Graph Decomposition and Recursive Closures

Yangjun Chen*

Dept. Business Computing, Winnipeg University,
515 Portage Ave. Winnipeg, Manitoba, Canada R3B 2E9
ychen2@uwinnipeg.ca

Abstract. In this paper, we propose a new algorithm for computing recursive closures. The main idea behind this is tree labeling and graph decomposition, based on which the transitive closure of a directed graph can be computed in O($e \cdot b$) time and O($n \cdot b$) space, where n is the number of the nodes of the graph, e is the numbers of the edges, and b is the graph's breadth. It is a better computational complexity than any existing algorithms for this problem.

1. Introduction

Let $G = (V, E)$ be a directed graph (*digraph* for short). Digraph $G^* = (V, E^*)$ is the reflexive, transitive closure of G if $(v, w) \in E^*$ iff there is a path from v to w in G. In this paper, we present a new algorithm for computing the transitive closure of a digraph efficiently.

2. Tree labeling

In this section, we mainly discuss the concepts of tree labeling and graph decomposition, based on which our algorithm is designed. For any directed tree T, we can label it as follows. By traversing T in *preorder*, each node v will obtain a number $pre(v)$ to record the order in which the nodes of the tree are visited. In a similar way, by traversing T in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to characterize the ancestor-descendant relationships of nodes as follows.

Proposition 1. Let v and v' be two nodes of a tree T. Then, v' is a descendant of v iff $pre(v') > pre(v)$ and $post(v') < post(v)$.
Proof. See [Kn73]. □
If v' is a descendant of v, then we know that $pre(v') > pre(v)$ according to the preorder search. Now we assume that $post(v') > post(v)$. Then, according to the postorder search, either v' is in some subtree on the right side of v, or v is in the subtree rooted at v', which contradicts the fact that v' is a descendant of v. Therefore, $post(v')$ must be less than $post(v)$.

The following example helps for illustration.

Example 1. See the pairs associated with the nodes of the directed tree shown in Fig. 1. The first element of each pair is the preorder number of the corresponding node and the second is its postorder number. Using such labels, the ancestor-descendant relationships of nodes can be easily checked. For instance, by checking the label associated with b against the label for f, we know that b is an ancestor of f in terms of Proposition 1. We can also see that since the pairs associated with g and c do not satisfy the condition given in Proposition 1, g must not be an ancestor of c and *vice versa*.
Let (p, q) and (p', q') be two pairs associated with nodes u and v. We say that (p, q) is subsumed by (p', q'), denoted $(p, q) \prec (p', q')$, if $p > p'$ and $q < q'$. Then, u is a descendant of v if (p, q) is subsumed by (p', q').

3. Branchings and graph decomposition

Now we discuss how to recognize the ancestor-descendant relationships w.r.t. a general structure: a DAG or a graph containing cycles. First, we address the problem of DAGs in 3.1. Then, cyclic graphs will be discussed in 3.2.

* The author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Can-

3.1 Recursion w.r.t. DAGs

What we want is to apply the technique discussed above to a DAG. To this end, we establish a *branching* of the DAG as follows.

Definition 2. (*branching* [Ta77]) A subgraph $B = (V, E')$ of a digraph $G = (V, E)$ is called a branching if it is cycle-free and $d_{indegree}(v) \leq 1$ for every $v \in V$.

Clearly, if for only one node r, $d_{indegree}(r) = 0$, and for all the rest of the nodes, v, $d_{indegree}(v) = 1$, then the branching is a directed tree with root r. Normally, a branching is a set of directed trees. Now, we assign each edge e a same cost (e.g., let cost $c(e) = 1$ for every edge). We will find a branching for which the sum of the edge costs, $\sum_{e \in E'} c(e)$, is maximum.

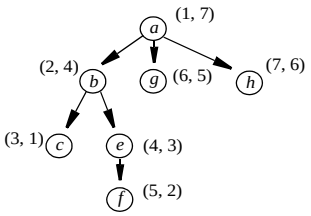

Fig. 1. Labeling a tree

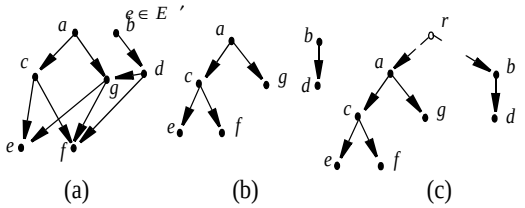

Fig. 2. A DAG and its branching

For example, the trees shown in Fig. 2(b) are a maximal branching of the graph shown in Fig. 2(a) if each edge has a same cost.

Assume that the maximal branching for $G = (V, E)$ is a set of trees T_i with root r_i ($i = 1, ..., m$). We introduce a *virtual root r* for the branching and an edge $r \rightarrow r_i$ for each T_i, obtaining a tree G_r, called the representation of G. For instance, the tree shown in Fig. 2(c) is the representation of the graph shown in Fig. 2(a). Using Tarjan's algorithm for finding optimum branchings [Ta77], we can always find a maximal branching for a directed graph in $O(|E|)$ time if the cost for every edge is equal to each other. Therefore, the representative tree for a DAG can be constructed in linear time.

