 ISBN：978-986-02-7580-3

Finding a Longest Increasing Subsequence on a Galled Tree

S. C. Chen, J. Y. Wu, G. S. Huang and R. C. T. Lee
Department of Department of Computer Science and Information Engineering
National Chi Nan University, Puli, Nantou Hsien, Taiwan, 54561
{s94321905, s97321509, shieng, rctlee}@ncnu.edu.tw

Abstract

Given an integer sequence $S = s_1 s_2 \cdots s_n$, the longest increasing subsequence (LIS for short) problem is to find a subsequence of S such that the subsequence is increasing and its length is longest. In this paper, we present an algorithm for finding a longest increasing subsequence on a galled tree in time $O(n \log\log n)$ and space O(n) where is the number of vertices in the galled tree.

1 INTRODUCTION

Given an integer sequence $S = s_1 s_2 \cdots s_n$, the longest increasing subsequence (*LIS* for short) problem is to find a subsequence of S such that the subsequence is increasing. That is, we are interested in subsequence $s_{i_1}, s_{i_2}, \cdots, s_{i_k}$ where $1 \leq i_1 < i_2 < i_k \leq n$, $s_{i_1} \leq s_{i_2} \leq \cdots \leq s_{i_k}$ and k is largest. For example, if S=4, 1, 3, 2, 5, we can see that (4, 5), (1, 3), (1, 2), (1, 5) and (1, 2, 5) are all increasing subsequences of and (1, 2, 5) is an *LIS*. Without loss of generality, we may assume that all numbers in the sequence are distinct.

The *LIS* problem, as well as its related problems, has drawn attentions from many researchers since 1960s [29]. A simple solution to this problem is by utilizing the longest subsequence problem (or *LCS* for short). Let T be the sorted list of S. It is easy to see that an *LCS* of S and T is the required *LIS*. Since the *LCS* problem can be solved in time $O(n^2)$ by using the dynamic programming, it implies that the LIS problem can also be solved in time $O(n^2)$. Conversely, we can also reduce the *LCS* problem to the *LIS* problem [16, 7]. Hence, these two problems are closely related. There is an $O(n \log n)$ -time algorithm for solving the *LIS* problem [14], and this algorithm is optimal under the comparison model [14, 22]. If the numbers in a sequence are restricted to , it is possible to design an $O(n \log\log n)$ -time algorithm [20, 6, 4] by using van Emde Boas trees [31, 32]. This time bound is further improved to $O(n \log\log n)$ where k is the length of an *LIS* in [7]. Studying the *LIS* problem arose many interesting research issues. For example, we may consider the average length of increasing subsequences in a random permutation, as well as studying the probability distribution of *LIS*s [23, 2, 8, 15, 27, 30]. The famous Erdos-Szekeres theorem [13] says that there always exists a length-$\lceil \sqrt{n} \rceil$ subsequence that is either increasing or decreasing, for any sequence of length n. There are many problems closely related to the *LIS* problem, such as the patience sorting [2], the Robinson-Schensted correspondence [29, 22], the maximum clique problem in circular-arc graphs and circle graphs, and finally, counting discrete points enclosed by an arbitrary triangle [3, 28]. Applications of the *LIS* problem to computational biology can also be found in [1, 19].

There exist many kinds of variations of the *LIS* problem. For example, we may consider the problem extracting an *LIS* for every sliding window of a fixed width on a given integer sequence [24, 10, 11]. Another variation considers finding *LIS*s on a circular integer list [25, 9]. The longest common increasing subsequence problem, which combines the requirements for the *LIS* problem and the *LCS* problem, can be found in [21]. Other variations (not a complete list here) can also be found in [26, 12, 5].

We propose a new kind of *LIS* problem. The general setting is to find out an *LIS* from a directed acyclic graph (DAG) whose vertices are labeled by integers. Here a subsequence in a DAG means a chain (i.e., a totally ordered subset with respect to the partial order defined by the DAG itself) on it, and we are interested in finding a longest increasing subsequence from all chains. We still cannot solve this problem efficiently, as compared with the *LIS* problem defined on a sequence. In this paper, we discuss a restricted version: the *LIS* problem on galled trees [18, 17]. Galled trees are invented by D. Gusfield in 2003

 ISBN：978-986-02-7580-3

for the purpose to represent evolutionary relations between species. However, a galled tree may not be a tree, and in fact, it is a DAG and can have confluent vertices (having two or more incoming edges). How to deal with confluent vertices on galled trees is critical to devising an efficient algorithm for solving the proposed problem.

The paper is organized as follows. In Section 2, we review a basic algorithm for solving the *LIS* problem on a sequence. In Section 3, we introduce the definition of galled trees, and in Section 4 we propose an algorithm for the *LIS* problem on galled trees. Finally, we give conclusion and some future work in Section 5.

