of S and T in Figure 4.4.

It is found that two closely related species should preserve the ordering of most conserved genes. For example, if we compare mouse chromosome 16 and human chromosome 16, we shall see that the ordering of 30 conserved gene pairs (out of 31 gene pairs) is preserved. Please refer to Figure 4.5 for visualization of this



(a) There are 31 conserved gene pairs found in Mouse chromosome 16 and Human chromosome 16. The genes are labeled according to the positions of genes in the mouse chromosome from the 5' end. The corresponding genes of human are drawn according to their relative positions from the 5' end of human chromosome.

Figure 4.5: Conserved genes in Mouse Chromosome 16 and Human Chromosome 16

example.

Definition 4.3 *Given two sequence S and T, we say that a sequence C is a Longest Common Subsequence (LCS) of S and T if C is a longest possible subsequence of both S and T. Note that the Longest Common Substring is contiguous while the Longest Common Subsequence need not be.*

As mentioned earlier, the idea of applying the concept of longest common subsequence(LCS) in algorithm MUMmer1 [3] arose from a key observation that the ordering of most of the conserved genes would be preserved in two related species. In 1999, Delcher et al [3], invented the MUMmer1 while working on Mycoplasma tuberculosis strains.

Hence, in MUMmer1, we compute the longest common subsequence of MUMs contained in both sequences and report only the MUMs in the LCS.

Compared to the brute force approach mentioned in the earlier section, MUMmer1 can sucessfully reduce the number of false positives reported at the cost of reducing the coverage.

4.3.2 Main Idea

As the output of the brute force approach contains too many irrelevant subsequences, MUMmer1 was constructed to decrease the number of false positives by outputting only MUMs contained in the LCS of both sequences. In additional to obtaining the MUMs of the two subsequence, we will now be required to determine the LCS of these MUMs, and then to output the MUMs in the LCS.

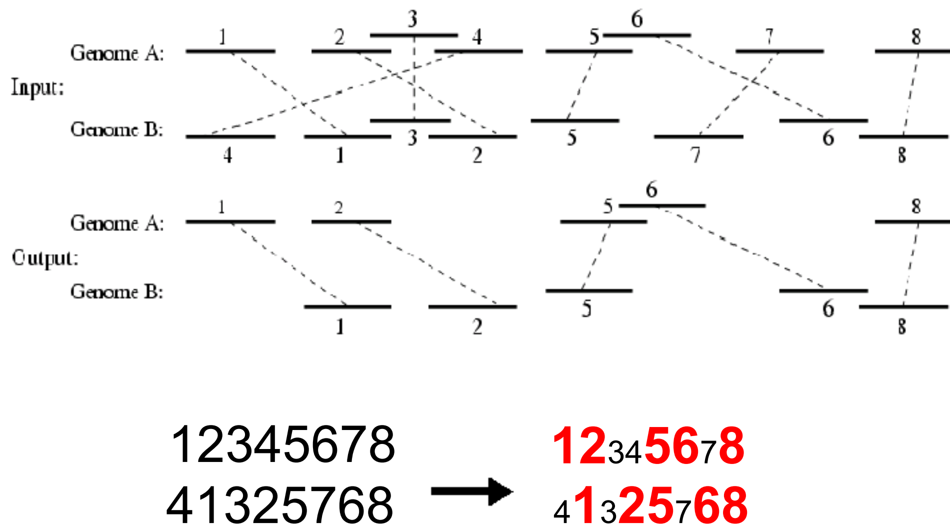


Figure 4.6: Example of LCS

4.3.3 Solution of LCS Problem

The naive way to solve large common subsequence problem is to use dynamic programming. Suppose there are K MUMs found, a dynamic programming algorithm requires $O(K^2)$ time and space to solve the problem. This approach quickly becomes infeasible as the size of the sequences increases.

Later, another method was derived based on a fact that all MUMs are distinct(by its definition). As the consequence of this fact, we can label each MUM by different character and we can limit the domain of the LCS problem to a set of sequences consisting of distinct characters, i.e. every single character occurs exactly once in the sequence. Using this approach,the LCS problem can be solved in $O(K \log K)$ time.

