An Algorithm for Enumerating all Directed Spanning Trees in a Directed Graph

Takeaki UNO

Department of Systems Science, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan. uno@is.titech.ac.jp

Abstract: A directed spanning tree in a directed graph $G = (V, A)$ is a spanning tree such that no two arcs share their tails. In this paper, we propose an algorithm for listing all directed spanning trees of G. Its time and space complexities are $O(|A| + N\mathrm{D}(|V|, |A|))$ and $O(|A| + \mathrm{DS}(|V|, |A|))$, where $\mathrm{D}(|V|, |A|)$ and $\mathrm{DS}(|V|, |A|)$ are the time and space complexities of the data structure for updating the minimum spanning tree in an undirected graph with $|V|$ vertices and $|A|$ edges. Here N denotes the number of directed spanning trees in G.

Keywords directed spanning tree, listing, enumerating algorithm

1 Introduction

Let $G = (V, A)$ be a directed graph with vertex set V and arc set A. An arc is specified by both of its endpoints. One of them is called its *head* and the other is called its *tail*. A directed spanning tree of G is a spanning tree in which no two arcs share their tails. Each vertex is the tail of exactly one arc of the directed spanning tree except for a special vertex r. We call r the *root* of the spanning tree.

Directed spanning trees have been studied in many fields. For instance, many problems on road and telephone networks have been formulated as some optimization problems of directed spanning trees. Some of them have complicated objective functions, and we can hardly solve them in efficient time. For those problems, one of the most simple approaches is to use enumerating. The branch-and-bound method, which is one of the most popular approaches, can be also considered as a kind of enumerating. In this method, the time complexity of the enumerating algorithm greatly influences to its speed. Therefore, improvements of enumerating algorithms largely enhance the efficiency for those problems.

In this paper, we consider the problem of enumerating all directed spanning trees with the root r of the given graph G. This problem has been studied for nearly 20 years. In 1978, H. N. Gabow and E. W. Myers proposed an algorithm which runs in $O(|A| + N|A|)$ time and $O(|A|)$ space [1]. Here N denotes the number of directed spanning trees in G. In 1992, H. N. Kapoor and H. Ramesh improved the time complexity to $O(|A| + N|V|)$

[2]. Since a directed spanning tree requires $O(|V|)$ time for outputting, it is the optimal algorithm in the sense of both time and space complexities if we have to output all of them explicitly. On the other hand, in undirected graphs, *compact output methods* have been studied to shorten the size of outputs [2, 3]. They output a spanning tree by differences from the previous outputted tree. By outputting all spanning trees in a special order, we can reduce the total size of outputs to $O(N)$. A. Shioura, A. Tamura and T. Uno proposed an optimal algorithm with the compact output method for enumerating all undirected spanning trees [3]. In their algorithm, they utilizes the reverse search. Our method in this note is also based on this reverse search technique. It can be also considered a kind of the back tracking method, which is used in [2].

Here we propose an algorithm for enumerating all directed spanning trees in $O(|A| + N\mathrm{D}(|V|, |A|))$ time. Our algorithm constructs and preserves an undirected graph with $|V|$ vertices and at most $|A|$ weighted edges. This graph is modified after each directed spanning tree is encountered. To update the data structure for finding new directed spanning trees, we delete, add or change the weight of an edge, and find the minimum spanning tree of the graph in each iteration of the algorithm. Since only few edges are changed in each iteration, we construct the minimum spanning tree of the changed graph from the previous one by some *updating spanning tree algorithm*.

Therefore the time complexity of our algorithm depends on the time complexity of the updating algorithm, which is denoted by $\mathrm{D}(|V|, |A|)$. The space complexity also depends on the updating algorithm. We denote its space complexity by $\mathrm{DS}(|V|, |A|)$. The update operations are adding, deleting and changing the weight of an edge. Some data structures are proposed for these updating operations of the minimum spanning tree. G. N. Frederickson proposed an $O(|A|^{1/2})$ time data structure [5], and D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig improved the time complexity into $O(|V|^{1/2} \log(|A|/|V|))$ [4]. Their time complexities are smaller than $O(|V|)$. There use only $O(|A|)$ space, thus we do not lose the optimality of space complexity by our improvement.

