Generalized dominators for structured programs

Stephen Alstrup[1] and Peter W. Lauridsen[1] and Mikkel Thorup[1]

Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark (e-mail : stephen,waern,mthorup@diku.dk, www : http://www.diku.dk/~stephen,~mthorup)

Abstract. Recently it has been discovered that control flow graphs of structured programs have bounded treewidth. In this paper we show that this knowledge can be used to design fast algorithms for control flow analysis. We give a linear time algorithm for the problem of finding the immediate multiple-vertex dominator set for all nodes in a control flow graph. The problem was originally proposed by Gupta (Generalized dominators and post-dominators, ACM Symp. on Principles of Programming Languages, 1992). Without the restriction of bounded treewidth the fastest algorithm runs in $O(|V| * |E|)$ on a graph with $|V|$ nodes and $|E|$ edges and is due to Alstrup, Clausen and Jørgensen (An $O(|V| * |E|)$ Algorithm for Finding Immediate Multiple-Vertex Dominators, accepted to Information Processing Letters).

1 Introduction

Constructing dominator trees for control flow graphs $G(V, E, s)$ has been investigated in many papers (see e.g. [8, 9, 11, 13, 14]) in connection with global flow analysis and program optimization. Recently Gupta [6, 7] extended the problem to finding generalized dominator trees which can be use for e.g. propagating loop invariant statements out of loops in cases, where no single node dominates the loop exit, but where a union of nodes together dominates the exit. The generalized dominator tree is constructed by adding to each node in the dominator tree, information about the immediate multiple-vertex dominator, *imdom*, for the node. In [6] an $O(n * 2^n * |V| + |V|^n)$ algorithm is given for computing the *imdom*-set for all nodes, where n is the largest cardinality of any *imdom*-set. Later Sreedhar and Gao have given an $O(|E|^2)$ algorithm [4] using a new representation for flow graphs [5]. The fastest algorithm so far runs in $O(|V| * |E|)$ time and is due to Alstrup, Clausen and Jørgensen [2]. In this paper we give a linear time algorithm for the same problem for structured programs which recently have been shown to have control flow graphs with bounded treewidth (see fact 1 below which is a result due to Thorup [15]). We use a general framework in this paper which is summarized in theorem 15 showing the strongness of using bounded treewidth. The theorem can thus be used as a basic tool for the design of control flow analysis algorithms.

2 Definitions

Let $G(V, E, s)$ be a control flow graph [1] with start node s. The nodes and the edges for G are denoted as $V(G)$ and $E(G)$ respectively. If Y is a subset of nodes of $V(G)$ then $G[Y]$ is the graph induced by the nodes in Y, hence $E(G[Y]) = \{(v, w)|v, w \in Y \wedge (v, w) \in E(G)\}$. If $(v, w) \in E(G)$ we say that v is a predecessor of w. Node v dominates node w if and only if all paths from s to w pass through v. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. The generalized dominator tree is constructed by adding an additional parent to each of the nodes in the dominator tree. The additional node for the node v is holding the immediate multiple-vertex dominator for the node v, $imdom(v)$, which is the minimum set of predecessors of v which together dominates v. More precisely $imdom(v)$ is defined by the following three conditions:

1. $imdom(v) \subseteq predecessors(v) = \{w|(w, v) \in E(G)\}$.
2. Any path from s to v contains a node $w \in imdom(v)$.
3. For each node $w \in imdom(v)$ a path from s to v exists which contains w and does not contain any other node in $imdom(v)$.

3 Structured programs have small treewidth

The usefulness of the linear algorithm given in this paper can be seen from the following fact due to Thorup [15]:

Fact 1 *Control flow graphs for structured programs have* (≤ 6) *treewidth, e.g.*

- *Goto-free Algol [12] and Pascal [16] programs have* (≤ 3) *treewidth control flow graphs.*
- *All Modula-2 [17] programs have* (≤ 5) *treewidth control flow graphs.*
- *Goto-free C [10] programs have* (≤ 6) *treewidth control flow graphs.*

Without short circuit evaluation, each of the above treewidths drops by one.

In the following we give the definition of treewidth and describe previous work with bounded treewidth, which will be used to decompose the control flow graph for the algorithm presented in the next section.

Definition 1. A *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T) where $T = (V(T), E(T))$ is a tree and X is a family $\{X_i|i \in V(T)\}$ of subsets of $V(G)$ such that

1. $\bigcup_{i \in V(T)} X_i = V(G)$.
2. for all edges $(v, w) \in E(G)$, there exists an $i \in V(T)$ with $v \in X_i$ and $w \in X_i$.
3. for all $i, j, k \in V(T)$: if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *width* of a tree decomposition is $\max_{i \in V(T)} |X_i| - 1$. The *treewidth* of a graph G is the minimum width over all possible tree decompositions of G.