4.3.3.1 Dynamic Programming Algorithm in $O(K^2)$

The crux of this algorithm lies in the fact that the LCS of $S[1..i]$ and $T[1..j]$ is related to the LCS of $S[1..(i-1)]$ and $T[1..(j-1)]$ by comparing the last characters of subsequence S and T.

Let us first define $C_i[j]$ as the length of the longest common subsequence of $S[1..i]$ and $T[1..j]$. Also define $\delta(i)$ as the index of the character in T such that $S[i] = T[\delta(i)]$.

Recall that $C_i[j]$ is the length of the longest common subsequence of $S[1..i]$ and

$T[1..j]$. There are two possibilies for the character i of S ($S[i]$) and the character j of T ($T[j]$):

1. $S[i] \neq T[j]$, in which case the LCS is the same as the LCS of $S[1..(i-1)]$, $T[1..j]$
2. $S[i] = T[j]$, in which case we recursively obtain the LCS of $S[1..(i-1)]$ and $T[1..(j-1)] + 1$, and select the larger of either 2.

With those observations, we can obtain:

$$C_i[j] = max \begin{cases} C_{i-1}[j], & \text{others} \\ 1 + C_{i-1}[\delta(i) - 1], & \text{if } j \leq \delta(i) \end{cases}$$

Hence from the above table filling algorithm, we can obtain the LCS of string S and T from $C_n[n]$. This is solved in $O(K^2)$ time.

4.3.3.2 Sparsification

By observation, we can see that -

$$C_i[1] \leq C_i[2] \leq C_i[3] \leq ... \leq C_i[n]$$

Instead of storing $C_i[0..n]$ explictly, we can store only the boundaries at which the values change. We will store (j, $C_i[j]$) for all j such that $C_i[j] \geq C_i[j-1]$. By storing these tuples in a binary search tree $\mathcal{T}_i$, every search, insert, delete operation takes O(log k) time.

4.3.3.3 Lemma And Proof

Lemma 4.4 *Note that* $C_i[\delta(i)] = C_{i-1}[\delta(i)]$ *or* $C_i[\delta(i)] > C_{i-1}[\delta(i)]$.
If $C_i[\delta(i)] = C_{i-1}[\delta(i)]$ *then* $C_i[1..n] = C_{i-1}[1..n]$
If $C_i[\delta(i)] > C_{i-1}[\delta(i)]$ *then* $C_i[1..\delta(i)] = C_{i-1}[1..(\delta(i) - 1)]$, $C_i[delta(i)..(j^1 - 1)] = C_{i-1}[\delta(i)] + 1$ *And* $C_i[j^1..n)] = C_{i-1}[j^1..n]$

When $C_i[\delta(i)] = C_{i-1}[\delta(i)]$, this means that for two strings S and T (Recall $S[i] = T[\delta(i)]$), the LCS of $S[1..i]$ and $T[1..\delta(i)]$ is not dependent on the last matching character, and hence will be the same as the LCS of $S[1..(i-1)]$ and $T[1..\delta(i)]$. Similarly, by defining j^1 to be the smallest integer greater then $\delta(i)$ such that $C_i[\delta(i)] < C_{i-1}[j^1]$, when $C_i[\delta(i)] > C_{i-1}[\delta(i)]$, this will infer that the

LCS of $S[1..i]$ and $T[1..\delta(i)]$ is dependent on the last character of S and T. Hence $C_i[\delta(i)..(j^1-1)] = C_{i-1}[\delta(i)-1]+1$ (By definition of $C_i[j]$), $C_i[1..(\delta(i)-1)] = C_{i-1}[1..(\delta(i)-1)]$ (as the characters before i are compared in the same way) and $C_i[(j^1-1)..n] = C_{i-1}[(j^1-1)..n]$.