2 A Review of a Simple Algorithm for the *LIS* Problem

D. Gusfield presented a simple algorithm for solving the *LIS* problem on a sequence, which is called the Nave Cover Algorithm. The algorithm splits the input sequence S into an ordered set of decreasing lists. The obtained set of decreasing lists is called a greedy cover. The number of decreasing lists in the greedy cover is exactly the length of an *LIS* for *S*. The Nave Cover Algorithm works as follows. Start from s_1 of *S*. Examine each successive element of S and place it at the end of the first (left most) list that it can extend. If there are no lists it can extend, then start a new list to the right end.

For example, we are given a sequence S=4, 2, 3, 8, 5, 6, 1, 7. The process of construction for the greedy cover of S is shown in Figure 1. Initially, we read 4 from S and make it alone as a list. Then we read 2, and append it to the end of 4. Next, we read 3, and it is larger than 2, the end of the first list. Hence, we create a new list for 3 alone and put the new list to the end of the greedy cover. Next, we read 8, and again, we find no list that can be extended. Hence, a new list for 8 is appended to the cover. The next number comes 5, and 5 is smaller than 8, the end of the third list. Notice that the end of the second list, which is 3, is smaller than 5. Hence, the third list is the required list that 5 can extend. This process is continuing, and at the end, we get a cover with five lists, which indicates that there is an *LIS* of length 5. We remark that numbers in each list are always decreasing, and the end elements in the lists are always increasing from left to right.

In the above illustration, we can only get the length of the *LIS*. If we want to obtain an *LIS*

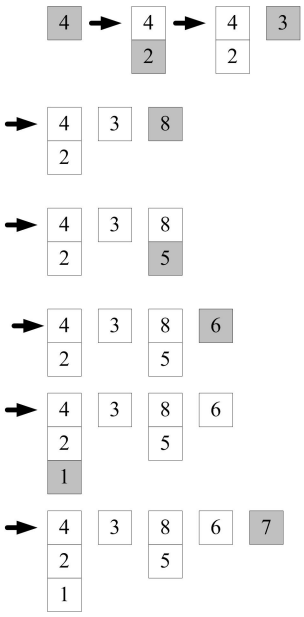

Figure 1: Building a greedy cover of S=4, 2, 3, 8, 5, 6, 1, 7.

for S instead of merely getting its length, we can create a back link for all but the first element in the lists. Let x be a newly added element. If we place x at the end of the kth list and suppose that y is the end element in (k-1)th list, we link back x to y. For example, all of the back links of the greedy cover for S=4, 2, 3, 8, 5, 6, 1, 7 is shown in Figure 2. After we create the back links, we can

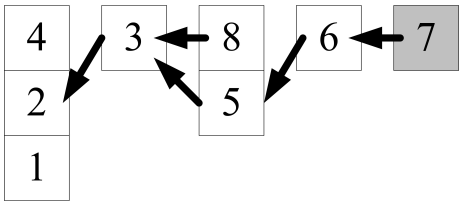

Figure 2: Back links in a greedy cover.

find a unique path from the end of the last list to an element in the first list. In Figure 2, this path is (7, 6, 5, 3, 2). Therefore, the required *LIS* for S=4, 2, 3, 8, 5, 6, 1, 7 is (2, 3, 5, 6, 7). Because all end elements in the cover list from left to right are increasing, the appropriate list for insertion can be identified by using a binary search in time

 ＩＳＢＮ：978-986-02-7580-3

$O(\log n)$, which makes the time complexity of the Nave Cover Algorithm to be $O(n \log n)$. If every element in S is limited between 1 to n, the van Emde Boas tree can reduce the time complexity to $O(n \log \log n)$.

If we only get the length of *LIS*, we could create the cover array (CA for short) to save the end element of each list. The cover array making algorithm[4] for S works as follows; Starting from s_1 of S and places it into CA(1). Examine each successive element s_i of S and place it at the CA(k) where CA(k) is the first larger than s_i from left to right and CA(k-1) $< s_i$. If s_i is the largest than all elements of CA, then start a new element of CA to the right of all existing CA and place it. For example, given a string S=4, 2, 3, 8, 5, 6, 1, 7. The process of construction for CA of S is in Figure 3.

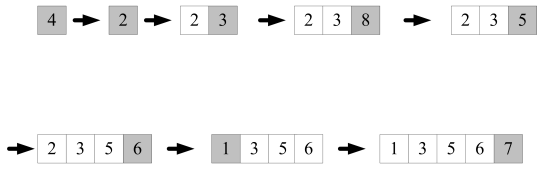

Figure 3: CA for S=4, 2, 3, 8, 5, 6, 1, 7.

3 Galled Trees

Galled trees are invented by D. Gusfield et al. [18, 17] in 2003, for the purpose to represent evolutionary relations between species. However, a galled tree may not be a tree, and in fact, it is a DAG and can have confluent vertices (having two or more incoming edges). In the following, we give some definitions of galled trees, which are modified from [18, 17].

