A FAST ALGORITHM TO DETERMINE MINIMALITY OF STRONGLY CONNECTED DIGRAPHS

by

JIANPING ZHU

(Under the Direction of Robert W. Robinson)

ABSTRACT

In this thesis, we consider the following problem: Given a strongly connected digraph $G = (V, E)$, where V is the set of vertices and E is the set of edges , "is it a minimal strong connected digraph?". A reducible edges e is one for which $G-e$ is strongly connected. A minimal strongly connected digraph is one with no reducible edges. Our approach is to apply depth first search on G to generate a depth first search tree and the sets of back, forward, and cross edges. Then we determine if there are any reducible non-tree edges. If not, we then check if there are any reducible tree edges based on an algorithm for finding immediate dominators. We have implemented the algorithm and report experimental results that show the algorithm can handle large digraphs quickly.

INDEX WORDS: Digraph, Minimal strong digraph, Strongly connected, Nearest common ancestor, Immediate dominators

A FAST ALGORITHM TO DETERMINE MINIMALITY OF STRONGLY CONNECTED DIGRAPHS

by

JIANPING ZHU

B.S., Nanjing Forestry University, China, 1986

M.S., Nanjing Forestry University, China, 1993

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2002

© 2002

Jianping Zhu

All Rights Reserved

A FAST ALGORITHM TO DETERMINE
MINIMALITY OF STRONGLY CONNECTED DIGRAPHS

by

JIANPING ZHU

Major Professor: Robert W. Robinson

Committee: E. Rodney Canfield
Thiab Taha

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2002

ACKNOWLEDGEMENTS

First of all, I would like to thank Dr. Robert W. Robinson for his guidance and support through out my thesis. Over the past year, he invested lots of time helping me with the algorihms, helping me debugging my program. Without his guidance, I could not have done this thesis.

I would also like to thank Dr. Canfield and Dr. Taha for severing as committee members and giving me valuable suggestions.

This thesis is dedicated to my families.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENT iv

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1 INTRODUCTION, NOTATION AND BASIC CONCEPTS 1

Introduction 1

Notation and basic concepts 2

2 DETECTING REDUCIBILITY OF NON-TREE EDGES 10

3 DETECTING REDUCIBILITY OF TREE EDGES 21

4 FINDING THE NEAREST COMMON ANCESTOR 25

Special case: a complete binary tree 26

A compressed tree 27

The balanced binary tree 28

Algorithm to compute nearest common ancestor 30

5 IMPLEMENTATION AND EXPERIMENT 32

REFERENCES 40

LIST OF TABLES

Page

Table 1: Running time of MSD algorithm 34

Table 2: Running time of NCA algorithm 36

Table 3: Running time in Sec of the immediate dominators algorithm 38

LIST OF FIGURES

Page

Figure 1: A strong digraph 4

Figure 2: A spanning in-tree 5

Figure 3: A Minimal Strong Digraph 5

Figure 4: Edge classification 8

Figure 5: A reducible forward edge 10

Figure 6: A reducible cross edge 11

Figure 7: Digraphs G , G_1 and G_2 12

Figure 8: Digraph G_1 13

Figure 9: Reducible back edges 15

Figure 10: Digraphs G and G_1 16

Figure 11: Digraphs G_2 and G_3 17

Figure 12: T^S and T in G 22

Figure 13: Symmetric-order numbering of a complete binary tree 25

Figure 14: An original tree 27

Figure 15: The compressed tree 28

Figure 16: Balanced binary tree corresponding to tree in Figure 14 30

Figure 17: Running time of MSD algorithm 35

Figure 18: Running time of NCA algorithm 37

Figure 19: Running time of the finding immediate dominators algorithm 39

CHAPTER 1

INTRODUCTION, NOTATION AND BASIC CONCEPTS

Introduction

One of the classic problems in computer science is to compute the connectivity and reachability of a digraph $G = (V, E)$, where V is the set of vertices and E is the set of edges. In this connection, finding a minimal storage representation of a digraph is a very important issue. The most straightforward method to solve this problem is to remove edges one by one and to test all vertices pair by pair to see if they can still reach each other after removing the edge. This method takes time $O(n^2m)$ in the worst case, which is inefficient. Here $n = |V|$ and $m = |E|$. Klause Simon developed an algorithm to find a minimal transitive reduction in a strongly connected digraph [1]. He claimed his algorithm to be a $O(m+n)$ time and space. But his algorithm relies on Harel's algorithm [2] for finding dominators in a flowgraph which Harel claimed to be linear. However the soundness of Harel's approach has been questioned for lack of details [3]. Therefore it could be very hard to implement Harel's algorithm , or perhaps impossible, in which case Klause Simon's algorithm would not be linear.

In this thesis, an $O(n \log n)$ time algorithm to find whether a digraph is a Minimal Strong Digraph (MSD) is developed and a successful implementation is described. This algorithm can be used to facilitate the process of generating MSDs. An $O(n^2)$ time algorithm is utilized by Kiran Bhogadi [4] to check whether a candidate digraph is an MSD and we think the performance of his algorithm can potentially be improved for large digraphs by utilizing our algorithm.

An edge *e* of a strong digraph *G* is said to be *reducible* if *G–e* is strongly connected. Clearly a strong digraph is an MSD if and only if it has no reducible edges.

This thesis is divided into several chapters. Following section contains basic concepts and notation, and a brief description of Depth First Search (DFS) on digraphs. In Chapter 2, an algorithm is developed to find if there is any reducible non-tree edge in a given strongly connected digraph. In Chapter 3, the result of Chapter 2 is built on to provide an algorithm to detect reducible tree edges. In Chapter 4, we will briefly introduce an $O(n \log n)$ time algorithm for finding nearest common ancestors in trees. Some experimental results of implementation of algorithm of determining minimality of strongly connected digraphs and some other related algorithms will be presented in Chapter 5. The algorithm for finding immediate dominator has been implemented successfully by T. Lengauer and R. E Tarjan. We will not describe this algorithm in this thesis.

Notation and basic concepts

A *directed graph* (or *digraph*) *G* is a pair (V, E), where *V* is a non-empty finite set and *E* is an antireflective binary relation on *V*, $V = \{1, 2, \ldots n\}$. The set *V* is called the *vertex set* of *G*, and its elements are called *vertices*. The set of *E* is called the *edge set* of *G*, and its elements are called edges $(E \subseteq V \times V)$. An edge joining vertex *u* to vertex *v* will be represented as (u, v), where *u* is said to be adjacent to *v*, and *v* is adjacent from u. A *path P* in *G* from vertex v_0 to vertex v_s is a sequence of vertices $v_0, v_1, \ldots, v_s$ such that $(v_{i-1}, v_i) \in E$ for $i \in (1, 2, \ldots, s)$. The path consists of the vertices $v_0, v_1, \ldots, v_s$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{s-1}, v_s)$. The *length* is the number of edges in the path. If a path joins vertex *u* to vertex *v*, *v* is *reachable* from *u*, hence the term *reachability*. If there is a path

P from v to u, we say that u is reachable from v via P which we denote by $v \text{–}(P)\rightarrow u$. The fact that v is reachable from u is denoted by $u \text{--}\rightarrow v$. A path is simple if the vertices in the path are distinct. A single edge (v, u) is denoted by $v\rightarrow u$.

A digraph is *strongly connected* if every two vertices are reachable from each other, i.e. $v\text{--}\rightarrow w\text{--}\rightarrow v$ for all $v, w \in V$ (See Figure 1). A graph $G'=(V', G')$ is a *subgraph* of $G = (V, E)$ if $V' \in V$ and $E' \in E$. *A reverse graph* G^R of G is formed by reversing every edge in G.

A *tree* or *out-tree* is an acyclic digraph with one distinguished vertex called the *root* r, such that $r\text{--}\rightarrow v$ for all vertices v, and no edges enters r. A tree vertex with no edges leaving it is a *leaf*. If (v, w) is a tree edge, v is the father of w and w is a son of v. If v is reachable from u via tree edges, then u is an *ancestor* of v and v is a *descendant* of u. A tree vertex z is called a *common ancestor* of tree vertex u and tree vertex w if and only if z is an ancestor for both w and u. The vertex z is called the *nearest common ancestor* of w and u if and only if there is no other common ancestor x of w and u which is a son of z. If a tree contains all vertices of G, then T is called a *spanning tree* of G. An *in-tree* or *sink-tree* S is the reverse of an out-tree. The father and son relations in an out-tree is the same as in the reversed tree. A spanning in-tree is an in-tree which contains all vertices of G. Figure 2 is a spanning in-tree corresponding to Figure 1.