2 The Reverse Search and the Parent-Child Relationship

We use a technique called reverse search for enumeration problems. The reverse search requires a parent-child relationship on those objects to be enumerated. The relationship has to satisfy: i) each object except for a specified object r_0 has a parent object and ii) each object is not a proper ancestor of itself.

Let us consider the graph representation of this parent-child relationship. An object corresponds to a node v of the graph and an edge (v, u) is included in the graph if and only if u corresponds to its parent. By the condition of the relationship, the graph contains no cycle and forms a rooted spanning tree. This tree is called an *enumeration tree*. In Figure 1, we show an example of an enumeration tree. The reverse search traverses all nodes

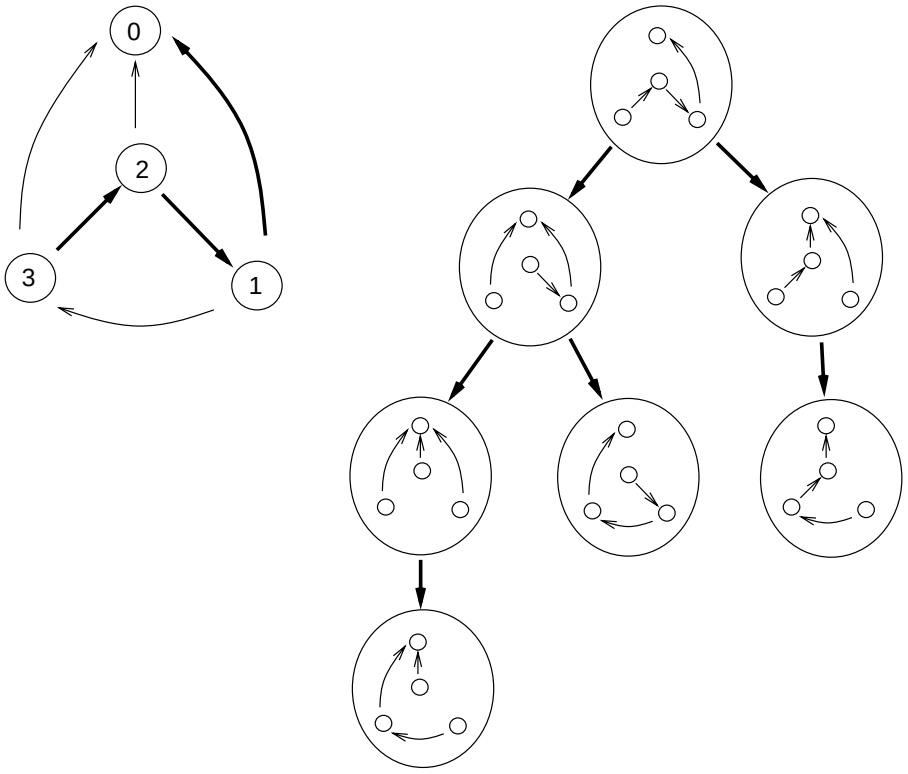

Fig1: The enumeration tree. The emphasized line of the graph is T_0.

of the enumeration tree by some depth-first-search scheme from the specified object r_0. Even if the size of the enumeration tree is huge, only a small memory space is required, since we can traverse it by only finding the parent and all children of the current object. Hence an important point in speeding up reverse search is "How to enumerate all children efficiently fast."

To enumerate all directed spanning trees, we define a parent-child relationship among them as below. Let T_0 be a depth-first-search tree of the given graph G. A depth-first-search tree is a directed spanning tree which arises from the traversal route of a depth-first-search. We assume that any search traverses an arc from its head to tail. Using this specified directed tree T_0, we introduce the parent-child relationship. In the relationship, T_0 corresponds to the specified object r_0. Let the indices of the vertices of G be as the order of the traversal of the depth first search. Similarly, we define the indices of arcs of G by indices of their tails. The parent of a directed spanning tree $T_c \neq T_0$ is defined by the directed spanning tree $T_p = T_c \setminus f \cup e$ where e is the minimum index arc e in $T_0 \setminus T_c$ and f is the arc of T_c sharing its tail with e. Since f shares its tail only one arc of T_c, T_p is uniquely defined. From the definition of the index, there always exists a path from the root to the head of e. Hence e is not contained in any cycle and T_p forms a directed spanning tree. In this parent-child relationship, T_c is not a proper ancestor of itself, as T_p contains just one more arc of T_0 than T_c.

