Match Chaining Algorithms for cDNA Mapping

Tetsuo Shibuya[1] and Igor Kurochkin[2]

[1] IBM Tokyo Research Laboratory, 1623-14, Shimo-tsuruma, Yamato, Kanagawa, 242-8502, Japan.
tshibuya@jp.ibm.com
[2] RIKEN Genomic Sciences Center,
1-7-22, Suehiro-cho, Tsurumi, Yokohama, Kanagawa 230-0045, Japan.
igork@gsc.riken.go.jp

Abstract. We propose a new algorithm called the MCCM (Match Chaining-based cDNA Mapping) algorithm that allows mapping cDNAs to the genomes efficiently and accurately, utilizing local matches called MUMs (maximal unique matches) or MRMs (maximal rare matches) obtained with suffix trees. From the MUMs (or MRMs), our algorithm selects appropriate matches which are related to the cDNA mapping. We call the selection the match chaining problem. Several $O(k \log k)$-time algorithms are known where k is the number of the input matches, but they do not permit overlaps of the matches. We propose a new $O(k \log k)$-time algorithm for the problem with provision for overlaps. Previously, only an $O(k^2)$-time algorithm existed. Furthermore, we also incorporate a restriction on the distances between matches for accurate cDNA mapping. We examine the performance of our algorithm through computational experiments using sequences of the FANTOM mouse cDNA database and the mouse genome. According to the experiments, the MCCM algorithm is not only very fast, but also very accurate: We achieved $> 95\%$ specificity and $> 97\%$ sensitivity at the same time against the mapping results of the FANTOM annotators.

1 Introduction

Since the complete and accurate human genome sequence has just been released, the interpretation and analysis of this large dataset has become a major task for the scientific community. One of the most fundamental analyses of a genome sequence is mapping a large number of cDNAs (mRNAs) or ESTs (expression sequence tags) to the regions of the genome they are transcribed from. Figure 1 illustrates the mapping. Precise mapping allows further analysis of the gene regulatory elements (promoters, transcription factor binding sites, etc.) to be performed. It also provides a basis for understanding how genomes evolve. However, the large size of the mammalian genomes, the existence of splice sites, and the numerous repetitive regions pose a serious problem for the alignment between cDNA and genome sequences. Alignments taking into account splice sites are called *spliced alignments*. Several spliced alignment algorithms based on $O(nm)$-time dynamic programming (DP) (where n and m are the lengths of

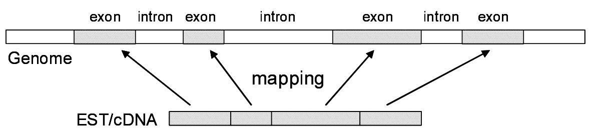

Fig. 1. Mapping a cDNA to the genome.

the two sequences to align) have been developed [11,18,21]. However, in practice, their time bound is too large for aligning large numbers of cDNAs to genome sequences of enormous size. Several other algorithms that heuristically map cDNAs to the genome have been proposed recently. These include sim4 [10], BLAT [14], and Squall [20], which are based on a hash or hash-like tools like BLAST [2].

Suffix trees [9,12,17,22,24] are known to be very powerful and flexible data structures for pattern matching. For example, exactly matching substrings can be found very efficiently using suffix trees, while an ordinary hash structure can deal with only fixed-length matches. Suffix trees require memory space that is several times larger than many of the other indexing structures, but the memory cost is going down at a remarkable rate these days and is becoming a less serious problem. Consequently many bioinformatics tools use suffix trees. For example, MUMmer [6,7] uses suffix trees for computing alignments of bacteria-size long sequences. They at first enumerate short exact matches called MUMs (Maximal Unique Matches) and align the sequences by chaining some of the MUMs. In this paper, we propose a new match chaining algorithm with better time complexity, and we also extend the strategy of the MUMmer to make it more suitable for cDNA mapping. We call the new algorithm the MCCM (Match Chaining-based cDNA Mapping) algorithm.

2 Algorithms

2.1 Preliminaries

We describe in this section several basic algorithms and data structures that will be used to describe our MCCM algorithm.