Suppose G is a strongly connected digraph with vertex s specified as tree root. If vertices x and y are distinct and x lies on every path form s to y then x is called a *dominator* of y. This is equivalent to say x dominate y if and only if y is not reachable for s in $G-\{x\}$. The vertex x is an *immediate dominator* of y if x is a dominator of y and x is dominated by every other dominator z of y. It is easy to see that each vertex other than

the root has a unique immediate dominator. Furthermore, we use the concept of dominator for an edge e in the same way as for a vertex. A particular edge e is a dominating edge of y if and only if s can't reach the vertex y in $G - e$. An edge or vertex is called *reverse dominating* if it is dominating in the reverse graph G^R.

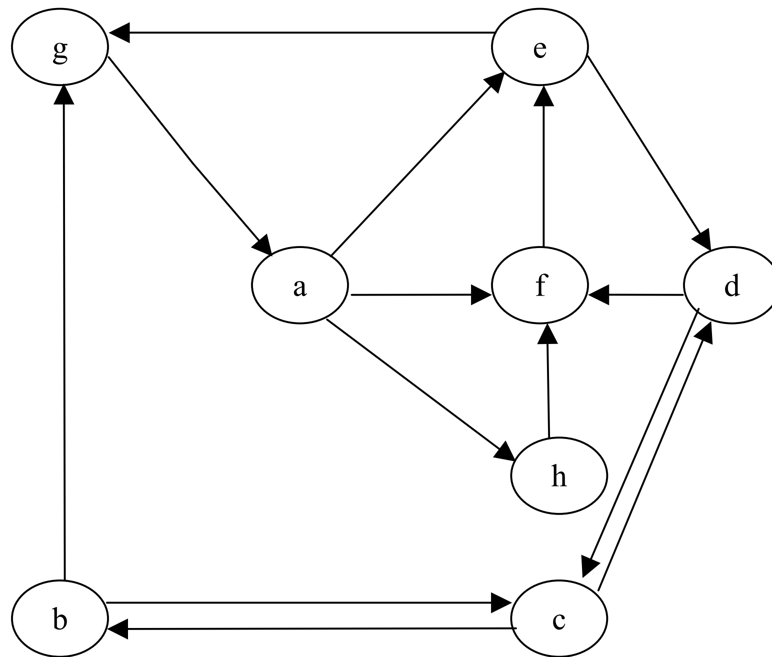


Figure 1. A strong digraph

A strong (or strongly connected) digraph is one in which every vertex is reachable from each other vertex, in this article we always suppose the input digraph G is a strong digraph.

A *Minimal Strong Digraph* (MSD) is a strong digraph which is no longer strong if any of its edges is removed (see Figure 3). An edge e in a strong digraph G is *reducible* if $G - e$ is strongly connected.

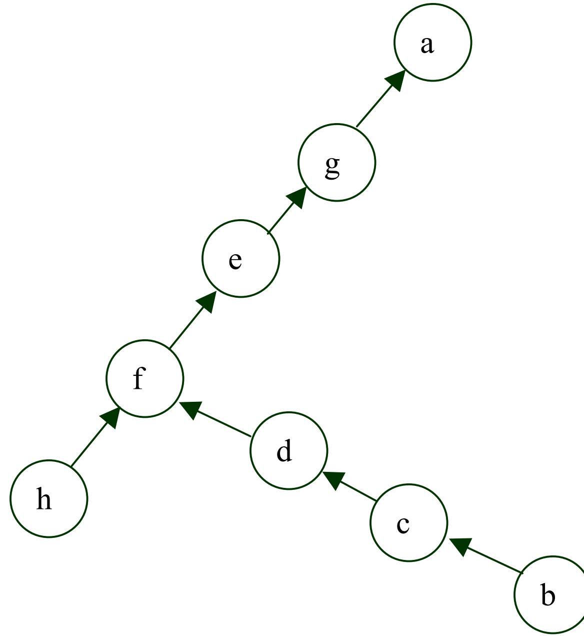


Figure 2. A spanning in-tree

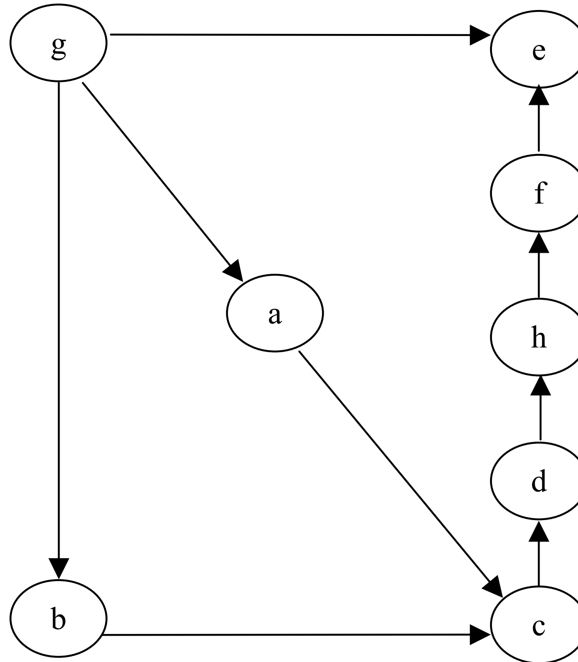


Figure 3. A Minimal Strong Digraph

Our algorithm is based on a systematic exploration of a digraph. In particular we rely on Depth First Search (DFS) as the basis of our algorithm. Therefore we need a short description of the algorithm of DFS. The strategy of depth first search is to search "deeper" in the graph when possible. In depth first search, edges are explored out of the most recently discovered vertex v that still have undiscovered edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process will continue until we have discovered all vertices that are reachable from the original source vertex.

DFS will create a depth first search tree (DFS-tree) and at the same time DFS also timestamps each vertex. Each vertex has two time stamps: The time stamp d[v] is related to the time when v is first discovered, and timestamp f[v] accounts for the time when search finishing examining v's adjacent list. The set REACH contains the explored vertices. For a given input strong digraph G, The DFS algorithm is displayed as following.

DFS (v, V, E)

REACH $\leftarrow \{v\}$

discover-time $\leftarrow$ 1

finish-time $\leftarrow$ 1

d[v] $\leftarrow$ 1

for $\forall$ w with $(v, w) \in E$

do if $w \notin$ REACH

then DFS-VISIT(w)

endif

```
        endfor

    DFS-VISIST(w)
            d[w] ← discover-time ++
            for each u ∈ adj[w]
                    do if u ∉ REACH
                    Then DFS-VISIT(u)
                     endif
            endfor
            f[w] ← finish-time++
```

Based on d[v] and f[v], we partition the edges into four classes: the tree edges TREE, the forward edges FORWARD, the cross edges CROSS and the back edges BACK (See Figure 4) . The classes are as follows:

(v, w) in TREE: d(v) < d(w)

and f(w)<f(v)

(v, w) is in the depth first search tree

(v, w) in FORWARD: d(v) < d(w)

and f(w) < f(v)

and *(v, w)* ∉ TREE

(v, w) in CROSS: d(w) < d(v)

and $f(v) < f(w)$

(v, w) in BACK: $d(w) < d(v)$

and $f(w) < f(v)$

Notice that $d(v) < d(w)$ and $f(v) < f(w)$ is impossible, and this is an elementary fact of DFS.

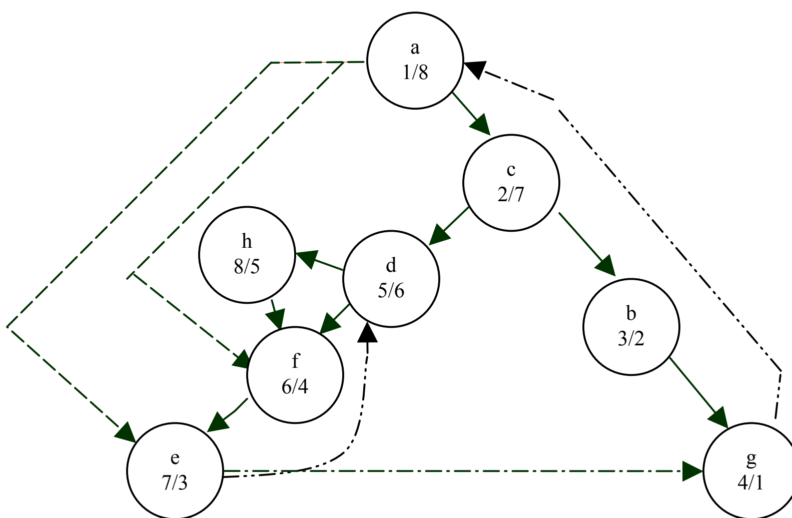


Figure 4: Edge classification

– – – – – –: FORWARD ———— : TREE

–·–·–·–·–: CROSS –··–··–··– : Back

··················: one or more tree edges

Note: This style applies to following Figures.

Figure 4 shows a DFS-tree and the corresponding edge classification of digraph in Figure 1. Now we define a new notation: if there is a path from v to u via tree edges, we denote this path by $v \rightarrow \text{(TREE)} \rightarrow u$.