To output directed spanning trees compactly, we consider differences between a directed spanning tree and the one outputted just before by the reverse search. Some of them may have $O(|V|)$ differences, but the total differences over all directed spanning trees does not reach $O(|V|N)$. Since the reverse search traverses the enumeration tree by the depth-first-search, the total amount is twice the sum of differences between a child and its parent over all directed spanning trees. Any directed spanning tree differs by only two arcs from its parent, thus the total the size of outputs can be reduced into $O(N)$ by outputting only differences.

Next we show the method of enumerating all children of a directed spanning tree T_p. Let $v^*(T_p)$ be the minimum index among arcs in $T_0 \setminus T_p$. Exceptionally, we define $v^*(T_0)$ by ∞. Let us construct T_c by removing an arc e from T_p whose index is less than $v^*(T_p)$ and adding an arc $f \neq e$ sharing its tail with e. If T_c forms a tree, it is a child of T_p from the definition. Conversely, in the case that $T_p = T_c \setminus f \cup e$ is the parent of T_c, the index of e is less than $v^*(T_p)$. Thus all children of T_p can be found by the above method. To find children, we deal with only vertices and arcs whose indices are less than $v^*(T_p)$. Hence we call them *valid*.

For a tree T, we call an arc not in T a *back-arc*, if its tail is an ancestor of its head, and otherwise a *non-back-arc*. In the above method, if f is a back-arc of T_p, then $T_c = T_p \setminus e \cup f$ contains a cycle and is not a directed spanning tree. If f is a non-back-arc of T_p, there always exists a path from r to the head of f in T_c and T_c contains no cycle. Hence each child of T_p is obtained by adding a valid non-back-arc f and removing an arc e sharing its tail with f. Valid non-back-arcs have a one-to-one correspondence with the children and the number of children is same as the number of valid non-back-arcs. By maintaining the set of all valid non-back-arcs, finding all children can be accomplished sufficiently easily.

We now describe the framework of our algorithm.

ALGORITHM: Enum_Directed_Spanning_Trees(G)
Step 1: Find T_0 by a depth-first-search.
Step 2: Assign indices for each vertex.
Step 3: Classify arcs not in T_0 into the back-arc set and the non-back-arc set.
Sort them in order of their indices.
Step 4: Call Enum_Directed_Spanning_Trees_Iter(T_0).

ALGORITHM: Enum_Directed_Spanning_Trees_Iter(T_p)
Step 1: For each valid non-back-arc f of T_p, do the following.
Step 2: Construct T_c by adding f and removing an arc e.
Output it by the difference from the previous outputted one.
Step 3: List all valid non-back-arcs of T_c in order of their indices.
Step 4: Call Enum_Directed_Spanning_Trees(T_c) recursively.

Step 1 to 3 of the algorithm ENUM_DIRECTED_SPANNING_TREES(G) runs in $O(|A|)$ time. The construction of a child of T_p in Step 2 of ENUM_DIRECTED_SPANNING_TREES_ITER may be done in $O(1)$ by swapping two arcs. Step 3 lists all valid non-back-arcs of T_c, and the number of them is equal to the number of children of T_c in the enumeration tree. Thus total time spent until Step 3 is the time to find one valid non-back-arc of T_c per one outputted directed tree.

The earlier algorithm [2] takes $O(|V|)$ time for finding one non-back-arc in the worst case and it is the bottle neck of the time complexity while other parts of the algorithm take only $O(1)$ time. Our improvement on this part reduces its time complexity to the time necessary to update the minimum spanning tree of an undirected graph. In the next section, we show a data structure which is the key to the improvement.

3 An Improved Data Structure

To enumerate all valid non-back-arcs sufficiently fast in **Step 3**, we show the following two conditions. They are also stated in [2]. We denote the tail of e by v and the nearest common ancestor of v and the head of f by w.