4.3.3.4 Reduction to O(K log K) time

From the binary search tree optimisation and the lemma above, we can compute $C_i[\delta(i)]$ in O(log K) time given C_{i-1}. We can construct the binary search trees by the following:

1. If $C_i[\delta(i)] = C_{i-1}[\delta(i)]$, $\mathcal{T}_i = \mathcal{T}_{i-1}$
2. If $C_i[\delta(i)] \leq C_{i-1}[\delta(i)]$, We first set $\mathcal{T}_i = \mathcal{T}_{i-1}$, then delete all tuples (j, $C_{i-1}[j]$) where $j \leq \delta(i)$ and $C_{i-1}[j] \leq C_i[\delta(i)]$; Finally we insert $(\delta(i), C_i[\delta(i)])$.

Complexity Analysis

- Step 1: Building $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$, suppose we delete α_i tuples, then $\mathcal{T}_i$ can be constructed in O(($\alpha_i + 1$)log K) time.
- Step 2: Repeat step 1 for all K trees. In total $\mathcal{T}_K$ can be constructed in O(($\alpha_1+\alpha_2+..+\alpha_K+K$)log K) time. Since we can delete at most K (since $\alpha_1+\alpha_2+..+\alpha_K \leq K$) tuples, $\mathcal{T}_n$ is in the order of O(K log K) time.

4.3.4 Analysis of Complexity

- Step 1 : According to Section 3.2.2, for two sequences of length m_1 and m_2 respectively, all MUMs can be computed in $O(m_1+m_2)$ time.
- Step 2 : After finding all the MUMs, we sort them according to their position in Genome A, and employ the LCS algorithm to find the largest set of MUMs whose sequences occur in the same order contained in both Genome A and B. As mentioned previously, this step requires O(K log K) time, where K is the number of MUMs. In general, K is much smaller than m and n.
- Step 3 : Once a global alignment is found, we can deploy several algorithms for closing the local gaps and completing the alignment. A gap is defined as an interruption in the MUM alignment which falls into one of the following four classes: (i) an SNP, (ii) an insertion, (iii) a highly polymorphic region or (iv) a repeat. Time and space complexity are dependent on the algorithm deployed.

4.3.5 Strength and Limitation

The strength of MUMmer1 lies on the speed at which it processes long alignments and that MUMmer1 produces much less false positives than its predecessors.

The limitation of this algorithm is that this approach assumes that there exist a single long alignment, which may not be always true, as they may be a huge chain of reversals between two genomes. See Figure 4.7

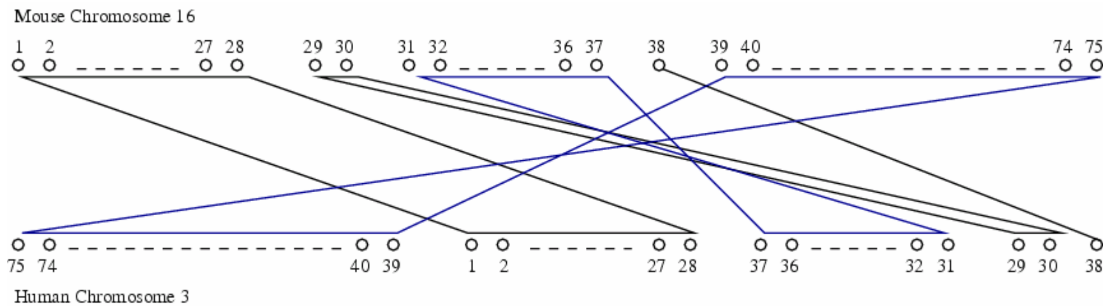


Figure 4.7: There are 31 conserved gene pairs found in Mouse chromosome 16 and Human chromosome 3. Note that the relative ordering of Genes 31 to 37 in Human chromosome is exectly the reverse of that in the mouse chromosome. The same occurs in Genes 39 to 75.

4.3.6 Applications

MUMmer1 does a pairwise alignment and comparison of two large scale DNA sequences, and outputs a base-to-base alignment of the input sequences, highlighting the exact differences between them. Thus, it can be used to identify all differences between two long DNA sequences or even genomes. It can also locate all single nucleotide polymorphisms (SNP), large inserts, significant repeats, tandem repeats and reversals, in addition to identifying the exact matches between the genomes. Another application of MUMmer1 is to compare two different versions of a genome at different stages of sequencing and to highlight precisely what has changed.