CHAPTER 2

DETECTING REDUCIBILITY OF NON-TREE EDGES

In order to find whether there is a reducible edge in a given strong digraph $G = (V, E)$, a DFS is performed and E is partitioned into sets TREE, FORWARD, CROSS, and BACK. For convenience, we use the d[v] to identify v. Thus the root is always 1, and n is the last vertex to be discovered among the n vertices of G.

Theorem 1. Each forward edge $e = (v, w)$ is reducible

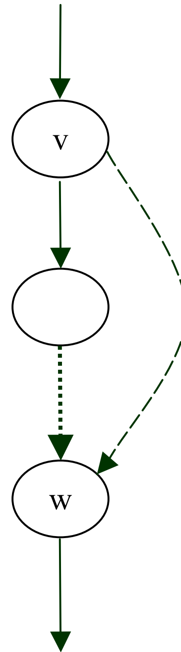


Figure 5. A reducible forward edge

Proof: By definition for forward edge, there must be a tree edge path form v to w with length equal or greater than 2 , therefore edge (v, w) is reducible (see Figure 5).

The following statement is the precondition for Theorems 2 and 3.

***e = (v, z)* is a cross edge and *w* is the nearest common ancestor of vertices *v* and *z* in the DFS-tree of *G*.**

Theorem 2: If edge $e_1 = (v, w)$ is in E then *e* is reducible.

Proof: There is a reducing path v→w −(TREE)→z of e. (see Figure 6).

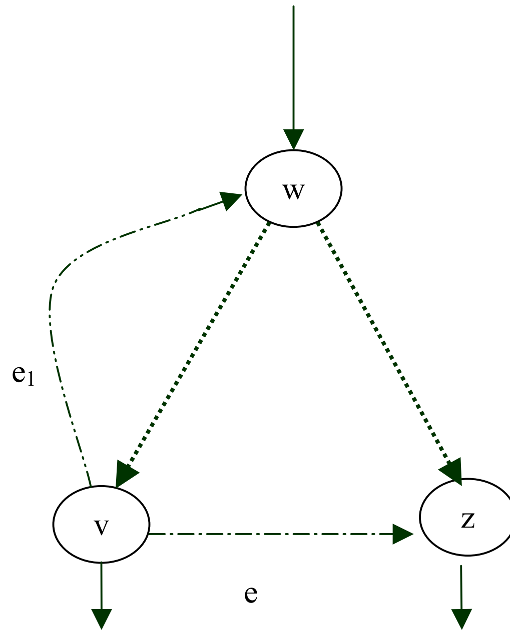


Figure 6. A reducible cross edge

Theorem 3: If edge *(v, w)* is not in *E*, let $G_1 = G - (v,z) + (v,w)$. Then:

1. The edge *e* is reducible in *G* if and only if the edge e_1 is reducible in the digraph G_1.

2. Let e_2 be an edge of *G* other than *e* which is not a tree edge. Then e_2 is reducible in *G* if and only if e_2 is reducible in G_1.

Proof of 1: From Theorem 2, we can conclude that G_1 is also a strong digraph. Now we need to form a new graph G_2 by deleting edge *e* from *G* (see Figure 7).

Let us first prove that if e is reducible in G, e_1 is reducible in G_1.

Because e is reducible in G and G_2 is formed by deleting e, we see that G_2 is still strongly connected, and so must contain a path P_1 from v to w. Now G_1 is G_2+e_1, so e_1 is reducible in G_1.

We can similarly prove that if e_1 is reducible in G_1, then e is reducible in G.

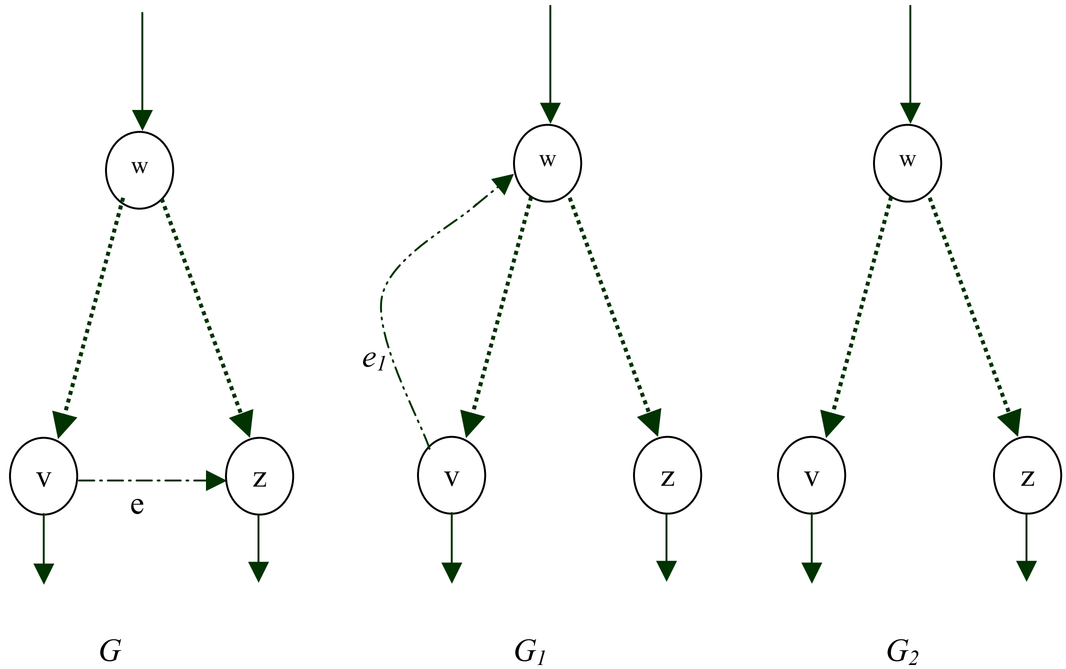


Figure 7. Digraphs G , G_1 and G_2

Proof of 2: Let us first prove that if e_2 is reducible in G, e_2 is reducible in G_1

Let P_1 be a reducing path for e_2 in G. The trivial case is given if e is not an edge of P_1, obviously P_1 is also in G_1 so e_2 is reducible in G_1. In the nontrivial case, the reducing path P_1 contain e, we see that G_1 has a path P_2 containing $v \to w$ — (TREE) $\to$ z. P_2 connects the same endpoints as P_1. This accounts for that there is also a reducing path P_2 in for e_2 in G_1.

Next we need to prove that if e_2 is reducible in G_1, it is also reducible in G.

Also let P_2 be a reducing path of e_2 in G_1. The trivial case is that e_1 is not an edge of P_2. Obviously in this case e_2 is also reducible in G. If e_1 is an edge of P_2, let u_1 be the first vertex after the starting edge w in the tree edges $w \rightarrow$ (TREE) $\rightarrow z$, u_2 is the first vertex after starting vertex w in the tree edges path $w \rightarrow$ (TREE) $\rightarrow v$ (see Figure 8). Here $v > u_2 > z > u_1 > w$. Because e_2 is reducible in G_1, $G_1 - e_2$ is still strongly connected. So there must be a path from z to w in $G_1 – e_2$. Now we want to prove that e_1 is not an edge of the path from z to w.

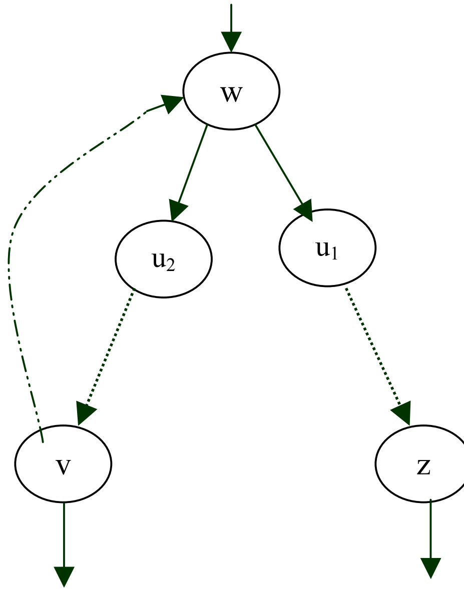


Figure 8. Digraph G_1

Assumption: (v, w) is an edge of the path from z to w.

Under this assumption, it is evident that there must exist a path which starts at u_1 and ends at u_2 via edge (v, w). So in order for this path to exist, there must exist such a path that starts from a vertex v_1 in subtree rooted at u_1 ends at a vertex v_2 which is in subtree rooted at u_2. But such an edge is impossible because according to the definition

of different class of edges, it is not a forward edge or tree edge because (f[v_1] < f[v_2]) and it is not a cross edges or back edge because (d[v_2] > d[v_1]). Therefore the assumption is not right and we can have the conclusion that (v, w) is not an edge of P_2. In this situation v can reach w in G via P_2. Therefore e_2 is also reducible in G.

Theorem 3 allows us to use the following strategy to find reducible edge. First we check if there are any forward edges. Then we check if there are any cross edges which are reducible by Theorem 2. Then we can use Theorem 3 to replace each cross edge by back edge.