Suffix Trees The suffix tree [9,12,17,22,24] of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S^+ = S\$$ where \$ is a character such that $\$ \notin \Sigma$. This data structure is known to be buildable in $O(n)$ time. Figure 2 shows an example of this data structure. Each leaf represents a suffix of the string S^+, and each node represents some substring. This data structure is very useful for various problems in sequence pattern matching. Using it, we can query a substring of length m in $O(m)$ time, we can find frequently appearing substrings in a given sequence in linear time, we can find a common substring of many sequences in linear time, and so on [12]. The list of leaves corresponds to a data structure called the suffix array [16] that is a list of the indices of the lexicographically

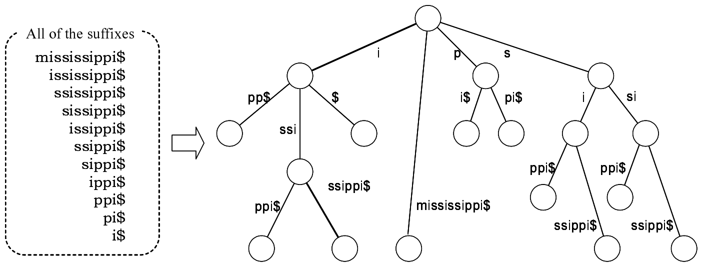

Fig. 2. The suffix tree of a string ‘mississippi’.

sorted suffixes. Note that the suffix array is also a very useful data structure that can be an alternative to the suffix tree for many purposes, especially when memory is limited.

Maximal Rare Matching We can easily find the longest prefix of a query string that matches somewhere in a given text sequence in a time linear to the query size if we have the suffix tree of the text sequence. Moreover, with the suffix tree, we can also find all the longest prefixes of all of the suffixes of the query string that matches somewhere in a given text sequence in the same time bound [7,12]. This is done by tracing the nodes of the suffix tree that correspond to the suffixes using suffix links. Note that a suffix link is a pointer from a suffix tree’s node that represents some substring T to another node that represents the suffix of T which is shorter than T by 1.

A ‘maximal matching substring’ of two sequences (a pattern and a text) is an exactly matching substring that occurs in a text sequence r times, but any other string that includes the substring appears less than r times in the text. The MUMmer [6,7] uses to compute the list of what they call MUMs (Maximal Unique Matching substrings), which is a list of the maximal matching substrings where $r = 1$. Using the algorithm above, we can compute all the MUMs in linear time relative to the query size if we already have the suffix tree of the text [7,12]. We will use them for cDNA mapping, considering the cDNA sequences to be pattern sequences and the genome sequence to be a text sequence.

If the genomic sequences contain many repetitive substrings, there can be cases that corresponding matching substrings are not ‘unique’, and it could be better to use a larger value of r. We call a maximal matching substring a maximal rare matching, or an MRM for short, if $r \leq l$ for some fixed l. All the MRMs for some l can be easily computed with suffix trees in $O(l \cdot m)$ time where m is the length of the query sequences, by using a minor extension of the above MUM algorithm. But in the case of the FANTOM cDNAs and the mouse genome, we do not need to use large ls according to our experiments.

Match Chaining Problem If we select appropriate matches from the large MUM or MRM list, we can build a sort of an alignment of the two sequences [6,13,19]. We call this the match chaining problem. To form an actual alignment, we must postprocess these selected matches, but we do not consider this in this paper. Let $M = \{m_1, m_2, \ldots, m_k\}$ be a list of input matches sorted by their start positions in the pattern sequence. Note that the MUM or MRM results are already sorted in this order if we use typical algorithms based on suffix trees. Let match m_i's start positions in the pattern sequence and in the text sequence be p_i and t_i respectively, and its length be l_i. An M's subset $\{m_{c_1}, m_{c_2}, \ldots, m_{c_f}\}$ is called a match chain of M if $p_{c_i} < p_{c_j}$ and $t_{c_i} < t_{c_j}$ for any i and j such that $i < j$. A match m_i is said to have an overlap with a match m_j on the pattern sequence, if the two regions $[p_i, p_i + l_i - 1]$ and $[p_j, p_j + l_j - 1]$ overlap with each other. Similarly, a match m_i is said to have an overlap with a match m_j on the text sequence, if the two regions $[t_i, t_i + l_i - 1]$ and $[t_j, t_j + l_j - 1]$ overlap with each other.

The match chaining problem is a problem to find a match chain that represents an appropriate alignment of the two sequences, and is defined as follows, letting $\mathit{score}(C)$ be some measure for the 'goodness' of a chain C, which we will discuss later.

Problem 1. Find the match chain $C = \{m_{c_1}, m_{c_2}, \ldots, m_{c_f}\}$ with the largest $\mathit{score}()$ value among all the possible match chains.

There can be several strategies of defining $\mathit{score}()$ values. The simplest measure is the number of matches, *i.e.*, $\mathit{score}(C) = f$. In this case, the best chain is known to be obtained in $O(k \log k)$ time using the longest increasing subsequence (LIS) algorithm [7,12]. We can also use as the $\mathit{score}()$ values the total lengths of the match chains instead of just the number of matches, *i.e.*, $\mathit{score}(C) = \Sigma_{1 \le i \le f} l_{c_i}$. If we do not permit the overlaps of the adjacent matches of the chain, it can also be computed in the same time bound $O(k \log k)$ using a tree structure called a range tree [5,19]. Let C_i be the chain that has the largest $\mathit{score}()$ value among the match chains that ends with m_i, and $\mathit{prev}(m_i)$ be the match previous to m_i in the chain C_i. Most previous algorithms for this problem can be described in the following form.