From fact 1 and definition 1 we have that a control flow graph can be decomposed for examination of small fractions of the graph, and the following fact due to Bodlaender [3] solves the problem of constructing the tree decomposition.

Fact 2 *1. For all constants $t \in \mathcal{N}$, there exist a linear time algorithm which tests whether a given graph G with n nodes has treewidth ($\leq t$) and if so, outputs a tree decomposition (X, T) of G with width ($\leq t$), where $|V(T)| = n - t$.*

2. We can in linear time convert (X, T) into another tree decomposition (X_b, T_b) of G with width t, where T_b is a binary tree and $|V(T_b)| \leq 2(n - t)$.

The second part follows by the usual binarization of an arbitrary tree.

In the following section we will use a binary tree decomposition, using any node from the tree which includes the start node as root (the rooted tree can be binarized by adding at most one more node to the tree). For a tree node i we use the names $Parent(i)$, $Sibling(i)$, $Left(i)$ and $Right(i)$ for the parent, sibling and children respectively of i. Furthermore we follow the convention that $X_j = \emptyset$ if $j \notin V(T)$.

4 Algorithm

In [2] the problem of finding immediate multiple-vertex dominator (*imdom*) is reduced to a reachability problem for each node in the graph. More specifically $imdom(v)$ is found by first marking all predecessors of v. Next a search method (e.g. depth first) is used to find all nodes which are reachable from s without passing marked nodes. The marked nodes found by this search equals $imdom(v)$. We will refer to this algorithm as the ACJ-algorithm. In this section we assume that a rooted binary tree decomposition of the graph, in which the start node belongs to the root, is given. We can therefore restrict the search in the ACJ-algorithm to a subgraph induced by a tree node to which is added edges containing the necessary reachability information from the rest of the graph. This additional information can be computed simultaneously for all tree nodes by a constant number of graph traversals. To be more specific we traverse the graph by examining the tree nodes first in a bottom-up fashion and then in a top-down fashion in order to find some restricted transitive closures for each tree node. Furthermore we traverse the graph in order to determine reachability from the start node under different constraints and finally we collect the *imdom*-sets. In the rest of this section we denote the tree decomposition of $G(V, E, s)$ as (X, T). We use the term "nodes reachable from v without *passing* nodes from Y", to restrict reachability to paths, on which only v and the node reached can belong to Y.

Definition 2. For a tree node i we define $X_i^{\uparrow}$ as the nodes in descendants of i, hence $X_i^{\uparrow} = \{u | u \in X_j \wedge j \text{ is a descendant of } i\}$. Similarly we define $X_i^{\downarrow}$ as the nodes in X_i and the nodes from tree nodes which are not a descendant of i, hence $X_i^{\downarrow} = \{u | u \in X_j \wedge (j = i \vee j \text{ is not a descendant of } i)\}$ (note that if i is the root node then $X_i^{\downarrow} = X_i$ and if i is a leaf then $X_i^{\uparrow} = X_i$).

4.1 Bottom-Up Closures

Definition 3. For a tree node i and a node $v \in X_i$ we define $TC^{\uparrow}(i,v)$ as the nodes in X_i reachable from v in the subgraph $G[X_i^{\uparrow}]$ without passing nodes from X_i. Furthermore we define $TC^{\uparrow}(i) = \{(v,w) | v,w \in X_i \wedge w \in TC^{\uparrow}(i,v)\}$.

Note that this definition and the following definitions introduce self-loops. These edges are only added to simplify the definitions and will be ignored in the following.

Lemma 4. *$TC^{\uparrow}(i,v)$ are the nodes in X_i reachable from v without passing nodes in X_i in the graph $G[X_{Left(i)} \bigcup X_{Right(i)} \bigcup X_i]$ to which are added the edges in $TC^{\uparrow}(Left(i)) \bigcup TC^{\uparrow}(Right(i))$.*

Proof. If i is a leaf the lemma is trivially true. Assume therefore i is not a leaf. We will first show that all nodes belonging to $TC^{\uparrow}(i,v)$ are reachable from v. Assume $w \in TC^{\uparrow}(i,v)$. If $(v,w) \in E(G[X_i])$ then w is obviously reachable, so let $P = v, y_1, ..., y_k, w$ be a path implying $w \in TC^{\uparrow}(i,v)$. By definition 3 no y_j is in X_i and therefore by definition 1.3 y_1 and y_k are in $X_{Left(i)}$ or $X_{Right(i)}$. Assume without loss of generality that y_1, y_k are in $X_{Left(i)}$. By induction we can find all nodes in $X_{Left(i)}$ reachable from y_1 in $G[X_{Left(i)}^{\uparrow}]$ and as a special case we can find y_k. The path P is therefore represented in the graph.
We now show that all nodes reachable from v are in $TC^{\uparrow}(i,v)$. The only way for this not to be the case would be if one of the additional edges concealed a node from X_i. This is however impossible by definition 1.3. □

