Algorithms for Identification of Patterns in Biogeography and Median Alignment of Three Sequences in Bioinformatics

by

Hai-Son Le

Supervisor: Vijaya Ramachandran

The University of Texas at Austin

December 2006

Acknowledgments

I am grateful for the supervision of Dr. Vijaya Ramachandran during my undergraduate research. I have learned from her the fundamentals of theoretical Computer Science and the joy of doing research. Her tireless passion and motivation enlightens my way through many difficult problems. It is my honor to work with Dr. Ganeshkumar Ganapathy and Rezaul Chowdhury, with generous help and kindness in responding to my multitude of questions. I would like to thank my collaborators: Rezaul Chowdhury, Ganeshkumar Ganapathy, Vijaya Ramachandran and Tandy Warnow in preparation of papers, where this work was presented. Special thanks to Dr. Tandy Warnow and Dr. Anna Gál for serving as committee members of this thesis. Finally, I am indebted to my parents for their support of my education and the endless encouragement during my stay at the University of Texas.

Hai-Son Le

The University of Texas at Austin

December 2006

Contents

Chapter 1

Introduction

Dynamic programming is a common technique used to solve variety of problems in many Computer Science fields. In bioinformatics, the technique finds wide usage due to the natural recursive structure of many computational problems. In my undergraduate research, I have looked at two different applications of dynamic programming in bioinformatics.

1. Tree comparison to infer common patterns in biogeography.
2. Median alignment of three DNA, RNA or protein sequences with affine gap cost.

My contribution to both of these problems have been to develop and apply techniques to improve the running time of the traditional algorithm. In the first application, we have developed a sparsified dynamic programming algorithm that improves the running time from $O(n^{2.5}\log n)$ to $O(n^2)$ in most common cases. For the second application, I applied the cache-oblivious method for dynamic programs, developed by Rezaul Chowdhury and Vijaya Ramachandran to obtain highly I/O efficient algorithms. This thesis shows how two very different approaches could improve the performance of traditional dynamic programming algorithms and their applications in bioinformatics.

Chapter 2

Identification of Patterns in Biogeography

2.1 Preliminaries

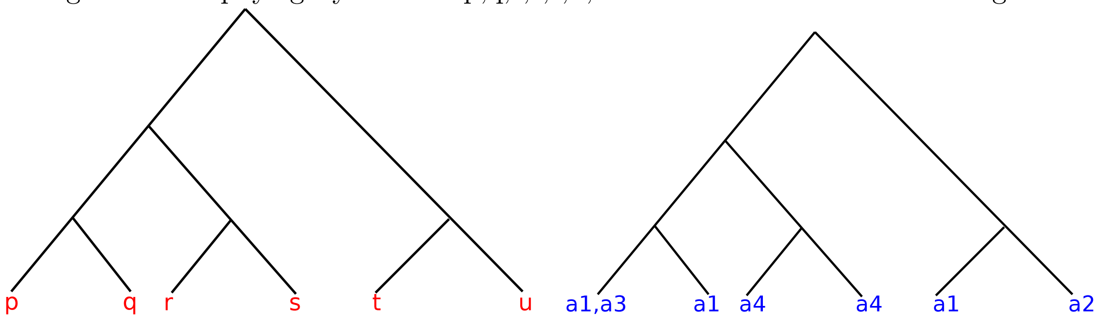
Biogeography is the science that attempts to understand spatial patterns of biodiversity. It is the study of distributions of organisms and of related patterns of variation over geographic areas in the numbers and kinds of living things. One of many ways to study the patterns is by looking at the evolutionary history of species. The evolutionary relationships are typically represented as branching tree structures called *phylogenetic trees*, or simply *phylogenies*. Biologists use area cladograms(Fig. 2.1) to represent the relationships and distribution of species in an area.

Definition 1. *Area cladograms are rooted or unrooted trees (as are phylogenies) whose leaves are labeled with geographic areas instead of taxa. They are obtained by replacing the taxon label in a phylogenic tree with the label of the area in which the taxon is found.*

The main difference between phylogenies and area cladograms relies on the uniqueness of leaf labels. Many leaves of an area cladogram could share the same label while leaf in a phylogeny is a unique species.

One technique to reconstruct the historial biogeographic distribution is *Indirect Inference.* Here, area cladograms of different groups of organisms which share their geographic distributions are compared. A consistent pattern observed in the evolutionary trees of species from different

Figure 2.1: A phylogeny on taxa p,q,r,s,t,u,v and the associated area cladogram



Geographic Distribution of Taxa: p → a1,a3 q → a1
r → a4 s → a4 t → a1 u → a2

genera in the same geographic area will imply stronger evidence for the particular hypotheses suggested by that pattern.

The problem of comparing phylogenies has been well studied for many years. Algorithms to solve this problem are formulated explicitly with the assumption that leaves on phylogenies are uniquely labeled. Therefore, these algorithms do not often apply to the context of area cladograms. Until recently, to solve this problem biologists have employed resolution techniques, which resolve area cladograms so that each leaf is uniquely labeled, and then used phylogeny-comparison algorithms. In 2005, Ganapathy et al. [GGJ+05] proposed a different approach to this problem. They invented the first rigorous metric, called the *a Maximum Agreement Area Cladogram (MAAC)* metric to directly compare area cladograms without any resolution steps. The new metric therefore is believed to yield stronger patterns.

We need some definitions before we can formally define the MAAC metric. Let T be an area cladogram on a set of leaves L. The *restriction* of T to a set of leaves L' is the cladogram obtained by deleting leaves in the set $L - L'$ from T and then suppressing internal nodes of degree two (except the root, if there is one). The formal definition of MAAC is:

Definition 2. *Maximum Agreement Area Cladogram (MAAC) and the MAAC metric [GGJ+05]*

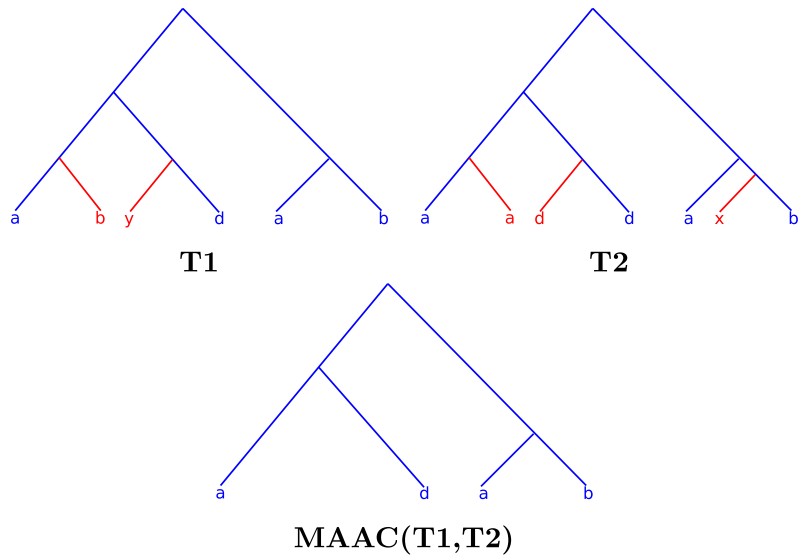
Let $\{T_1,\ T_2, \ldots, T_k\}$ be a set of rooted area cladograms, with L_i the leaf set of tree T_i, for $i = 1, 2, \ldots, k$. Let $\lambda_1 \subseteq L_1$ through $\lambda_k \subseteq L_k$ be sets of leaves of maximum cardinality such that the respective restrictions of the trees $T_1, \ldots, T_k$ to the sets $\lambda_1 \ldots \lambda_k$ are all isomorphic, with the isomorphisms preserving leaf labels. A restriction of any tree T_i to such a subset of leaves λ_i is a maximum agreement area cladogram *(MAAC) for the cladograms T_1 through T_k. The size of the*

MAAC is defined to be the number of leaves in the maximum agreement area cladogram, and is denoted by $size_{maac}(T_1, T_2, \ldots, T_k)$.

The MAAC distance *between two trees* T_1 *and* T_2 *is* $d_M(T_1, T_2) = max(n_1,\ n_2) - size_{maac}(T, T')$, *where* n_1 *and* n_2 *are the number of leaves in* T_1 *and* T_2 *respectively.*

Figure 2.2 shows an example of T1, T2 and their MAAC.

Figure 2.2: Delete minimum number of leaves to obtain the common sub-cladogram MAAC(T1,T2)



The organization of the rest of this chapter is as follows: section 2.2 presents the basic dynamic programming presented in [GGJ+05] by Ganapathy et al.; in section 2.4, we develop a faster dynamic program for the MAAC problem and in the last section we prove that computing the MAAC of k ($k > 2$) area cladograms is NP-hard.

2.2 Basic Dynamic Programming algorithm for MAAC

Overview: The basic algorithm for MAAC, presented by Ganapathy et al. in [GGJ+05] is based on a dynamic programming algorithm for the phylogenetic rooted maximum agreement subtree algorithm (called *the MAST problem*) from [SW93]. We present here the outline of the algorithm and its correctness.

Throughout this presentation, we employ following definitions. We are given two area cladograms T_1 and T_2 on the set of leaves L. We let n_1, n_2 be the respective number of leaves in trees T_1 and T_2 , $n = max\{n_1, n_2\}$ and $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ be the set of areas with which the leaves of T_1 and T_2 are labeled. More specifically, we let $\pi_{i,j}$, $j = 1, 2$, be the number of leaves in tree T_j which are labeled with a_i; hence $\sum_{i=1}^{k} \pi_{i,j} = n_j$ for $j = 1, 2$. For a given node v in a rooted tree T, we let $L(v)$ be the set of descendent leaves of v, $p(v)$ be the parent of v, $c(v)$ be the set of children of v and $d_v = |c(v)|$ the degree of v. Also, we let $V(T)$ be the set of internal nodes of the tree T and $T(v)$ be the rooted subtree of T at the node v. For simplicity, we denote MAAC($T_1(v_1)$, $T_2(v_2)$) as MAAC(v_1, v_2).

Pseudocode ([GGJ+05]):

```
MATCH($v_1, v_2$)
1   Construct $G_{v_1,v_2}$
2   return MWBM ($G_{v_1,v_2}$)


DIAG($v_1, v_2$)
1   $t_1 \leftarrow \max_{w \in c(v_2)}\{$ MAAC $(v_1, w)\}$
2   $t_2 \leftarrow \max_{w \in c(v_1)}\{$ MAAC $(w, v_2)\}$
3   return $\max\{t_1, t_2\}$


ALGORITHM MAAC($T_1, T_2$)
1   Let $\mathcal{O}$ be an ordering of $V(T_1) \times V(T_2)$
2     such that if $(v_1, v_2)$ is before $(w_1, w_2)$,
3     then $v_1$ is not a parent of $w_1$ and $v_2$ is not a parent of $w_2$.
4   for $(v_1, v_2)$ in increasing order of $\mathcal{O}$
5         do if $v_1$ or $v_2$ is a leaf
6               then MAAC $(v_1, v_2) = L(v_1) \cup L(v_2)$
7               else MAAC $(v_1, v_2) = \max\{$MATCH$(v_1, v_2)$, DIAG$(v_1, v_2)\}$
8   return MAAC $(r(T_1), r(T_2))$
```

Analysis: The algorithm is a dynamic program based on the recursive recurrence relation on all pairs of nodes from T_1 and T_2.

Theorem 1.

$$MAAC(v_1, v_2) =$$
$$\begin{cases} L(v_1) \cap L(v_2) & , \textit{if } v_1,\ v_2 \in L \\ \max\Big\{ \max_{x\in c(v_1)}\{MAAC(x, v_2)\}, \max_{y\in c(v_2)}\{MAAC(v_1, y)\}, \mathcal{U}(MWBM)(G_{v1,v2}) \Big\} & , \textit{otherwise} \end{cases} \quad (2.1)$$

where $G_{v_1,v2}$ is a weighted complete bipartite graph with bipartitions $(c(v_1), c(v_2))$ where the weight of the edge (x, y) is the number of leaves in $MAAC(T_1(x), T_2(y))$ and $MWBM(G_{v1,v2})$ is the maximum weighted bipartite matching of G_{v_1,v_2}. Also, the operation $\mathcal{U}$ on the set of pairs of node $\{(v_1^1, v_2^2), \ldots, (v_1^k, v_2^k\}$ is the construction of the tree with a new root s such that $MAAC(v_1^i, v_2^i)$ is the i^{th} children of s.

For completeness, I present my proof of equation 2.1. Note that this is not the original proof presented in [GGJ+05].

Proof. We proceed by induction.

- **Initial Step:** If both v_1 and v_2 are leaves, the MAAC of them is either a singleton tree of root v_1 if $v_1 = v_2$ or an empty tree.
- **Induction Hypothesis:** To calculate the MAAC(v_1, v_2), assume that we have computed:
  - MAAC(x, y) with $x \in c(v_1)$ and $y \in c(v_2)$.
  - MAAC(x, v_2) with $x \in c(v_1)$.
  - MAAC(v_1, y) with $y \in c(v_2)$.
- **Inductive Step:** Because of the definition of MAAC and the existence of MAAC(v_1, v_2) there exits the leaf sets $\lambda_1 \in L(v_1)$ and $\lambda_2 \in L(v_2)$ such that the restriction of $T_1(v_1)$ on λ_1 and $T_2(v_2)$ on λ_2 are isomorphic. Consider the following cases:
  1. $\lambda_1 \in L(x)$ *with x is a child of v_1:* The restriction of $T_1(v_1)$ on λ_1 is exactly the restriction of $T_1(x)$ on λ_1. Therefore MAAC(v_1, v_2) = MAAC(x, v_2).
  2. $\lambda_2 \in L(y)$ *with y is a child of v_2:* The restriction of $T_2(v_2)$ on λ_2 is exactly the restriction of $T_2(y)$ on λ_2. Therefore MAAC(v_1, v_2) = MAAC(v_1, y).