Definition 1: In a directed acyclic graph, if a vertex v has two outgoing paths that meet at another vertex v, those two paths are called a recombination cycle. In this *recombination cycle*, we say that the vertex v is a *coalescent vertex* and the vertex v is a *recombination vertex*. If a vertex w has two outgoing path in the recombination cycle but it is not a *coalescent vertex*, we say that vertex w is a *split vertex*.

Definition 2: A recombination cycle is called a gall if it shares no vertex with any other recombination cycle.

Definition 3: A directed acyclic graph is called a galled tree if this graph is connected and contains exactly one vertex (which is called a root) with no incoming edges and every recombination cycle is a gall.

For example in Figure 4, $\{S_2, S_3, S_6, S_5\}$ and $\{S_{10}, S_{11}, S_{13}, S_{14}, S_{15}\}$ are both *recombination cycles*. S_2 and S_{10} are coalescent vertices, S_6 and S_{15} are *recombination vertices* and S_3, S_{11} and S_{14} are *split vertices*. This tree is a galled tree because these *recombination cycles* are galls.

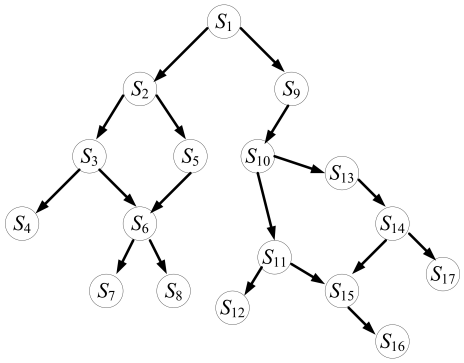

Figure 4: A galled tree.

Definition 4: Suppose that we are given a galled tree $\mathcal{G}$ where each vertex of it is associated with an integer. A chain on $\mathcal{G}$ is a sequence of integers such that (1) each number on the chain comes from some vertex of $\mathcal{G}$; and (2) for every two numbers on the chain there is a path from the first number to the second number on $\mathcal{G}$.

Our *LIS* problem can be defined as follows.

The *LIS* Problem on Galled Tree: Given a galled tree where each vertex is associated with an integer, our goal is to find a longest chain such that numbers in this chain are increasing.

4 Computing an *LIS* from a Galled Tree

In the *LIS* problem on a galled tree, we have to confront two phases. The first is on how to traverse all vertices in the galled tree and the second is on how to compute an *LIS* in a gall.

ＩＳＢＮ：978-986-02-7580-3

For the first phase, because a galled tree is like an ordinary tree, we can take a gall to a big vertex in the tree. Hence, we can use the depth first search (*DFS* for short) to traverse all vertices. For example, a tree is illustrated in Figure 4.

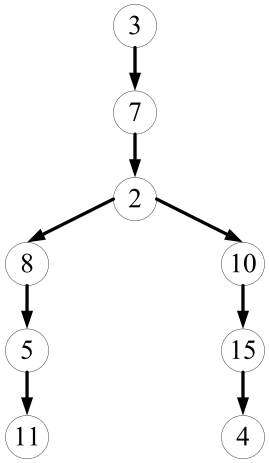

Figure 5: A tree shrinked from a galled tree.

In this tree in Figure 5, 3 is a root, 11 and 4 are leaves and 2 is a split vertex. The *DFS* order of this tree is 3, 7, 2, 8, 5, 11, 5, 8, 2, 10, 15, 4, 15, 10, 2, 7, 3. If there is a split vertex w on the gall, we can copy a path from the coalescent vertex to w to a new path and split an outgoing path which leaves the gall to connect the new path, as shown in Figure 5.

If we trace to meet a gall, we also use *DFS* to trace this gall. For example in Figure 6, we first trace from 3 to 6 (one of the path) and then back trace to 2. Finally, we trace from 2 to 11. The *DFS* order of this galled tree is 3, 7, 10, 3, 10, 2, 8, 5, 11.

In order to compute the *LIS* in a more complex gall, we can simplify the gall. For example, in Figure 7, S_3, S_{11}, S_{14} are in the recombination cycle and they also have an outgoing edge leaving the cycle. These vertexes are split vertices. If there is a split vertex w on the gall, we can copy a path from coalescent vertex to w to a new path and split an outgoing path which leaves the gall to connect the new path.

If we traverse from the root to a leaf, we would add numbers on this path to modify the cover or delete them from the cover. The time complexity of adding or deleting an element in the cover is $O(\log \log n)$ by using the van Emde Boas tree.

The second phase is to find the *LIS* in galled tree when we meet a gall. Because we use the *DFS* order to trace the galled tree in the first phase,

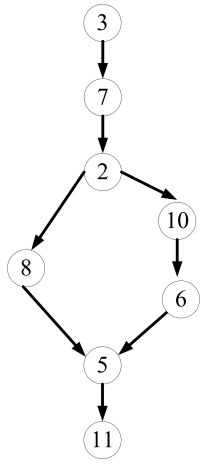

Figure 6: A gall.

