A 6-Approximation Algorithm for Computing Smallest Common AoN-supertree With Application to the Reconstruction of Glycan Trees

Kiyoko F. Aoki-Kinoshita⋆, Minoru Kanehisa⋆⋆, Ming-Yang Kao⋆⋆⋆, Xiang-Yang Li†, and Weizhao Wang§

Abstract. A node-labeled rooted tree T (with root r) is an all-or-nothing subtree (called *AoN-subtree*) of a node-labeled rooted tree T' if (1) T is a subtree of the tree rooted at some node u (with the same label as r) of T', (2) for each internal node v of T, *all* the neighbors of v in T' are the neighbors of v in T. Tree T' is then called an *AoN-supertree* of T. Given a set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ of n *node-labeled rooted* trees, smallest common AoN-supertree problem seeks the smallest possible *node-labeled rooted* tree (denoted as **LCST**) such that every tree T_i in $\mathcal{T}$ is an *AoN-subtree* of **LCST**. It generalizes the smallest superstring problem and it has applications in glycobiology. We present a polynomial-time greedy algorithm with approximation ratio 6.

1 Introduction

In smallest AoN-supertree problem we are given a set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ of n node-labeled rooted trees and we seek the smallest possible node-labeled rooted tree **LCST** such that every tree T_i in $\mathcal{T}$ is an all-or-nothing subtree (called *AoN-subtree*) of **LCST**. Here a tree T_i is an AoN-subtree of another tree T if (1) T_i is a subtree of T, and (2) for each node v of tree T_i, either all children nodes of v in T are also children of v in T_i, or none of the children nodes of v in T is a child node of v in T_i. The widely studied shortest superstring problem (*e.g.*, [1–7]), which is known to be NP-hard and even MAX-SNP hard [5], is a special case of smallest supertree problem where each string can be viewed as a unary rooted tree. The best known approximation ratio for shortest superstring problem is $2\frac{1}{2}$ [6]. The simple greedy algorithm was also proven to be effective [4, 5], with the best proven approximation ratio $3\frac{1}{2}$ [4]. Here, we present a polynomial-time 6-approximation algorithm for smallest supertree problem.

The superstring problem has application in data compression and in DNA sequencing, while the supertree problem also has vast applications in glycobiology. In the field

⋆ Kanehisa Laboratory, Bioinformatics Center, Institute for Chemical Research, Kyoto University. Email: kiyoko@kuicr.kyoto-u.ac.jp

⋆⋆ Bioinformatics Center, Institute for Chemical Research, Kyoto University, and Human Genome Center, Institute of Medical Science, University of Tokyo. Email: kanehisa@kuicr.kyoto-u.ac.jp

⋆⋆⋆ Dept. of Electrical Engineering and Computer Science, Northwestern University. Email: kao@cs.northwestern.edu. Supported in part by NSF Grant IIS-0121491.

† Dept. of Computer Science, Illinois Institute of Technology. Email: xli@cs.iit.edu, wangwei4@iit.edu. Xiang-Yang Li is partially supported by NSF CCR-0311174.

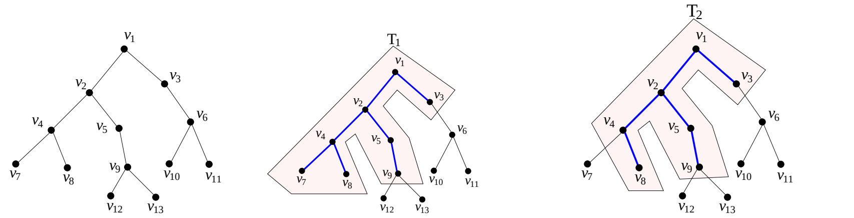
of glycobiology, for the study of glycans, or carbohydrate sugar chains (called glycome informatics), much work pertains to analyzing the database of known glycan structures themselves. Glycans are considered the third major class of biomolecules next to DNA and proteins. However, they are not studied as much as DNA or proteins due to their complex tree structure; they are branched structures. In recent years, databases of glycans [8] have taken off, and the application of theoretical computer science and data mining techniques have produced glycan tree alignment [9,10], score matrices [11] and probabilistic models [12] for the analysis of glycans. In this work, we look at one of the current biggest challenges in this field, which is the characterization of glycan tree structures from mass spectrometry data. The retrieval of what glycan structures these data represent still remains a major difficulty. In this work, we will assess this problem theoretically in application to any glycan structure. By doing so, it would be straightforward to apply algorithms to quickly annotate any mass spectrometry data with accurate glycan structures, thus enabling the rapid population of glycan databases and resulting biological analysis.