3. *Otherwise:* The isomorphism between the two restrictions matches v_1 to v_2. Consider the sets $c(v_1) = \{v_1^1, v_1^2, \ldots, v_1^\alpha\}$ and $c(v_2) = \{v_2^1, v_2^2, \ldots, v_2^\beta\}$. We define the two sets $S_1 = \{\lambda_1(v_1^i) = \lambda_1 \cap c(v_1^i) \text{ for } 1 \leq i \leq \alpha\}$ and $S_2 = \{\lambda_2(v_2^i) = \lambda_2 \cap c(v_2^i) \text{ for } 1 \leq i \leq \beta\}$. Because there is an isomorphism between the two restrictions, there must exist a one-to-one matching m from elements S_1 and S_2. Hence $\text{MAAC}(v_1, v_2) = \mathcal{U}_{x \in c(v_1)}\{\text{MAAC}(x, m(x))\}$. It is easy to see that m is one weighted matching of G_{v_1,v_2}. And since the λ_1 and λ_2 are of maximum cardinality, m is the $\text{MWBM}(G_{v_1,v_2})$.

The $\text{MAAC}(v_1, v_2)$ would be the maximum value in all cases. Hence,

$$\text{MAAC}(v_1, v_2) = \max\left\{ \max_{x \in c(v_1)}\{\text{MAAC}(x, v_2)\}, \max_{y \in c(v_2)}\{\text{MAAC}(v_1, y)\}, \mathcal{U}(\text{MWBM})(G_{v1,v2})\right\}$$

□

The correctness of the algorithm is followed from the correctness of the equation 2.1.

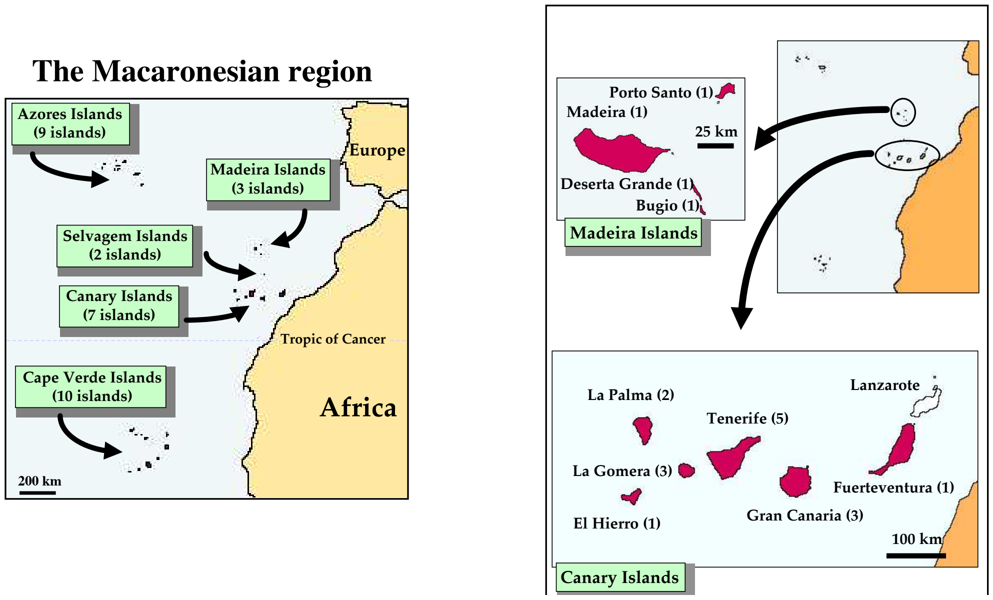
Running-time Analysis ([GGJ$^+$05]). The MWBM computation is performed using the Gabow-Tarjan algorithm [GT89]. This algorithm runs in $O(\sqrt{|V|}|E|\log|V|)$ on a graph $G = (V, E)$. Therefore, the running time of the algorithm is:

$$\begin{aligned}
\text{time}_{MAAC}(n_1,\ n_2) &= \sum_{u \in T_1}\sum_{v \in T_2} O(d_u d_v \sqrt{(d_u + d_v)}\log(d_u + d_v)) \\
&= O(\sqrt{(n_1 + n_2)}\log(n_1 + n_2)) \sum_{u \in T_1} d_u \sum_{v \in T_2} d_v \\
&= O(\ n_1 n_2 \sqrt{(n_1 + n_2)}\log(n_1 + n_2)) \\
&= O(n^{2.5}\log n)
\end{aligned}$$

2.3 Implementation

We have implemented the basic MAAC algorithm in a JAVA package. The code uses the Hungarian algorithm to solve the MWBM sub-problem. We have cooperated with biologists (Dr. Robert Jansen and Barbara Goodson) to run the program on area cladograms containing areas of Canary islands(Figure 2.3).

Figure 2.3: The map of Canary islands

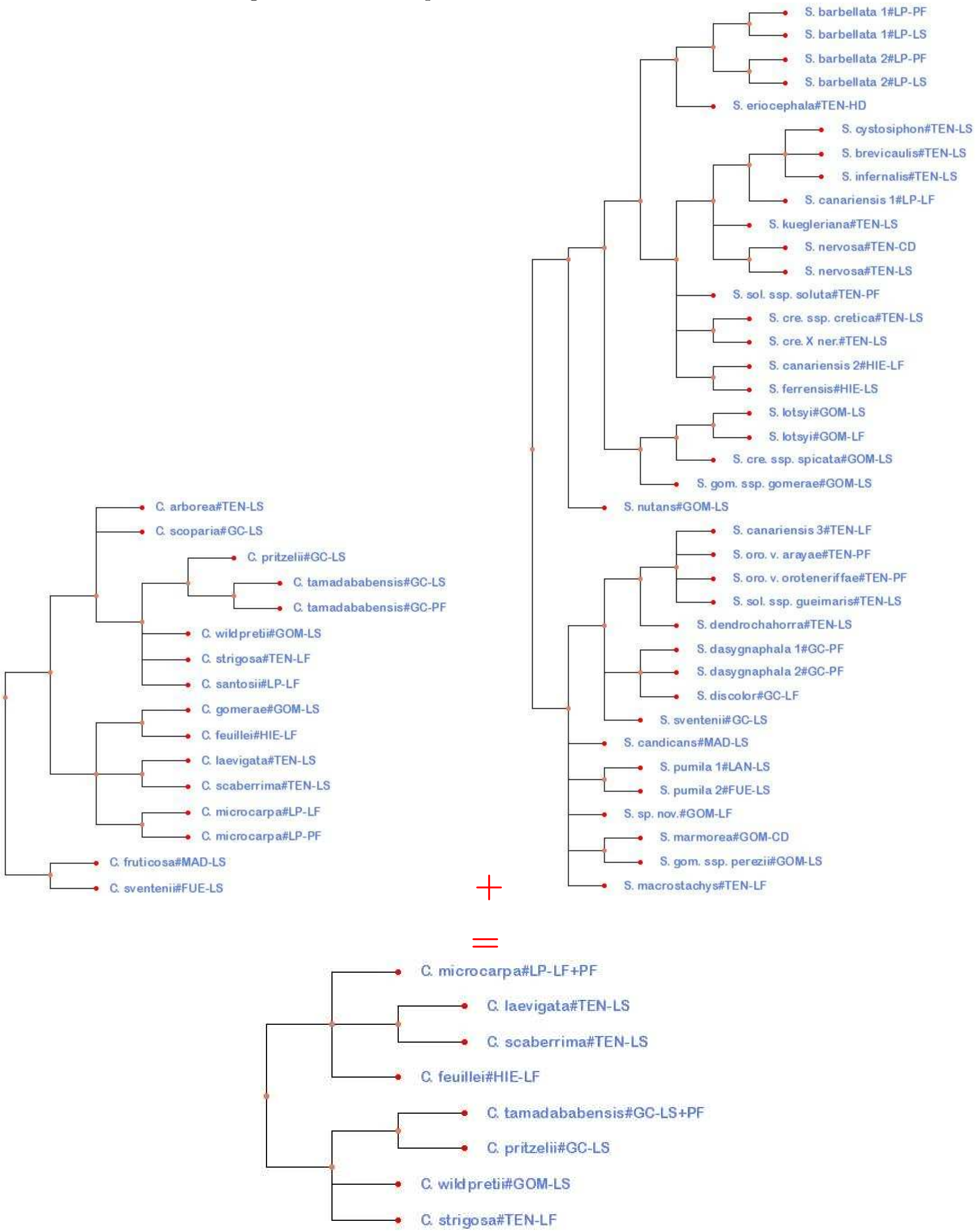


Two area cladograms for two families of organisms in the dataset and their MAAC are shown in Figure 2.4.

2.4 Sparse Dynamic Program for MAAC

The MAAC algorithm given in Section 2.2 spends most of its time computing maximum weighted bipartite matchings in complete bipartite graphs, where the weight of each edge in the bipartite graph represents the size of a MAAC between some pair of rooted subtrees. For the MAST problem a faster version of this algorithm is presented in [FCT94] by Farach and Thorup. The speed-up is achieved by eliminating many edges in many of the bipartite graphs constructed by the algorithm. In particular, observe that if two subtrees do not share any leaf label, the size of their MAAC is zero. Thus, if there are only a few leaves with any given label, it is highly likely that many edge weights are zero. Further, it turns out that several edges can be deleted in many of the bipartite graphs without affecting the optimum solution. Farach and Thorup's sparse dynamic programming algorithm for MAST incorporates these feature into the Steel-Warnow algorithm, thereby achieving

Figure 2.4: An example of data in Section 2.3



a running time of $O(n^2)$ [FCT94].

In this section we adapt the Farach-Thorup MAST method [FCT94] to MAAC. As opposed to Steel-Warnow MAST method, this method could not readily apply to the MAAC problem because it assumes the uniqueness of leaf labels of the input trees. We show that as long as the number of leaves with any given label is $O(n^{1/2-\epsilon})$, the algorithm runs in $O(n^2)$ time, which matches the bound in [FCT94] for MAST, where there is only one leaf with a given label in each tree. The worst case running time of our algorithm, however, is $O(n^{2.5} \log n)$, matching that of the straightforward dynamic programming algorithm given in the previous section.

Key Lemmas. The following discussion uses notation from the straightforward MAAC algorithm given in the previous section: for a given node v in a rooted tree T, we let $A(v)$ be the set of all labels of leaves that descend from v, $p(v)$ be the parent of v, and $c(v)$ be the set of children of v. For a given rooted tree T, we let $V(T)$ be the set of all nodes of a tree. Finally, let $\pi_{i,j}$ be the number of leaves labeled with area a_i in tree T_j.

For each internal node v of T_1 and T_2, among all children of v, we choose the child having the most number of descendent leaves to be the "heavy" child, and all the remaining children are "light" children. If there are many nodes that have the same maximum number of descendants, we designate one of them as the heavy child arbitrarily. A node is "heavy" if it is the heavy child of its parent, and otherwise it is "light".

For vertices $u_1 \in T_1$, $u_2 \in T_2$, consider the weighted bipartite graph G_{u_1,u_2} constructed by the basic MAAC algorithm (see Section 2.2). In G_{u_1,u_2} we will let h_1 and h_2 denote the heavy child of u_1 in T_1 and the heavy child of u_2 in T_2 respectively. An edge in G_{u_1,u_2} will be called "heavy-heavy" if it is between h_1 and h_2; similarly we will refer to "heavy-light" and "light-light" edges.

We will denote by $\mathcal{M}$ the set of all bipartite graphs encountered throughout the course of the algorithm. Also from now on, we will assume that we have modified all the bipartite graphs in $\mathcal{M}$ to get rid of all zero-weight edges.

We first bound the total number of light-light edges across all the bipartite graphs in $\mathcal{M}$ in Lemmas 1 and 2. In Lemma 3 we show how to delete most of the heavy-light edges in each bipartite graph in $\mathcal{M}$ without affecting the value of the MWBM solution. Thus we create for each bipartite graph G in $\mathcal{M}$ a bipartite graph G' with a fewer number of edges. We will call the set of all such reduced bipartite graphs $\mathcal{M}'$. All MWBM computations are performed only on these reduced bipartite graphs in $\mathcal{M}'$. Finally in Lemma 4 we bound the total number of edges across all bipartite graphs in $\mathcal{M}'$, and this helps us bound the total running time of the algorithm.