Lemma 1 *Let $T_c = T_p \setminus e \cup f$ be a child of T_p. A valid non-back-arc of T_p is a valid non-back-arc of T_c if its index is less than $v^*(T_c)$.*

Proof : Since T_0 is a depth-first-search tree, the index of the tail of f is larger than $v^*(T_c)$. Thus for all valid vertices of T_c, all its descendants in T_c are also descendants T_p. Therefore any valid vertex is not an ancestor of a vertex v in T_c if it is not an ancestor in T_p. ∎

Lemma 2 *Let P be the vertex set of the interior points of the path from the head of e to w on T_p, where interior points of the path are vertices of the path which are not its endpoints. A back-arc of T_p is a non-back-arc of T_c if and only if its head is a descendant of v and its tail is in P.*

Proof : A descendant of v is also a descendant of all vertices of P but not in T_c. Thus if the head of a back-arc is a descendant of v and its tail is in P, it is a non-back-arc of T_c. Conversely, only descendants of v have ancestors which are not ancestors in T_c and only vertices of P have descendants which are not descendants in T_c. Thus any back-arc of T_p has the head in descendants of v and a tail in P if it is a non-back-arc of T_c. ∎

From these lemmas, the valid non-back-arc set of T_c is composed by those back-arcs of T_p and valid non-back-arcs of T_p whose indices are less than $v^*(T_c)$. By listing both of them in order of their indices, we can obtain the non-back-arc set sorted by their indices. The former can be listed easily if we have the set of valid non-back-arcs of T_p sorted by

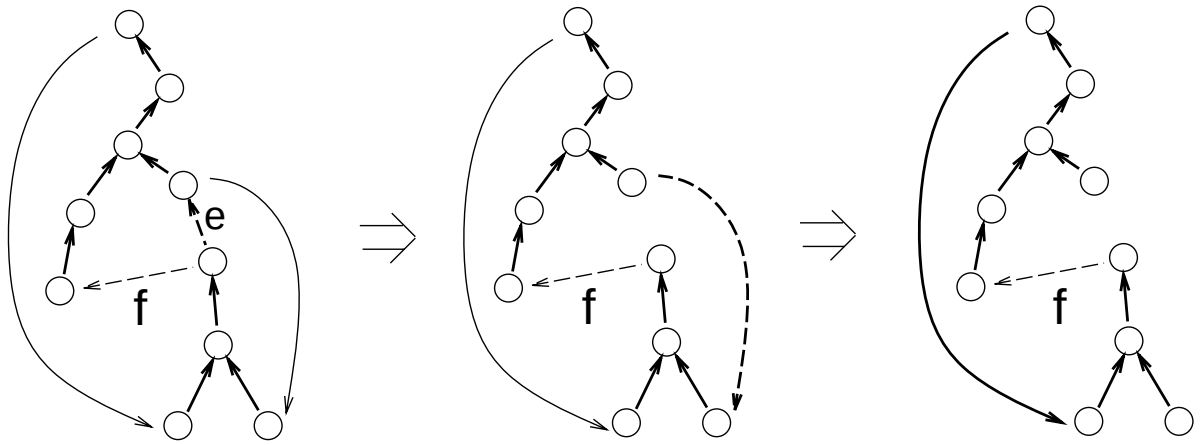

Fig2: The way of $B(T)$ changes.

their indices. But the latter is hard to list with only simple data structures. We use the following data structure shown as below.

For a directed spanning tree T_p, let $B(T_p)$ be an undirected graph with vertex set V and edge set composed by arcs of T_p, valid back-arcs of T_p and some of other rest back-arcs. $B(T_0)$ is an undirected graph composed by T_0 and all of its back-arcs. The weight of edges in the graph $B(T_p)$ is defined as 0 for arcs of T_p and ($|V|+1-$ index) for others. The minimum spanning tree of $B(T_p)$ is T_p.

By removing a valid arc e of T_p from the undirected graph $B(T_p)$, the minimum spanning tree of $B(T_p)$ is split into two. One of them contains all descendants of v and the other contains the rest. The minimum spanning tree of the graph $B(T_p)\setminus e$ is obtained by adding the minimum weight cut edge b, which has endpoints in both of those two trees. From above lemmas, b is a valid non-back-arc of T_c if and only if the tail of b is in P. In the case that b does not exist, no back-arc of T_p is a non-back-arc of T_c. Since b is a back-arc of T_p, the tail of b is in the path from v to r and its head is a descendant of v. From the definition of the weight, no back-arc connects a descendant of v and an interior point of the path from the tail of b to v. We show an example of the way of $B(T)$ changes in Figure 2. Emphasized lines are a part of the minimum spanning tree. Doted lines are eliminated and a new edge will be inserted.

Therefore we can find a new valid non-back-arc of T_c by updating the minimum spanning tree. By removing b and applying the method repeatedly, we can enumerate all new valid non-back-arcs of T_c. The algorithm is showed as follows. It outputs all new non-back-arcs and updates the graph from $B(T_p)$ to $B(T_c)$.