2 Preliminaries and Problem Definition

In the remainder of this paper, unless explicitly stated otherwise, a tree is rooted. The relative positions of the children could be significant or non-significant. The tree is called an *ordered* tree if the relative positions of the children of each node is significant, that is, there is the first child, the second child, the third child, *etc.*, for each internal node. Otherwise it is called a *non-ordered* tree. The *size* of a tree T, denoted as $|T|$, is the number of nodes in T. The *distance* between nodes u and v in a tree T is the number of edges on the unique path between u and v in T. Given a node u in a tree T rooted at node r, the *level* of u is the distance between u and the root r. The *height* of a tree T is the maximum level over all nodes in the tree. A node w is an *ancestor* of a node u if it is on the path between u and r; the node u is then called a *descendant* of w. If all leaf nodes are on the same level, the tree is called *regular*. Given a rooted tree T, we use $r(T)$ to denote the root node of T.

In this paper, we consider the trees composed of nodes with *label*s that are not necessary to be unique. We assume that the labels of nodes are selected from a *totally ordered set*. Each node hasx a unique ID. Given a tree T and a node u of T, a tree T' rooted at u is an **AoN-subtree** (representing *All-or-Nothing subtree*) of T if for each node v that is a descendant of u, either all children of v in tree T are in T' or none of the children of v in T is in T'. Note that the definition of the AoN-subtree is different from the traditional subtree definition. For example, consider a tree T in Figure 1 (a) and tree T_1 in Figure 1 (b). Tree T_1 is an AoN-subtree of T. Tree T_2 in Figure 1 is not an AoN-subtree of T since tree T_2 only contains one of the two children of node v_4. Given two trees T_1 and T_2, if T is an AoN-subtree of both T_1 and T_2, then T is the *common AoN-subtree* of T_1 and T_2. If T has the maximum number of nodes among all common AoN-subtrees, then T is the *maximum common AoN-subtree*. Given a tree T and an internal node u of T, let $T(u)$ be the tree composed of node u and all descendants of u in T. Obviously, $T(u)$ is an AoN-subtree of T.

If tree T' is an AoN-subtree of T, then T is an *AoN-supertree* of T'. In this paper, we assume that there is a set $\mathcal{T}$ of n rooted trees $\{T_1, T_2, \cdots, T_n\}$, where $r_i = r(T_i)$ is the



(a) Tree T (b) T_1 is an AoN-subtree of T (c) T_2 is not an AoN-subtree of T

Fig. 1. Illustration of AoN-subtree Notation

root of the tree T_i. Here trees T_i could be *ordered* or *non-ordered*. If tree T is an AoN-supertree for every tree T_i for $1 \le i \le n$, then T is called a *common AoN-supertree* of $T_1, T_2, \cdots, T_n$. If T has the smallest number of nodes among all common AoN-supertrees, then T is *smallest common AoN-supertree* and is denoted as $\mathbf{LCST}(\mathcal{T})$. In smallest AoN-supertree problem we are given a set $\mathcal{T}$ of n node-labeled rooted trees and we seek smallest common AoN-supertree $\mathbf{LCST}(\mathcal{T})$.

3 Find the Maximum Overlap AoN-subtree

Our algorithm for finding smallest common AoN-supertree is based on greedy merging of two trees that have the largest overlap. Given two trees T_1 and T_2, with root r_1 and r_2 respectively, if an internal node u of T_1 satisfying that (1) $u = r_2$ and (2) $T_1(u)$ is an AoN-subtree of T_2, then $T_1(u)$ is an *overlap AoN-subtree* of tree T_2 over T_1, denoted by $T_1(u) = T_1 \Cap T_2$. Note that if tree T is an overlap AoN-subtree of T_2 over T_1, it is not necessary that T is an overlap AoN-subtree of T_1 over T_2. If T has the largest number of nodes among all overlap AoN-subtrees of T_2 over T_1, then T is the *largest overlap AoN-subtree*. Let $\mathcal{L}(T_1, T_2)$ be the largest overlap AoN-subtree of T_2 over T_1 and note that $\mathcal{L}(T_1, T_2)$ is not necessarily symmetric. If we remove $\mathcal{L}(T_1, T_2)$ from T_2, then the remaining forest is denoted as $T_2 - T_1$.

Here, we assume that the tree is non-ordered. If the tree is ordered, then find the largest overlap AoN-subtree is trivial. Without loss of generality, we assume that the labels of the tree are integers from $[1, m]$. We abuse the notations little bit here by using u to also denote the label of a node u with ID u if it is clear from context. Given two trees T_1 and T_2, we define a *total order-relation* $\prec$ of two trees as follows.