4.4 MUMmer2

4.4.1 Algorithmic Improvements

Three significant technical improvements in the core algorithms of MUMmer2 are listed in the following:

4.4.1.1 Reducing Memory Usage

By employing techniques described by Kurtz [8], the amount of memory used to store the suffix tree is reduced to at most 20 bytes/base pair (or amino acid, or other character). The maximum memory usage occurs when each internal node in the suffix tree has only two children. In practice, however, many nodes have more than two children (particularly in the case of polypeptide sequences), which reduces the actual memory requirement.

4.4.1.2 Employing new alternative algorithm for finding exact matches

The MUMmer1 [3] employs an algorithm that builds a generalized suffix tree for two input sequences to find all MUMs. Although this algorithm is still available in the MUMmer2 system, a new alternative algorithm was introduced as the default algorithm to find all MUMs. Instead of storing the generalized suffix tree for the two input sequences, this new algorithm stores only one input sequence in the suffix tree. This sequence is called the reference sequence. The other sequence, which is called the query, is then 'streamed' against the suffix tree, exactly as if it were being added without actually adding it. This technique was introduced by Chang and Lawler [2] and is fully described in [7].

Using this 'streaming' process, we can identify where the query sequence would branch off from the tree, thus we can find all matches to the reference sequence. Whenever a branch occurs at an internal node with just a single leaf beneath it, the matching subsequence is unique in the reference sequence. Note that these matches are not necessarily unique in the query sequence. Then, by checking the character immediately preceeding the start of this match, we can determine whether it is maximal. As the consequence of streaming through the query, i.e. outputting matches as we find them, we do not know what sequence will occur later in the query.

By using this new algorithm, all maximal matches between query sequence and a unique substring of the reference sequence can be identified in time proportional to the length of the query sequence.

The advantage of this method is that only one of the two sequences is stored in the suffix tree, reducing the memory requirement by at least half. Furthermore,

because of the streaming nature of the algorithm, once the suffix tree has been built for an arbitrarily long reference sequence, multiple queries can be streamed against it. Therefore, we don't have to re-build the suffix tree if one of the input sequences is changed. In fact, Delcher et al [4] have used these programs to compare two assemblies of the entire human genome (each approximately of 2.7 billion characters long), using each chromosome as a reference and then streaming the other entire genome past it.

4.4.1.3 Clustering matches

Biologists observed that a pair of conserved genes are likely to correspond to a sequence of MUMs that are consecutive and close in both genomes. At the same time, this sequence of MUMs generally has a sufficient length. The set of such MUMs is called a cluster.

For example, as shown in Figure 4.8, for the given two sequences, there are two clusters: 123 and 56.

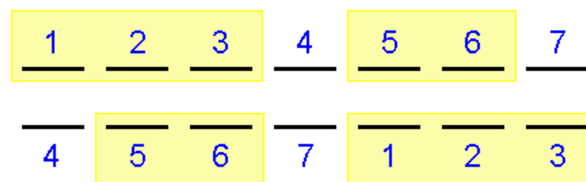


Figure 4.8: Clusters of MUMs

MUMmer1 presumed that two complete sequences were to be aligned, and that no major rearrangements would have occurred between them. Hence, it computed a single longest alignment between the sequences. In order to facilitate comparisons involving unfinished assemblies and genomes with significant rearrangements, a module has been added in MUMmer2. This module first groups all the close consecutive MUMs into clusters and then finds consistent paths within each cluster. The clustering is performed by finding pairs of matches that are sufficiently close and on sufficiently similar diagonals in an alignment matrix (using thresholds set by the user), and then computing the connected components for those pairs. Within each component, a longest common subsequence is computed to yield the most consistent sequence of matches in the cluster.