Lemma 1. *Each leaf in trees T_1 and T_2 has $O(\log n)$ ancestors that are light nodes.*

Proof. Consider a leaf l in T_1. Let r be the root of T_1. Suppose that l has more than $\log_2(n)$ ancestors that are light nodes. It is easy to see that if a node v is the light child of $p(v)$ then $|L(p(v))| \geq 2|L(v)|$. Thus if a node has more than $\log_2(n)$ ancestors that are light, we would have $|L(r)| > 2^{\log_2(n)}$ or $|L(r)| > n$, which is a contradiction. Therefore, l has at most $\log_2(n)$ light ancestors. The same argument holds for a leaf l in T_2. $\square$

Lemma 2. *Across all bipartite graphs in $\mathcal{M}$, the total number of light-light edges is $O\left(\left(\sum_{a_i \in \mathcal{A}} \pi_{i,1}\pi_{i,2}\right) \log^2 n\right)$.*

Proof. The weight of an edge (x, y) is non-zero if and only if the two sets of descendent leaves $L(x)$ and $L(y)$ intersect. Consider a label $a_i \in \mathcal{A}$, and let S_1 and S_2, respectively, be the sets of leaves of T_1 and T_2 which are labeled with a_i. Note that $|S_j| = \pi_{i,j}, j = 1, 2$. A light ancestor of a pair of leaves, one in S_1 and one in S_2, accounts for one light-light edge across all graphs in $\mathcal{M}$. By Lemma 1, there are $O(\pi_{i,j} \log n)$ ancestors of elements of S_j, $j = 1, 2$, that are light. Therefore, there are at most $O(\pi_{i,1}\pi_{i,2} \log^2 n)$ light-light edges produced by elements of S_1 and S_2. Summing the quantity over all labels a_i, we get the desired upper bound on the number of light-light edges. $\square$

Lemma 3. *For each bipartite graph $G = (V, E)$ in $\mathcal{M}$ with α light-light edges, we can reduce the number of edges in G to get $G' = (V, E')$ such that $MWBM(G) = MWBM(G')$ and $|E'| \leq 3\alpha + 3$.*

Proof. Let V_1 and V_2 be the two parts of V and let h_1 and h_2 be the heavy nodes in V_1 and V_2 respectively. Let E^* be the set of light-light edges, with $|E^*| = \alpha$. We partition the sets $V_1 \setminus \{h_1\}$ and $V_2 \setminus \{h2\}$ into two disjoint subsets as follows: $V_1 \setminus \{h_1\} = V_1^\alpha \cup V_1^\beta$ and $V_2 \setminus \{h_2\} = V_2^\alpha \cup V_2^\beta$ such that $v \in V_j^\alpha$ iff there exists no edge $e \in E^*$ such that v is in e. Among all edges connecting h_1 and an element of V_2^α, we can delete all except the heaviest edge since no maximum matching can contain them. The same reasoning applies for h_2 and an element of V_1^α. So we can construct a new graph G' with the same set of vertices: the new set of edges E' contains α light-light edges; one possible edge between h_1 and h_2; two possible edges between h_1 and V_2^α, and h_2 and V_1^α; and at most $|V_1^\beta| + |V_2^\beta| \leq 2\alpha$ edges between h_1 and V_2^β, and h_2 and V_1^β. Therefore, $|E'| \leq 3\alpha + 3$ or the graph G' contains at most $3\alpha + 3$ edges. $\square$

We now present SP-MAAC, our sparse dynamic programming algorithm for the MAAC problem. The differences between this algorithm and the earlier MAAC algorithm are italicized.

```
SP-MATCH($v, w$)
1  Construct $G_{v,w}$
2  *Remove all zero-weight edges from G*
3  *For each heavy child, remove all edges incident to it except for the heaviest one.*
4  Construct $E_0 = \text{MWBM}(G_{v,w})$
5  Let $E_0 = \{(v_1, w_1), (v_2, w_2), \ldots, (v_k, w_k)\}$
6  Construct tree $M$ with root $s$ such that SP-MAAC($T_{v_i}, T_{w_i}$) is the $i$-th child of $s$.
7  **return** $M$
```

```
DIAG($v, w$)
1  $t_1 \leftarrow$ largest SP-MAAC($T_v, T_x$) such that $x \in c(w)$
2  $t_2 \leftarrow$ largest SP-MAAC($T_y, T_w$) such that $y \in c(v)$
3  **return** the larger of $t_1$ and $t_2$
```

```
ALGORITHM SP-MAAC($T_1, T_2$)
1  *Choose a heavy child for each internal node of $T_1$ and $T_2$*
2  Let $\mathcal{O}$ be an ordering of $V(T_1) \times V(T_2)$
3     such that if $(v_1, w_1)$ is before $(v_2, w_2)$,
4     then $v_1$ is not an ancestor of $v_2$ and $w_1$ is not an ancestor of $w_2$.
5  **for** $(v, w)$ in increasing order of $\mathcal{O}$
6        **do if** $v$ or $w$ is a leaf
7              **then** SP-MAAC($T_v, T_w$) $\leftarrow$ a node with label in $S = A(v) \cap A(w)$ if $S \neq \emptyset$; else $\emptyset$
8              **else** SP-MAAC($T_v, T_w$) $\leftarrow$ larger of SP-MATCH($v, w$) and DIAG($v, w$)
9  **return** SP-MAAC($T_{r_1}, T_{r_2}$) ; $r_1$ is the root of $T_1$ and $r_2$ is the root of $T_2$.
```

Lemma 4. *Across all graphs in $\mathcal{M}'$, the total number of edges is $O\big(\min\big\{(\sum_{a_i \in \mathcal{A}} \pi_{i,1}\pi_{i,2}) \log^2 n, n^2\big\}\big)$.*

Proof. By lemmas 2 and 3, the total number of edges across all graphs in $\mathcal{M}'$ is $O((\sum_{a_i \in \mathcal{A}} \pi_{i,1}\pi_{i,2}) \log^2 n)$. This summation can be shown to be $\Omega(n^2 \log^2 n)$ in worst case. However, the total number of edges is at most the number of edges in all complete bipartite graphs, which is

$$\begin{aligned}\sum_{x\in V(T_1)}\sum_{y\in V(T_2)} |c(x)|\cdot|c(y)| &= O(n_1 n_2)\\ &= O(n^2)\end{aligned}$$

Hence, the total number of edges across all graphs in $\mathcal{M}'$ is $O\big(\min\big\{(\sum_{a_i\in\mathcal{A}}\pi_{i,1}\pi_{i,2})\log^2 n, n^2\big\}\big)$.

□

As presented, the algorithm uses $O(n^2)$ time to remove all zero-weight edges from the bipartite graphs. However, it is not difficult to maintain $|V(T_2)|$ queues of non-zero edges incident to each internal node of T_2 and update this queue as we compute the MAAC in the ordering of $\mathcal{O}$.

Running-time Analysis.

Theorem 2. *Algorithm SP-MAAC computes the MAAC in*

$$O\big((\sqrt{n}\log n)\min\big\{(\sum_{a_i\in\mathcal{A}}\pi_{i,1}\pi_{i,2})\log^2 n, n^2\big\} + n^2\big)$$

Proof. Let the total running time of SP-MAAC be $time_{\text{SP-MAAC}}$. The algorithm spends a total of $O(n)$ time in step 1 choosing a heavy child for each node, and it spends a total of $O(n^2)$ time computing the ordering $\mathcal{O}$. Each call to DIAG$(v,\ w)$ takes $O(|c(v)|+|c(w)|)$ time. Over all the calls to DIAG, the total running time is therefore $O(n^2)$. Let the time spent in all calls to SP-MATCH be $time_{\text{SP-MATCH}}$. Therefore,

$$\text{time}_{\text{SP-MAAC}} = O(n^2) + \text{time}_{\text{SP-MATCH}}$$

We now show how to bound $time_{\text{SP-MATCH}}$. We let $time_{\text{MWBM}}$ be the running time of a single call to procedure MWBM in SP-MATCH. The MWBM computation is performed using the Gabow-Tarjan algorithm [GT89]. This algorithm runs in $O(\sqrt{|V|}|E|\log|V|)$ on a graph $G=(V,E)$. We have:

$$\begin{aligned}
\text{time}_{\text{SP-MATCH}} &= \sum_{G=(V,E)\in\mathcal{M}'} \text{time}_{\text{MWBM}}(G) \\
&= \sum_{G=(V,E)\in\mathcal{M}'} O(\sqrt{|V|}|E|\log|V|) \\
&= \sum_{G=(V,E)\in\mathcal{M}'} O(\sqrt{n}|E|\log n) \\
&= O(\sqrt{n}\log n \sum_{G=(V,E)\in\mathcal{M}'} |E|) \\
&= O\big((\sqrt{n}\log n)\min\big\{(\sum_{a_i\in\mathcal{A}} \pi_{i,1}\pi_{i,2})\log^2 n, n^2\big\}\big) \quad \text{from Lemma 4}
\end{aligned}$$

Hence,

$$\text{time}_{\text{SP-MAAC}} = O\big((\sqrt{n}\log n)\min\big\{(\sum_{a_i\in\mathcal{A}} \pi_{i,1}\pi_{i,2})\log^2 n, n^2\big\} + n^2\big)$$

□

Lemma 5. *If every $\pi_{i,j}$ is $O(n^{1/2-\epsilon})$, then time$_{SP\text{-}MAAC} = O(n^2)$.*

Proof.

$$\begin{aligned}
\sum_{a_i\in\mathcal{A}} \pi_{i,1}\pi_{i,2} &\le \frac{1}{2}\sum_{a_i\in\mathcal{A}} \pi_{i,1}^2 + \pi_{i,2}^2 \\
&= \frac{1}{2}(\sum_{a_i\in\mathcal{A}} \pi_{i,1}^2 + \sum_{a_i\in\mathcal{A}} \pi_{i,2}^2)
\end{aligned}$$

For any $a, b \ge 0$, $(a+b)^2 \ge a^2 + b^2$. Hence if every $\pi_{i,j}$ is $O(n^{1/2-\epsilon})$, then for $j = 1, 2$

$$\begin{aligned}
\sum_{a_i\in\mathcal{A}} \pi_{i,j}^2 &\le \lceil \frac{n}{n^{1/2-\epsilon}} \rceil (O(n^{1/2-\epsilon}))^2 \\
&= O(n^{3/2-\epsilon})
\end{aligned}$$

Together,

$$\sum_{a_i \in \mathcal{A}} \pi_{i,1}\pi_{i,2} = O(n^{3/2-\epsilon})$$

Therefore, by theorem 2, $time_{\text{SP-MAAC}} = O(n^2)$. □

Finally, we summarize the following two bounds for the running time of the algorithm.

- If every $\pi_{i,j}$ is $O(n^{1/2-\epsilon})$, then $time_{\text{SP-MAAC}} = O(n^2)$. Thus, as long as no leaf label occurs a huge number of times, the algorithm is as efficient as the Farach-Thorup MAST algorithm, where it is assumed that every leaf label occurs exactly once.
- In the worst case, the running time $time_{\text{SP-MAAC}}$ remains $O(n^{2.5}\log n)$, matching the time bound of the basic MAAC algorithm given in the previous section.

2.5 MAAC for k Trees.

In this section, we study the complexity of computing the MAAC of many area-labeled trees. Amir and Keselman [AK97] show that computing the MAST of just three trees with unbounded degrees is NP-hard. They present a reduction from the 3-Dimensional Matching problem (3DM) to 3-HUT, which is the decision problem version of the MAST problem on three unbounded degree trees (the Homeomorphic Agreement Subtree of three unbounded degree Trees). Since the MAAC problem is a less restricted version of the MAST problem, computing the MAAC of more than three unbounded degree trees is also NP-hard.

We prove a new result. We show that computing the MAAC of a set of k binary trees is NP-hard (recall that the corresponding problem for MAST is solvable in polynomial time). Therefore, it seems that most natural generalizations of the approaches used in computing the MAST of two trees and k trees with maximum degree d (in [AK97] and [FCPT95]) would run in time exponential in both k and d.

NP-completeness of k-tree MAAC. The proof of NP-completeness will be by reduction of the VERTEX-COVER problem. The proof ideas come from the NP-completeness proof of the Largest

Common Subsequences (LCS) problem presented by David Maier in [Mai78]. We will use the following description of VERTEX-COVER.

- VERTEX-COVER
- Input: Graph $G = (V, E)$ and an integer k
- Question: is there a subset $S \subseteq V$ of at most k vertices, such that for every edge $e = (x, y) \in E$, $\{x, y\} \cap S \neq \emptyset$?

We state here a decision problem of computing the MAAC of many binary area-labeled trees, which we call BIN-MAAC.

- BIN-MAAC
- Input: set $\mathcal{T}$ of binary area-labeled trees, and an integer k.
- Question: is $|\text{MAAC}(\mathcal{T})| \geq k$?

Theorem 2.5.1. *BIN-MAAC is NP-complete.*

Proof. BIN-MAAC is in NP since a naive algorithm can simply guess a subset of leaves of each tree in $\mathcal{T}$, and check if all induced trees are isomorphic in polynomial time. Hence it will suffice to show that VERTEX-COVER reduces to BIN-MAAC.

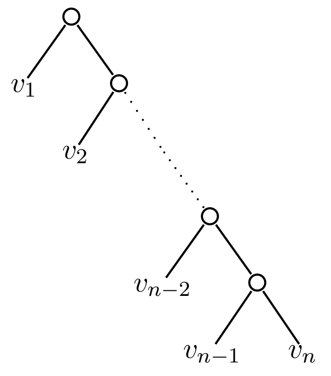
Consider an instance $(G = (V, E), k)$ of VERTEX-COVER. Let $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$. We will construct a set $\mathcal{T} = \{T_0, T_1, \ldots, T_m\}$ of $(m + 1)$ binary area-labeled trees such that G has a vertex cover of size k if and only if $|\text{MAAC}(\mathcal{T})| \geq n - k$.

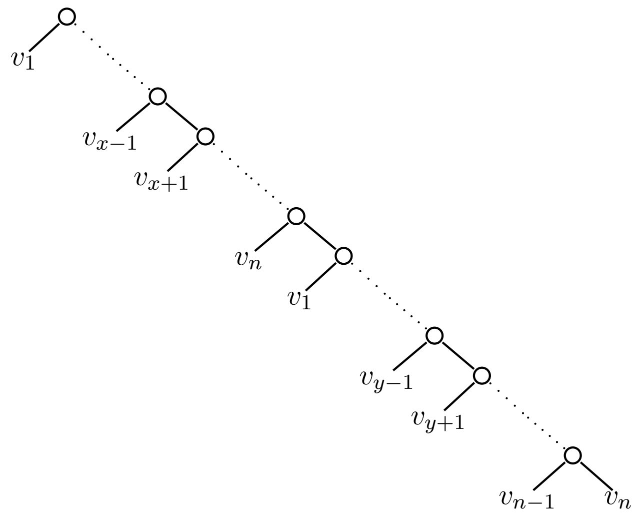
The set of areas with which the leaves of tree are labeled is $\mathcal{A} = \{v_1, v_2, \ldots, v_n\}$.

The tree T_0 is a binary tree on leaf set $v_1, v_2, \ldots, v_n$, with no non-trivial left subtrees. Thus, T_0 is a rooted "caterpillar" tree defined by the ordering on its leaves, which we will assume is given by $v_1, v_2, \ldots, v_n$. We use the notation $T_0 \setminus X$ to denote the tree obtained by deleting the leaves in X from the tree T_0, and suppressing nodes with only one child.

Now consider an edge $e = (v_x, v_y)$ in the graph G. We will define the tree T_e as follows. T_e is obtained by "concatenating" the trees $T_0 \setminus \{v_x\}$ and $T_0 \setminus \{v_y\}$, where by "concatenation" we mean replacing the deepest leaf of the first tree by a branching node whose children are the second tree and the old leaf. Figure 2.5 illustrates this construction.