Algorithm 1 (Basic Match Chaining Algorithm).

1. For all i from 1 to k in this order, do the following.
   – Find $\mathit{prev}(m_i)$ for m_i.
2. Select the match chain with the largest $\mathit{score}()$ value among the match chains obtained in step 1 for all the matches. The actual match chain is constructed by tracing $\mathit{prev}()$ entries.

If $\mathit{prev}(m)$ can be obtained in $O(g(k))$-time in step 1 for a match m, the total computation time is $O(k \cdot g(k))$ time, as step 2 can be done in linear time.

However, no algorithm has been known to exist that considers overlaps and runs in $O(k \log k)$ time. Let $\mathit{pattern\_overlap}(m_i, m_j) = \max\{0, p_i + l_i - p_j\}$. When $p_i \le p_j$ and $p_i + l_i \le p_j + l_j$, this represents the actual overlap length of the two matches on the pattern sequence. Similarly, we let $\mathit{text\_overlap}(m_i, m_j) = \max\{0, t_i + l_i - t_j\}$ and $\mathit{overlap\_length}(m_i, m_j) =$

$\max\{\textit{text\_overlap}(m_i, m_j), \textit{pattern\_overlap}(m_i, m_j)\}$. Then the total length of a match chain $\{m_{c_1}, m_{c_2}, \ldots, m_{c_f}\}$ is described as $\sum_{1 \leq i \leq f} l_{c_i} - \sum_{1 \leq i < f} \textit{overlap\_length}(m_{c_i}, m_{c_{i+1}})$ if we consider overlaps. In this paper, we use this value as the *score*() measure. The MUMmer uses the same measure, but their algorithm requires $O(k^2)$ time, which is inefficient for large k's. We will improve this bound later.

2.2 Dynamic Range Maximum Query

In this section, we deal with yet another problem, the range maximum query (RMQ) problem [4], which plays an important role in our match chaining algorithm. Given a length n array A of numbers, the RMQ problem is the problem of finding the index of the maximum value in the subarray $A[i \ldots j]$ for any query of i and j. For a static array, we can find the index in a constant time with linear-time preprocessing [4]. We consider the dynamic version of this problem as follows.

Problem 2 (Dynamic RMQ). Let S be a set of items that is a null set ϕ at first. Items that are not known in advance are inserted to or deleted from S one by one. An item I has two given values $a(I)$ and $b(I)$, and the problem is to find the item with the maximum b value among the items $I \in S$ whose a value is within the range $[p, q]$ for any query of p and q at any time.

We present a dynamic RMQ algorithm which allows $O(\log k)$ query time with $O(\log k)$ update time for both insertions and deletions, where k is the size of S at the time of the query or the update. In our dynamic RMQ algorithm, we maintain a queue of items sorted by the a values. Such queues are often implemented with balanced binary tree data structures. The height of a balanced tree data structure is always $O(\log k)$, where k is the current number of nodes or the current queue length. For any node v in these balanced trees, the left child of a node v always has a key (the a value in this case) that is no larger than the key of v, while the right child of v always has a key that is no smaller than the key of v. We use the AVL tree [3] for this purpose. The details of the AVL tree are described in many textbooks such as [1]. The AVL tree can maintain such a queue implicitly with the update (insertion and deletion) time of $O(\log k)$. Each node of the AVL tree corresponds to an item and the in-order traversal of the tree corresponds to the sorted list of the items. Each item can be accessed in $O(\log k)$ time using its a value as its key.

In our dynamic RMQ algorithm, we maintain a pointer from each node v to the item with the maximum b value among the items whose corresponding nodes are in the subtree under v (including v). We call this a max pointer. We show that maintenance of the max pointers for insertion and deletion of an item can also be done in $O(\log k)$ time as follows. The update procedure of the AVL tree consists of five procedures: position lookup, leaf deletion, leaf addition, node replacement, and a procedure called rotation, each of which can be done in a constant time except for the lookup procedure. The lookup procedure finds the node position to add, delete or replace, and it can be done in $O(\log k)$ time. This

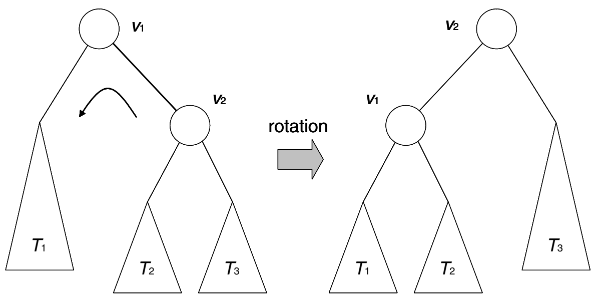

Fig. 3. The rotation procedure for the AVL tree.