1. If $r(T_1) < r(T_2)$, we say $T_1 \prec T_2$. If $r(T_2) < r(T_1)$, we say $T_2 \prec T_1$.
2. If $r(T_1) = r(T_2)$, we further let that $\{u_1, u_2, \cdots, u_p\}$ be all the children of $r(T_1)$ in T_1 and $\{v_1, v_2, \cdots, v_q\}$ be all the children of $r(T_2)$ in T_2. W.l.o.g., we also assume that the children are sorted in an order such that $T_1(u_i) \succeq T_1(u_j)$ for any $1 \le i < j \le p$ and $T_2(v_i) \succeq T_2(v_j)$ for any $1 \le i < j \le q$. Let k the smallest index such that either $T_1(u_k) \prec T_2(v_k)$ or $T_2(v_k) \prec T_1(u_k)$. We have three subcases: a) If $T_1(u_k) \prec T_2(v_k)$, we say $T_1 \prec T_2$; b) If $T_1(u_k) \prec T_2(v_k)$, we say $T_1 \prec T_2$; c) Such k does not exist. If $p < q$, then $T_1 \prec T_2$; if $p > q$ then $T_2 \prec T_1$; if $p = q$, then $T_1 \asymp T_2$.

Notice that here $T_1 \preceq T_2$ if $T_1 \prec T_2$ or $T_1 = T_2$; $T_1 \succeq T_2$ if $T_1 \succ T_2$ or $T_1 = T_2$; $T_1 \succ T_2$ if $T_2 \prec T_1$. More formally, Algorithm 1 summarizes how to decide the order-relation between two non-ordered trees.

Algorithm 1 Decide the relationship of two trees.

Input: Two trees T_1 and T_2.
Output: The relationship between T_1 and T_2.

```
Label all internal nodes in T_1 WHITE and all leaf nodes BLACK.
repeat
    Pick any internal node in T_1 such that all children nodes are marked BLACK, say u. Sort
    all children nodes of T_1(u) in the order as {u_1, u_2, ..., u_p} such that T_1(u_i) ⪰ T_1(u_j)
    for any 1 ≤ i < j ≤ p.
    Mark u BLACK.
until all internal nodes in T_1 are BLACK.
Mark all internal nodes in T_2 WHITE and all leaf nodes BLACK.
repeat
    Pick any internal node in T_2 such that all children nodes are with marked BLACK, say
    u. Sort all children nodes of T_2(u) in the order as {v_1, v_2, ..., v_p} such that T_2(v_i) ⪰
    T_2(v_j) for any 1 ≤ i < j ≤ p.
    Mark u BLACK.
until all internal nodes in T_2 are BLACK.
If r(T_1) < r(T_2) then return T_1 ≺ T_2. end if
If r(T_1) > r(T_2) then return T_1 ≻ T_2; end if
Assume {u_1, u_2, ..., u_p} are children nodes of r(T_1) and {v_1, v_2, ..., v_p} are children
nodes of r(T_2).
for i = 1 to min(p, q) do
    If T_1(u_i) ≺ T_2(v_i) return T_1 ≺ T_2; if T_1(u_i) ≻ T_2(v_i) return T_1 ≻ T_2.
If p < q return T_1 ≺ T_2; if p > q return T_1 ≻ T_2; if p = q return T_1 = T_2.
```

In Algorithm 1, we first compute a lexicographic ordering of a tree and the compute the order-relation of two trees. Note for any two siblings of a common parent, we can compare the order of them by a breadth first search. Thus, the worst case happens when the tree is a complete binary tree and all nodes have the same label, which takes time $O(n^2)$. Thus, for a tree T of n nodes, we have

Lemma 1. *Algorithm 1 computes the ordering of a tree T in time $O(n^2)$.*

We present a recursive method (Algorithm 2) that decides whether one tree is an AoN-subtree of another. Given two trees T_1 and T_2, we then show how to find the largest overlap tree of T_2 over T_1. First, we order the trees T_1 and T_2, and then find the internal node u such that $T_1(u)$ is an AoN-subtree of T_2 and $|T_1(u)|$ is maximum. From Lemma 1, the ordering of trees T_1 and T_2 need $O(|T_1|^2 + |T_2|^2)$. Notice that for any internal node u of T_1, checking whether $T_1(u)$ is an AoN-subtree of T_2 takes time $O(|T_1(u)|$. Thus, the total time needed is $\sum_{u \in T_1} |T_1(u)| \leq |T_1|^2$. Thus, we have

Lemma 2. *Finding largest overlap tree has time complexity $O(|T_1|^2 + |T_2|^2)$.*

We expect a better algorithm to find the largest overlap AoN-subtree based on the fact that there exists efficient linear time algorithm that can find a largest common substring of a set of strings. However, designing such efficient algorithm is not the scope of this paper. We leave it as a future work.

Algorithm 2 Decide whether a tree T_2 is an AoN-subtree of T_1.