Figure 2.5: Tree T_0 and T_e corresponding to $e = (v_x, v_y)$





Now, we show that G has a vertex cover of size k if and only if $|\text{MAAC}(T)| \geq n - k$:

- (only if:) Suppose G has a vertex cover S of size k. Let $T^* = T_0 \setminus S$. It is clear that $|L(T^*)| = n - k$. Therefore it is enough to show that T^* is an agreement subtree. Obviously T^* is a subtree of T_0. For each tree T_{e_i} corresponding to the edge $e_i = (v_x, v_y), x < y$, which is the concatenation of $T_0 \setminus \{v_x\}$ and $T_0 \setminus \{v_y\}$, T^* is either a subtree of the first upper half or the second lower half of T_i because either v_x or v_y is in S, hence both of them cannot be labels of leaves of T^*.

- (if:) If $|\text{MAAC}(\mathcal{T})| \geq n-k$, then let $S = V \setminus L(\text{ MAAC}(\mathcal{T}))$. Because $\text{MAAC}(\mathcal{T})$ is a subtree of T_0, every v_i labels only one leaf. That implies $|S| = |V| - |L(\text{MAAC}(\mathcal{T})| \leq k$. It is sufficient to show that S is a vertex cover of G in order for G to have a vertex cover of size k. Consider any edge $e_i = (v_x, v_y), x < y$, and assume that both vertices are not in S, or that both are labels of leaves of $\text{MAAC}(\mathcal{T})$. In the trees T_0 and T_i, there is only one instance of label v_x and one instance of label v_y. Because of the structure of T_0, the leaf labeled v_x is above the leaf labeled v_y. However, in T_i, the leaf labeled v_x is strictly below the leaf labeled v_y. By that reason, both labels cannot coexist in $L(\text{MAAC}(\mathcal{T}))$.

  This contradicts the initial assumption. Therefore, for every edge e, S contains at least one of its vertices. In other words, S is a vertex cover of G.

□

Chapter 3

Median Alignment of Three Sequences: Theory

3.1 Preliminaries

Multiple sequence alignment is a crucial step in phylogenetic analysis to infer relationships of a set of organisms. In sequence alignment, the task is to place the sequences into a data matrix where the rows represent the sequences and the columns represent positional homology, which implies the evolution of these sequences from a common ancestor. Evolution can happen from deletion,insertion and substitution events. Insertion and deletion events are called indels. The matrix represents this evolution process and therefore is an important step in phylogenetic analysis. Figure 3.1 shows an example of how the three related sequences are placed into the the data matrix.

Figure 3.1: An example of three sequences alignment

Sequences	Data matrix
`CATTTCCTAGAA`	`CATTTCCTAGAA-`
`ATATCAGAGG`	`-ATATC--AGAGG`
`ATCGCTTAA`	`-ATCGC--TTAA-`

The evolution model has three operations: insertion, deletion and substitution. An edit transformation from a sequence A to a sequence B is a sequence of operations transforming A to B. The cost of a transformation is calculated based on the cost of each operation and a function to score the indels. In phylogenetic analysis, the relationship between two sequences A and B is expressed by the edit distance, which is a minimum cost (or optimal) transformation from A to B.

For the sequence alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ of size k, the cost of substitution is described by a $k \times k$ matrix M, where $M[a_i, a_j]$ is the cost of substituting a_i with a_j. Normally, the diagonal of the matrix, i.e entries $M[a_i, a_i]$ are 0. For the events of insertion and deletions, there are two common models to score the alignment: the linear and affine gap model. In the linear gap model, each insertion and deletion incurs one constant cost c. Hence, the total cost for a transformation with l insertions and deletions is $G(l) = c \times l$. The affine gap model considers the relative positions of the indels in the matrix and more realistically models the evolution. The cost of indels is based on a gap introduction cost ($g_i > 0$) and a gap extension cost ($g_e > 0$), giving the total cost of $G(l) = g_i + g_e l$ for a gap of length l. This new cost function distinguishes long gaps from short gaps and therefore, yields more accurate alignments since it is more likely to have a long gap than many short gaps in real sequences.

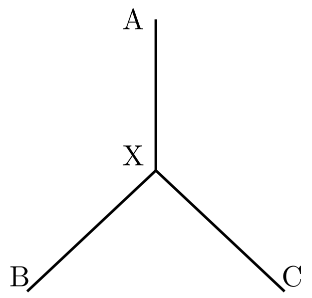
Formally, for an optimal transformation from a sequence A to a sequence B with n_g gaps of length $l_1, l_2, \ldots, l_{n_g}$, and the number of substitutions of a_i with a_j $N[a_i, a_j]$, the edit distance could be defined as:

$$d(A, B) = \sum_{i=1}^{n_g} G(l_i) + \sum_{i,j \in [1..k]} N[a_i, a_j] M[a_i, a_j]$$

For the rest of this chapter, we consider a special type of multiple alignment of *three* sequences, called the median alignment with affine gap cost. The median alignment assume the evolutionary tree connecting the sequences is a Steiner or star tree, where the sequence X at the internal node is called the median sequence(see Figure 3.2). The method computes the optimal alignment cost as well as the median sequence. The cost of the median alignment is the sum of the edit distances of the three sequences to the median sequence.

Knudsen [Knua] presented a dynamic program to find multiple alignment of N sequences, each of length n in $O(16.81^N n^N)$ time and $O(7.442^N n^N)$ space. It is also mentioned in [Knua] that the Hirschberg memory reduction technique ([Hir75]) can be used to improve the space complexity of the algorithm by a factor of n. For the median problem, this gives an $O(n^3)$ time, $O(n^2)$ space algorithm. Powell et al. [DRPD00] presented an Ukkonen-based algorithm, which performs well especially for sequences whose (three-way) edit distance d is small. On average, it requires $O(d^3 + n)$ time and $O(d^3 + n)$ space to compute the alignment [DRPD00]. In section 3.2, we describe the

Figure 3.2: Steiner tree



Knudsen's algorithm without going into the details of its correctness. Section 3.3 outlines the Hirschberg memory reduction technique and discusses the new algorithm MED-H. Finally, in the last section we show our I/O efficient algorithm MED-CO.

3.2 Knudsen's Dynamic Programming algorithm MED-Knudsen

Overview: We assume three sequences $A = a_1a_2 \dots a_{n_a}$, $B = b_1b_2 \dots b_{n_b}$ and $C = c_1c_2 \dots, c_{n_c}$ and the sequence alphabet Σ of size k, the gap introduction g_i, the gap extension cost g_e and the matching cost matrix $M[1..k, 1..k]$. Knudsen's algorithm [Knua] is a dynamic program over a three-dimensional matrix $\mathcal{C}$. The value of each entry $\mathcal{C}[i, j, k]$ of the matrix is the optimal alignment cost of the three subsequences $A[1..i]$, $B[1..j]$ and $C[1..k]$.

Residue and Indel Configurations: Recall that an alignment of the sequences is equivalent to a data matrix whose rows represent sequences and column entry is either a sequence character, called a residue or an indel. Therefore, for the median alignment we are filling a table of 4 rows correspondent to the sequences A, B, C and the median sequence X. A column configuration is called a residue configuration $e =< e_A, e_B, e_C, e_X >$ where each index could be either a 0 for an indel or an 1 for a residue. Out of 16 possible such configurations, only 10 are acceptable, due to the biological interpretation of the indels(see [Knua]).

In order to keep track of the ongoing indels, each entry of $\mathcal{C}$ has 23 fields. Each field, called an indel configuration is a triple $< I_A, I_B, I_C >$ with $I_A, I_B, I_C \in \{\mathrm{M}, \mathrm{I}, \mathrm{D}\}$. For $S \in \{A, B, C\}$, I_S represents the matching state of the sequence S with the median sequence X. M denotes that the last characters of the two sequences are matched; I denotes an insertion in the median sequence and D denotes a deletion in the median sequence. Knudsen explained in his paper why there are only 23 acceptable configurations.

In summary, the residue configuration represents the state of the next column while the indel configuration represents the ongoing matching state between input sequences and the median sequence. Table 3.1 lists the acceptable residue and indel configurations.

Table 3.1: Acceptable residue and indel configurations

Residue Configurations	Indel Configurations
Acceptable	
$<1,0,0,0>, <0,1,0,0>, <0,0,1,0>$	$<\mathrm{D,M,M}>, <\mathrm{M,D,M}>, <\mathrm{M,M,D}>$
$<0,0,1,1>, <0,1,0,1>, <1,0,0,1>$	$<\mathrm{I,M,M}>, <\mathrm{M,I,M}>, <\mathrm{M,M,I}>$
$<1,1,0,1>, <0,1,1,1>, <1,0,1,1>$	$<\mathrm{D,D,M}>, <\mathrm{D,M,D}>, <\mathrm{M,D,D}>$
$<1,1,1,1>$	$<\mathrm{I,D,M}>, <\mathrm{M,I,D}>, <\mathrm{I,M,D}>$
	$<\mathrm{D,I,M}>, <\mathrm{M,D,I}>, <\mathrm{D,M,I}>$
	$<\mathrm{I,I,M}>, <\mathrm{M,I,I}>, <\mathrm{I,M,I}>$
	$<\mathrm{D,I,D}>, <\mathrm{D,D,I}>, <\mathrm{I,D,D}>$
	$<\mathrm{D,D,D}>, <\mathrm{M,M,M}>$
Non-acceptable	
$<1,1,0,0>, <1,0,1,0>, <0,1,1,0>$	$<\mathrm{I,I,I}>, <\mathrm{I,I,D}>, <\mathrm{D,I,I}>$
$<0,0,0,1>, <1,1,1,0>, <0,0,0,0>$	$<\mathrm{I,D,I}>$

Notation:

Each residue configuration is a 4-tuple $< e_A, e_B, e_C, e_X >$, each field is :
0: an indel ; 1: a residue.
Each residue configuration is a triple $< I_A, I_B, I_C >$, where $I_S, S \in \{A, B, C\}$ is:
M: matching state ; D: deletion state ; I: insertion state

The recursion: The algorithm initializes the "empty" indel configuration $<\mathrm{M,M,M}>$ of $\mathcal{C}[0,0,0]$ to 0 while the remaining indel configurations of $\mathcal{C}[0,0,0]$ are ∞. The recursive step builds the data matrix column by column and keeps track of the ongoing indel configurations.

The recurrence relation: We define $\texttt{next}(e,d) = d'$ for applying the residue configuration $e =< e_A, e_B, e_C, e_X >$ to the indel configuration $d =< I_A, I_B, I_C >$. For $t = \{A, B, C\}$, the value of $d' =< I'_A, I'_B, I'_C >$ is shown in the table 3.2.

Hence, the recurrence relation of Knudsen's algorithm is:

$$\mathcal{C}[i,j,k]_d = \begin{cases} 0 & \text{,if } i,j,k = 0; d = d_o \\ \min\limits_{e,d' \text{ s.t. } d=\texttt{next}(e,d')} \left\{ \mathcal{C}[i',j',k']_{d'} + \mathcal{G}_{e,d} + \mathcal{M}_{(i',j',k') \to (i,j,k)} \right\} & \text{, otherwise.} \end{cases} \quad (3.1)$$

Table 3.2: Values of I'_t from e_T and e_X

e_t (0 = gap, 1 = residue)	e_X (0 = gap, 1 = residue)	I'_t
0	0	I_t
0	1	I
1	0	D
1	1	M

where $d_o =< \mathrm{M,M,M} >$, $i' = i - e_1$, $j' = j - e_2$ and $k' = k - e_3$, $\mathcal{M}_{(i',j',k')\to(i,j,k)}$ is the matching cost between characters of the sequences, and $\mathcal{G}_{e,d}$ is cost for introducing or extending the gap.

Note that both $\mathcal{M}$ and $\mathcal{G}$ do not depend on the value of $\mathcal{C}[i,j,k]_d$ but depend on e, d and d'. Hence, the values of $\mathcal{M}$ and $\mathcal{G}$ can be pre-processed before the execution of the program.

We show here our pseudocode developed based on Knudsen's description of his algorithm.

Pseudocode:

```
MED-KNUDSEN(COST)($A[1..n_a], B[1..n_b], C[1..n_c]$)
1  Pre-compute the arrays $\mathcal{M}$ and $\mathcal{G}$.
2  $\mathcal{C}[0,0,0]_{<\mathrm{M,M,M}>} \leftarrow 0; \forall d \neq < \mathrm{M,M,M} >, \mathcal{C}[0,0,0]_d \leftarrow \infty$
3  for $i \leftarrow 0$ to $n_a$
4      do for $j \leftarrow 0$ to $n_b$
5          do for $k \leftarrow 0$ to $n_c$
6              do if $i \neq 0$ or $j \neq 0$ or $k \neq 0$
7                  then Compute $\mathcal{C}[i,j,k]_d, \forall d$ based on equation 3.1.
8  return $(\delta, \mathcal{C}[n,n,n]_\delta)$ such that $\mathcal{C}[n,n,n]_\delta = \max_{\forall d} \mathcal{C}[n,n,n]_d$
```

```
MED-KNUDSEN($A[1..n_a], B[1..n_b], C[1..n_c]$)
1  $(\delta, cost) \leftarrow$ MED-KNUDSEN(COST)$(A, B, C)$
2  $i \leftarrow n_a, j \leftarrow n_b, k \leftarrow n_c$
3  $X$ (the median sequence) $\leftarrow NIL$
4  while $i \neq 0$ or $j \neq 0$ or $k \neq 0$
5      do Add a character to $X$ accordingly.
6         $(i,j,k,\delta) \leftarrow (i',j',k',d')$ such that $\left\{\mathcal{C}[i,j,k]_\delta = \mathcal{C}[i',j',k']_{d'} + \mathcal{G}_{e,d} + \mathcal{M}_{(i',j',k')\to(i,j,k)}\right\}$
7  return $(X, cost)$
```

The algorithm computes the matrix $\mathcal{C}$ to find the optimal alignment cost at $\mathcal{C}[n_a, n_b, n_c]$. To compute the median sequence, the algorithm traces back from $\mathcal{C}[n_a, n_b, n_c]$ to $\mathcal{C}[0, 0, 0]$ by the same recurrence relation. Therefore, Knudsen's algorithm runs in $O(n^3)$ time and $O(n^3)$ space.