procedure does not change the data structure, so we do not have to change any of the max pointers related to the lookup procedure. The next three procedures are executed at most once for an update, while the rotation procedure is executed $O(\log k)$ times. If we add or delete a leaf, or replace a node, we must also update the max pointers of the ancestors of the updated node. As the number of the ancestors is $O(\log k)$, we can do it in $O(\log k)$ time. Figure 3 shows the rotation procedure. Notice that the max pointers of nodes except for nodes v_1 and v_2 in the figure do not change at all. Furthermore, the new max pointers for the two nodes can be obtained in a constant time using the max pointers of the root nodes of the subtrees T_1, T_2, and T_3. Consequently, we need only a constant time for one rotation procedure to maintain the max pointers. In this way, we can maintain the max pointers in $O(\log k)$ time for each update.

We can compute the dynamic RMQ in $O(\log k)$ time using this data structure as follows. Using the AVL tree, we can easily find the item with the smallest a value that is not smaller than p in $O(\log k)$ time. Let the item be I_{left} and the corresponding node be v_{left}. Similarly, we can also find the item with the largest a value that is not larger than q in the same time bound $O(\log k)$. Let the item be I_{right} and the corresponding node be v_{right}. By checking the max pointers of the children of the nodes on the path from v_{left} to v_{right} as follows, we can find the item with the maximum b value among the items whose a value is not smaller than p and not larger than q. Let v_{lca} be the lowest common ancestor of the two nodes v_{left} and v_{right}. A constant-time query for the lowest common ancestor is possible with linear-time preprocessing [4], but we do not have to use the algorithm, as it can be easily computed in a time linear to the size of the path from v_{left} to v_{right}, that is $O(\log k)$. For each node v on the path P from v_{left} to v_{lca} including v_{left} but not including v_{lca}, we check the item pointed to by the max pointer of the right child of v and also the item of node v, if the right child of v is not on the path P. Similarly, for each node v on the path P' from v_{right} to v_{lca} including v_{right} but not including v_{lca}, we check the item pointed to by the max pointer of the left child of v and also the item of node v, if the left child of v is not on the path P'. Add to these, we check the item of v_{lca}. Then we choose the item with the maximum b value among these checked items, which is the answer to the query. Since the path length is $O(\log k)$, we can find the desired item in $O(\log k)$ time. Note that this algorithm can use a set of multiple keys as a b value that will be compared lexicographically.

2.3 Match Chaining Algorithms

Let us set out several definitions before describing our MCCM algorithm. Let R be a dynamic RMQ data structure as described in the previous section. Let $insert(R, I, a_I, b_I)$ be a function that inserts an item I whose a value is a_I and whose b value is b_I into R. Let $delete(R, I)$ be a function that deletes the item I from R. Let $rmq(R, p, q)$ be a function that returns the item that has the maximum b value among the items in R such that $p \leq a \leq q$. If such an item does not exist, it returns nil. Each function can be executed in $O(\log k)$ time, where k is the number of current items in R.

Maximum Inter-match Region Length The genome sequence has many repetitive regions, and copies of a gene subset can be seen in the genome. As a result, an ordinary match chaining algorithm described in the preliminary section often chains matches that are too far away from each other (several hundred millions bp apart in some cases). This can be avoided easily if we set a maximum inter-match region length and we do not permit chaining matches whose distance in the text sequence is larger than that. It can easily be incorporated without increasing the time complexity by using the following algorithm. Note that this algorithm does not permit overlaps of the matches. Let max_len be the maximum inter-match region length.

Algorithm 2 (Match Chaining Algorithm with Maximum Inter-match Region Length).

```
1. Let $R$ be an empty dynamic RMQ data structure.
2. For all $i$ from 1 to $k$ in order, do the following.
   (a) For all matches $m_j$ such that $j < i$, $p_j + l_j \leq p_i$ and $m_j \notin R$, execute
       the function $insert(R, m_j, t_j + l_j - 1, score(m_j))$. Note that $score(m_j)$
       is described in the next step.
   (b) Let $prev(m_i) = rmq(R, t_i - max\_len - 1, t_i - 1)$. If $prev(m_i) = nil$, let
       $score(m_i) = l_i$. Otherwise let $score(m_i) = score(prev(m_i)) + l_i$.
3. Find the match $m$ with the largest value of $score(m)$, and construct a chain
   from $m$ by tracing back the $prev()$ information.
```

This algorithm runs in $O(k \log k)$ time in total, as $insert()$ and $rmq()$ routines are executed k times at most.