By traversing G_r in *preorder*, each node v will obtain a number $pre(v)$; and by traversing G_r in *postorder*, each node v will get another number $post(v)$. These two numbers can be used to recognize the ancestor-descendant relationships of all G_r's nodes as discussed in Section 2.

In a G_r (for some G), a node v can be considered as a representation of the subtree rooted at v, denoted $T_{sub}(v)$; and the pair (pre, $post$) associated with v can be considered as a pointer to v, and thus to $T_{sub}(v)$. (In practice, we can associate a pointer with such a pair to point to the corresponding node in G_r.) In the following, what we want is to construct a pair sequence: (pre_1, $post_1$), ..., (pre_k, $post_k$) for each node v in G, representing the union of the subtrees (in G_r) rooted respectiivey at (pre_j, $post_j$) ($j = 1, ..., k$), which contains all the descendants of v. In this way, the space overhead for storing the descendants of a node is dramatically reduced. Later we will shown that a pair sequence contains at most $O(b)$ pairs, where b is the breadth of G.

Example 2. The representative tree G_r of the DAG G shown in Fig. 2(a) can be labeled as shown in Fig. 3(a). Then, each of the generated pairs can be considered as a representation of some subtree in G_r. For instance, pair (3, 3) represents the subtree rooted at c in Fig. 3(a).

If we can construct, for each node v, a pair sequence as shown in Fig. 3(b), where it is stored as a link list, the descendants of the nodes can be represented in an economical way. Let $L =$ (pre_1, $post_1$), ..., (pre_k, $post_k$) be a pair sequence and each (pre_i, $post_i$) is a pair labeing v_i ($i = 1, ..., k$). Then, L corresponds to the union of the subtrees $T_{sub}(v_1)$, ..., and $T_{sub}(v_k)$. For example, the pair sequence (4, 1)(5, 2)(6, 4)(8, 6) associated with d in Fig. 3(b) represents a union of 4 subtrees: $T_{sub}(e)$, $T_{sub}(f)$, $T_{sub}(g)$ and $T_{sub}(d)$, which contains all the descendants of d in G. □

The question is how to construct such a pair sequence for each node v so that it corresponds to a union of subtrees in G_r, which contains all the descendants of v in G.

First, we notice that by labeling G_r, each node in $G = (V, E)$ will be initially associated with a pair

as illustrated in Fig. 4. That is, if a node v is labeled with (pre, $post$) in G_r, it will be initially labeled with the same pair (pre, $post$) in G.

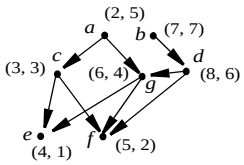

Fig. 4. Graph labeling

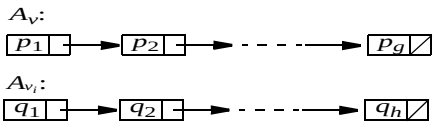

Fig. 5. Link lists associated with nodes in G

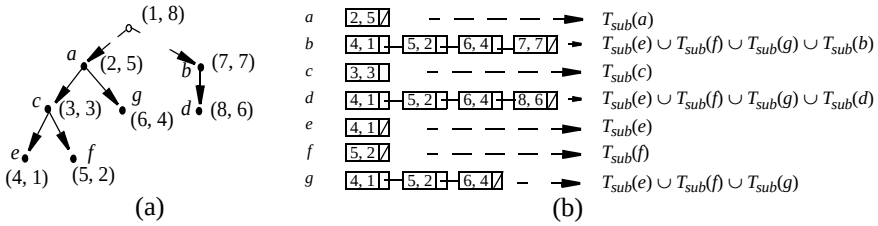

Fig. 3. Tree labeling and illustration for transitive closure representation

To compute the pair sequence for each node, we sort the nodes of G topologically, i.e., $(v_i, v_j) \in E$ implies that v_j appears before v_i in the sequence of the nodes. The pairs to be generated for a node v are simply stored in a link list A_v. Initially, each A_v contains only one pair produced by labeling G_r.

We scan the topological sequence of the nodes from the beginning to the end and at each step we do the following:

> Let v be the node being considered. Let v_1, ..., v_k be the children of v. Merge A_v with each A_{v_i} for the child node v_i (i = 1, ..., k) as follows. Assume $A_v = p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_g$ and $A_{v_i} = q_1 \rightarrow q_2 \rightarrow ... \rightarrow q_h$, as shown in Fig. 5. Assume that both A_v are A_{v_i} increasingly ordered. (We say a pair p is larger than another pair p', denoted $p > p$' if $p.pre > p'.pre$ and $p.post > p'.post$.)

We step through both A_v and A_{v_i} from left to right. Let p_i and q_j be the pairs encountered. We'll make the following checkings.