Flag $\leftarrow$ FALSE;
For each internal node in T_1, order its p children from left to right as $u_1, u_2, \cdots, u_p$ such that for any pair of children u_i and u_j, $T_1(u_i) \preceq T_1(u_j)$ for $i < j$. Similarly, we also order the children of each internal node in T_2 similarly. Assume that the children of $r(T_2)$ from left to right is $\{v_1, v_2, \cdots, v_q\}$.
for each internal node u of T_1 such that $u = r(T_2)$ and Flag==FALSE **do**
    Assume that the set of "sorted" children nodes of u is $\{u_1, u_2, \cdots, u_p\}$.
    Flag $\leftarrow$ TRUE if (1) $p = q$, and (2) tree $T_2(u_i)$ is an AoN-subtree of $T_1(v_i)$ for evey u_i with $1 \leq i \leq p$.
Return Flag;

4 Approximate Smallest Common AoN-Supertree

We then consider how to find smallest common AoN-supertree given a set $\mathcal{T}$ of n regular trees $\{T_1, T_2, \cdots, T_n\}$. Here, we assume that no tree T_i is an AoN-subtree of another tree T_j. It is known that the problem of computing smallest common superstring, given n strings, is NP-Hard and even MAX-SNP hard [5]. Notice that computing the smallest common superstring is a special case of computing smallest common AoN-supertree when all trees are restricted to a rooted unary tree. Thus, we have

Theorem 1. *Computing smallest common AoN-supertree is NP-Hard.*

4.1 Understanding the Structure of LCST

Notice that if a tree T is a common AoN-supertree of $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$, then for each tree T_i, we can find an internal node u of T such that there is an AoN-subtree $T(u)$ of T root at u that matches T_i. When multiple such internal nodes u exist, we choose any node with the lowest level, denoted by $r_i(T)$. For notational simplicity, we also denote the AoN-subtree of T that equals to T_i rooted at $r_i(T)$ as T_i if it is clear from the context. If $r_i(T)$ is an ancestor of $r_j(T)$, then we also say that T_i is an *ancestor* of T_j. Similarly, if $r_i(T)$ is a descendant of $r_j(T)$, then we also say that T_i is a *descendant* of T_j. If T_i is an ancestor of T_j and there does not exist a tree T_k such that T_k is an ancestor of T_j and T_i is ancestor of T_k, then T_i is the *parent* of T_j and T_j is a *child* of T_i. Lemma 3 and 4 (whose proofs are omitted due to space limit) showed that the notation of child and parent is well defined in smallest common AoN-supertree **LCST**$(\mathcal{T})$.

Lemma 3. *If T_i is T_j's parent in tree **LCST**$(\mathcal{T})$, then either $r_j($**LCST**$(\mathcal{T}))$ is a node in T_i or a child of some leaf node of T_i.*

Lemma 4. *There is a unique tree T_i such that $r_i($**LCST**$(\mathcal{T}))$ is the root of tree **LCST**$(\mathcal{T})$.*

Given a tree set $\mathcal{T}$ and a common AoN-supertree T, if any node in tree T is in a tree T_i for some index i, then we call this common AoN-supertree *condensed common AoN-supertree*. If a common AoN-supertree T is not a condensed common AoN-supertree, then recursively apply the following process will generate a condensed common AoN-supertree. First, we pick any node $u \in T$ that is not in any tree T_i. Remove u, and let all children of u in T become the children of u's parent. Notice that this will not violate the all-or-nothing property of the AoN-supertree. Thus, we will only consider condensed common AoN-supertrees when we approximate smallest common AoN-supertree. Notice Lemma 3 and Lemma 4 implies the following lemma.

Lemma 5. *The optimum tree* ***LCST***$(\mathcal{T})$ *is a condensed common AoN-supertree.*

Notice that if T is a common AoN-supertree of $\mathcal{T}$, then for any tree T_i, its parent is unique. Together with Lemma 4, we have the following lemma.

Lemma 6. *Given $\mathcal{T}$ and a condensed common AoN-supertree T, for any node $r_i(T)$, either $r_i(T)$ is the root of T or there is a unique j where $r_j(T)$ is the parent of $r_i(T)$.*

If we treat each tree T_i as a node, then Lemma 6 reveals that we can construct a unique *virtual overlap tree* $\mathbb{VT}(T)$ as follows. Each vertex of the virtual overlap tree corresponds to a tree T_i. If $r_i(T)$ is the root of tree T, then T_i is the root. Otherwise, T_i's unique parent in T, denoted by $\mathcal{P}(T_i)$, becomes its parent in $\mathbb{VT}(T)$ and all children in T becomes its children in $\mathbb{VT}(T)$. When T_i is T_j's parent, from Lemma 3, the root of T_j is either in T_i or a child of a leaf node of T_i. If T_p and T_q are both children of T_i, then T_p and T_q are *siblings*. Following lemma reveals a property of the siblings.

Lemma 7. *If T_p and T_q are siblings, then T_p and T_q do not share any common nodes.*

Thus, given a virtual overlap tree $\mathbb{VT}(T)$, the size of the condensed common AoN-supertree is $|T| = |T_i| + \sum_{T_j\in\mathcal{T}-T_i} |T_j - \mathcal{P}(T_j)| = \sum_{T_j\in\mathcal{T}} |T_j| - \sum_{T_j\in\mathcal{T}-T_i} |\mathcal{P}(T_j) \Cap T_j|$, where T_i is the root in $\mathbb{VT}(T)$. Algorithm 3 will reduce the size of a condensed tree T.