Incorporating Overlaps We next incorporate overlaps without increasing the time complexity, as maximal matching substrings often overlap with each other. Our problem is to find the match chain with the largest total length considering overlaps. The MUMmer uses the same metric, but their algorithm requires $O(k^2)$ time. We break this barrier by proposing an $O(k \log k)$ algorithm. Moreover, at

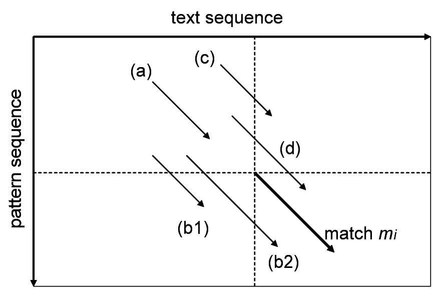

Fig. 4. Types of overlapping matches.

the same time, we use the restriction of the maximum inter-match region as stated above.

In our algorithm, we classify the match candidates for $prev(m_i)$ into 4 groups as follows. (a) A match that does not have an overlap with m_i. (b) A match m_j that has an overlap with m_i on the pattern sequence and satisfies $pattern\_overlap(m_j, m_i) > text\_overlap(m_j, m_i)$. (c) A match that has an overlap on the text sequence but not on the pattern sequence. (d) A match m_j that has overlaps with m_i both on the pattern and on the text and satisfies $pattern\_overlap(m_j, m_i) < text\_overlap(m_j, m_i)$. Figure 4 shows these match types graphically. In the figure, Type (b) is divided into the following two subtypes: (b1) A match of Type (b) that has no overlap with m_i on the text sequence. (b2) A match of Type (b) that also has an overlap with m_i on the text sequence. Note that we have to consider only the case (a) if we do not permit overlaps.

Our algorithm finds the best match candidates among the matches of Type (a), among those of Types (b), among those of Type (c), and among those of Type (d) separately, each of which is computed in $O(\log k)$ time. After that, it determines the actual best match from the four candidates. As for the matches of Type (a), we do not have to consider about any overlaps, so it is easy to deal with them with the dynamic RMQ data structure. To compute scores for the matches of Type (b), we must take the overlap lengths on the pattern sequence into account. Thus we construct a dynamic RMQ data structure based on the scores minus the pattern positions instead of the scores. As for those of Types (c) and (d), we must concisder the overlap lengths on the text sequence. Thus we use the scores minus the text positions instead of the scores as the measure. For the matches of Type (c), we use a different data structure for queuing and two functions for it, $insert\_queue()$ and $find\_best()$, which we will describe later. Letting $t(m)$ denote the start position of a match m (*i.e.* t_i for m_i), our match chaining algorithm is described as follows. In the algorithm, R_1, R_2, R_3, and Q maintain the data for querying matches of types (a), (b), (d), and (c) respectively.

Algorithm 3 (Algorithm for Match Chaining with Overlaps).

```
1. Let $R_1$, $R_2$ and $R_3$ be empty dynamic RMQ data structures, and $Q$ be an
   empty queue represented by a balanced tree data structure.
2. For all $i$ from 1 to $k$ in order, do the following.
   (a) For all matches $m_j$ such that $j < i$, $p_j + l_j \leq p_i$ and $m_j \notin R$, do the
       following.
       – Execute the functions $insert(R_1, m_j, t_j + l_j - 1, score(m_j))$,
         $delete(R_2, m_j)$, $delete(R_3, m_j)$, and $insert\_queue(Q, m_j, t_j, t_j + l_j -
         1, score(m_j) - t_j - l_j)$.
   (b) Let $m^{(a)} = rmq(R_1, t_i - max\_len - 1, t_i - 1)$, $m^{(b)} = rmq(R_2, t_i - p_i -
       max\_len - 1, t_i - p_i - 1)$, $m^{(c)} = find\_best(Q, t_i)$, and $m^{(d)} = rmq(R_3, t_i -
       p_i, t_i - 1)$.
   (c) If $m^{(d)} \neq nil$ and $t(m^{(d)}) > t_i$, let $m^{(d)} = nil$.
   (d) If the four candidates $m^{(a)}$, $m^{(b)}$, $m^{(c)}$ and $m^{(d)}$ are all $nil$, let $prev(m_i)$
       be $nil$ and $score(m_i) = l_i$. Otherwise, let $prev(m_i)$ be the match $m$ that
       is selected from the four (or less when some of them are $nil$) matches so
       that $score(m_i) = score(m) + l_i - overlap(m, m_i)$ is maximized.
   (e) Execute the functions $insert(R_2, m_i, t_i - p_i, score(m_i) - p_i - l_i)$ and
       $insert(R_3, m_i, t_i - p_i, key(score(m_i) - t_i - l_i, t_i - p_i))$, where $key(x, y)$
       denotes a multiple key that is compared lexicographically, *i.e.*, $x$ value
       is compared first and $y$ is used only when $x$ is same.
3. Find the match $m$ with the largest value of $score(m)$, and construct a chain
   from $m$ by tracing back the $prev()$ information.
```