(1) If $p_i.pre > q_j.pre$ and $p_i.post > q_j.post$, insert q_j into A_v after p_{i-1} and before p_i and move to q_{j+1}.
(2) If $p_i.pre > q_j.pre$ and $p_i.post < q_j.post$, remove p_i from A_v and move to p_{i+1}. (*p_i is subsumed by q_j.*)
(3) If $p_i.pre < q_j.pre$ and $p_i.post > q_j.post$, ignore q_j and move to q_{j+1}. (*q_j is subsumed by p_i; but it should not be removed from A_{v_i}.*)
(4) If $p_i.pre < q_j.pre$ and $p_i.post < q_j.post$, ignore p_i and move to p_{i+1}.
(5) If $p_i = p_j$' and $q_i = q_j$', ignore both (p_i, q_i) and (p_j', q_j'), and move to (p_{i+1}, q_{i+1}) and (p_{j+1}', q_{j+1}'), respectively.

In terms of the above discussion, we have the following algorithm to merge two pair sequences together.

Algorithm *pair-sequence-merge*(A_1, A_2)

Input: A_1 and A_2 - two link lists associated with v_1 and v_2.

Output: A - modified A_1, containing all the pairs in A_1 and A_2 with all the subsumed pairs removed.

begin

1 $p \leftarrow$ *first-element*(A_1);

```
2    q ← first-element(A2);
3    while p ≠ nil do{
4      while q ≠ nil do{
5       if (p.pre > q.pre ∧ p.post > q.post) then
6          {insert q into A1 before p;
7            q ← next(q);} (*next(q) represents the pair next to q in A2.*)
8       else if (p.pre > q.pre ∧ p.post < q.post) then
9            {p' ← p; (*p is subsumed by q; remove p from A1.*)
10            remove p from A1;
11            p ← next(p');} (*next(p') represents the pair next to p' in
                                  A1.*)
12          else if (p.pre < q.pre ∧ p.post > q.post) then
13                  {q ← next(q);} (*q is subsumed by p; move to the
                                       next element of q.*)
14                else if (p.pre < q.pre ∧ p.post < q.post) then
15                       {p ← next(p);}
16                     else if (p.pre = q.pre ∧ p.post = q.post)
17                          then {p ← next(p); q ← next(q);}
18   if p = nil ∧ q ≠nil then {attach the rest of A2 to the end of A1;}
end
```

The following example helps for illustration.

Example 3. Assume that A_1 = (7, 7)(11, 8) and A_2 = (4, 3)(8, 5)(10, 11). Then, A = *pair-sequence-merge*(A_1, A_2) = (4, 3)(7, 7)(10, 11). Fig. 6 shows the entire merging process.

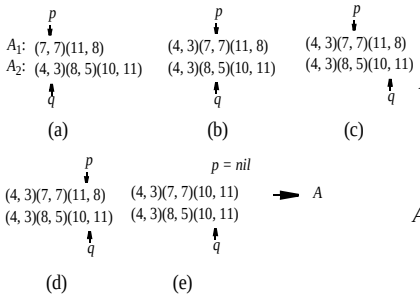

Fig. 6. An entire merging process

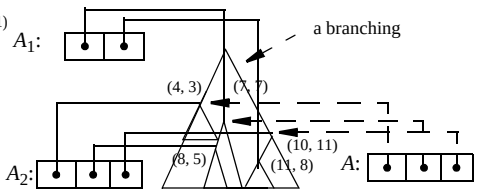

Fig. 7. Illustration of mering two pair sequencees

In each step, the A_1-pair pointed by p and the A_2-pair pointed by q are coompared. In the first step, (7, 7) in A_1 will be checked against (4, 3) in A_2 (see Fig. 6(a)). Since (4, 3) is smaller than (7, 7), it will be inserted into A_1 before (7, 7) (see Fig. 6(b)). In the second step, (7, 7) in A_1 will be checked against (8, 5) in A_2. Since (8, 5) is subsumed by (7, 7), we move to (10, 11) in A_2 (see Fig. 6(c)). In the third step, (7, 7) is smaller than (10, 11) and we move to (11, 8) in A_1 (see Fig. 6(d)). In the fourth step, (11, 8) in A_1 is checked against (10, 11) in A_2. Since (11, 8) is subsumed by (10, 11), it will be removed from A_1 and p becomes *nil* (see Fig. 6(e)). In this case, (10, 11) will be attached to A_1 (see line 18 of Algorithm *pair-sequence-merge*( )), forming the result A = (4, 3)(7, 7)(10, 11) (see Fig. 6(e)). Fig. 7 is a pictorial illustration of the result of merging A_1 and A_2.

Along the topological order of a graph, we can generate the pair sequences for all the nodes, which computes the transitive closure of the graph using $O(e \cdot b)$ time.

References

Kn73 D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley Pub. London, 1973.

Ta77 J. Tarjan, Finding Optimum Branching, *Networks*, 7. 1977, pp. 25 -35.