Theorem 4. Let {(v, w_1) … (v, w_S)} be the set of back edges emanating from vertex v in such a way that $w_1 < w_2 < \ldots < w_S$ then all edges (v, w_2) … (v, w_S) are reducible.

Proof: This situation is illustrated in Figure 9. We can easily see that edge (v, w_2) is reducible because there is a path $v \rightarrow w_1 \rightarrow w_2$. Edges (v, w_3), … (v, w_s) are similarly seen to be reducible.

Theorem 5. Let $e = (v, w)$ be a back edge in G, x be a descendant of v with $v \neq x$ and z be a vertex such that there is an edge from x to z (illustrated by G in Figure 10).

1. If $e_1 = (z, w)$ is an edge in G_1, then e is reducible (illustrated by G_1 in Figure 10).

2. If e_1 is not an edges in G, then e is reducible in G if and only if e_1 is reducible in the digraph $G_2 = G - e + e_1$.

3. Let $e_2 = (a, b)$ be an edge in G with $e \neq e_2 \neq e_1$ and e_2 is a not a tree edge. Then e_2 is reducible in G if and only if e_2 is reducible in G_2.

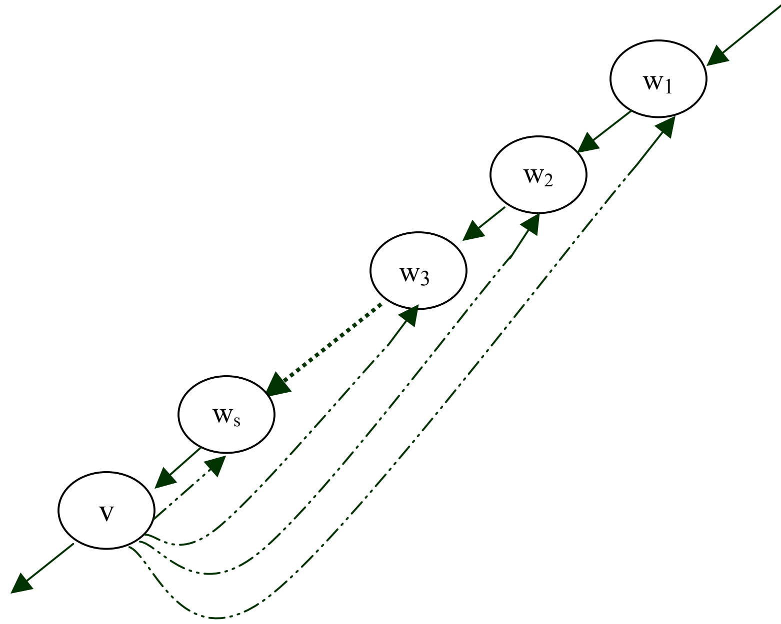


Figure 9. Reducible back edges

Proof of 1: Because there is a reducing path $v \rightarrow$ (TREE) $\rightarrow x \rightarrow z \rightarrow w$ of *(v, w)* in *G*, so *(v, w)* is reducible.

From 1, we can see that if e_1 is not an edge in *G*, then $G_2 = G - e + e_1$ is strongly connect.

Proof of 2: Let us first prove that if *e* is reducible in *G*, e_1 is reducible in G_2.

Because *e* is reducible in *G*, after removing *e* from *G* we get G_3 (see Figure 10, G_3) which is still strongly connected. Therefore there must exist a path from z to w in G_3. We notice that G_2 has one more edge e_1 than G_3. Thus e_1 is reducible in G_2.

We can similarly prove that if e_1 is reducible in G_2, then *e* is reducible in *G.*

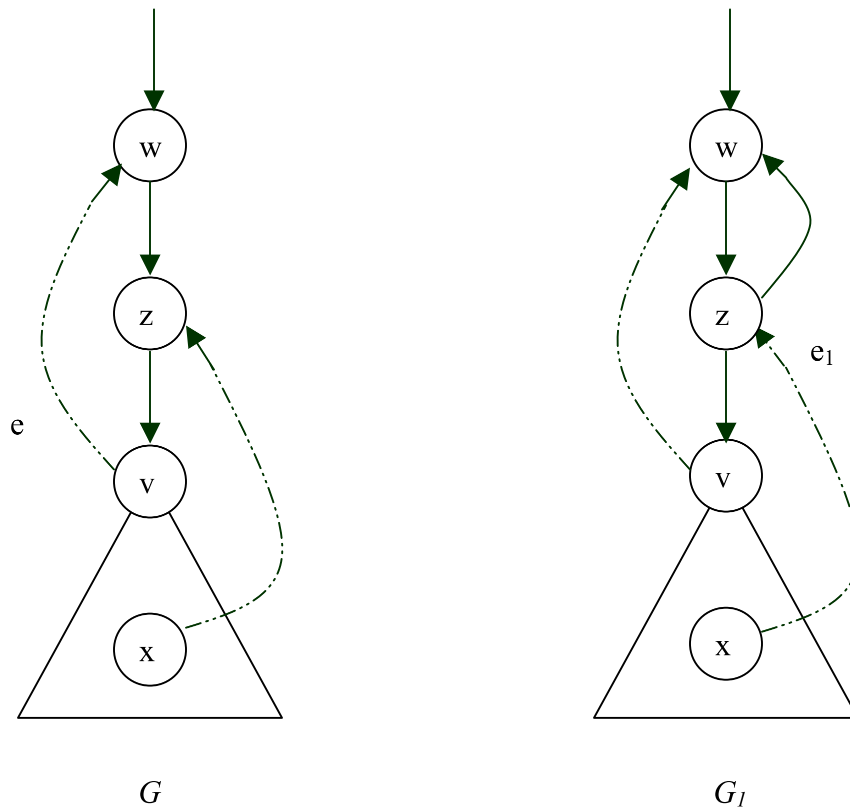


Figure 10. Digraphs G and G_1

Proof of 3: Let us prove if e_2 is reducible in G, it is reducible in G_2.

Let P_1 be a reducing path of e_2 in G. The trivial case is that P_1 does not contain e. It is evident that e_2 is also reducible in G_2. In the nontrivial case, P_1 contains e. G_2 has no has a path $v \rightarrow$ (TREE) $\rightarrow x \rightarrow z \rightarrow w$, therefore there exist a reducing edge for e_2 and thus e_2 is also reducible in G_2. Next we will prove that if e_2 is reducible in G_2, it is also reducible in G. Let P_2 is a reducing path of edge e_2 in G_2. The trivial case is that P_2 does not contain edge e_1. It is evident that e_2 is also reducible in G. In the nontrivial case, P_2 contains $e_1 = (z, w)$. We notice that, in G there exists a path $z \rightarrow v \rightarrow w$ which connects z and w. Therefore, there exist a reducing edge for e_2 and thus e_2 is also reducible in G.

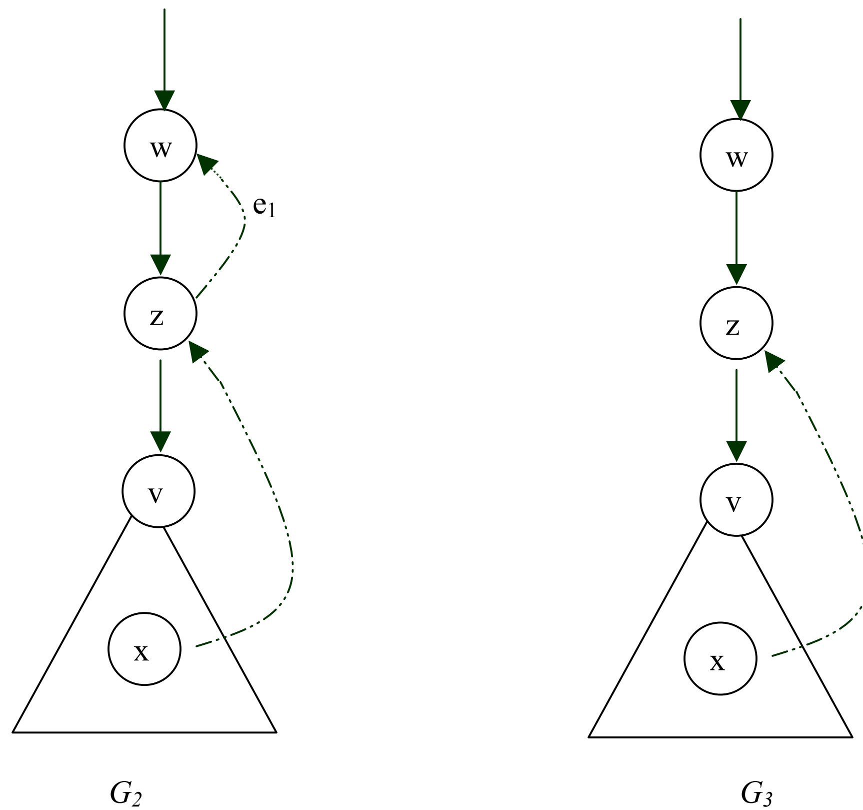


Figure 11. Digraphs G_2 and G_3

Based on the above discussion, we have following algorithm to deal with non-tree edge of a given strongly connected digraph *G*. Our implementation use array backpoint(*v*) to indicate the back point of vertex *v*.