In step 3(b), RMQ against R_3 sometimes returns a match whose text position is larger than t_i and we dismiss it in step 3(c). There is no problem in doing so because we always have a candidate of another type that is not worse than any candidates of Type (d) in this case. Next we explain the two functions for the matches of Type (c). Letting $\{I_i\}$ be a set of items for each of which we will give three values a_i, b_i, and c_i, Q be a queue of the items sorted by their b_i values, the function $insert\_queue(Q, I_i, a_i, b_i, c_i)$ is described as follows.

Algorithm 4 ($insert\_queue(Q, I_i, a_i, b_i, c_i)$).

```
1. Find the item $I_e$ in $Q$ that has the smallest $b$ value such that $b > b_i$.
2. Insert the item $I_i$ into $Q$ if $b_i < a_e$ or $c_i > c_e$. Otherwise, the algorithm is
   finished.
3. For the items $I_j$ such that $a_i \leq b_j \leq b_i$ from the items that have larger $b_j$,
   do the following.
   – If $c_j > c_i$, the algorithm is finished. Otherwise delete $I_j$ from $Q$.
```

This procedure runs in $O((d + 1) \log k)$ time where d is the number of items deleted in step 3, and k is the current size of Q, if we implement the queue Q with a balanced tree data structure like the AVL trees. The total number of deletions can be bounded by the number of matches k, thus the computation time for this function in total is $O(k \log k)$. Using this data structure, we can maintain the match candidates of Type (c). Note that this data structure does not maintain some of the matches of Type (c) if a better match candidate or that with the same score exists. We can obtain the best candidate with the following function, $find\_best(Q, b)$.

Table 1. Computation time for mapping all the 60,770 FANTOM cDNAs to three mouse chromosomes.

Chromosome #	1	15	19
Genome size (bp)	196,842,934	104,633,288	61,356,199
Query time in total (sec)	1565.48	1431.37	1333.87
Suffix tree construction time (sec)	2294.28	1159.37	665.88

Algorithm 5 ($find\_best(Q, b)$).

1. Find the item I_e in Q that has the smallest b value such that $b_j > b$.
2. If $b > a_e$, return the item I_e. Otherwise, return nil.

This can also be done in $O(\log k)$ time. Thus our match chaining algorithm runs in $O(k \log k)$ time in total.

There are several possible extensions of the MCCM algorithm. We can easily extend our algorithm to use different scores for different bases instead of just using the total length of the match chain. In our match chaining algorithm, we find the $previous(m)$ for a match m from the matches only from the region specified by the maximum inter-match region length on the same strand. By selecting $previous(m)$ in our algorithm from both strands, without increasing the time complexity, we can extend our algorithm for finding irregular mapping sites when a protein is encoded from both DNA strands [15]. An extension to multiple alignments as in [13] is also interesting, but the analysis of overlaps will be very difficult and remains as a future task.

3 Computational Experiments

In this section, we examine the performance of the MCCM algorithm through computational experiments in which we mapped cDNA sequences of the FANTOM 2.0 database [8] to the mouse chromosome 1 genome sequence and several others taken from the whole genome sequence assembly version 3 of Mouse Genome Sequencing Consortium (MGSC) (ftp://wolfram.wi.mit.edu/pub/mouse_contigs/MGSC_V3). We used this data because the FANTOM annotators used it for mapping the cDNAs. The FANTOM 2.0 database consists of 60,770 full-length cDNA clones whose total size is around 120 Mbp. Chromosome 1 is the largest chromosome of the mouse with a length of 196,842,934 bp. In the following experiments, we used a Power4 CPU running at 1.3 GHz.

Table 1 shows the speed of the MCCM algorithm for the genome sequences of three mouse chromosomes. In the experiments, we let $l = 1$ for the MRM computation (*i.e.* we use MUMs) and set the maximum inter-match to 3 Mbp, which is larger than any intron size as far as the authors know. We did not use a larger l because using a larger l leads to decrease of specificity according to our preliminary experiments. It may be because the mouse genome does

Table 2. Numbers of cDNAs mapped to the positive strand of Chromosome 1.