ALGORITHM: ENUM_NON-BACK-ARCS($T_p, B(T_p), e, f$)

Step 1: Remove e from $B(T_p)$.

Step 2: Update the minimum spanning tree by adding some edge b.
If b does not exist, then add f to $B(T_p)$, and stop.

Step 3: If b is not a back-arc of $T_c = T_p \setminus e \cup f$, then remove b and add f to $B(T_p)$. Stop.
Step 4: Output b. Remove b from $B(T_p)$. Go to **Step 2**

Lemma 3 *The algorithm* ENUM_NON-BACK-ARCS *outputs all back-arcs of T_p which are valid non-back-arcs of T_c in the order of their indices. It takes $O(D(|V|, |A|))$ time and $O(D(|V|, |A|))$ time for one output. Its space complexity is $O(|A| + DS(|V|, |A|))$.*

Proof : Since the algorithm finds the maximum index one among those back-arcs, back-arcs are outputted in the order of their indices. Therefore merging operation is done in linear time of the number of non-back-arcs. To identify both the endpoints of the path P, we have to find the nearest common ancestor of the head of e and the head of f. It can be found in $O(\log |V|)$ time by D. D. Slater and R. E. Tarjan's dynamic tree data structure [6]. The updating operation, which is adding an edge, deleting an edge and changing the root, of it is also done in $O(\log |V|)$ time. The time complexity of the rest of the algorithm is $O(\mathrm{D}(|V|, |A|))$ for one iteration. Because of the number of iterations, the time complexity is $O(\mathrm{D}(|V|, |A|))$ per one output. ∎

By using this algorithm, we obtain the following theorem.

Theorem 3.1 *Algorithm* ENUM_DIRECTED_SPANNING_TREES *enumerates all directed spanning trees in $O(|A| + ND(|V|, |A|))$ time. It uses $O(|A| + DS(|V|, |A|))$ space.*

Proof : By above lemmas, the time complexity of algorithm is $O(|A|)$ for preprocessing, $O(\log |V| + \mathrm{D}(|V|, |A|))$ time for one directed spanning tree and $O(\mathrm{D}(|V|, |A|))$ time for one new non-back-arc. Thus the time complexity satisfies the condition. The algorithm changes the data structure for updating the minimum spanning tree if a recursive call occurs. When the recursive call ends, we have to restore it. The restoring operation is done by adding the deleted edge, deleting the added edge and changing the modified weight. The total amount of these changes may increase by recursive calls, but the total size of each of the added edges, deleted edges and weight changed edges does not exceed $|A|$. Therefore they are stored in $O(|A|)$ space.

The set of valid non-back-arcs and back-arcs are treated similarly. After a recursive call, we can restore them by adding and removing the removed or added arcs. The total space for storing them is $O(|A|)$. Hence the space complexity is $O(|A| + \mathrm{DS}(|V|, |A|))$. ∎

Acknowledgment

We greatly thank to Associate Professor Akihisa Tamura of University of Electro-Communications for his kindly advise. We also owe a special debt of gratitude of Research Assistant Yoshiko T. Ikebe of Science University of Tokyo.

References

[1] H. N. Gabow, E. W. Myers, "Finding All Spanning Trees of Directed and Undirected Graphs," SIAM J. Comp., 7, 280-287, 1978.

[2] H. N. Kapoor and H. Ramesh, "Algorithms for Generating All Spanning Trees of Undirected, Directed and Weighted Graphs," Lecture Notes in Computer Science, Springer-Verlag, 461-472, 1992.

[3] A. Shioura, A. Tamura and T. Uno, "An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs, " SIAM J. Comp., to be appeared.

[4] D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig, "Sparsification - A Technique for Speeding up Dynamic Graph Algorithms, " FOCS 33, 60-69, 1992.

[5] G. N. Fredrickson, "Data Structure for On-line Updating of Minimum Spanning Trees, with Applications, " SIAM J. Comp., 14, No 4, 781-798, 1985.

[6] D. D. Sleator and R. E. Tarjan, "A Data Structure for Dynamic Trees," J. Comp. Sys. Sci. 26, 362-391, 1983.

[7] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithm," SIAM J. Comp. 1, 146-169, 1972.