Algorithm 3 Find the largest overlap AoN-subtree.

Input: A tree set $\mathcal{T}$ and a condensed common super tree T.
Output: A new tree T.

for each tree T_j in $\mathbb{VT}(T)$ that is not a root **do**
  if the overlap of T_i on T_j in tree T is not equal $\mathcal{L}(T_i, T_j)$ where T_i is $\mathcal{P}(T_j)$ **then**
    Find the the node $u \in T_i$ such that $T_i(u)$ is $\mathcal{L}(T_i, T_j)$.
    For each tree T_p who is a sibling of T_j and r_p is an descendant of u, we construct a new tree T_p^{new} that equals $T(r_p)$. Similarly, we also construct the tree T_j^{new}.
    For each tree T_p^{new}, remove $T_p^{new} - T_i$ from T. Tree T becomes T^{temp}.
    We overlap tree T_j^{new} at node u. For each tree T_p^{new}, we let root of T_p^{new} be the child of any leaf node in T^{temp}. Update tree T as T^{temp}.

Theorem 2. *Algorithm 3 always maintains tree T as a condensed AoN-supertree. Moreover, for any tree T_j that is not a root, if T_j does not have a maximum overlap with its parent, then Algorithm 3 decreases the size of T by at least* 1.

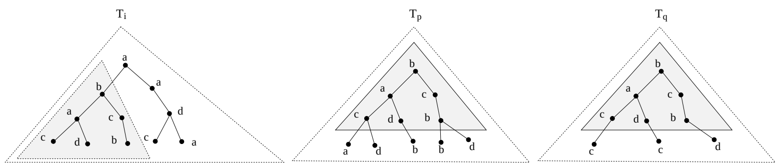
PROOF. It is not difficult to observe that T^{temp} is a condensed common AoN-supertree. Thus, we focus on the second part. Without loss of generality, we assume that T_j, which is not the root, does not have the maximum overlap with its parent T_i. Notice that for each tree T_k who is a child of T_i and r_k is a descendant of u (including T_j), there is an overlap of T_k over T_i. It is not difficult to observe that the sum of the overlap is smaller than the size of $T_i(u)$. On the other hand, the maximum overlap of T_j over T_i is exactly $T_i(u)$. Thus, the overall overlap is increased by 1 at least. In order words, the size of T^{temp} is decreased at least by 1. □

Theorem 2 shows that for any condensed tree T, we can decrease its size by applying Algorithm 3 as long as some tree does not have a maximum overlap with its parent. Therefore, we can focus on the tree in which each non-root AoN-subtree always has a maximum overlaps with its parent. We denote this kind of common AoN-supertree as *maximum condensed common AoN-supertree* (MCCST). We then have

Theorem 3. *Smallest common AoN-supertree* $\mathbf{LCST}(\mathcal{T})$ *is indeed a MCCST.*

4.2 Compute Good MCCST

With understanding of structures of LCST in Section 4.1, we are now ready to present our algorithm with constant approximation ratio. Notice that our focus now is to choose maximum condensed common supertree (MCCST) with smaller size among all MCCSTs. Given a tree set $\mathcal{T}$, the naive way is to first compute $\mathcal{L}(T_i, T_j)$ for each pair of T_i and T_j in $\mathcal{T}$. After that, for each T_j, we choose the T_i such that $\mathcal{L}(T_i, T_j)$ is maximum as its parent, which we call *treelization*. However, this solution does not guarantee a valid virtual overlap tree due to two reasons.



(a) Tree T_i (b) T_p (c) T_q

Fig. 2. Illustration of confliction.

- First, it is possible that T_p and T_q choose the same tree T_i as their parent, and it is not possible for T_p and T_q to have maximum overlap with T_i simultaneously. In this case, we call tree T_p *conflicts* with T_q regarding T_i. One such example is shown in Figure 2. The maximum overlap of tree T_p and T_q over T_i are shown in Figure 2 (b) and (c) respectively. It is not difficult to observe that T_p and T_q can not maximally overlap with T_i simultaneously.
- Second, it is possible that T_j chooses T_i as its parent and T_i chooses a tree T_k who is a descendant of T_j as its parent. It thus creates a cycle in the virtual overlap graph, which we called *cycled tree*.

If we ignore the second problem, then any treelization avoids the first problem in the virtual overlap graph is a special forest with possibly several disconnected components such that each component is a tree whose root may have one backward edge toward its descendant. We call the forest a *cycled* forest.