Algorithm	MIML = 3 Mbp					MIML not used				
Ratio (%)	90	80	70	60	50	90	80	70	60	50
#mapped	1804	1907	1964	2032	2112	1808	1914	1983	2057	2153
#annotated	1773	1824	1846	1860	1864	1774	1824	1846	1861	1864
Specificity (%)	98.3	95.6	94.0	91.5	88.3	98.1	95.3	93.1	90.5	86.6
Sensitivity (%)	94.6	97.3	98.5	99.3	99.5	94.7	97.3	98.5	99.3	99.5

not have many long copies of genes in the same chromosome. We set the minimum MRM length threshold to 20, because extremely short matches can appear randomly and they also cause the decrease of specificity. The table shows our results are much faster than previous tools like BLAT [14] and sim4 [10], which require about 2 to 20 seconds per query for a problem of the same size with the same CPU. The MCCM algorithm requires only 0.026 seconds per query on average. Note that the query time is roughly proportional to the query size. The Squall [20] algorithm runs at a speed similar to ours, but the query time of Squall is proportional to the genome size. On the other hand, the query time of our algorithm is not much influenced by the the genome size. Table 1 reveals that the query time increases only about 17% even if the genome size becomes more than 3 times larger. This means that the query time will be still reasonable even if we apply our algorithm to an entire genome with a size of 2 Gbp size or larger. A suffix tree for a sequence of that size is difficult to construct in the main memory of today's typical machines, but we believe it will be easy to do so in the very near future and then our algorithm will work very efficiently. The time for constructing the suffix tree is proportional to the genome size, but we only have to build it once.

Next, we examine the accuracy of the MCCM algorithm. We compared the results of our algorithms with the FANTOM annotations. We also did experiments with the same algorithm but without setting the maximum inter-match length (MIML) to see the importance of the MIML. Table 2 shows the results. In the table, the 'MIML = 3 Mbp' columns show the results of our algorithm where we set the maximum inter-match length to 3Mbp, while the 'MIML not used' columns show the results of the algorithm without MIML. Note that we did experiments only against the positive strand of the chromosome. In the experiment, we accept that the cDNAs are mapped to the chromosome if the total match chain length exceeds ratio r of the cDNA length for some r. The 'Ratio (%)' row shows the ratio r, which we adjusted from 90% to 50%. The '#mapped' row shows the numbers of cDNAs that are determined to be mapped to the positive strand of the chromosome by our algorithms. FANTOM annotators mapped 1,874 cDNAs to the positive strand of this chromosome. We call these 1,874 cDNAs 'annotated cDNAs'. The '#annotated' row shows the numbers of annotated cDNAs among our results such that the annotated regions are the same as ours or overlap with ours. The 'Specificity (%)' row shows the ratio of them among our results, while

Table 3. An example of a cDNA mapping result.

Exon	Annotated positions		Maximal match positions	
	cDNA	Genome	cDNA	Genome
1	1 : 220	65168568 : 65168788	2 : 224	65168570 : 65168792
2	221 : 342	65186935 : 65187056	220 : 346	65186934 : 65187060
3	343 : 489	65194686 : 65194832	341 : 489	65194684 : 65194832
4	490 : 590	65202019 : 65202119	485 : 591	65202014 : 65202120
5	591 : 731	65205936 : 65206078	589 : 628	65205934 : 65205973
			633 : 702	65205978 : 65206047
			712 : 731	65206059 : 65206078
6	732 : 915	65209875 : 65210057	732 : 779	65209875 : 65209922
			848 : 886	65209990 : 65210028
			887 : 914	65210028 : 65210055
7	916 : 1067	65211981 : 65212131	925 : 1068	65211989 : 65212132
8	1068 : 1768	65240140 : 65240840	1066 : 1768	65240138 : 65240840

the 'Sensitivity (%)' row shows the ratio of them among the annotated 1,874 cDNAs. Both the sensitivity and the specificity of our algorithm are remarkably high. For $r = 80\%$, we achieved 95.6% specificity and 97.3% sensitivity for example. The table also shows that we succeeded in increasing the specificity without decreasing the sensitivity by setting the maximal inter-match length, especially in the case of low ratio thresholds.

Finally, we show an example of the output of the MCCM algorithm in Table 3. This table shows the result for the cDNA whose ID is '1110063D23' in the FANTOM database. It is annotated as 'Camp-response element binding protein-homolog' and mapped to the chromosome 1. Our algorithm also maps this cDNA to the chromosome 1. The 'Annotated positions' columns show the positions given by the FANTOM annotators, while the 'Maximal match positions' show the corresponding maximal match positions. We can see that several matches overlap with each other, which means consideration of overlaps is mandatory, though the overlaps are very short. Our algorithms do not consider biological information like donor and acceptor signals. As a result, the boundaries are not correct in many cases as seen in this example. Add to these, two of the exons are divided into three matches, as our algorithm only outputs exact matches, which is caused by indels or substitutions. But, all in all, the match boundaries are quite close to the actual annotated boundaries, and we succeeded in mapping this cDNA to the correct site of the genome.