Algorithm 1:

Input A strongly connected digraph *G(V, E)*

Output true: the input digraph may be an MSD

(need to check if there are reducible tree edges)

false: the input digraph is not an MSD

1. do depth first search to get the partitions of edges TREE, BACK, CROSS, FORWARD, The depth first search tree DFS-Tree.
2. for $\forall\ v \in V$ backpoint(v) $\leftarrow v$

   endfor
3. if the set FORWARD is not empty

   return false;

   endif
4. for $\forall\ e = (v, w) \in$ BACK

   backpoint(v) $\leftarrow$ min (backpoint(v), w);

   if there exists an $e = (v, w) \in$B with w $\neq$ backpoint(v)

   return false;

   endif

   endfor
5. for $\forall\ e = (v, w) \in$ CROSS do

   let n be the nearest common ancestor of v and w

   if (w < backpoint(v))

   add (v, n) to G;

   delete (v, backpoint(v)) from G;

   backpoint(v) $\leftarrow w$;

   else

   return false

   endif

   endfor

```
6. //R-test (root of DFS-tree)
R-test(v)
for ∀ w with (v, w) ∈ T do R-test(w)
 if w is not a leaf;
        z ← min (backpoint(w),(v, w) ∈ TREE);
        if (v ≠ z);
                if ( backpoint(v) < backpoint(z))
                        backpoint(z) ← backpint(v);
                else if (v ≠ backpoint(v))
                      return false
                else
                      backpoint(v) ← z;
                 endif
        endif
7.  return false
```

Let us take a look at the running time and space of algorithm 1 complexites. It is well known that depth first search can be done with linear time and storage complexity. Therefore step 1 takes linear time and storage. It is trivial to see that steps 2, 3, and 4 can be implemented in $O(1)$ time per edge considered. This is also true for step 5 without calculation of the nearest common ancestor w for the vertices v and z. In Chapter 4 it will be shown that finding nearest common ancestor can be done in $O(n \log n)$ time and $O(n)$

storage. For any fixed vertex v the step 6 takes linear time in the number of edges emanating form v. So the total cost of 6 is linear. Because we have $O(n \log n)$ time and linear storage complexity in finding nearest common ancestor and linear time and space complexity in each other of our steps, we reach $O(n \log n)$ time and linear storage complexity for the whole algorithm 1.

CHAPTER 3

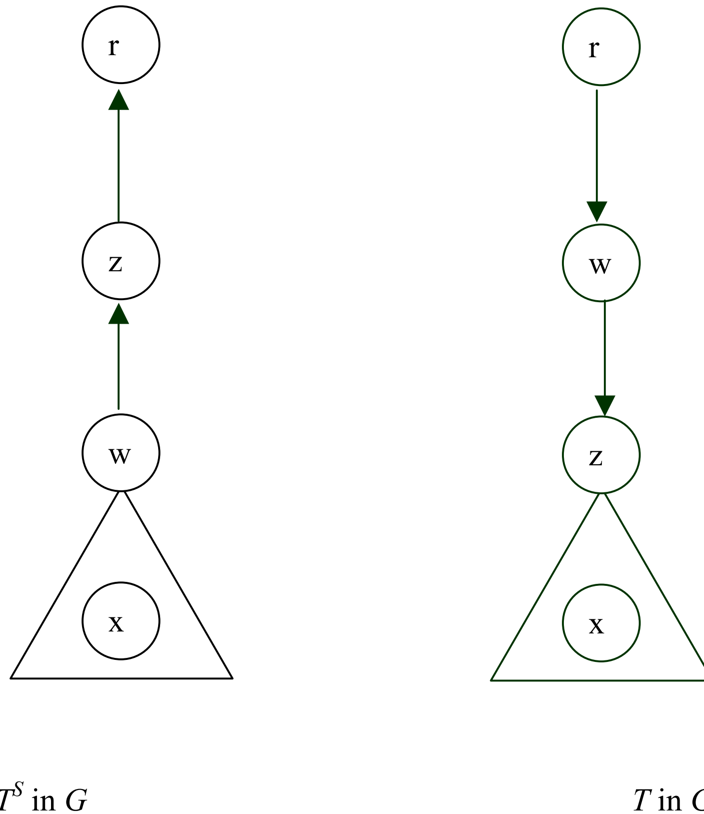
DETECTING REDUCIBILITY OF TREE EDGES

If we can not find any reducible edge by running Algorithm 1, we have to test the reducibility of tree edges. Our strategy is to find reducible tree edges in the process of finding a subgraph containing a spanning in-tree which does not have any reducible edges. It is a well-known fact that if an edge (u, v) is reducible in a strong digraph G, then the edge (v, u) is reducible in the reverse digraph G^R. We reverse every edge of a strong digraph G to obtain the reverse graph of G^R, then we apply algorithm 1 to G^R. If we find a reducible non-tree edge, we conclude that G is not an MSD. Otherwise we know that there is no reducible non-tree edge in G^R. In this process we calculate DFS tree T^R of G^R and the spanning sink tree T^S which is given by the reversal of T^R. Next we run algorithm 1 on the original graph G and, if we find a reducible non-tree edge, it follows that G is not an MSD. Otherwise let T be the DFS tree formed by running algorithm 1 on G. We observe that every edge which might be reducible is both an edge of T^S and an edge of T. We call such edges critical.

Then we will check each critical edge (x, y) to see if it is a reverse dominating edge for x or a dominating edge for y, if we find any critical edge (w, z) which is not a reverse dominating edge for w and not a dominating edge for z, we mark this edge as reducible.

Now we will prove that such an edge (w, z) must be reducible.

From the above discussion, we notice that *G* contains a spanning sink-tree T^S and a spanning tree *T* (see Figure 12). Let r be the root of T^S and *T*. Because (*w, z*) is not a reverse dominating edge for *w*, in the reversal of *G* there is a path from *r* to w which avoids edge (*z, w*). Thus in *G*, there is a path from w to r which avoids (*w, z*). On other hand, because (*w, z*) is not a dominating edge for *z*, there exists a path $r\text{--} \rightarrow z$ in *G* avoiding (*w, z*). Concatenating theses two paths gives a path w to *z* in *G* which avoids (*w, z*) and thus (*w, z*) is reducible.



T^S in *G*　　　　*T* in *G*

Figure 12 T^S and *T* in *G*

Obviously any tree edge (*x, y*) which is either a dominating edge for *y* or a reverse dominating edge for x is not reducible.

Now we can state the final algorithm for finding whether a strong digraph is an MSD or not.

Algorithm 2:

```
1.  reverse G to get G^R,
    run Algorithm 1 on G^R
    if  (Algorithm 1 returns false)
          return false;
    else
        let T^R be the DFS tree associated with G^R
        let T^S be the reverse graph of T^R
        go to step 2.
     endif
  2. run Algorithm 1 on G
     if ( Algorithm 1 return false)
          return false
     else
          let T be the DFS tree associated tree with G
          for edges (x, y) which are in both T and T^S
               if (x is not dominating y and y is not reverse dominating x )
                    return false
               endif
          endfor
    endif
```

In Algorithm 2, in step 1 has $O(n \log n)$ time complexity and linear storage complexity. In step 2 the process for finding an immediate dominator takes $O(n \log n)$ time and linear space. Therefore Algorithm 2 has $O(n \log n)$ time complexity and linear storage complexity.

CHAPTER 4

FINDING THE NEAREST COMMON ANCESTOR

Let $T = (V, E)$ be a tree with root r, and let $P \subseteq V \times V$ be a set of pairs of vertices of T. we wish to computer the Nearest Common Ancestor NCA(x, y) for each pair $\{x, y\} \in P$. In this chapter, we will introduce at first how to deal with NCA issue in a complete binary tree and then to deal with common tree. In order to find NCA on an arbitrary tree T, our plan is to convert the NCA problem on T into an NCA problem of a complete binary tree. This transformation proceeds by a sequence of steps, which involves solving on two auxiliary trees, a compressed tree C and a balanced binary tree B. We will discuss C and B in following sections too.