As the result of the additional module, the system outputs a series of separate, independent alignment regions. This improvement is the main key for MUMmer2 to achieve a better coverage.

4.4.2 Applications

MUMmer2 has been applied to solve "alignment of incomplete genomes" and "comparative genome annotation" problems. Two new solvers with MUMmer as the core algorithm are developed to solve this two problems, they are NUCmer(nucleotide MUMmer) and PROmer(protein MUMmer). Experiments showed that MUMmer is useful to solve the two problems stated above. Detailed description for these two applications can been found in [4].

4.5 MUMmer3

Basically, the structure of MUMmer3 is pretty similar to that of MUMmer2, but some amendments were done to further improve its time and space complexity. In terms of space, MUMmer3 is slightly more efficient as it uses approximately 16 bytes only to store a base pair of the reference sequence in the suffix tree. The process of 'streaming' the query sequence past the reference suffix tree is maintained, so that the memory requirements do not depend on the size of the query sequence at all.

In MUMmer3, the concept of MUM is slightly relaxed that it need not to be unique in both input sequences. We can opt to find all non-unique maximal matches, all matches that are unique only in reference sequence, or all matches that are unique in both sequences. This feature was added as it was observed that the uniqueness constraint would prevent MUMmer from finding all matches for a repetitive substring.

The most critical improvement in MUMmer 3 is a complete re-write of the core suffix tree library, based on the compact suffix tree representation. It was implemented by Stefan Kurtz [8] and explained in his various publications. The improvements resulting from the use of this library can be seen in the table below. All statistics are from test runs on a 3.0 GHz Pentium 4 computer running Linux. Resulting output includes both forward and reverse matches.

MUMmer3 requires approximately 25% less memory than MUMmer2 and it runs slightly faster. Compared to MUMmer1, MUMmer3 is more than twice faster and uses less than half of the memory. Another additional feature that is only available in MUMmer3 system, is a new Java viewer, DisplayMUM. It is a new graphical output program to generate images in fig-format or PDF, showing the alignment of a set of contigs to a reference chromosome.

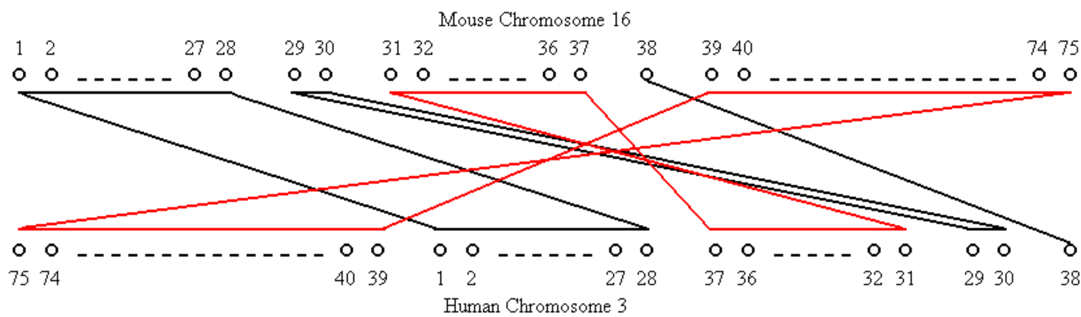
	MUMmer2	MUMmer3
E.coli K12 vs. E.coli O157:H7	102 MB / 18 s	77 MB / 17 s
S.cerevisiae vs. S.pombe	261 MB / 51 s	204 MB / 47 s
A.fumigatus vs. A.nidulans	578 MB / 128 s	459 MB / 120 s
	NUCmer 2	NUCmer 3
D.melanogaster arm 2L vs. D.pseudoobscura	684 MB / 879 s	485 MB / 835 s
	PROmer 2	PROmer 3
P.falciparum vs. P.yoelii	752 MB / 1109 s	522 MB / 975 s

Table 4.3: Performance Comparison between MUMmer2 and MUMmer3

4.6 Mutation Sensitive Alignment

4.6.1 Introduction

The Mutation Sensitive Alignment (MSA) algorithm [1] is based on the observation that if two genomes are closely related, then it is likely that they can be transformed from each other using a few transposition/reversal operations. For example, it can be seen in Figure 4.9 that Mouse Chromosome 16 can be transformed into Human Chromosome 3 in two operations (on sequences 31-37 and 39-75).