By lemma 4 we can find $TC(i)$ for each tree node i by performing simple searches in all tree nodes in a bottom-up fashion. This information can now be used for finding a Restricted Transitive Closure (RTC) for each tree node bottom-up. We define RTC as follows:

Definition 5. For a tree node i and nodes $v,w \in X_i$ we define $RTC^{\uparrow}(i,v,w)$ as the nodes in X_i reachable from w in the subgraph $G[X_i^{\uparrow}]$ without passing v or any predecessors of v. Furthermore we define $RTC^{\uparrow}(i,v) = \{(w,u) | w,u \in X_i \wedge u \in RTC^{\uparrow}(i,v,w)\}$.

Lemma 6. *$RTC^{\uparrow}(i,v,w)$ are the nodes in X_i reachable from w without passing v or any predecessors of v in the graph $G[X_{Left(i)} \bigcup X_{Right(i)}) \bigcup X_i]$ to which are added the edges:*
$TC^{\uparrow}(Left(i))$, if $v \notin X_{Left(i)}$ and $RTC^{\uparrow}(Left(i),v)$ otherwise
$TC^{\uparrow}(Right(i))$, if $v \notin X_{Right(i)}$ and $RTC^{\uparrow}(Right(i),v)$ otherwise

Proof. If i is a leaf the lemma is trivially true. Assume therefore that i is not a leaf. We will first prove that the nodes found belong to $RTC^{\uparrow}(i,v,w)$. To prove this we only need to prove that none of the additional edges can conceal a predecessor of v. If $v \notin X_{Left(i)} \wedge v \notin X_{Right(i)}$ then all predecessors of v

in $X_i^{\uparrow}$ will belong to X_i and none of the additional edges can conceal a predecessor of v by definition 1.3. If $v \in X_{Left(i)}$ by induction we have that the edges in $RTC^{\uparrow}(Left(i), v)$ do not conceal predecessors of v. The case where $v \in X_{Right(i)}$ is analogous. In order to prove that all nodes are found assume that $u \in RTC^{\uparrow}(i, v, w)$. If $v \notin X_{Left(i)} \wedge v \notin X_{Right(i)}$ then all paths from w to u in $G[X_i^{\uparrow}]$ are represented in the graph by lemma 1. If $v \in X_{Left(i)}$ then by induction all paths from w in $G[X_{Left(i)}^{\uparrow}]$ which avoids predecessors of v are represented. Since the only possible remaining paths have to be in $G[X_i]$ all paths which meets the requirements are represented in the graph. The case where $v \in X_{Right(i)}$ is again analogous. □

According to lemma 6 the set $RTC^{\uparrow}(i, v, w)$ can be found by performing the ACJ-algorithm with w as the start node on the graph described in the lemma.

4.2 Top Down Closures

Definition 7. For a tree node i and a node $v \in X_i$ we define $TC^{\downarrow}(i, v)$ as the nodes in X_i reachable from v in the subgraph $G[X_i^{\downarrow}]$ without passing nodes in X_i. Furthermore we define $TC^{\downarrow}(i) = \{(v, w) | v, w \in X_i \wedge w \in TC^{\downarrow}(i, v)\}$.

Lemma 8. *$TC^{\downarrow}(i, v)$ are the nodes in X_i reachable from v in the graph $G[X_{Parent(i)} \bigcup X_{Sibling(i)} \bigcup X_i]$ to which are added the edges $TC^{\downarrow}(Parent(i)) \bigcup TC^{\uparrow}(Sibling(i))$ without passing nodes in X_i.*

Proof. The lemma is analogous to lemma 4 except that in lemma 4 $X_{Left(i)} \bigcap X_{Right(i)} \subseteq X_i$ whereas $(X_{Parent(i)} \bigcap X_{Sibling(i)}) \backslash X_i$ is not necessarily empty. It is however easily verified that these nodes does not affect the correctness. □

Analogously to the bottom-up closure we define a restricted transitive closure top-down:

Definition 9. For a tree node i and nodes $v, w \in X_i$ we define $RTC^{\downarrow}(i, v, w)$ as the nodes in X_i reachable from w in the subgraph $G[X_i^{\downarrow}]$ without passing v or any predecessors of v. Furthermore we define $RTC^{\downarrow}(i, v) = \{(w, u) | w, u \in X_i \wedge u \in RTC^{\downarrow}(i, v, w)\}$.