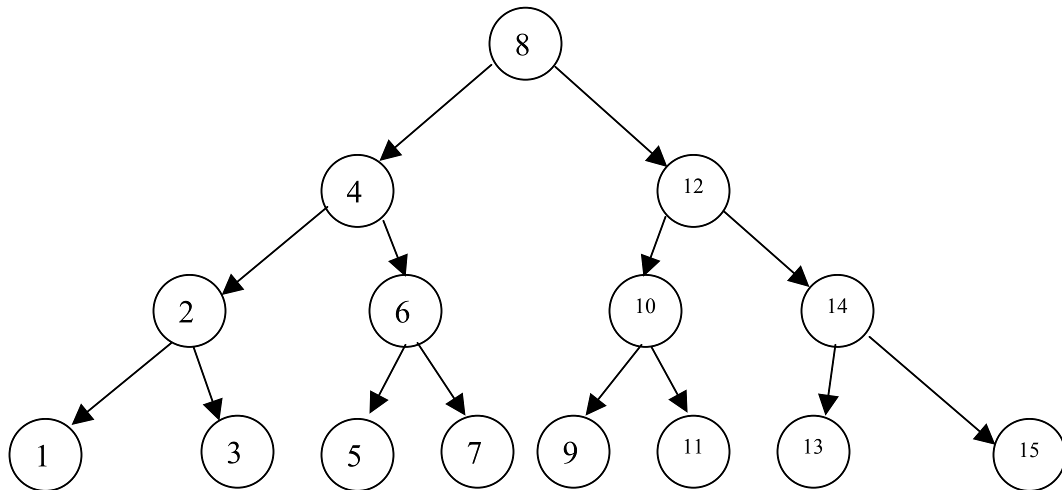


Figure 13. Symmetric-order numbering of a complete binary tree

<u>Special case: a complete binary tree</u>

Let us begin our process for finding NCA (x, y) by considering a complete binary tree CB.

We number the n nodes in CB in such a way that an inorder traversal gives the natural number sequence 1, 2, 3, … n (see Figure 13). This is called the *symmetric-order* numbering. We use sym(v) to denote the number of vertex v, $sym^{-1}(i)$ to denote the vertex whose number is I, h(v) to denote the height of vertex v, d_0 to denote the depth of the nearest common ancestor, and d to denote the depth of the tree . In a complete binary tree, the nearest common ancestor can be solved in $O(1)$ time by direct calculation.

The algorithm to compute the nearest common ancestor of two given vertices v and w can be given in two steps. First note that $d = \lfloor \log n \rfloor$ where log denotes the logarithm to base 2. This is the length of the binary representation of n, which is assumed to be determinable in $O(1)$ time. Similarly the height h(m) of the vertex v with sys(v) = m is the number of trailing 0's in the binary representation of m which is also assumed to be determinable in $O(1)$.

Step 1. Compute the d_0, the depth of NCA(v, w)

Step 2. Compute the ancestor of v at $d - d_0$ steps above.

Step 1 can be accomplished in $O(1)$ time as follows.

If v is an ancestor of w ($sym(w) \in [sym(v) - 2^{h(v)} + 1, sym(v) + 2^{h(v)} - 1]$),

let $d_0 = d - h(v)$. If w is an ancestor of v (($sym(w) \in [sym(v) - 2^{h(v)} + 1, sym(v) + 2^{h(v)} - 1]$),

let $d_0 = d - h(w)$. If v and w is unrelated let $d_0 = d - \lfloor \log(sym(v) \oplus sym(w)) \rfloor$

Step 2 can be accomplished in $O(1)$ time as follows.

Given a vertex v of depth d_1 let $d_2 < d_1$, $h = d - d_2$, the ancestor of v whose depth is d_2 is $sym^{-1}(2^{h+1} \lfloor sym(v)/ 2^{h+1} + 2^h \rfloor)$.

A compressed tree

Let T be an arbitrary n-vertex tree rooted at r. A compressed tree C of T is defined as following. For each vertex v in T, let $\text{size}_T(v)$ be the number of descendants of v (including v itself) in T, $p_T(v)$ be parent of v, Define a n edge $p_T(v) \rightarrow v$ to be *light* if $2\text{size}_T(v) \leq \text{size}_T(p_T(v))$ and *heavy* otherwise. Since the size of a vertex is one greater than the sum of sizes of its children, at most one heavy edge enters each vertex. Thus the heavy edges partition the vertices of T into a collection of *heavy paths* (see Figure 14).

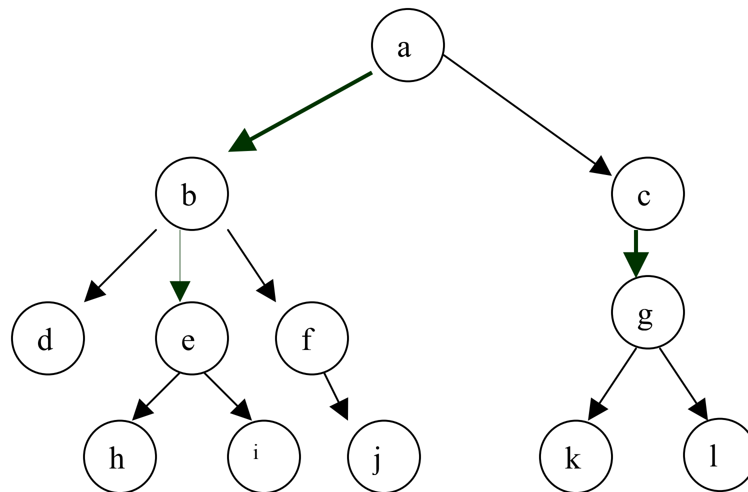


Figure 14. An original tree (heavy edges are shown as thick)

The *apex* of a heavy path is the vertex on the path of smallest depth. For any vertex v we denote by apex(v) the apex of the heavy path containing v, and by hp_size(v) the number of descendants of v on the same heavy path as v. The compressed tree C is defined by the set of edges:

$$\{\text{apex}(p_T(v)) \rightarrow v \mid v \text{ is a vertex of } T \text{ other than } r\}$$

It takes $O(n)$ time to transform an ordinary tree with n vertices into a compressed tree. In $O(n)$ time, We can also compute the following information for each vertex v: $p_T(v)$, $p_C(v)$, apex(v), hp size(v), $d_C(v)$ (the depth of v in C) and $size_C(v)$ (the size of v in C).

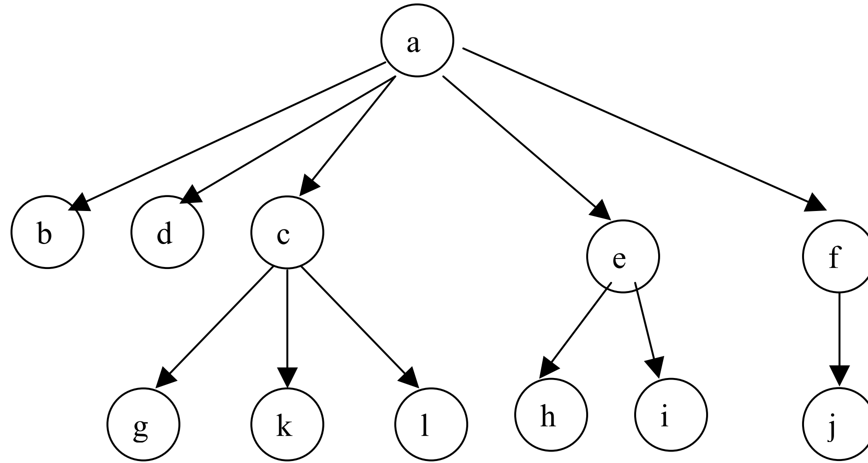


Figure 15. The compressed tree

The balanced binary tree

Now we have the compressed tree. We need to solve the nearest common ancestor problem on the compressed tree. For this purpose we transform C to another auxiliary tree called a balanced binary tree B. The tree B will contain all vertices of C and possibly some additional vertices. Let u equal to $NCA_C(v, w)$ which equals to $NCA_B(v, w)$. If u is on T or the nearest ancestor of u which is on T if u is not on T.

The algorithm for constructing the balanced binary tree is listed below. We suppose C contains a parent v and a set of children W such that $|W| \geq 2$. The binarize process can guarantees that B has depth $O(\log n)$.

binarize (v, W)

Step 1. Let W = $(w_1, w_2, \ldots w_k)$, and $s = \sum_{i=1}^{k} size_C(w_i)$. Let j be the

minimum index such that $\sum_{i=1}^{j} size_C(w_i) \geq s/2$. If j = k, replace j by k−1.

Step2. If j = 1, attach w_1 as the left child of v, otherwise, let x_1 be a new node. attach x_1 as the left child of v and execute binarize(x_1, W_1), where W_1 = (w_1, ...w_j).

Step3. If j = k−1, attach w_k as the right child of v. Otherwise let x_2 be a new vertex, attach x_2 as the right child of v and execute binarize(x_2, W_2) where W_2 = ($w_{j+1}, \ldots, w_k$) .

This method can be implemented to run in $O(|W|)$ time[6]. But in this implementation, we use binary search in step 1 to find j, thus we use O(n log n) time to finish the process of binarizing. To construct B (see Figure 16), we binarize each family of C using the method above. The total run time to construct B is $O(n \log n)$.

In order to find NCA on C, we need to embed B in a complete binary tree B_1, and use the direct calculation as described before. All we need to know for each vertex in B is its symmetric-order number and height, we use following algorithm to number the tree (v is a vertex, h is the height of this vertex in B, and i is the number of the vertex).

number(v, h, i)

Step 1. Assign number i and height h to v

Step 2. If v has a left child w_1, execute number(w_1, h −1, i − 2^{i+1}).