4 Concluding Remarks

We proposed a new cDNA mapping algorithm called the MCCM (Match Chaining-based cDNA Mapping) algorithm based on suffix trees and a match chaining technique. For it, we proposed a new $O(k \log k)$ match chaining algorithm which considers overlaps and distances between matches. We also examined the speed and accuracy of our algorithm through computational experiments using FANTOM cDNA sequences and the mouse genome. According to the experiments, our algorithm runs very fast, and we succeeded in finding almost the same ($> 95\%$) set of cDNAs that were given by the FANTOM annotators.

Several tasks remain as future work. The suffix tree is a very large data structure. An implementation with suffix arrays or compressed suffix arrays is very attractive. As shown in the experiments, the match boundaries obtained with our algorithms are a little bit different from the actual boundaries of the exons. Correcting the boundaries does not seem to be a complex problem and this will be addressed in our future work.

Acknowledgment

The authors thank Prof. Kunihiko Sadakane of Kyushu University and Prof. Akihiko Konagaya of RIKEN for various interesting discussions on related problems.

References

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *Data Structures and Algorithms*, (1983) Addison-Wesley.
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. (1990) Basic local alignment search tool. *J. Mol. Biol.*, 215, 403-410.
3. Adelson-Velskil, G.M. and Landis, E.M. (1962) *Soviet Math. (Dokl.)* 3, 1259-1263.
4. Bender, M.A. and Farach, M. (2000) The LCA problem revisited. *Proc. Latin American Theoretical Informatics*, LNCS 1776, 88-94.
5. Bentley, J. and Maurer, H. (1980) Efficient worst-case data structures for range searching. *Acta Informatica*, 13, 155-168.
6. Delcher, A.L., Kasif, S., Fleischmann, D., Paterson, J., White, O. and Salzberg, S.L. (1999) Alignment of whole genomes. *Nucleic Acids Res.*, 27(11), 2369-2376.
7. Delcher, A.L., Phillippy, A., Carlton, J. and Salzberg, L. (2002) Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, 30(11), 2478-2483.
8. FANTOM Consortium and the RIKEN Genome Exploration Research Group Phase I & II Team. (2002) Analysis of the mouse transcriptome based on functional annotation of 60,770 full-length cDNAs. *Nature*, 420, 563-573.
9. Farach, M. (1997) Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symp. Foundations of Computer Science,* 137-143.
10. Florea, L., Hartzell, G., Zhang, Z., Rubin, G.M. and Miller, W. (1998) A computer program for aligning a cDNA Sequence with a Genomic DNA Sequence. *Genome Res.*, 8, 967-974.
11. Gelfand, M.S., Mironov, A.A. and Pevzner, P.A. (1996) Gene recognition via spliced sequence alignment. *Proc. Natl. Acad. Sci. USA*, 93, 9061-9066.
12. Gusfield, D. (1997) *Algorithms on strings, trees, and sequences: computer science and computational biology,* Cambridge University Press.
13. Hoehl, M., Kurtz, S. and Ohlebusch, E. (2002) Efficient multiple genome alignment. *Bioinformatics*, 18(Suppl. 1), S312-320.

14. Kent, W. J. (2002) The BLAST-like alignment tool. *Genome Res.*, 12, 656-664.
15. Labrador, M., Mongelard, F., Plata-Rengifo, P., Bacter, E.M., Corces, V.G. and Gerasimova, T.I. (2001) Protein encoding by both DNA strands. *Nature*, 409, 1000.
16. Manber, U. and Myers, G. (1993) Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5), 935-948.
17. McCreight, E.M. (1976) A space-economical suffix tree construction algorithm. *J. ACM,* 23, 262-272.
18. Mott, R. (1997) EST_GENOME: A program to align spliced DNA sequences to unspliced genomic DNA. *Comput. Applic. Biosci.*, 13(4), 477-478.
19. Myers, E and Miller, W. (1995) Chaining multiple-alignment fragments in subquadratic time. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 38-47.
20. Ogasawara, J. and Morishita, S. (2002) Fast and sensitive algorithm for aligning ESTs to Human Genome. *Proc. 1st IEEE Computer Society Bioinformatics Conference*, Palo Alto, CA, 43-53.
21. Sze, S-H. and Pevzner, P.A. (1997) Las Vegas algorithms for gene recognition: suboptimal and error-tolerant spliced alignment. *J. Comp. Biol.*, 4(3), 297-309.
22. Ukkonen, E. (1995) On-line construction of suffix-trees. *Algorithmica,* 14, 249-260.
23. Usuka, J., Zhu, W. and Brendel, V. (2000) Optimal spliced alignment of homologous cDNA to a genomic DNA template. *Bioinformatics*, 16(3), 203-211.
24. Weiner, P. (1973) Linear pattern matching algorithms. *Proc. 14th Symposium on Switching and Automata Theory*, 1-11.