3.3 Hirschberg's space reduction technique and MED-H

Quadratic-space cost computation: For just computing the optimal alignment cost, it is straight-forward to reduce the space complexity of Knudsen's algorithm to $O(n^2)$. Since the computation of $\mathcal{C}[i, j, k]$ in the Knudsen's algorithm depends on $\mathcal{C}[(i-1)..i, (j-1)..j, (k-1)..k]$. instead of a 3-dimensional array $\mathcal{C}$, we can use two 2-dimensional arrays $CC[1..n_a, 1..n_b]$ and $CC'[1..n_a, 1..n_b]$. CC' and CC entries are the $(k-1)$-surface and k-surface of the cuboid $\mathcal{C}$.

$$\begin{aligned} CC'[i,j]_d &= \mathcal{C}[i,j,k-1]_d \\ CC[i,j]_d &= \mathcal{C}[i,j,k]_d \end{aligned}$$

Hence, the equation 3.1 can be rewritten as:

$$CC[i,j]_d = \min_{e,d' \text{ s.t. } d=\texttt{next}(e,d')} \left\{ \left\{ \begin{array}{ll} CC'[i',j']_{d'} & \text{, if } e_C = 1 \\ CC[i',j']_{d'} & \text{, otherwise} \end{array} \right\} + \mathcal{G}_{e,d} + \mathcal{M}_{(i',j',k')\to(i,j,k)} \right\} \tag{3.2}$$

Given the three subsequences $A[x_a..y_a], B[x_b..y_b], C[x_c..y_c]$, the procedure Compute Cost computes the y_c surface of $\mathcal{C}$, i.e entries $\mathcal{C}[i, j, y_c], x_a - 1 \le i \le y_a, x_b - 1 \le j \le y_b$ of the cuboid space and returns $CC[0..(y_a - x_a + 1), 0..(y_b - x_b + 1)]$. The procedure also takes a value c_o as the initial value for entry $\mathcal{C}[x_a - 1, x_b - 1, x_c - 1]$. This additional flexibility will be helpful in the next discussion of MED-H.

```
Compute Cost($A[x_a..y_a], B[x_b..y_b], C[x_c..y_c], c_o$)
1  $CC[0,0] \leftarrow c_o$
2  for $k \leftarrow 0$ to $(y_c - x_c + 1)$
3        do for $i \leftarrow 0$ to $(y_a - x_a + 1)$
4              do for $j \leftarrow 0$ to $(y_b - x_b + 1)$
```

```
5                       do if $i \neq 0$ or $j \neq 0$ or $k \neq 0$
6                           then Compute $\mathcal{CC}[i,j]_d, \forall d$ based on equation 3.2.
7           Swap $CC$ and $CC'$.
8   return $CC$
```

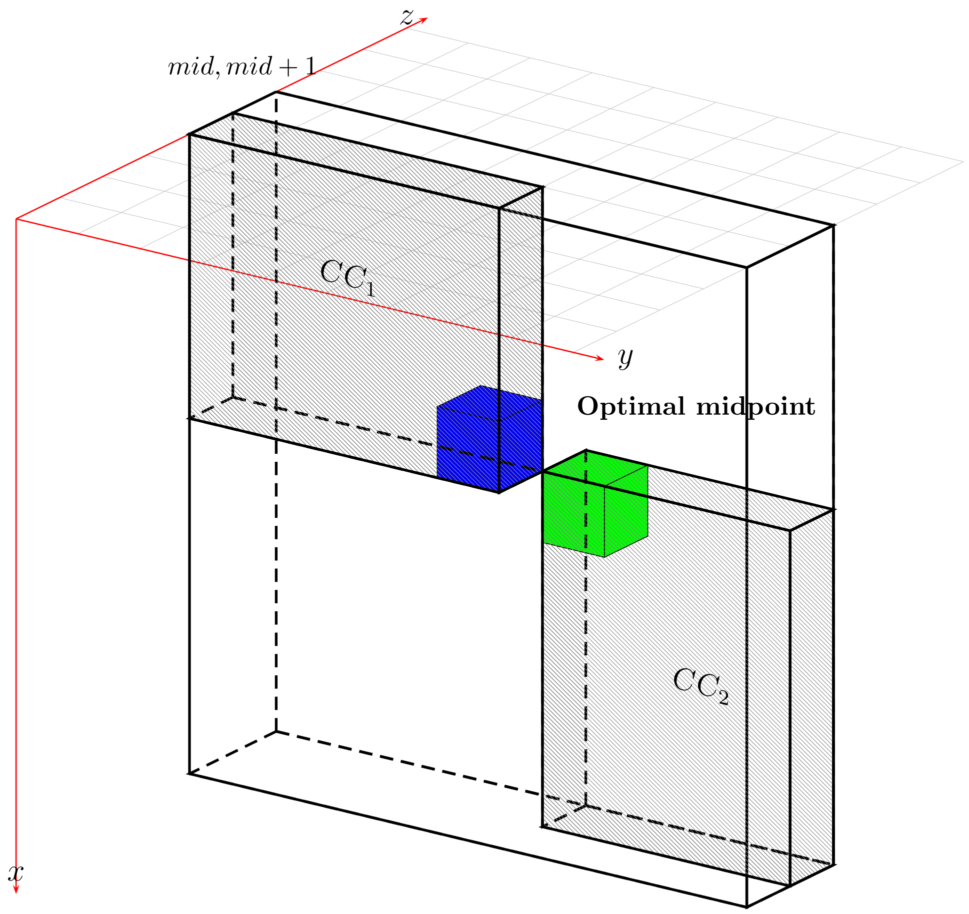
Hirschberg's technique and its adaption to MED-Knudsen: Unfortunately, with only CC and CC', it is not possible to compute the median sequence since we do not have information about the trace-path from $\mathcal{C}[n_a, n_b, n_c]$ to $\mathcal{C}[0,0,0]$. Hirschberg [Hir75] presented a recursive divide-and-conquer technique to compute the maximum common subsequence of two strings in linear space. Hirschberg's insights are that computing the score could be done in linear space easily and that the common subsequence could be broken into two parts: one part is the common subsequence of string 1 and the first half of string 2, one part is the common subsequence of string 1 the second half of string 2. However, the original formula of Hirschberg is too restrictive to apply to the context of sequence alignment directly. Myer and Miller [MM88] later took the ideas and extended it into the pairwise sequence alignment using linear space ([Got82]). The maximum slow-down of the new algorithm is at most two times the traditional algorithm. Since then, the technique has been adapted to many dynamic programs to reduce the space complexity ([CHM94], [GHS97]). Knudsen mentioned briefly in [Knua] that the technique reduces the space complexity of MED-Knudsen to $O(n^2)$. In the rest of this section, we elaborate on this idea and describe formally the equations and pseudocode of the new algorithm, called MED-H. However, we omit the formal proof of correctness.

The recurrence relation for MED-H:

MED-H is a recursive algorithm, whose recursive step calls the procedure COMPUTE COST twice to compute the optimal cost for the "forward" and "backward" halves of the cuboid space $\mathcal{C}$, and calculates the optimal alignment from the intersection of the two outputs.

More formally, suppose we compute the alignment of $A[x_a..y_a]$, $B[x_b..y_b]$ and $C[x_c..y_c]$ starting with indel d_1 and ending with indel d_2. For a sequence S, let S^* be the reverse of S and $S[i..j]$ be the subsequence of S if $i < j$ or the empty sequence if $i \geq j$. Let $mid = (x_c + y_c)/2$. Also let $F_{\alpha,\beta}(i,j,k)$ be the optimal cost to align $\{A[x_a..i]$, $B[x_b..j]$ and $C[x_c..k]\}$ starting with indel α and ending with indel β. and let $F^*_{\alpha,\beta}(i,j,k)$ be the optimal cost to align $\{A^*[i..y_a]$, $B^*[j..y_b]$ and $C^*[k..y_c]\}$ starting with indel α and ending with indel β, Therefore, we want to find $F_{d1,d2}(y_a, y_b, y_c)$.

Figure 3.3: CC_1, CC_2 arrays and the optimal midpoint mentioned in Observations 1 and 2.



In the "forward" part, we compute optimal cost alignment of $\{A[x_a..y_a], B[x_b..y_b], C[x_c..mid]\}$ starting at indel configuration d_1 with the procedure COMPUTE COST. The entries of the output CC_1 are:

$$CC_1[i, j] =< F_{d_1,\beta}(x_a+i-1, x_b+j-1, mid),\ \forall\beta > \quad 0 \le i \le y_a - x_a + 1, 0 \le j \le y_b - x_b + 1 \quad (3.3)$$

Similarly, optimal alignment cost of $\{A^*[x_a..y_a], B^*[x_b..y_b], C^*[(mid+1)..y_c]\}$ starting at indel d_2 is obtained in CC_2 in the "backward" part. The entries of the output CC_2 are:

$$CC_2[i, j] =< F^*_{d_2,\beta}(y_a-i+1, y_b-j+1, mid+1),\ \forall\beta >, \quad 0 \le i \le y_a - x_a + 1, 0 \le j \le y_b - x_b + 1 \quad (3.4)$$

Given the CC_1 and CC_2, the overall optimal cost can be computed by the following two observations.

Observation 1:

An alignment $A[x_a..y_a]$, $B[x_b..y_b]$ and $C[x_c..y_c]$ starting with indel d_1 and ending with indel d_2 can be broken down (for some $(x_a - 1) \leq i \leq y_a, (x_b - 1) \leq j \leq y_b$) into two alignments of $\{A[x_a..i], B[x_b..j], C[x_c..mid]\}$ starting at indel d_1 and $\{A^*[i+1..y_a], B^*[j+1..y_b], C^*[mid+1..y_c]\}$ starting at indel d_2.

Moreover, the alignment cost is $F_{d1,d2}(y_a, y_b, y_c)$, as given in Observation 2 below.

Observation 2:

$$\begin{gathered} F_{d1,d2}(y_a, y_b, y_c) = \\ \min_{(x_a-1)\leq i\leq y_a,(x_b-1)\leq j\leq y_b,\beta_1,\beta_2} \left\{ F_{d1,\beta_1}(i, j, mid) + F^*_{d2,\beta_2}(i+1, j+1, mid+1) + \texttt{indel-diff}(\beta_1, \beta_2) \right\} \end{gathered} \tag{3.5}$$

where `indel-diff` is defined in table 3.3.

Table 3.3: $\texttt{indel-diff}(d_1, d_2)$

$\texttt{indel-diff}(d_1 =< I_1, I_2, I_3 >, d_2 =< I'_1, I'_2, I'_3 >) = \sum_{i=1}^{3} \texttt{diff}(I_i, I'_i)$

I_i	I'_i	$\texttt{diff}(I_i, I'_i)$
I	I	$g_e - g_i$
I	D	0
I	M	0
D	D	$g_e - g_i$
D	M	0
M	M	0

The justification for `indel-diff` is that the gap introduction is counted twice in the sum and one gap extension cost needed to be added in the case that both alignment ends with a gap.

From equations 3.3, 3.4 and 3.5 we have:

$$\begin{gathered} F_{d1,d2}(y_a, y_b, y_c) = \\ \min_{0\leq i\leq (y_a-x_a+1),0\leq j\leq (y_b-x_b+1),\beta_1,\beta_2} \left\{ CC_{1,\beta_1}(i, j) + CC_{2,\beta_2}(y_a - i + 1, y_b - j + 1) + \texttt{indel-diff}(\beta_1, \beta_2) \right\} \end{gathered} \tag{3.6}$$

Pseudocode We are ready to show the pseudo-code of MED-H.

MED-H(HELPER)$(A[x_a..y_a], B[x_b..y_b], C[x_c..y_c], c_o, c'_o)$

```
1  if $y_c = x_c$
2     then $CC_1 \leftarrow$ Compute Cost$(A[x_a..y_a], B[x_b..y_b], C[x_c..y_c], c_o)$
3          $X \leftarrow$ nil
4          Compute $X$ by tracing from $CC_1[y_a - x_a + 1 y_b - x_b + 1]$ back to $c_o$.
5  $mid \leftarrow (x_c + y_c)/2$
6  $CC_1 \leftarrow$ Compute Cost$(A[x_a..y_a], B[x_b..y_b], C[x_c..mid], c_o)$
7  $CC_2 \leftarrow$ Compute Cost$(A^*[x_a..y_a], B^*[x_b..y_b], C^*[mid + 1..y_c], c_o')$
8  $cost \leftarrow \infty, ni \leftarrow 0, nj \leftarrow 0$
9  for $i \leftarrow 0$ to $(y_a - x_a + 1)$
10       do for $j \leftarrow 0$ to $(y_b - x_b + 1)$
11             do for All indels $d_1$
12                   do for All indels $d_2$
13                         do $tmp \leftarrow CC_1[i, j] + CC_2[y_a - i + 1, y_b - j + 1] +$
14                               indel-diff$(d_1, d_2)$
15                            if $tmp < cost$
16                              then $cost \leftarrow tmp, ni \leftarrow i, nj \leftarrow j$
17 return MED-H(helper)$(A[x_a..ni], B[x_b..nj], C[x_c..mid], c_o, CC_2[y_a - ni + 1, y_b - nj + 1]) +$
18       MED-H(helper)$(A[ni + 1..y_a], B[nj + 1..y_b], C[mid + 1..y_c], CC_1[ni, nj], c_o')$
```

```
MED-H$(A[1..n], B[1..n], C[1..n])$
1  Pre-compute the arrays $\mathcal{M}$ and $\mathcal{G}$.
2  $c_{<\mathrm{M,M,M}>} \leftarrow 0; \forall d \neq <\mathrm{M, M, M}>, c_d \leftarrow \infty$
3  return MED-H(helper)$(A[1..n_a], B[1..n_b], C[1..n_c], c, c)$
```

Remarks:

In recursive subproblem calls, we restrict the alignment to start and end at specific indel configurations by passing the correct initial values for c_o and c_1. This step is critical to ensure the consistency between many subproblem alignments. An optimal alignment of $\{A[1..n_a], B[1..n_b], C[1..n_c]\}$ is equivalent to an alignment starting and ending both at $< M, M, M >$.