Step 3. If v has a right child w_2, execute number(w_2, h −1, i + 2^{i+1})

It is easy to see that the procedure number(v, h, i) takes linear time and space.

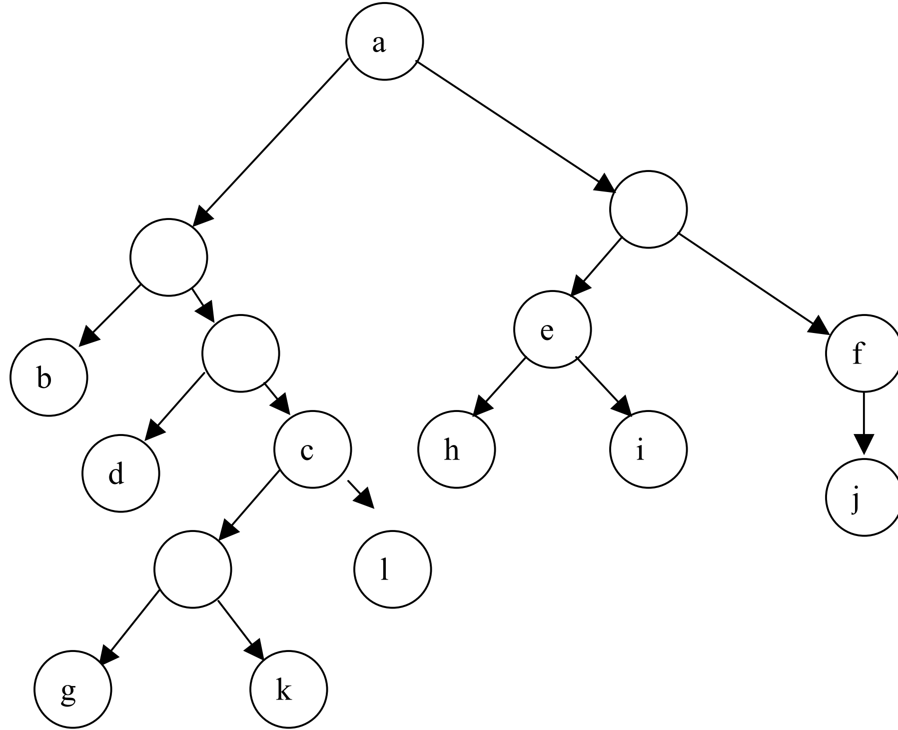


Figure 16. Balanced binary tree corresponding to tree in Figure 14

After we find the nearest common ancestor *v* in B_1, it is possible that v is not a vertex in the original tree *T*. We need to trace the parent of *v* in *B* until we find a real vertex in the *T*. Because *B* has a height of $O(\log n)$, therefore the run time per vertex is $O(\log n)$, thus the run time for find NCA_C for all vertices is $O(n \log n)$.

<u>Algorithm to compute nearest common ancestor</u>

To compute $NCA_T(v, w)$, we first compute the nearest common ancestor $NCA_C(v, w)$ in *C* by using the balanced tree *B*. Recall that for each vertex v we have computed $p_T(v)$, $p_C(v)$, apex(*v*), hp_size(*v*), $d_C(v)$ (the depth of *v* in *C*) and $size_C(v)$ (the size of *v* in *C*) in the process of building the compressed tree *C*.

The algorithm is described as following.

Step 1: Compute $NCA_C(v, w)$

1a compute NCA_B (*v, w*)

1b look up NCA_C (*v, w*)

Step 2: Look up the depth in C of $NCA_C(v, w)$, Compute NCA_T (*v, w*)

Step 2 is composed of following procedures:

2a. Let u ← NCA_C (*v, w*). If (*u* = *v* or *u* = *v*), return *u*, otherwise lookup $d_C(u)$.

2b. Compute the ancestor v_1 of v whose depth is $d_C(u)$+1. If (apex(v_1) = *u*),

let v_2 ← *v*, and otherwise let v_2 ← $p_T(v_1)$

2c. Compute the ancestor w_1 of w_2 whose depth is $d_C(u)$+1, if (apex(w_1) = *u*),

let w_2 ← w_1, and otherwise let w_2 ← $p_T(w_1)$

Return whichever of v_2, w_2 has the larger value of hp_size

It takes O(*n* log *n*) time and linear space to finish step 1 and it takes linear time and space to finish step 2. Thus the time complexity for this algorithm is *O*(*n* log *n*) and storage complexity for this algorithm is linear.

CHAPTER 5

IMPLEMENTATION AND EXPERIMENT

We implemented this algorithm with C++. The input for the whole algorithm is a given strong digraph *G*. Four classes are used in our program. The first is for finding nearest common ancestor, the second is for finding the immediate dominators, the third is for dealing with non-tree edges in *G*. The fourth is to deal with tree edges in *G*.

We tested the running time for randomly generated digraphs up to 1000 vertices. We generated 20 digraphs for each number of vertices from 4 starter digraphs, and we tested running time on the 20 digraphs and then took average.

The algorithm for generating digraph G_1 (with N vertices) from starter MSD digraph *G*:

1. Randomly choose two vertices until we find two vertices *a* and *b* which satisfy two conditions: first, there is no edge (*a, b*), second if we add another vertex between *a* and *b*, the resulted digraph is still MSD
2. Let n be the umber of vertices of current digraph

```
while n < N
    pick a random integer i < 10 ,
     if  i+n < N then
          add i vertices between  a,  b.
     else
        add N –n vertices between a, b
```

endif

endwhile

We test the run time of finding whether a given strong digraph is MSD on the generated digraph (see Table1. Figure 17).

We use the DFS tree of the generated graph to test the running time of finding nearest common ancestor of two arbitrary vertices (see Table 2, Figure 18).

We use the same graph to test the running time of finding immediate dominators (see Table 3, Figure 19).

The three algorithms are tested independently.

The algorithm can surely be implemented in linear time. Adam L. Buchsbaum claimed a new simpler linear time dominator algorithm [7] which has been implemented successfully. There is also a linear algorithm for finding nearest common ancestor [2] but which is very hard to implement. Obviously if the linear time algorithms can be implemented successfully, the performance of finding if a give strong digraph is MSD for sufficient large digraph will be greatly improved.

Table 1. Running time of MSD algorithm

No. of vertices	Seconds
20	0.301
40	0.353
60	0.364
80	0.388
100	0.415
200	0.476
300	0.518
400	0.560
500	0.602
600	0.659
700	0.707
800	0.758
900	0.795
1000	0.829

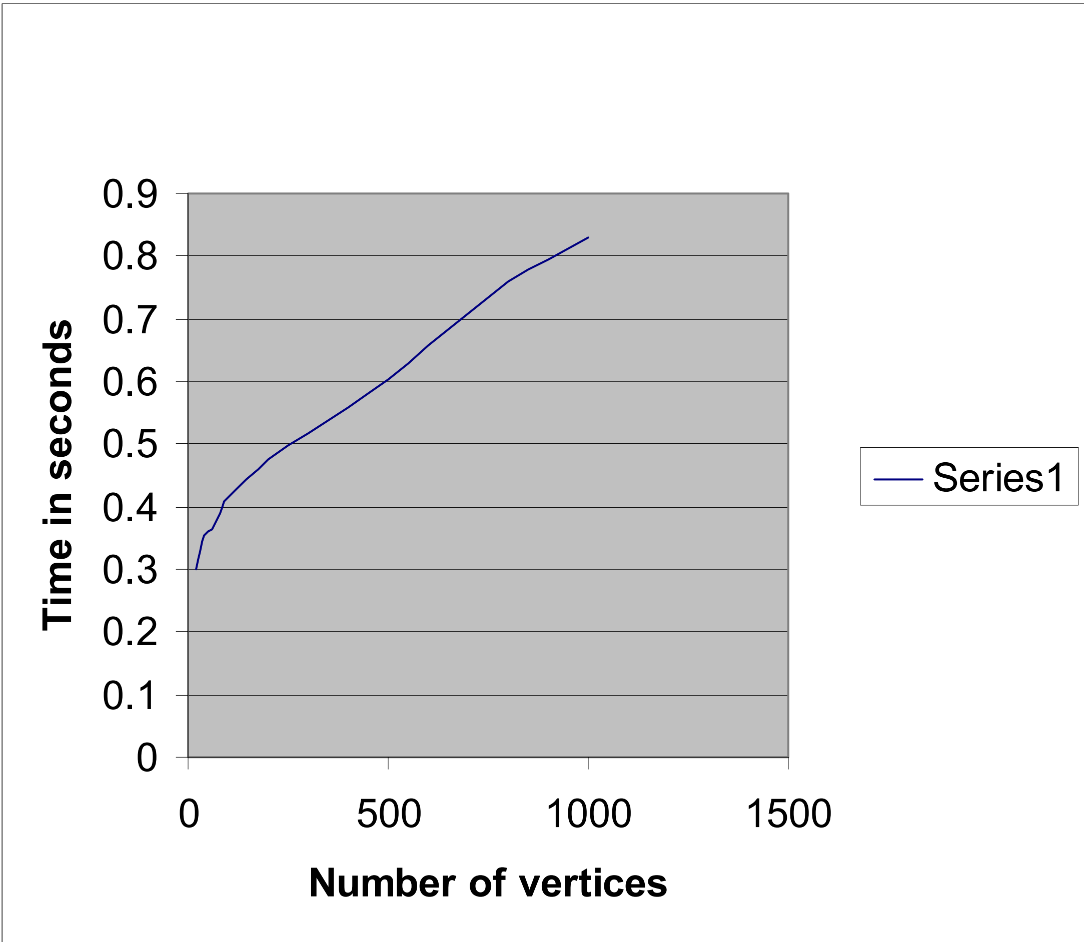


Figure 17. Running time of MSD algorithm

Table 2. Running time of NCA algorithm

No. of vertices	Seconds
20	0.171
40	0.192
60	0.198
80	0.178
100	0.201
200	0.210
300	0.221
400	0.235
500	0.241
600	0.298
700	0.325
800	0.343
900	0.356
1000	0.368