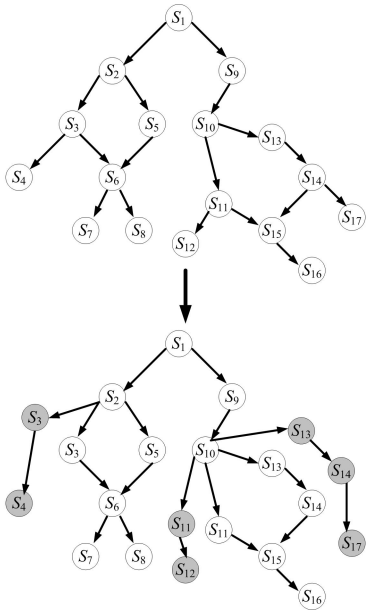

Figure 7: Simplifying the galls

 ＩＳＢＮ：978-986-02-7580-3

there are two different states to face, from the root to a leaf and from a leaf to the root. Suppose that the kth element of CA is CA(k) and the end of element of kth list is CA(k). When we read t_i in the galled tree in the state of the root to a leaf, our tactic is that we insert t_i into CA to replace CA(k) and put t_i into the kth list if CA(k-1) $< t_i <$ CA(k). We also create back link to CA(k-1) in the (k-1)th list and last link to CA(k) in the kth list. It is shown in Figure 8. If t_i is larger than CA(k) for all k, we insert t_i into CA(k+1) and put t_i into (k+1)th list. At the same time, we create a back link to CA(k) in the kth list. When we are tracing in the gall, beside to the above method, we record pairwise number (t_i, k) of each element in the gall to the temp list where t_i is the number of this element and t_i is in the kth list in the gall. If we back trace one of path in the gall, we use the element tl_j in the temp list to delete in CA and insert the last element of tl_j in the list but do not remove any element in all lists. After we finish tracing the gall, we use the element tl_j in the temp list to update the CA. If tl_j=(t_i, k) and the kth element of CA is CA(k), CA(k)=min{CA(k), t_i}.

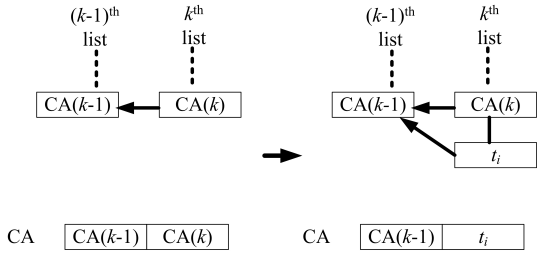

Figure 8: The base idea of our method.

In the other state, tracing from a leaf to the root, if we read t_i, we delete t_i in the kth list and CA. Moreover, we insert the element of last link of t_i into CA. When we meet a gall, we use each element in the temp list to delete the element in the list and CA. If we read t_i in the gall, we can easily find the information in temp list about where t_i is in the kth list. For this reason, we delete t_i in the kth list and CA and insert the element of last link of t_i into CA. Each step to modify CA only need $O(\log \log n)$ using van Emde Boas tree.

For example in Figure 9, we first read 3 and get CA and lists in the Figure 10.

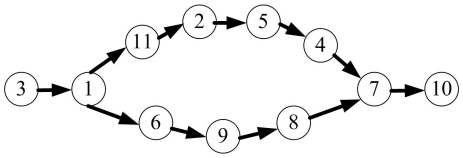

Figure 9: A path with a gall.

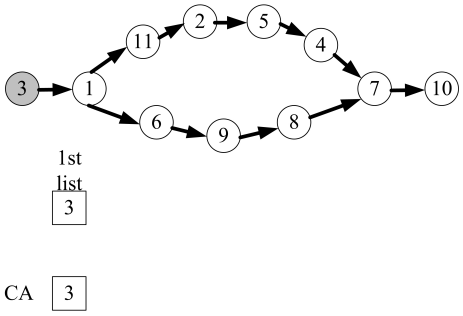

Figure 10: The CA and lists for S=3

We next read 1. Because 1 <CA(1), we insert 1 to replace CA(1). At the same time, we put 1 in the end of 1st list, create last link to 3 and shown in Figure 11.

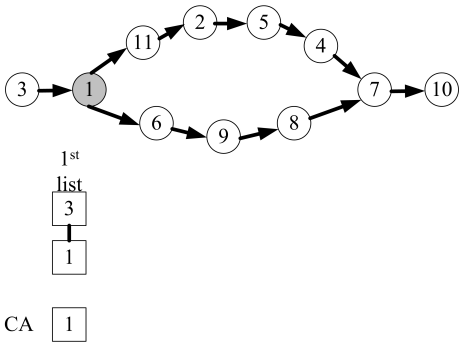

Figure 11: The CA and lists for S=3, 1.