3.4 Cache-Oblivious Method and the Cache-Oblivious algorithm MED-CO

Two-level I/0 and cache-oblivious memory model:

In modern computer architecture, the storage is organized into many levels. This organization is called the memory hierarchy. Each level of the hierarchy is faster and smaller than lower levels. Moreover, data in one level is the cache of lower levels. Computer programs are designed to take advantage of this caching and minimize memory transfers between levels of the hierarchy. With the development of modern computers, the size of dataset becomes much larger than before. Therefore, the memory hierarchy becomes the performance bottleneck, which limits the potential of processing power.

It is however difficult to work with this multi-level memory model because of many configuration possibilities. Aggarwal and Vitter [AV88] proposed the *two-level I/0 or external-memory model.* The model organizes the computer storage into two levels:

1. the cache or internal memory close to the processor, which is fast but has a limited size M.
2. the arbitrarily large but costly to access external memory partitioned into blocks of size B.

Traditionally, many algorithms have been developed in this model. The model is simple and captures a variety of other memory models. However, algorithms relies crucially on B and M. Therefore, these algorithms are called *cache-aware* . The model also lacks the transfer management of data between two levels and hence algorithms needs to make explicit data requests.

The *cache-oblivious model* was introduced by Frigo et al. in [FLPR99] to accommodate the in-flexibilities of the two-level model. As before, the model has 2 levels but algorithms designed in the cache-oblivious do not know the block B and the cache M sizes. Therefore they are flexible and adapt well to any two levels of the multi-level memory hierarchy. Moreover, the model assumes an optimal page replacement strategy, which specifies that the evicted page will be accessed farthest in the future. Algorithms in this model, called cache-oblivious algorithms do not have to explicitly manage the cache. The advantage of this model comes from the fact that optimal cache-oblivious algorithms are also optimal in multi-level memory model ([Pro99], [FLPR99]).

Cache-aware algorithms often require fine-tuning of cache parameters for performance on different architectures. In contrast, cache-oblivious algorithms work well on most machines without modifications. Of course, the code still requires some tuning, e.g different base-case size of a recursion, ... but these optimizations are not results of cache configurations ([Dem], [Pro99]).

Cache-Oblivious method for dynamic program: A methodology to develop efficient cache-oblivious algorithms for several fundamental dynamic programs was proposed by R. Chowdhury and V. Ramachandran ([CR06], [CR05]). In the rest of this section, we will describe our work on applying this method to MED-Knudsen to yield the cache-oblivious algorithm MED-CO for the median alignment of three sequences problems.

We first present the two helper procedures Compute Boundary and Trace Path, then later we show how to use them in the pseudocode of MED-CO.

Procedure Compute Boundary: Recall from the previous sections that the computation of the entry $\mathcal{C}[i, j, k]$ in the Knudsen's algorithm depends only on $\mathcal{C}[(i-1)..i, (j-1)..j, (k-1)..k]$. Accordingly, we could compute all the entries in the cuboid $\mathcal{C}[i, j, k]$, for $x_a < i \leq x_b, x_a < j \leq x_b, x_a < k \leq x_b$ provided that following entries are available:

1. $\mathcal{C}[x_a - 1, j, k]$ for $x_b - 1 \leq j \leq y_b, x_c - 1 \leq k \leq y_c$.

2. $\mathcal{C}[i, x_b - 1, k]$ for $x_a - 1 \leq i \leq y_a, x_c - 1 \leq k \leq y_c$.

3. $\mathcal{C}[i, j, x_c - 1]$ for $x_a - 1 \leq i \leq y_a, x_b - 1 \leq j \leq y_b$.

We refer to these three sets of entries as the front F, top T and left L boundaries respectively. The procedure take the three sequences and the three boundaries F, T, L and returns the B (back), D (down), R (right) boundaries, which are following entries:

1. $\mathcal{C}[y_a, j, k]$ for $x_b \leq j \leq y_b, x_c \leq k \leq y_c$.

2. $\mathcal{C}[i, y_b, k]$ for $x_a \leq i \leq y_a, x_c \leq k \leq y_c$.

3. $\mathcal{C}[i, j, y_c]$ for $x_a \leq i \leq y_a, x_b \leq j \leq y_b$.

Figure 3.4 shows how the boundaries are laid out and used in the procedure. For clarity of presentation, the rest of this section assumes the three sequences have the same length $n = 2^p$.

The procedure Compute Boundary computes recursively by dividing the space into subsequently smaller-size cuboid, specifically $s/2 \times s/2 \times s/2$ size. Intuitively, the advantage of this divide-and-conquer approach is that in subsequent calls of the procedure, computation is confined

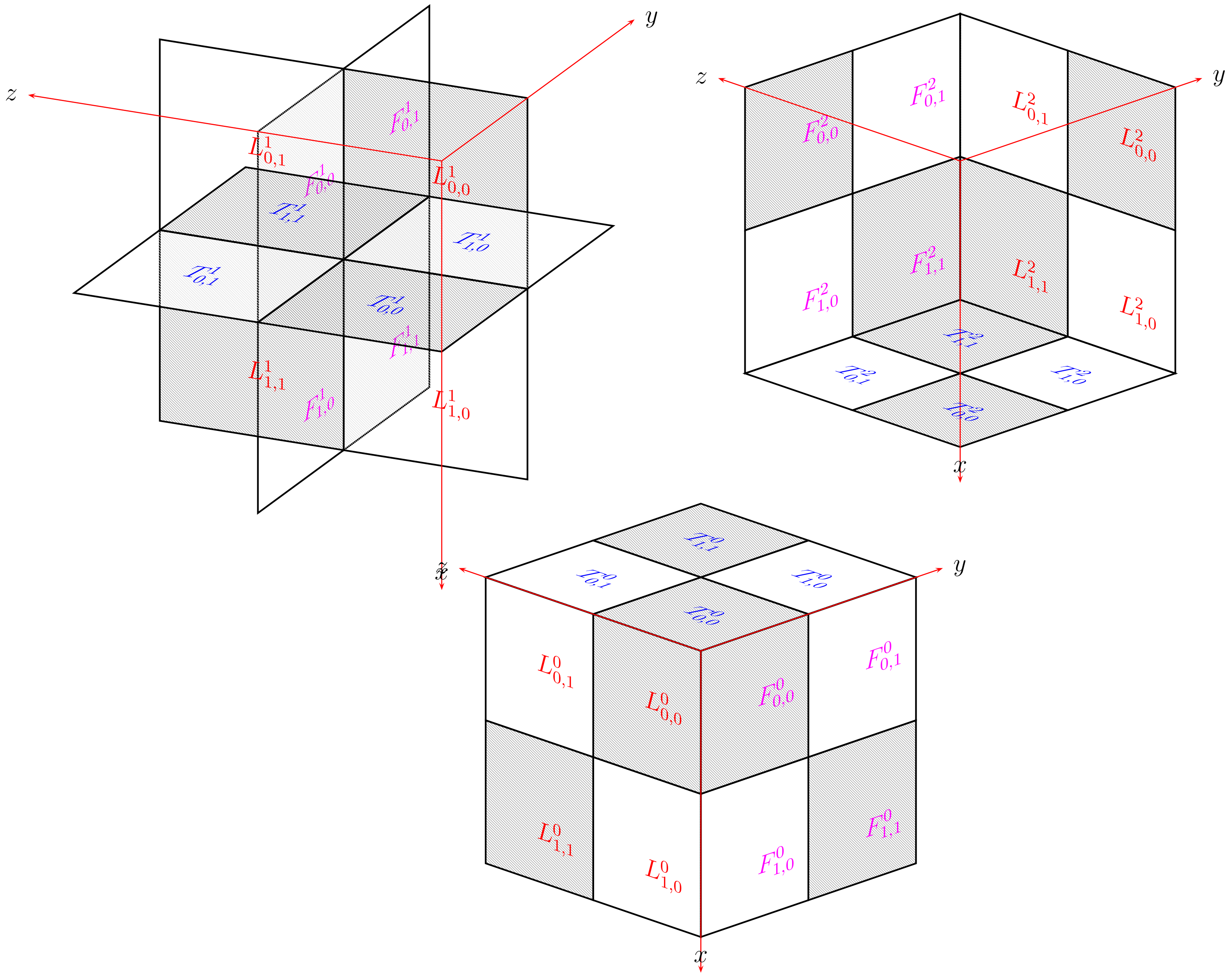


Figure 3.4: F,T,L,B,D,R boundaries of procedures Compute Boundary and Trace Path

within a smaller space, and if data is laid out in memory so that these entries are contiguous in memory, the spatial locality of memory takes full advantage of the cache.

Procedure Trace Path:

Procedure Trace Path computes an optimal alignment and the median sequence by tracing along the trace-path. Similarly to Compute Boundary, the procedure is a divide-and-conquer method with the computing space divided into 8 smaller cuboides. The main difference is in the base case where we extract a character to the median sequence X.

Pseudocode: Pseudo code of Compute Boundary, Trace Path and MED-CO are shown below.

```
Compute Boundary($A[x_a..(x_a+s)], B[x_b..(x_b+s)], C[x_c..(x_c+s)], s, F, T, L$)
 1  if $s = 1$
 2     then Using the equation 3.1 to compute the entry $c$ from F, T, L.
 3          return $< c, c, c >$
 4  Extract $F^0_{i,j}, T^0_{i,j}, L^0_{i,j}$ for $0 \le i, j \le 1$ from $F, T, L$ respectively.
 5  $\epsilon[1..8] = \{<0,0,0>, <0,0,1>, <0,1,0>, <1,0,0>, <0,1,1>, <1,0,1>, <1,1,0>, <1,1,1>\}$
 6  for $l \leftarrow 1$ to 8
 7          do $< i, j, k > \leftarrow \epsilon[l]$
 8              $< F^{k+1}_{i,j}, T^{i+1}_{j,k}, L^{j+1}_{i,k} > \leftarrow$ Compute Boundary($A_i, B_j, C_k, s/2, F^k_{i,j}, T^i_{j,k}, L^j_{i,k}$)
 9  Combine $F^2_{i,j}, T^2_{i,j}, L^2_{i,j}$ for $0 \le i, j \le 1$ into $B, D, R$ respectively.
10  return $< B, D, R >$
```

```
Trace Path($A[x_a..(x_a+s)], B[x_b..(x_b+s)], C[x_c..(x_c+s)], s, F, T, L, < ti, tj, tk, \delta >, X$)
 1  if $s = 1$
 2     then Add a character to $X$ accordingly.
 3          $(ti, tj, tk, \delta) \leftarrow (i', j', k', d')$ where $(i', j', k', d')$ is the preceding index in the trace-path.
 4          return $< ti, tj, tk, \delta >$
 5  Extract $F^0_{i,j}, T^0_{i,j}, L^0_{i,j}$ for $0 \le i, j \le 1$ from $F, T, L$ respectively.
 6  $\epsilon[1..8] = \{<0,0,0>, <0,0,1>, <0,1,0>, <1,0,0>, <0,1,1>, <1,0,1>, <1,1,0>, <1,1,1>\}$
 7  for $l \leftarrow 1$ to 7
 8          do $< i, j, k > \leftarrow \epsilon[l]$
 9              $< F^{k+1}_{i,j}, T^{i+1}_{j,k}, L^{j+1}_{i,k} > \leftarrow$ Compute Boundary($A_i, B_j, C_k, s/2, F^k_{i,j}, T^i_{j,k}, L^j_{i,k}$)
10  if $\delta =$ NIL
11     then Initialize $\delta$ such that the optimal cost is at the indel configuration $\delta$.
12  for $l \leftarrow 8$ to 1
```

```
13          do $< i, j, k > \leftarrow \epsilon[l]$
14              if $< ti, ti, tk >$ lies on one of the boundaries $F^{k+1}_{i,j}, T^{i+1}_{j,k}, L^{j+1}_{i,k}$
15                  then $< ti, tj, tk, \delta > \leftarrow$ TRACE PATH($A_i, B_j, C_k, s/2, F^{k}_{i,j}, T^{i}_{j,k}, L^{j}_{i,k}, < ti, tj, tk, \delta >, X$)
16  return $X$
```

```
MED-CO($A[1..n], B[1..n], C[1..n]$)
1   Pre-compute the arrays $\mathcal{M}$ and $\mathcal{G}$.
2   Initialize $F, T, L$ based on equation 3.1.
3   return TRACE PATH($A[1..n], B[1..n], C[1..n], n, F, T, L, < n, n, n,$ NIL $>$, NIL)
```

Analysis: The correctness of the procedure COMPUTE COST and TRACE PATH could be shown by induction. We do not go into details of the proofs since the proofs are mechanic. Both procedures take $O(n^3)$ time and $O(n^2)$ space. Therefore, the time and space complexities of MED-CO are $O(n^2)$ and $O(n^3)$ respectively.

Theorem 3. *The cache complexity of MED-CO is $O(n^3/M + n^3/(B\sqrt{M}))$.*

For completeness, we provide the steps to prove this result. This proof is the work of Rezaul Chowdhury and we thank him for this representation.

Proof. Let $I_1(n)$ be the cache-complexity of COMPUTE COST on an $n \times n$ input. Then

$$I_1(n) = \begin{cases} O(1 + \frac{n^2}{B}) & \text{if } n^2 \leq \alpha M, \\ 8I_1\left(\frac{n}{2}\right) + O(1 + \frac{n^2}{B}) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that computation on an input of size $\sqrt{\alpha M} \times \sqrt{\alpha M}$ can be performed completely inside the cache. Solving the recurrence we obtain $I_1(n) = O((n^3/M) + (n^3/(B\sqrt{M})))$.

Let $I_2(n)$ be the cache-complexity of TRACE PATH on an input of size $n \times n$ ($n = 2^p$). We observe that though the algorithm calls itself recursively 8 times in the backward pass, at most 4 of those recursive calls will actually be executed since the traceback path cannot intersect more than 4 sub-cubes. Then,

$$I_2(n) = 4I_2\left(\frac{n}{2}\right) + 7I_1\left(\frac{n}{2}\right) + O(1 + \frac{n^2}{B})$$

with $I_2(n) = O(1 + n^2/B)$ if $n^2 \leq \beta M$, where β is the largest constant sufficiently small that computation on an input of size $\sqrt{\beta M} \times \sqrt{\beta M}$ can be performed completely inside the cache. Solving the recurrence we obtain $I_2(n) = O((n^3/M) + (n^3/(B\sqrt{M})))$.