In order to find the cycled forest with minimum size, we model it as a linear programming problem. Here, $x_{i,j} = 1$ if tree T_j chooses T_i as its parent; otherwise $x_{i,j} = 0$. Notice that for each tree T_j, it has exactly one parent, thus $\sum_{i:i \neq j} x_{i,j} = 1$ for each tree T_j. On the other hand, if $x_{i,j} = 1$, then in order to avoid the first problem, any tree T_k conflicting with T_j with respect to T_i should satisfy that $x_{i,k} = 0$. The objective is to minimize $\sum_{i \neq j} x_{i,j} \cdot (|T_j - \mathcal{L}(T_i, T_j)|)$, *i.e.*, the total size of the trees with cycles. Following is the Integer Programming we aim to solve, which is denoted as **IP1**.

$$\sum x_{i,j} \cdot (|T_j - \mathcal{L}(T_i, T_j)|). \tag{1}$$

$$\text{Subject to} \begin{cases} \sum_{i \neq j} x_{i,j} = 1 \ \forall \ T_j \\ x_{i,j} + x_{i,k} \leq 1 \ \forall \ i, j, k \text{ such that } T_j \text{ conflicts } T_k \text{ regarding } T_i \\ x_{i,j} = \{0, 1\} \ \forall \ i \neq j \end{cases} \tag{2}$$

Algorithm 4 Greedy Method To Compute A Cycled Forest.

Input: A tree set $\mathcal{T}$ and a condensed common AoN-supertree T.
Output: A cycled forest.

1: Compute $\mathcal{L}(T_i, T_j)$ for each pair of trees and sort them in a descending order. Initialize the tree set $S = \{\mathcal{L}(T_i, T_j) \mid \forall i \neq j\}$ and $\mathbb{TC} = \emptyset$.
2: **while** S is not empty **do**
3: Choose the tree in S with the maximum size, say $\mathcal{L}(T_i, T_j)$.
4: Add $\mathcal{L}(T_i, T_j)$ in $\mathbb{TC}$ and remove $\mathcal{L}(T_p, T_j)$ from S for any $p \neq i$ in S. Remove $\mathcal{L}(T_i, T_q)$ from S if T_q conflicts with T_j regarding T_i.
5: Set $x_{i,j} = 1$ if $\mathcal{L}(T_i, T_j)$ is in $\mathbb{TC}$, and $x_{i,j} = 0$ otherwise.

Algorithm 4 greedily selects the $\mathcal{L}(T_i, T_j)$ and it finds one solution to **IP1**.

Theorem 4. *Algorithm 4 computes a solution to the Integer Programming (1).*

PROOF. It is not difficult to verify that the solution does satisfy the constraints. Thus, we focus on the proof that it minimizes $\sum x_{i,j} \cdot (|T_j - \mathcal{L}(T_i, T_j)|)$. Since, $\sum_{i:i \neq j} x_{i,j} = 1$ for each T_j, $\sum x_{i,j} \cdot (|T_j - \mathcal{L}(T_i, T_j)|) = \sum_{T_i \in \mathcal{T}} |T_i| - \sum_{i \neq j} x_{i,j} \cdot |\mathcal{L}(T_i, T_j)|$. Thus, we only need to show that $\sum_{i \neq j} x_{i,j} \cdot |\mathcal{L}(T_i, T_j)|$ is maximized.

Without loss of generality, we assume that Algorithm 4 chooses $\mathcal{L}(T_{i_1}, T_{j_1})$, $\mathcal{L}(T_{i_2}, T_{j_2})$, $\cdots, \mathcal{L}(T_{i_n}, T_{j_n})$ in that order. We also assume that the solution to IP1 is x^{opt}, and all $\mathcal{L}(T_i, T_j)$ such that $x^{opt}_{i,j} = 1$ are ranked in a descending order $\mathcal{L}(T_{p_1}, T_{q_1})$, $\mathcal{L}(T_{p_2}, T_{q_2})$, $\cdots, \mathcal{L}(T_{p_n}, T_{q_n})$. Obviously, $\mathcal{L}(T_{i_1}, T_{j_1}) \geq \mathcal{L}(T_{p_1}, T_{q_1})$. Let k be the smallest index such that $i_k \neq p_k$ or $j_k \neq q_k$. If such k does not exist, then Algorithm 4 does compute a solution to IP1. Otherwise, such k must exist. Since greedy method always chooses the maximum $\mathcal{L}(T_i, T_j)$ that does not violate the constraint (2) of the Integer Programming **IP1**, $\mathcal{L}(T_{i_k}, T_{j_k}) \geq \mathcal{L}(T_{p_k}, T_{q_k})$. Without loss of generality, we assume that $x^{opt}_{a,j_k} = 1$ and $T_{b_1}, T_{b_2}, \cdots, T_{b_y}$ are the trees such that $x^{opt}_{i_k,b_\ell} = 1$ and T_{j_k} conflicts with T_{b_ℓ} regarding T_{i_k}. Let r_{b_i} be the root of tree $\mathcal{L}(T_{i_k}, T_{b_i})$, then r_{b_i} must be the descendant of root of tree $\mathcal{L}(T_{i_k}, T_{j_k})$. Since T_{b_i} does not conflict with T_{b_j} for any pair of i, j, then r_{b_i} is neither an ancestor nor a descendant of r_{b_j}. Now we modify the solution x^{opt} as follows. First, let $x^{opt}_{i_k,j_k} = 1$ and $x^{opt}_{i_k,b_\ell} = 0$ for $1 \leq \ell \leq y$. Then, set $x^{opt}_{a,b_\ell} = 1$ if it did not violate Constraint (2) and $x^{opt}_{z,b_\ell} = 0$ for any z that does not violate Constraint (2). The modified solution x^{opt} must satisfy Constraint (2). After this modification, the only difference between original solution and modified solution is the threes that overlap T_{i_k} and T_a. Let δ_1 be the increase of the overlap by replacing T_{j_k} with trees