We arbitrarily trace one of the path and read 11. 11 is the biggest than all elements in CA, we put 11 into CA(2). We put 11 into 2nd list and create back link to 1. At the same time, we record (11, 2) to temp list and shown in Figure 12.

When we read 2, because CA(1) = 1 < 2 <

ＩＳＢＮ：978-986-02-7580-3

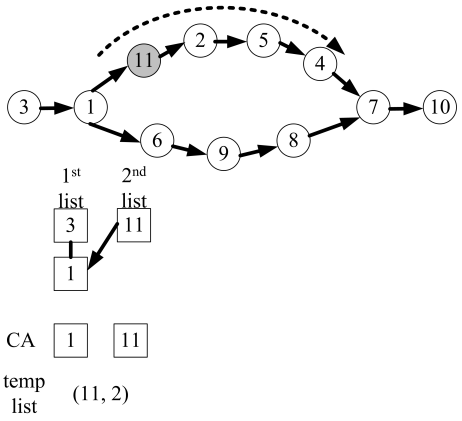

Figure 12: The CA and lists for tracing one path to 11.

11 = CA(2), we insert 2 to replace CA(2). We also put 2 in the 2nd list, create back link to 1 (CA(1)) in the 1st list and last link to 11 (original CA(2)) in the 2nd list. At the same time, we record (2, 2) to temp list. We go on reading 5. Because 5 is the biggest than all elements of CA, we put 5 into CA(3). We also put 5 in the 3rd list, create back link to 2 (CA(2)) in the 2nd list. At the same time, we record (5, 3) to temp list. Finally, we read 4. Because CA(2) = 2 < 4 < 5 = CA(3), we insert 4 to replace CA(3). We also put 4 in the 3rd list, create back link to 2 (CA(2)) in the 2nd list and last link to 5 (original CA(3)) in the 3rd list. At the same time, we record (4, 3) to temp list. The above process is shown in Figure 13.

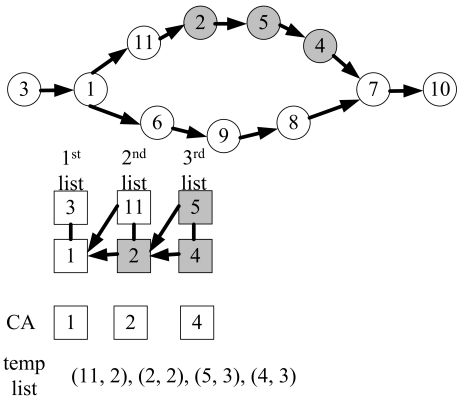

Figure 13: The CA and lists for tracing one path to 14.

When we finish tracing this path, we back trace from 4 to 11. We use all elements in the temp list and lists to modify CA. First we read 4. Because we get that 4 is in the 3rd list from the temp list and the element of last link of 4 is 5 from the 3rd list, we delete 4 in CA ,insert 5 (the element of last link of 4) into CA and shown in Figure 14.

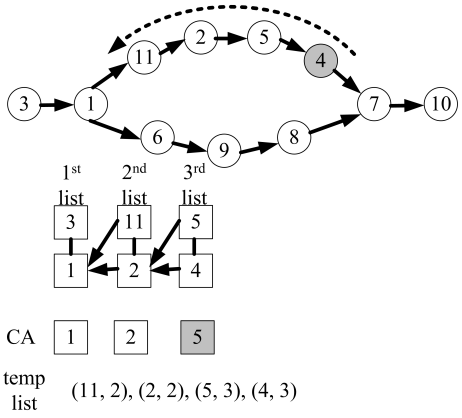

Figure 14: The CA and lists for back tracing the path.

We proceed to read 5. Because we get that 5 is in the 3rd list from the temp list and the element of last link of 5 is null from the 3rd list, we delete 5 in CA and shown in Figure 15.

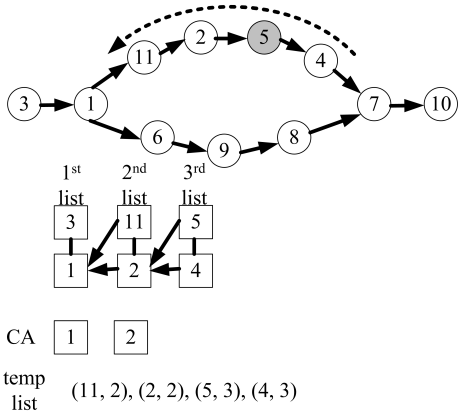

Figure 15: The CA and lists for back tracing the path to 5.

Finally, the CA and lists is shown in Figure after we back trace to 11 and shown in Figure 16.