(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 the relative ordering of genes 31 to 37 in human chromosome is exactly the reverse of that in the mouse chromosome. The same situation occurs in genes 39 to 75.

Figure 4.9: Comparison of Mouse Chromosome 16 and Human Chromosome 3

The MSA algorithm makes use of this observation by looking for subsequences in the two input genome strings that differ by at most k transposition/reversal oper-

ations (the authors of the algorithm set k=4). It then returns these subsequences as possible conserved genes.

4.6.2 Concepts and Definitions

In this section, we explain the similar subsequence problem (or *k-similar subsequence problem*), which is the basis of the MSA algorithm.

Each MUM is assigned a number as its weight. Normally, the weight of an MUM is equal to its length. The *weight* of a common subsequence is the total weight of all the MUMs that make up the subsequence. The *maximum weight common subsequence (MWCS)* is the common subsequence with maximum weight.

Definition 4.5 *Let k be a non-negative integer. Consider a subsequence X of A and a subsequence Y of B. (X,Y) is a pair of* ***k-similar subsequences*** *if X can be transformed into Y by performing k transposition/reversal operations on k disjoint subsequences in X.*

Let (X,Y) be a pair of k-similar subsequences. Intuitively, both X and Y consist of k disjoint blocks and a remaining common subsequence, which is called the *backbone*. Each of the k blocks is either a common subsequence or reversed common subsequence of X and Y. These k blocks are parts of the subsequence on which the k transposition/reversal operations are performed. An example of a 2-similar subsequence is presented in Figure 4.10. In this example, the two blocks are (2,3) and (4,5,6), whereas the backbone is (1,7,8).

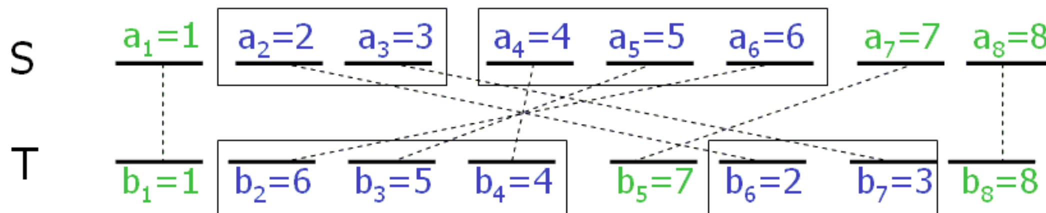


Figure 4.10: A Pair of 2-Similar Subsequences

Given two sequences A and B, and a non-negative parameter k, the *Similar Subsequence Problem* is defined as the problem in finding a k-similar subsequence of A and B with the greatest weight.

4.6.3 Algorithm Details

The MSA algorithm perceives two sequences A and B, and a non-negative parameter k as its input. It returns the set of greatest-weight k-similar subsequences as output. The rationale is that conserved genes between two species are likely to be k-similar for some small user-defined value of k.

The MSA algorithm consists of 4 steps, they are as follows:

1. Find all MUMs of A and B (as shown in Section 3.2.2, it can be done in linear time)
2. Find the k-similar subsequences of A and B
3. Return the set of k-similar subsequences

The Similar Subsequences Problem that forms the basis of the MSA algorithm in Step 2 has been shown to be NP-Complete [1], by reduction from the **MAX-2SAT Problem** [6]. Therefore, an optimal solution to the Similar Subsequences Problem will take exponential time (assuming P $\neq$ NP).

Using dynamic programming, this k-similar subsequences problem can be solved in O(n^{2k+1} log n) time, where n is the number of MUMs. Unfortunately, this is too slow for practical purposes. However, using a heuristic algorithm, this problem

can be approximated in O(n^2(log n + k)) time [1].