$T_{b_1}, T_{b_2}, \cdots, T_{b_y}$, and δ_2 be the decrease of the overlap by replacing $T_{b_1}, T_{b_2}, \cdots, T_{b_y}$ with T_{j_k}. Since, $\mathcal{L}(a, T_{T_{j_k}})$ is an AoN-subtree of $\mathcal{L}(T_{i_k}, T_{T_{j_k}})$, then $\delta_1 \geq \delta_2$. Thus, $\sum_{i \neq j} x_{i,j} \cdot |\mathcal{L}(T_i, T_j)|$ does not increase. This is a contradiction, which proves that Algorithm 4 computes a solution to the Integer Programming (1). $\Box$

Since $\sum x_{i,j}^{opt} \cdot (|T_j - \mathcal{L}(T_i, T_j)|) \leq |\mathbf{LCST}(\mathcal{T})|$, we found a cycled forest that is smaller than the size of **LCST**. Notice that cycled forest is not a valid tree because it violates the tree property. Following we will show that simple modification based on the cycled tree that was found by Algorithm 4 does output a valid common AoN-supertree. In the meanwhile, we also will show that the increase of the size is at most a constant time of the size of the original cycled forest.

Algorithm 5 Modify the cycled forest.

Input: Cycled Forest $\mathbb{CF}$.
Output: A valid virtual overlap tree.

1: Rank all cycled tree in cycled forest $\mathbb{CF}$ in arbitrary order, say $\mathbb{CF}_1, \mathbb{CF}_2, \ldots, \mathbb{CF}_k$.
2: For a cycled tree $\mathbb{CF}_i$, find the unique cycle C_i in tree $\mathbb{CF}_i$. Let r_i be any node in C_i and $\mathcal{P}(r_i)$ be its parent, then we remove the edge between r_i and $\mathcal{P}(r_i)$.
3: Concatenate the tree $\mathbb{CF}_i$ to $\mathbb{CF}_i$ without conflict for $i = 2, \cdots, k$, *i.e.*, let r_i be a child of some node in $\mathbb{CF}_{i-1}$.
4: Output the final tree as a valid virtual overlap tree.

Borrowing some ideas from the construction of shortest common super-string (see [5] for more details), we have the following lemma

Lemma 8. *For any two cycles C_i and C_j in two different cycle tree $\mathbb{CF}_i$ and $\mathbb{CF}_j$, let s_i and s_j be any node in C_i and C_j respectively, then $\mathcal{L}(s_i, s_j) \leq |C_i| + |C_j|$.*

Theorem 5. *Algorithm 5 finds a common AoN-supertree of $\mathcal{T}$ with size $\leq 6 \cdot |$**LCST**$(\mathcal{T})|$.*

PROOF. Let $\mathbb{CF}_1, \mathbb{CF}_2, \ldots, \mathbb{CF}_k$ be all the cycled trees computed by Algorithm 4. Then, $\sum_{i=1}^{k} |\mathbb{CF}_i| \leq |\mathbf{LCST}(\mathcal{T})|$. Let C_i be the cycle in cycled tree $\mathbb{CF}_i$, and s_i be the node whose corresponding tree has the largest size in cycle C_i. Lemma 8 shows that $\mathcal{L}(s_i, s_j) \leq |C_i| + |C_j|$ for any pair of cycles C_i and C_j. Unlike the string case, it is possible that two or more trees overlap with the same tree. However, if nodes $s_{j_1}, s_{j_2}, \ldots, s_{j_k}$ overlap with the tree s_i in $\mathbf{LCST}(\mathcal{T})$, then $\sum_{\ell=1}^{k} |\mathcal{L}(s_i, s_{j_\ell})| \leq \sum_{\ell=1}^{k} |C_{j_\ell}| + |C_i| + |\mathbb{CF}_i|$. Thus, $\sum_{\ell=1}^{k} |\mathcal{L}(s_i, s_{j_\ell})| \leq \sum_{\ell=1}^{k} |C_{j_\ell}| + |C_i| + |\mathbb{CF}_i|$. For each s_i, let $\mathcal{P}(s_i)$ be its nearest ancestor in the virtual overlap graph of $\mathbf{LCST}(\mathcal{T})$, then $\sum_{s_i} |\mathcal{P}(s_i) \Cap s_i| \leq \sum_{s_i} |\mathcal{L}(\mathcal{P}(s_i), s_i)| \leq 2 \sum_{s_i} |C_i| + |\mathbb{CF}_i| \leq 4 \sum_{s_i} |\mathbb{CF}_i|$. Thus, $|\mathbf{LCST}(\mathcal{T})| \geq \sum_{s_i} |s_i| - \sum_{s_i} |\mathcal{P}(s_i) \Cap s_i| \geq \sum_{s_i} |s_i| - 4 \sum_{s_i} |\mathbb{CF}_i|$. Recall the virtual overlap tree computed by Algorithm 5 has the size at most $\sum_{s_i} |s_i| + \sum_{s_i} |\mathbb{CF}_i|$. Thus, $\sum_{s_i} |s_i| + \sum_{s_i} |\mathbb{CF}_i| \leq |\mathbf{LCST}(\mathcal{T})| + 5 \sum_{s_i} |\mathbb{CF}_i| \leq 6 |\mathbf{LCST}(\mathcal{T})|$. $\Box$