We trace the other path in the gall. We first read 6. 6 is the biggest than all elements of CA, we insert 6 in CA(2). We also put 6 in the 2nd list and create back link to 1 (CA(1)) in the 1st

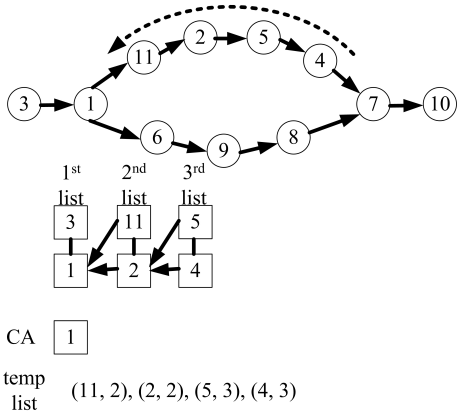

Figure 16: The CA and lists for back tracing the path to 11.

list. In this moment, we record (6, 2) into temp list and shown in Figure 17.

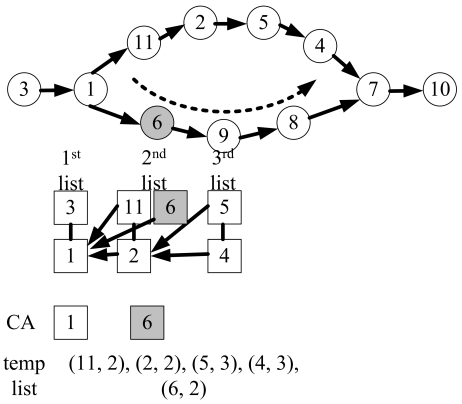

Figure 17: The CA and lists for tracing the other path to 6.

We proceed to read 9. 9 is the biggest than all elements of CA, we insert 9 in CA(3). We also put 9 in the 3rd list and create back link to 6(CA(2)) in the 2nd list. In this moment, we record (9, 3) into temp list and shown in Figure 18.

We proceed to read 8. Because CA(2) = 6 < 8 < 9=CA(3), we insert 8 to replace CA(3). We also put 8 in the 3rd list, create back link to 6 (CA(2)) in the 2nd list and last link to 9 (original CA(3)) in the 3rd list. At the same time, we save (8, 3) into temp list and shown in Figure 19.

Finally, we finish tracing two path of the gall. Because we want to get the CA including the information of two paths in the gall, we use all ele-

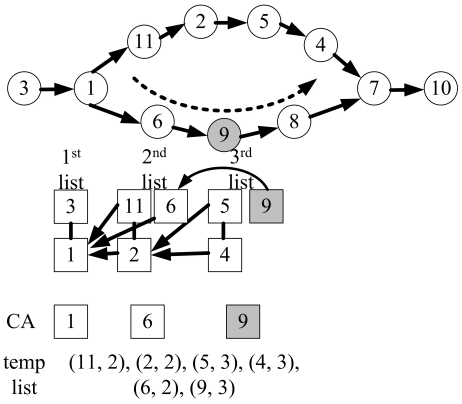

Figure 18: The CA and lists for tracing the other path to 9.

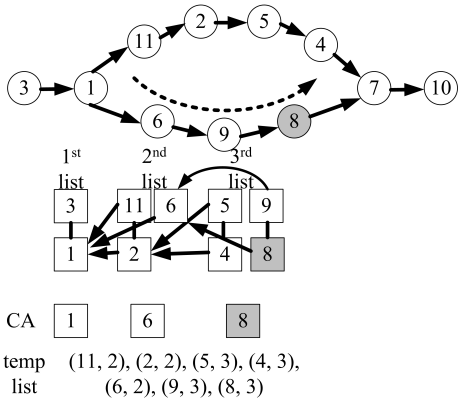

Figure 19: The CA and lists of finishing tracing the gall.

 ＩＳＢＮ：978-986-02-7580-3

ments of temp list to get the CA where CA(k)= min(CA(k), t_i). We first read (11, 2). Because CA(2) = 6 < 11, CA is still equal to 6. The second element in the temp list is (2, 2). Because 2 is smaller than CA(2)=6, we replace 2 into CA(2). The next element is (5, 3). Because 5 is smaller than CA(3)=8, we replace 5 into CA(3). The next element is (4, 3). Because 4 is smaller than CA(3)=5, we replace 4 into CA(3). We continually use (6, 2), (9, 3) and (8, 3) to update CA. Finally, CA=1, 2, 4.

We proceed to read 7. Because 7 is the biggest than all elements of CA, we put 7 into CA(4) and 4th list. We also create back link to 4 in the 3rd list and shown in Figure 20.

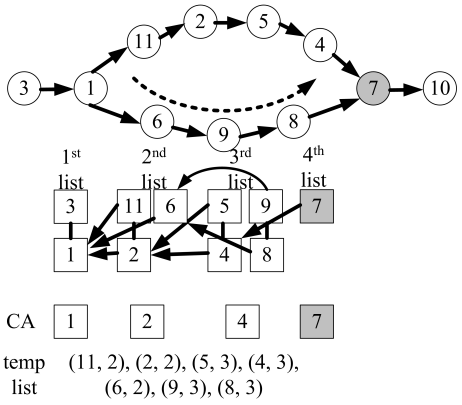

Figure 20: The CA and lists for tracing to 7.