4.6.4 Experimental Results

The performance of MSA was compared to that of MUMmer3 on 15 pairs of mouse and human chromosomes (Table 4.1). Included in this set of data is the known conserved genes between the pairs of chromosomes. This data was obtained from *GenBank* [10], the largest public DNA sequence database maintained by the National Center for Biotechnology Information (NCBI).

For these experiments, the parameter k for MSA was set at 4, while the parameter gap for MUMmer3 was set at 2000. For each of the chromosome pairs, both techniques were rated on their *coverage* and *preciseness*. The value of coverage indicates the percentage of published genes discovered, whereas that of preciseness indicates the percentage of results returned that matches some published gene pairs. The results are given in Figure 4.11.

It can be seen from these results that in general MSA provides a much better coverage and a slightly better preciseness over MUMmer3. MSA is currently

	Coverage		Preciseness	
Exp.No.	MUMmer3	MSA	MUMmer3	MSA
1	76.50%	92.20%	21.70%	22.70%
2	71.40%	91.70%	21.30%	25.10%
3	87.00%	100.00%	24.80%	25.50%
4	76.30%	94.70%	27.40%	26.70%
5	92.50%	96.30%	32.50%	32.00%
6	72.20%	95.80%	31.20%	32.90%
7	67.70%	87.10%	13.50%	17.80%
8	78.10%	90.60%	37.20%	36.70%
9	80.00%	86.70%	40.70%	49.70%
10	82.00%	92.00%	30.90%	32.10%
11	65.20%	89.10%	30.50%	36.00%
12	60.00%	80.00%	27.50%	41.90%
13	89.10%	95.30%	18.20%	18.40%
14	72.70%	86.40%	10.40%	12.60%
15	78.50%	91.40%	30.00%	29.70%
average	76.60%	91.30%	26.50%	29.30%

Figure 4.11: Performance Comparison between MUMmer3 and MSA

the best technique for discovering conserved genes between two closely related genomes.

References

[1] Chan, H.L. et al. (2003). Improved Whole Genome Alignment via Mutation-Sensitive Sequence Similarity, in preparation. In Proceedings of Fourth IEEE Symposium on Bioinformatics and Bioengineering (Taichung, Taiwan, May 19-21, 2004), BIBE 2004, pp.545-552.

[2] Chang, W.I. and Lawler, E.L. (1994). Sublinear Expected Time Approximate String Matching and Biological Applications. Algorithmica, 12, 1994, pp. 327-344.

[3] Delcher, A.L. et al. (1999). Alignment of Whole Genomes. Nucleic Acids Research, Vol. 27, No.11, 1999, pp. 2369-2376.

[4] Delcher, A.L. et al. (2002). Fast Algorithms for Large-scale Genome Alignment and Comparison. Nucleic Acids Research, Nuclei Acids Research, 30(11), 2002, pp. 2478-2483.

[5] Eisen, J.A. et al. (2000). Evidence for Symmetric Chromosomal Inversions around the Replication Origin in Bacteria. Genome Biol., 1, 2000, pp. 1101-1109.

[6] Garey, M.R. and Johnson, D.S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Co., 1979.

[7] Gusfield, D. (1997). Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, 1997.

[8] Kurtz, S. (1999). Reducing the Space Requirement of Suffix Trees. Software Pract. Experience, 29, 1999, pp. 1149-1171.

[9] Lin, X. et al. (1999). Sequence and Analysis of Chromosome 2 of the Plant Arabidopsis Thaliana. Nature, 402, 1999, pp. 761-768.

[10] National Center for Biotechnology Information. (July 15, 2004). Comparative Maps. [Online]. Available: http://www.ncbi.nlm.nih.gov/Homology

[11] Perna, N.T. et al. (2001). Genome Sequence of Enterohaemorrhagic Escherichia Coli O157:H7. Nature, 409, 2001, pp. 529-533.

[12] The Arabidopsis Genome Initiative. (2000). Analysis of the Genome Sequence of the Flowering Plant Arabidopsis Thaliana. Nature, 408, 2000, pp. 796-815.