Theorem 6. *The time complexity of our approach is $O(n \cdot m^2)$, where n is the number of trees and m is the number of total nodes in these n trees.*

PROOF. Note that $m = \sum_{T_i \in \mathcal{T}} |T_i|$, and the time complexity to find the maximum overlap of T_j over T_i is $O(|T_i|^2 + |T_j|^2)$. Thus, finding the maximum overlap between each pair of trees is of time $O(n \cdot m^2)$. Algorithm 4 takes time $O(n^2 + n \log n)$ and Algorithm 5 only takes time $O(n)$. Thus, the overall time complexity is $O(n \cdot m^2)$. $\Box$

5 Conclusion

In this paper, we gave a 6-approximation algorithm for smallest common AoN-supertree problem. It has applications in glycobiology. There are several interesting problems left for future research. It is known that the simple greedy algorithm will have an approximation ratio 3.5 (conjectured to be 2). It remains to be proved whether a similar technique as of [4] can be used to reduce the approximation ratio of our method to 5.5. Further, it remains an open problem what is the lower bound on the approximation ratio of the greedy method when all trees of the tree set $\mathcal{T}$ are k-nary trees. Secondly, currently the best approximation ratio for superstring problem is 2.5 [6] (not using the greedy method). Since superstring is a special case of the AoN-supertree problem, it remains an open question whether we can get similar approximation ratio for AoN-supertree problem. The last but not least important problem is to improve the time-complexity of finding the maximum overlapping subtree of two trees. Is there a linear time algorithm that can find the maximum overlap AoN-subtree of two trees?

References

1. Turner, J.S.: Approximation algorithms for the shortest common superstring problem. Information and Computation (1989) 1–20
2. Teng, S., Yao, F.: Approximating shortest superstrings. In: Annual Symposium on Foundations of Computer Science. (1993)
3. Weinard, M., Schnitger, G.: On the greedy superstring conjecture. In: FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science. (2003) 387–398
4. Kaplan, H., Shafrir, N.: The greedy algorithm for shortest superstrings. Inf. Process. Lett. **93** (2005) 13–17
5. Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. Journal of the ACM **41** (1994) 630–647
6. Sweedyk, Z.: A $2\frac{1}{2}$-approximation algorithm for shortest superstring. SIAM Journal of Computing **29** (1999) 954–986
7. Armen, C., Stein, C.: A 2 2/3-approximation algorithm for the shortest superstring problem. In: Combinatorial Pattern Matching. (1996) 87–101
8. Hashimoto, K., Goto, S., Kawano, S., Aoki-Kinoshita, K., Ueda, N., Hamajima, M., Kawasaki, T., Kanehisa, M.: Kegg as a glycome informatics resource. Glycobiology (2005).
9. Aoki, K., Yamaguchi, A., Ueda, N., Akutsu, T., Mamitsuka, H., Goto, S., Kanehisa, M.: Kcam (kegg carbohydrate matcher): A software tool for analyzing the structures of carbohydrate sugar chains. Nucleic Acids Research (2004) W267–W272
10. Aoki, K., Yamaguchi, A., Okuno, Y., Akutsu, T., Ueda, N., Kanehisa, M., Mamitsuka, H.: Efficient tree-matching methods for accurate carbohydrate database queries. In: Proceedings of the Fourteenth International Conference on Genome Informatics (Genome Informatics, 14). (2003) 134–143 Universal Academy Press.
11. Aoki, K.F., Mamitsuka, H., Akutsu, T., Kanehisa, M.: A score matrix to reveal the hidden links in glycans. Bioinformatics **8** (2005) 1457–1463
12. Ueda, N., Aoki-Kinoshita, K.F., Yamaguchi, A., Akutsu, T., Mamitsuka, H.: A probabilistic model for mining labeled ordered trees: capturing patterns in carbohydrate sugar chains. IEEE Transactions on Knowledge and Data Engineering **17** (2005) 1051–1064