We read 10. Because 10 is the biggest than all elements of CA, we put 10 into CA(5) and 5th list. We also create back link to 7 in the 4th list and shown in Figure 21.

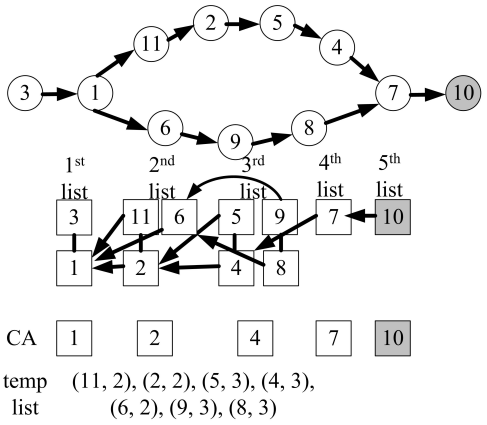

Figure 21: The CA and lists for tracing to 10.

Therefore, we get the length of *LIS* is 5 and *LIS* is (1, 2, 4, 7, 10) from the lists.

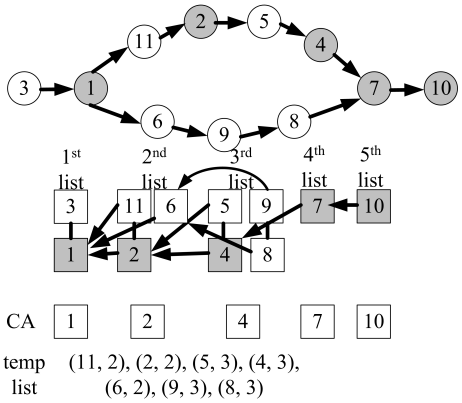

Figure 22: The CA and lists for the path with a gall.

If we want to back trace from 10 to 1, we delete the element in the lists and synchronous update CA. We first read 10, we delete 10 in the 5th list and CA. The process for 7 is the same as 10. When we back trace to meet a gall, we use all elements of temp list (from the end) to delete the elements in the list and update CA. We first read (8, 3) in the temp list. From this element, we get 8 is end of 3rd list. However, 3rd list has two end, we only check these two ends which 8 is and delete it. At the same time, we check weather 8 is in 3rd list or not. If 8 does not exist in CA, we can ignore to delete 8 in CA. This checking only need O(1) using the van Emde Boas tree. The process for (9, 3) and (6, 2) is the same as the process for (8, 3). We go on reading (4, 3). Because 4 is in the end of 3rd list and exists in CA, we delete 4 in the 3rd list and CA and insert 5 (the element of last link of 4) into CA. After we process (5, 3), (2, 2) and (11, 2), the result of lists is the same as Figure 17.

5 Conclusion and Future Work

In this paper, we propose an algorithm for finding an LIS on a galled tree in time $O(n \log\log n)$and space O(n) where n is its size. It is not difficult to see that our technique can also be applied to more general graphs rather than galled trees. It is interesting to answer two questions. The first is to identify the largest class of graphs that our technique could apply. The second problem is to conquer the LIS problem on a general DAG.

ＩＳＢＮ：978-986-02-7580-3

Algorithm 1: Computing an *LIS* from a galled tree

Input: A galled tree $\mathcal{G}$ where each vertex is associated with an integer
Output: Find an LIS for $\mathcal{G}$
Preprocessing
Detect all galls in the galled tree and locate them. If there is a split vertex w on the gall, copy a path from its coalescent vertex v to w to form a new path.
begin
Traverse $\mathcal{G}$ from the root using the *DFS*. While Read t_i in $\mathcal{G}$
Step 1: Starting from t_1 of $\mathcal{G}$ and places it into CA(1) and 1st list. Examine each successive element t_i of $\mathcal{G}$ and place it at the CA(k) and kth list where CA(k) is the first bigger then s_i from left to right and CA(k-1) $< s_i$. If s_i is the biggest then all elements of CA, then place it in the starting a new element of CA and a list to the right of all existing CA and lists. Create back link to CA(k-1) in the (k-1)th list and last link to original CA(k) in the kth list.
Step 2: If we meet a gall in $\mathcal{G}$, trace one of the paths in the gall (not included *recombination vertex*) and update CA and then lists. At the same time, we save each element in the gall and its location in the lists to temp list.
Step 3: Back trace this path to *coalescent vertex* and move the elements of this path in the cover to the temp list. Use the elements in temp list and lists to update CA.
Step 4: Trace the other path and update CA and lists.
Step 5: Compare these elements in the temp list and CA after finishing tracing the path. If this element in kth list is smaller than CA(k), we insert it into CA.
Step 6: When we arrive a leaf, we save the length of CA to *LLIS* if this length $>$ *LLIS*.
Step 7: If we back trace from leaf to another vertex t_j, delete t_j in the lists and CA and insert the element of last link of t_j. If we meet a gall, use the elements in the temp list to delete the list and CA.
After we finish traversing $\mathcal{G}$, *LLIS* is the length of *LIS* in this galled tree. Trace $\mathcal{G}$ again until the length of CA is equal to *LLIS*. Track the back link from an element of last list to an element of the first list in this cover. We can trace from this element of the first list to the list to find the *LIS*.
end

References

[1] R. D. Fleischmann J. Peterson O. White A. L. Delcher, S. Kasif and S. L. Salzberg. Alignment of whole genomes. *Nucleic acids research*, 27(11):2369–2376, 1999.