Therefore, The cache complexity of MED-CO is $O(n^3/M + n^3/(B\sqrt{M}))$. $\square$

Chapter 4

Median Alignment of Three Sequences: Experiments

This chapter discusses the performance of 5 programs solving the median alignment of three sequences problem. Existing implementations include Knudsen's program ([Knub]) and David Powell 's implementation of two Ukkonen-based algorithms ([Pow]). We coded programs in C++ of two algorithms MED-H(section 3.3) and MED-CO(section 3.4). Performance of these 5 programs are compared on dataset of both random sequences and synthetic sequences. In section 4.2, we consider options in implementing MED-H and MED-CO. We also talk about our design and our experience in writing the code. Section 4.1 gives an overview of computing medium for performance comparison of these programs. Finally, sections 4.2 summarizes the results. Briefly, MED-CO outperforms other programs dramatically (by more than 50%) on most of the experiments. It also works and scales with larger dataset much better than Knudsen and Powell 's programs.

4.1 Experimental set-up

We ran our experiments on the following three architectures:

- **Intel Xeon.** A dual processor 3.06 GHz Intel Xeon shared memory machine with 4 GB of RAM and running Ubuntu Linux 5.10. Each processor had an 8 KB L1 data cache (4-way set associative) and an on-chip 512 KB unified L2 cache (8-way). The block size was 64 bytes for both caches.

- **AMD Opteron.** A dual processor 2.4 GHz AMD Opteron shared memory machine with 4 GB of RAM and running Ubuntu Linux 5.10. Each processor had a 64 KB L1 data cache (2-way) and an on-chip 1 MB unified L2 cache (8-way). The block size was 64 bytes for both caches.

- **SUN Blade.** A 1 GHz Sun Blade 2000/1000 (UltraSPARC-III+) with 1 GB of RAM and running SunOS 5.9. The processor had an on-chip 64 KB L1 data cache (4-way) and an off-chip 8 MB L2 cache (2-way). The block sizes were 32 bytes for the L1 cache and 512 bytes for the L2 cache.

The caching data on the Sun Blade was obtained using the *cputrack* utility that keeps track of hardware counters. All our algorithms were implemented in C++ using a uniform programming style and compiled using *g++* 3.3.4 with optimization parameter -O3. Implementations of all algorithms we collected for comparing against our algorithms were written in C, and we compiled them using *gcc* 3.3.4 with optimization level -O3. Each machine was exclusively used for experiments (i.e., no other programs were running on them), and on multi-processor machines only a single processor was used.

4.2 Implementation

We refer to our implementations of algorithms in section 3.3 and 3.4 simply as MED-H and MED-CO. We performed experimental evaluations of the following algorithms for finding the median of the three sequences. We used $g_i = 3$, $g_e = 1$ and a mismatch cost of 1 in all experiments.

1. **MED-Knudsen:** Knudsen's implementation of his $O(n^3)$ time & space algorithm [Knub].
2. **MED-ukk.alloc & MED-ukk.checkp:** Two Ukkonen-based algorithms [DRPD00] authored by David Powell [Pow]: ukk.alloc (denoted MED-ukk.alloc) and ukk.checkp (denoted MED-ukk.checkp). The time and space complexities of MED-ukk.alloc are $O(n + d^3)$ (avg.) and $O(n + d^3)$, respectively, while for MED-ukk.checkp they are $O(n \log d + d^3)$ (avg.) and $O(d^2)$, respectively, where d is the edit distance of the sequences.
3. **MED-H:** Our implementation of Knudsen's algorithm using Hirschberg's technique to reduce the space complexity to $O(n^2)$.
4. **MED-CO:** Our cache-oblivious median-finding algorithm.

MED-H and MED-CO implementation:

The programs compose of the driver to read,write input/output and C++ classes that encapsulates the algorithms.

Pre-computation of $\mathcal{G}$ and $\mathcal{M}$: MED-H and MED-CO share the same code snippet to programmatically generate the 2-d array `next`(e, d) based on table 3.2. Using `next`(e, d), two arrays $\mathcal{G}(e, d)$ and $\mathcal{M}$ based on the cost constants, i.e g_i, g_e, M are computed.

Command-line support and input/output sequence formats: The code supports command-line options following the standard UNIX format. There are options to either read input sequences from standard input (`stdin`) or from a specific file. Three standard sequence formats was supported by current implementations: `FASTA`, `PHYLIP INTERLEAVED` , `PHYLIP SEQUENTIAL`. Users can also specify the cost constants through the command-line options. The programs can either just compute the optimal alignment cost or compute the median sequence.

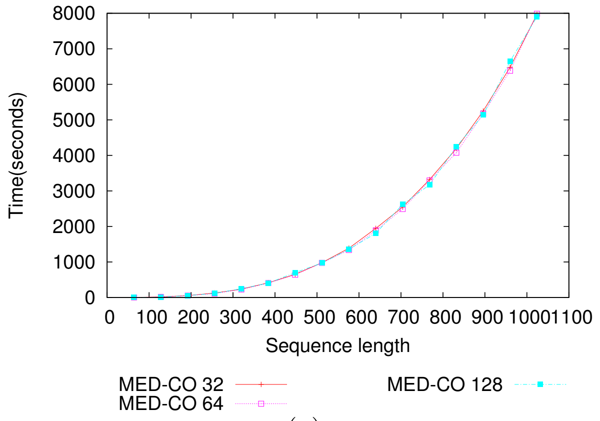
Most of the code for parsing command-line options, reading the input and writing the output was written by Rezaul Chowdhury in a driver file. I glued the driver with my implementation.

Base-case procedure of the recursion: The pseudocode of MED-CO (section 3.4) stops the recursion at base-case $s = 1$. Our implementation allows user to specify the base-case size `BASE-SIZE`. However, we restrict the value to be a power of 2. The rationality is simply that it is easier to code and faster to compute the index of data, which we will go into more details later. We believe that this restriction does not hinder any potential performance gain.

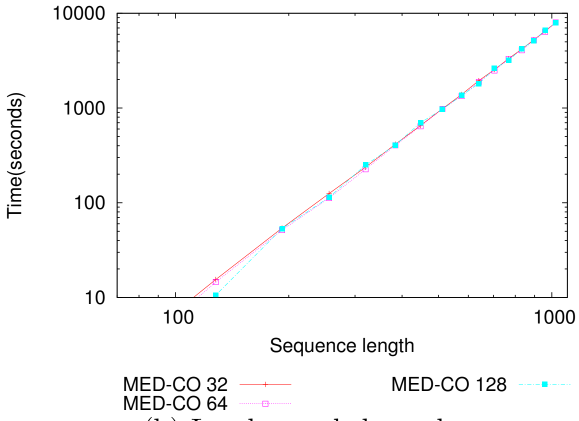
Different base-case sizes have pros and cons. On one hand, overhead in recursive calls, parameter passing, stack pushes and pops could potential slowdown recursive algorithms. Therefore, for smaller `BASE-SIZE`, MED-CO will take more space on the stack, more overhead for function calls. However, it is simpler for coding and optimizing. We could make the base-case as efficient as possible as well as taking advantage of compiler options such as loop-unrolling. Another pro is that the memory-footprint of small base-case size is small hence the cache efficiency is high. On the other hand, large `BASE-SIZE` reduces the number of recursive calls, hence includes less overhead. However, if `BASE-SIZE` is too large, the code would incur more cache misses since it uses much more memory. So it is best to find a value of `BASE-SIZE` so that the memory-footprint could fit into the cache.

It is clear that the base-case procedure is the computational core of MED-CO. Each architecture may have different best value of `BASE-SIZE` to balance listed pros and cons. We spent much time experiment with different `BASE-SIZE` values. The summary of the result is represented in figure 4.1. On both Intel Xeon and AMD Opteron, the best value is 64; on SUN Blade it is 32.

Figure 4.1: Running time of MED-CO with different `BASE-SIZE` values

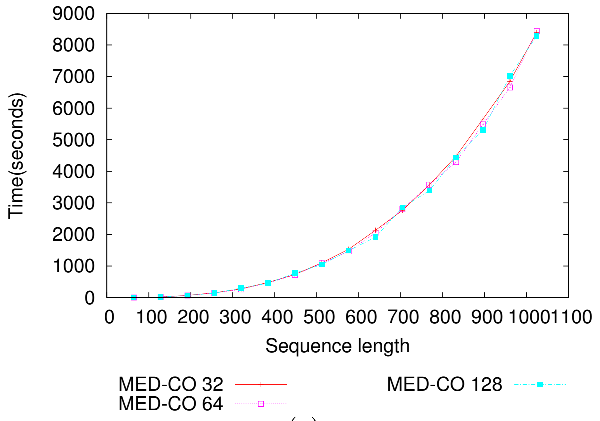


(a)

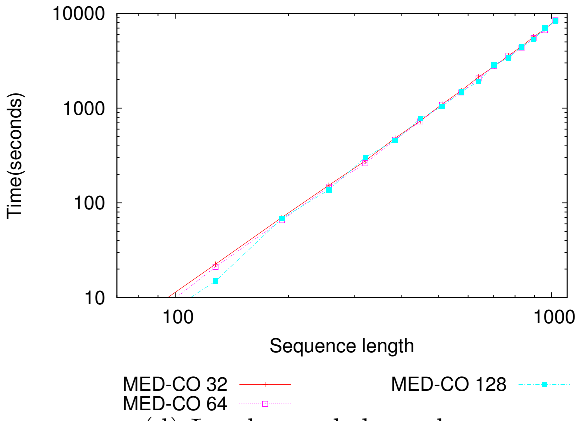


(b) Log-log scaled graph

On AMD Opteron machines



(c)



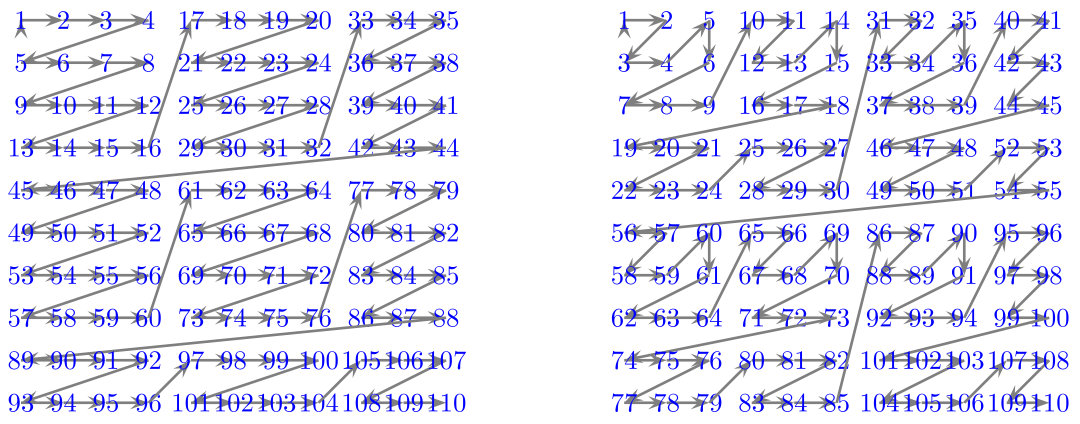
(d) Log-log scaled graph

On Intel Xeon machines

Moreover, the flexility of different `BASE-SIZE` allows us to have many options for the base-case procedure. Our initial implementation uses a 3-dimensional arrays of size $\texttt{BASE-SIZE}^3$ and computes the entire array using the traditional MED-Knudsen procedure (three nested for loops,...). It is fast and simple but consumes memory space. We tried next to adapt MED-H into the procedure. However, it is quite tricky to do so because the original MED-H, at each recursive step, knows exactly the starting indel and ending indel for the alignment(recall that we pass these information in term of c_o and c'_o in calls to MED-H (HELPER)). This information however is hard to extract from recursive calls of MED-CO. We managed to get this working. Unfortunately, the code does not perform better as we expected. We believe that because the code is too complicated, both in term of code size and expensive instructions, these extra complexities swallow our expected performance gain. In future, we could investigate on fine-tuning this code and probably trying advanced compiler options.

Data layout of F, T, L and B, D, R boundaries: In the pseudocode, all boundaries are regarded as a 2-d arrays. At each recursive step, the code divides each boundaries into four sub-arrays and passes each sub-arrays to the recursive calls. It is totally not cache-efficient to use a simple 2-d arrays in the implementation. Therefore, we linearize all boundaries into one-dimensional arrays and use index/pointer to access individual data fields. We consider two ways to layout data, illustrated in Figure 4.2:

Figure 4.2: Data layout



(a) 4x4 blocked layout (b) Bit-interleaved layout

- `BASE-SIZE` x `BASE-SIZE` blocked layout: The 2-d array is divided into blocks of size `BASE-SIZE` x `BASE-SIZE`. Data in each block is contiguous in memory. This is simple to calculate the data index and we need not use pointers but an integer for the index. To handle non-power of 2 array size, we consider blocks on right and bottom edges incomplete and ignore the missing entries. This layout requires many tuning parameters in recursive calls.

- Bit-interleaved layout: Data is recursively laid out in the array such that data in each sub-arrays is contiguous in memory. We could use pointers to directly offset into the data arrays and hence no extra tuning parameters are required. One of the practical advantages of bit-interleaved layout is that pointer calculations on conventional microprocessors can be costly.

Both layouts has the advantage that data in base-case procedure is contiguous in memory. Our code uses the blocked layout. We made this decision because our previous experience with cache-oblivious algorithm implementation suggests that for the computer architectures we run experiments on, the bit-interleaved layout does not perform better.

Programming experience:

Experience from both implementations convinces us that MED-CO is more programming-friendly than MED-H. Even though MED-CO 's code is longer, the algorithm has a nice recursive structure. It is somehow tricky to implement the TRACE PATH but MED-H has quite many cumbersome corner cases. We also believe that MED-CO is more compiler-friendly due to its "balanced" recursion: at each step, the cube is evenly divided while each step of MED-H may divide the space into unequal sub-spaces.