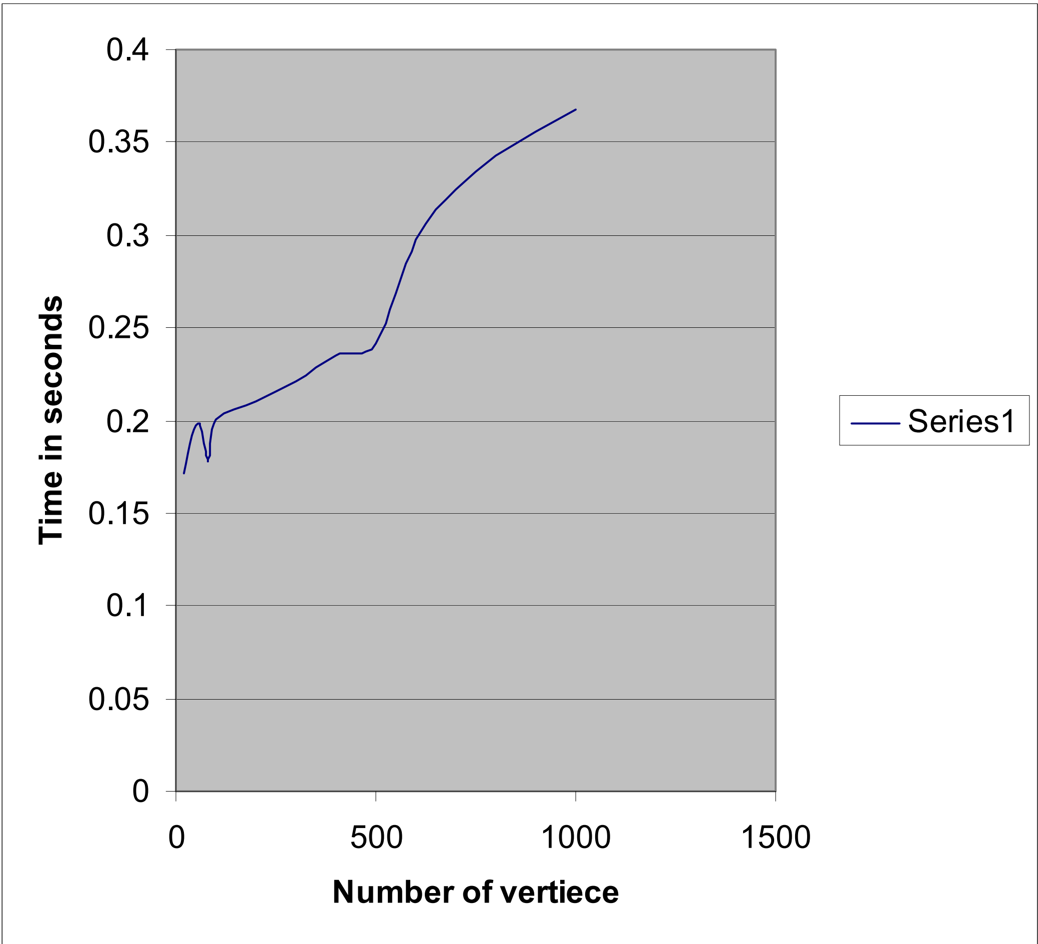


Figure 18. Running time of NCA algorithm

Table 3 Running time in Sec of the immediate dominators algorithm

No. of vertices	Seconds
20	0.013
40	0.014
60	0.016
80	0.017
100	0.019
200	0.022
300	0.020
400	0.025
500	0.026
600	0.032
700	0.036
800	0.039
900	0.042
1000	0.047

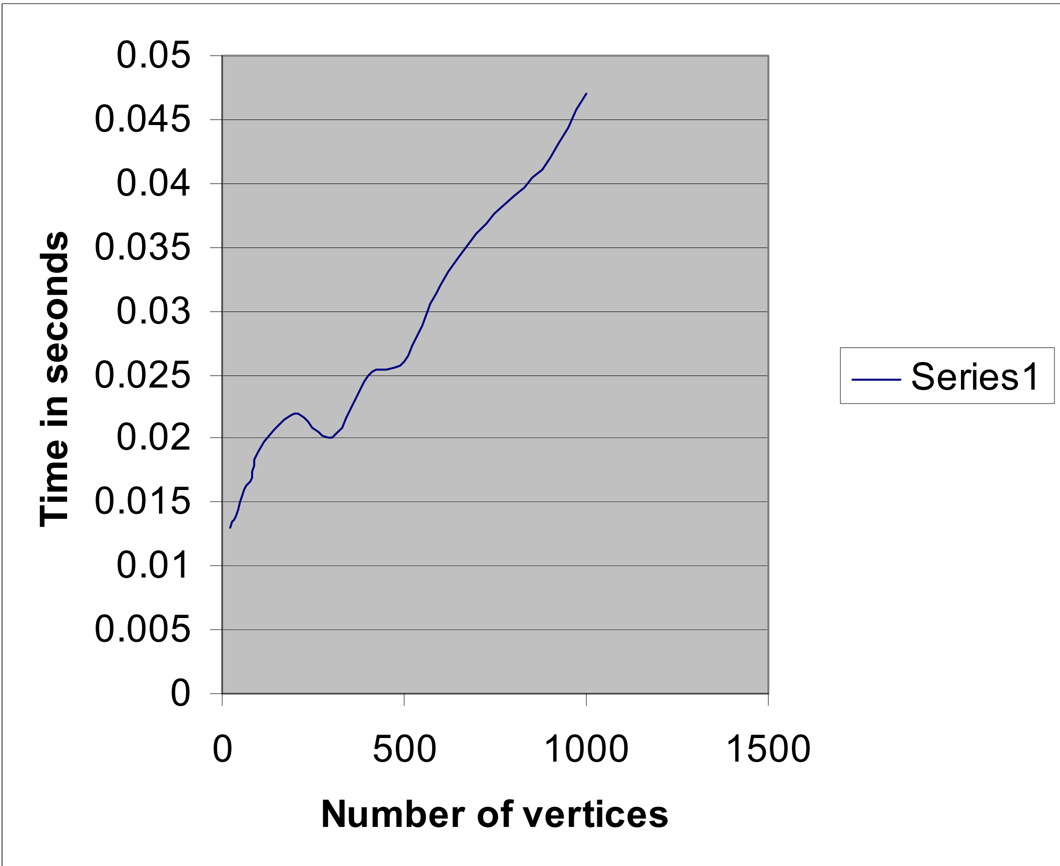


Figure 19. Running time of the finding immediate dominators algorithm

REFERENCES

[1] K. Simon. Finding a minimal transitive reduction in a strongly connected digraph within linear time. Graph-theoretic Concepts in Computer Science Proceedings, 15th International Workshop WG' 89 Castle Rolduc, The Netherlands., 245-259, 1989.

[2] D. Harel and R.E. Tarjan. Fast algorithm for finding nearest common ancestor. SIAM J. Comput., 13: 338-355, 1984.

[3] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thomrup. Dominators in linear time SIAM J. Comput., 28: 2117-32, 1999.

[4] K. K. Bhogadi. Decomposition and generation of minimal strongly connected digraphs. Master's thesis, The University of Georgia, Athens, 1999.

[5] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM Trans. Prog. Lang. Sys., 1: 121-41, 1979.

[6] M. L. Fredman. Two applications of a probabilistic search technique: sorting X+Y and building balanced search trees. Proc. Seventeenth ACM Symposium on Theory of Computing., 2: 240-244, 1975.

[7] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new simpler linear-time dominator algorithm. AT&T Labs-Research., TR 97.31.2., 1998.