[2] D. Aldous and P. Diaconi. Longest increasing subsequences: From patience sorting to the baik-deift-johansson theorem. *Bull. Amer. Math. Soc*, 36:413–432, 1999.

[3] R. Bar-Yehuda and Fogel S. Maximal sets of k-increasing subsequences with applications to counting points in triangles. technical report 596. Technical report, Department of Computer Science, Technion, 1989.

[4] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.

[5] C. B. Yang C. T. Tseng and H. Y. Ann. Minimum height and sequence constrained longest increasing subsequence. *Journal of Internet Technology*, 10:173–178, 2009.

[6] M. S. Chang and F. H. Wang. Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. *Information Processing Letters*, 43(6):293 – 295, 1992.

[7] M. Crochemore and E. Porat. Fast computation of a longest increasing subsequence and application. *Information and Computation*, 208:1054–1059, September 2010.

[8] J. Baik P. Deift and K. Johansson. On the distribution of the length of the longest increasing subsequence of random permutations. *Journal of the American Mathematical Society*, 12:1119–1178, 1999.

[9] S. Deorowicz. An algorithm for solving the longest increasing circular subsequence problem. *Information Processing Letters*, 109:630–634, 2009.

[10] H. Yuan E. Chen and L. Yang. Longest increasing subsequences in windows based on canonical antichain partition. In *In Proceedings of ISAAC*, pages 1153–1162, 2005.

[11] H. Yuan E. Chen and L. Yang. Longest increasing subsequences in windows based on canonical antichain partition. *Theoretical Computer Science*, 378(3):223–236, 2007.

[12] A. Elmasry. The longest almost-increasing subsequence. *Information Processing Letters*, 110(16):655 – 658, 2010.

[13] P. Erdos and G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[14] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[15] P. Groeneboom. Hydrodynamical methods for analyzing longest increasing subsequences. *Journal of Computational and Applied Mathematics*, 142:83–105, May 2002.

[16] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1999.

[17] D. Gusfield, S. Eddhu, and C. Langley. The fine structure of galls in phylogenetic networks. *INFORMS Journal on Computing*, 16:459–469, September 2004.

[18] D. Gusfield, E. Satish, and C. Langley. Efficient reconstruction of phylogenetic networks with constrained recombination. In *Proceedings of the IEEE Computer Society Conference on Bioinformatics*, CSB '03, pages 363–374, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Y. T. Tsai H. P. Lee and C.Y. Tang. A seriate coverage filtration approach for homology search. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 180–184, New York, NY, USA, 2004. ACM.

[20] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, May 1977.

[21] C. P. Huang I. H. Yang and K. M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5):249–253, 2005.

[22] D. E. Knuth. Permutations, matrices, and generalized young tableaux. *Pacific Journal of Mathematics*, 34:709–727, 1970.

[23] V. Lifschitz and B. Pittel. The number of increasing subsequences of the random permutation. *Journal of Combinatorial Theory, Series A*, 31(1):1–20, 1981.

[24] A. M. Hamel A. Lpez-Ortiz S. S. Rao M. H. Albert, A. Golynski and M. A. Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321(2-3):405–414, 2004.

[25] D. Nussbaum J.-R. Sack M. H. Albert, M. D. Atkinson and N. Santoro. On the longest increasing subsequence of a circular list. *Information Processing Letters*, 101:55–59, 2007.

[26] M. D. Atkinson-H. Ditmarsch B. Handley C. C. Handley M. H. Albert, R. A. Aldred and J. Opatrny. Longest subsequences in permutations. *Australasian Journal of Combinatorics*, 28:225–238, 2003.

[27] M. Loebl M. Kiwi and J. Matousek. Expected length of the longest common subsequence for large alphabets. *Advances in Mathematics*, 197(2):480–498, 2005.

[28] J. Matousek and E. Welzl. Good splitters for counting points in triangles. *Journal of Algorithms*, 13:307–319, June 1992.

[29] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

[30] H. Thomas and A. Yong. Longest increasing subsequences, plancherel-type measure and the hecke insertion algorithm. *Advances in Applied Mathematics*, 46(1-4):610–642, 2011.

[31] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.

[32] P. van Emde Boas R. Kaas and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976.