4.3 Results

Overall MED-Knudsen ran about 30%-80% slower than MED-CO on Intel Xeon, but there was virtually no slowdown on SUN Blade for problem sizes it could handle. However, MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp crashed for sequences longer than 256. We summarize our results below.

Random Sequences We ran all implementations on random (equal-length) sequences of length $l = 64k$ for $1 \leq k \leq 16$. Running time on Intel Xeon and AMD Opteron is plotted in Figure 4.3. Selected data points for length 2^i, $6 \leq i \leq 10$ on Intel Xeon are shown in Table 4.1 and on SUN Blade in Tables 4.2 and 4.3.

Due to lack of memory space MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp crashed on both machines for sequences longer than 448, 256 and 640, respectively.

On Intel Xeon and AMD Opteron, MED-CO was the fastest. It ran 1.79 times on Intel Xeon and 2.5 times on AMD Opteron faster than MED-Knudsen for length 256. It also outperformed MED-H by 58% for length 1024 and even more for smaller sequence lengths. Both MED-ukk.alloc and MED-ukk.checkp ran $4-5$ times slower than MED-CO for length 256.

On SUN Blade, the difference in running times of MED-Knudsen and MED-CO was insignificant while MED-Knudson incurred significantly more L2 cache misses than MED-CO. We believe this happens because MED-Knudsen's code is much simpler, hence it executed less instructions. MED-CO ran dramatically faster than MED-ukk.alloc and MED-ukk.alloc since it incurred far fewer cache misses. Though MED-CO incurred more cache misses than MED-H for smaller lengths and it incurred significantly fewer cache misses than MED-H for lengths larger than 128. This happens because for small lengths, the entire space fits in the L2-cache but for large sequences, it does not.

Synthetic Sequences We used ROSE [SEM98] to generate a set of 100 sequences of average length 200 under the HKY evolutionary model [HKaY85] with equal probability for A, C, G and T. We then ran all implementations on three sequences randomly selected from the set. Table 4.4 shows the results of 5 such runs.

MED-CO was the fastest implementation in all runs except for Run 5. It outperformed both MED-Knudsen and MED-H by at least 30%, even by 80% on Run 5. MED-ukk.alloc and MED-ukk.checkp again performed quite badly except for the last run. The score of the alignment of Run 5 is 106 which explains why Ukkonen-based algorithms ran faster than MED-CO.

	Running Times (in sec) for Random Sequences on Intel Xeon (runtime w.r.t. MED-CO)				
Length (n)	MED-Knudsen	MED-H	MED-ukk.alloc	MED-ukk.checkp	MED-CO
64	3.727 (1.65)	6.115 (2.70)	6.288 (2.78)	7.602 (3.36)	2.264 (1.00)
128	29.171 (1.70)	34.090 (1.99)	46.379 (2.70)	55.377 (3.23)	17.157 (1.00)
256	215.392 (1.79)	213.104 (1.77)	501.385 (4.16)	613.717 (5.10)	120.404 (1.00)
512	– (–)	1,493.426 (1.62)	– (–)	4,724.386 (5.13)	921.250 (1.00)
1,024	– (–)	10,847.156 (1.58)	– (–)	– (–)	6,850.470 (1.00)

Table 4.1: Each figure is the average of 3 independent runs on randomly generated strings over $\{ A, T, G, C \}$. The ratio of the runtime of the implementation to that of MED-CO is within parentheses.

	Running Times (in sec) for Random Sequences on SUN Blade (runtime w.r.t. MED-CO)				
Length (n)	MED-Knudsen	MED-H	MED-ukk.alloc	MED-ukk.checkp	MED-CO
32	1.194 (1.37)	2.046 (2.34)	5.782 (6.62)	6.063 (6.95)	0.873 (1.00)
64	8.814 (1.08)	14.614 (1.79)	53.262 (6.52)	57.920 (7.09)	8.165 (1.00)
128	69.400 (1.04)	112.801 (1.68)	394.553 (5.89)	419.371 (6.26)	67.012 (1.00)
256	549.787 (1.01)	893.426 (1.63)	– (–)	4,569.631 (8.36)	546.480 (1.00)
512	– (–)	7,112.395 (1.49)	– (–)	– (–)	4,781.163 (1.00)
1,024	– (–)	56,678.730 (1.47)	– (–)	– (–)	38,567.742 (1.00)

Table 4.2: Each figure is the average of 3 independent runs on randomly generated strings over $\{ A, T, G, C \}$. The ratio of the runtime of the implementation to that of MED-CO is within parentheses.

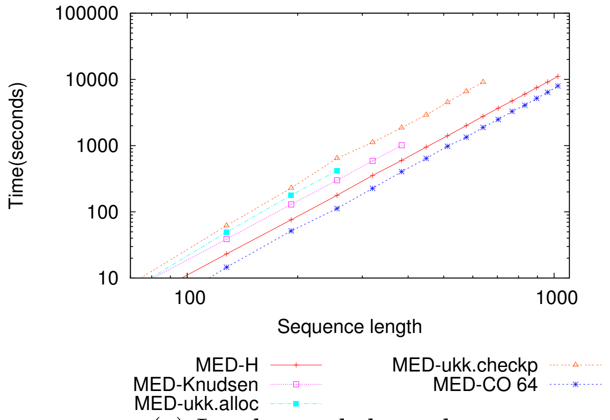
	Ratios of L2 Misses (w.r.t. MED-CO) for Random Sequences on SUN Blade			
Length (n)	MED-Knudsen	MED-H	MED-ukk.alloc	MED-ukk.checkp
32	0.59	0.24	5.86	4.78
64	2.44	0.27	28.47	26.92
128	3.74	0.57	47.66	50.79
256	4.08	4.24	–	88.20
512	–	5.13	–	–
1,024	–	5.65	–	–

Table 4.3: Each figure is the average of ratios obtained from 3 independent runs.

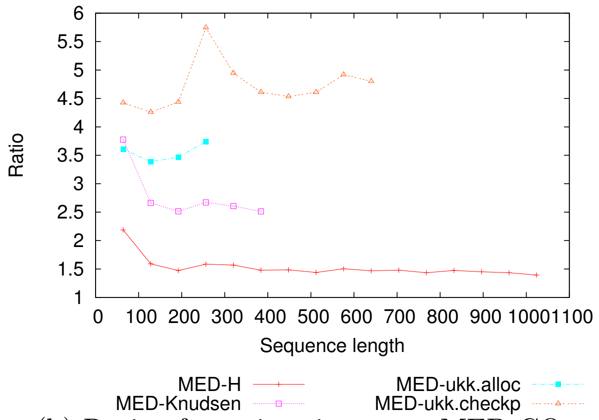
	Running Times (in sec) for Synthetic Sequences on Intel Xeon (runtime w.r.t. MED-CO)						
Run	Lengths	Score	MED-Knudsen	MED-H	MED-ukk.alloc	MED-ukk.checkp	MED-CO
1	154 174 203	224	72.165 (1.30)	80.657 (1.45)	161.822 (2.92)	207.401 (3.74)	55.439 (1.00)
2	236 152 152	247	76.961 (1.49)	82.321 (1.59)	186.236 (3.60)	227.118 (4.39)	51.731 (1.00)
3	154 235 238	279	111.771 (1.45)	116.387 (1.51)	280.726 (3.64)	670.455 (8.69)	77.161 (1.00)
4	219 151 282	295	121.148 (1.44)	125.484 (1.49)	307.207 (3.66)	392.977 (4.68)	83.965 (1.00)
5	167 146 146	106	46.695 (1.60)	53.435 (1.84)	13.633 (0.47)	16.965 (0.58)	29.098 (1.00)

Table 4.4: Each figure is a run on three strings chosen randomly from a set of 100 strings of average length 200 generated using ROSE [SEM98]. The ratio of the runtime of the implementation to that of MED-CO is given within parentheses.

Figure 4.3: Running time of MED-CO, MED-Knudsen, MED-H, MED-ukk.alloc and MED-ukk,checkp

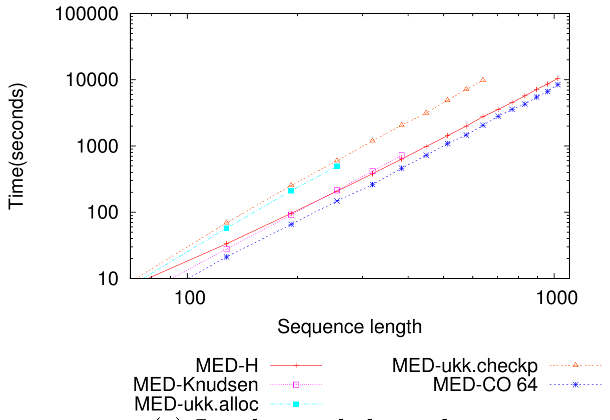


(a) Log-log scaled graph

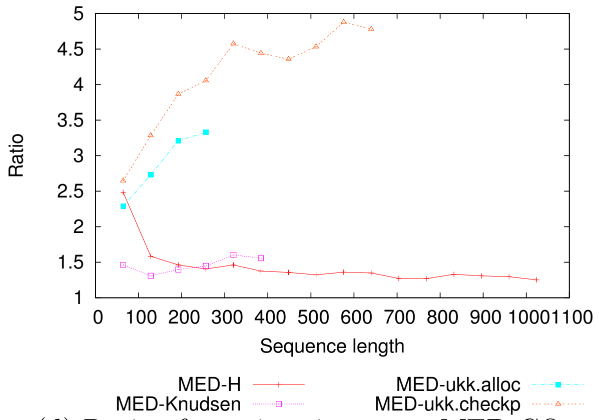


(b) Ratio of running time w.r.t MED-CO

On AMD Opteron machines



(c) Log-log scaled graph



(d) Ratio of running time w.r.t MED-CO

On Intel Xeon machines

Chapter 5

Conclusion

This thesis has presented two approaches to speed up dynamic programs: one for computing the MAAC of two area cladograms and the other for computing the median alignment of three sequences. In the first approach, we reduced the complexity of the bottle-neck subproblem by sparsifying the weighted graph. In the latter, we reduced the I/O complexity without degrading the space and time complexity of the old algorithm. Besides theoretical work, we implemented the new algorithms and experimentally show the performance win. For the median alignment problem, the speed-up over traditional algorithms is significant. The new cache-oblivious algorithm could also handle much larger datasets and scales well with different architectures.

There are many avenues for further research. The MAAC metric of Ganapathy et al.([GGJ+05], [GGJ+06]) is the first rigorous method to identify patterns in biogeography. It will be fruitful to explore this further. As we mentioned before, we have implemented a JAVA package for computing the MAAC of two area cladograms and conducted some experiments. However, much still remains to be done.

Our new algorithm for the median alignment problem opens the possibility of using the median alignment as a tuning step in heuristics to solve computationally hard problems in bioinformatics. This was not possible before because of the limit on the size of input that could be handled by previous methods and the running time of existing implementations. Moreover, we believe the speed-up of the cache oblivious can be even larger with more fine-tuning and optimization of the code. We also think that MED-CO can potentially be multi-threaded, therefore taking advantage of multi-processor machines.

Bibliography

[AK97] A. Amir and D. Keselman. Maximum Agreement Subtrees in a Set of Evolutionary Trees: Metrics and Efficient Algorithms. *SIAM Journal of Computing*, 26(6):1656–1669, 1997. A preliminary version of this paper appeared in FOCS '94.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[CHM94] Kun-Mao Chao, Ross C. Hardison, and Webb C. Miller. Recent developments in linear-space alignment methods: a survey. *J. Computational Biology*, 1:271–291, 1994.

[CR05] Rezaul Alam Chowdhury and Vijaya Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 735–744, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[CR06] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 591–600, New York, NY, USA, 2006. ACM Press.

[Dem] Erik D. Demaine. Cache-oblivious algorithms and data structures. in Lecture Notes from the EEF Summer School on Massive Data Sets, Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27-July 1, 2002, to appear.

[DRPD00] Lloyd Allison David R. Powell and Trevor I. Dix. Fast, optimal alignment of three sequences using linear gap cost. *Journal of Theoretical Biology*, pages 207(3):325–336, 2000.

[FCPT95] M. Farach-Colton, T.M. Przytycka, and M. Thorup. On the Agreement of Many Trees. *Information Processing Letters*, 55:297–301, 1995.

[FCT94] M. Farach-Colton and M. Thorup. Fast Comparison of Evolutionary Trees. In *Proc. of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 481–488, 1994.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. page 285, 1999.

[GGJ$^+$05] Ganeshkumar Ganapathy, Barbara Goodson, Robert Jansen, Vijaya Ramachandran, and Tandy Warnow. Pattern identification in biogeography. *Lecture Notes in Computer Science*, 3692:116–127, October 2005.

[GGJ$^+$06] Ganeshkumar Ganapathy, Barbara Goodson, Robert Jansen, Hai son Le, Vijaya Ramachandran, and Tandy Warnow. Pattern identification in biogeography. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):334–346, 2006.

[GHS97] J. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment, 1997.

[Got82] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, pages 162:705–708, 1982.

[GT89] H. Gabow and R. R. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM Journal of Computing*, 18(5):1013–1036, 1989.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[HKaY85] Masami Hasegawa, Hirohima Kishino, and Taka aki Yano. Dating of human-ape splitting by a molecular clock of mitochondrial dna. *Journal of Molecular Evolution*, pages 22:160–174, 1985.

[Knua] Bjarne Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. 2812/2003:433–446.

[Knub] Bjarne Knusen. Multiple parsimony alignment with "affalign". Software package `multalign.tar` containing files: affalign.cc, readme.txt and codon_dist.

[Mai78] D. Maier. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[MM88] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. pages 4(1):11–17, 1988.

[Pow] David R. Powell. Software package `align3str_checkp.tar.gz`. Containing 4 programs: ukk.dpa, ukk.noalign, ukk.alloc and ukk.checkp.

[Pro99] Harald Prokop. Cache-oblivious algorithms, 1999.

[SEM98] J. Stoye, D. Evers, and F. Meyer. Rose: generating sequence families, 1998.

[SW93] M. Steel and T. Warnow. Kaikoura Tree Theorems: Computing the Maximum Agreement Subtree. *Information Processing Letters*, 48:77–82, 1993.