Lemma 10. *$RTC^{\downarrow}(i, v, w)$ are the nodes in X_i reachable from w without passing v or any predecessors of v in the graph $G[X_{Parent(i)} \bigcup X_{Sibling(i)} \bigcup X_i]$ to which are added the edges:*
$TC^{\downarrow}(Parent(i)) \bigcup TC^{\uparrow}(Sibling(i))$, if $v \notin X_{Parent(i)}$.
$RTC^{\downarrow}(Parent(i), v) \bigcup TC^{\uparrow} Sibling(i)$, if $v \notin X_{Sibling(i)} \wedge v \in X_{Parent(i)}$.
$RTC^{\downarrow}(Parent(i), v) \bigcup RTC^{\uparrow}(Sibling(i), v)$, if $v \in X_{Sibling(i)}$.

Proof. Apart from the difference mentioned in the proof for lemma 8, lemma 10 differs only from lemma 6 in that the case $v \in X_{Sibling(i)} \wedge v \notin X_{Parent(i)}$ cannot occur. The proof for lemma 10 is otherwise analogous to the proof of lemma 6 and is therefore omitted. □

4.3 Reachability

With both the top-down and bottom-up closure edges in hand, all information about paths between nodes in a tree node, which does not pass through nodes from the tree node is obtainable. We therefore only need to know which nodes are reachable from s in each tree node.

Definition 11. For a tree node i we define $STC(i)$ as the nodes in X_i reachable from s in the subgraph $G[X_i^{\downarrow}]$ by a path, on which the last node is the only node from X_i.

Lemma 12. *$STC(i) = \{s\}$, if i is the root node. Otherwise let H be the graph $G[X_{Parent(i)} \bigcup X_{Sibling(i)}]$ to which are added the edges $TC^{\downarrow}(Parent(i)) \bigcup TC^{\uparrow}(Sibling(i))$. Then $STC(i)$ are the nodes in X_i reachable from nodes in $STC(Parent(i))$ by a path P in H, such that the last node on P is the only node in X_i.*

Proof. The proof is similar to the proof of lemma 8 and is therefore omitted. □

Definition 13. For a tree node i and a node $v \in X_i$ we define $RSTC(i, v)$ as the nodes in X_i reachable from s in the graph G by a path on which only the last node allowed to be v or any predecessor of v.

Lemma 14. *If $v \notin X_{Parent(i)}$ then $RSTC(i, v)$ are the nodes reachable from a node in $STC(i)$ by a path, on which only the last node is allowed to be v or any predecessor of v, in the graph $G[X_i]$ to which are added the edges $RTC^{\uparrow}(i, v) \bigcup RTC^{\downarrow}(i, v)$. Otherwise if $v \in X_{Parent(i)}$ then $RSTC(i, v)$ are the nodes reachable from a node in $RSTC(Parent(i), v) \bigcap X_i$ by a path, on which only the last node is allowed to be v or any predecessor of v, in the graph $G[X_i]$ to which are added the edges $RTC^{\uparrow}(i, v)$.*

Proof. If $v \notin X_{Parent(i)}$ then all predecessors of v in $G[X_i^{\uparrow}]$ are in X_i and therefore a path exists from s to all nodes in $STC(i)$ which avoids v and predecessors of v. This observation together with definitions 5, 9 and 11 establishes the first case. The second case follows directly from the same definitions and the definition of $RSTC$ by induction. □

4.4 Closure of a tree node

In this subsection we will summarize the information defined in the previous subsections. The theorem below shows that it is possible in linear time to preprocess a graph with bounded treewidth, so as to "close" each tree node. In other words we can ensure that certain paths, restricted in different ways, between or to nodes of a tree node, are represented in the tree node. The theorem can thus be used for the development of efficient algorithms in graphs with bounded treewidth.

Theorem 15. *Given a graph $G(V, E, s)$ with treewidth t and a binary tree decomposition of G we can preprocess G in linear time, so that for each preprocessed tree node i:*

1. *For each node $v \in X_i$ the set of nodes in X_i reachable from v without passing other nodes in X_i is known. The preprocessing has complexity $O(|V|t^3)$*
2. *For each node pair $v, w \in X_i$ the set of nodes in X_i reachable from w without passing v or any predecessors of v is known. The preprocessing has complexity $O(|V|t^4)$.*
3. *All nodes in X_i reachable from the start node s by a path on which the last node is the only node from X_i are known. The preprocessing has complexity $O(|V|t^3)$*
4. *For each node $v \in X_i$ all nodes in X_i reachable from the start node s by a path, on which only the last node is allowed to be v or any predecessor of v, are known. The preprocessing has complexity $O(|V|t^4)$.*

Proof. To obtain the information in 2 and 3, the information in 1 has to be available. Similarly the information in 4 requires the information in 1,2 and 3.

1. By lemma 4 we can find all paths between nodes of X_i which passes through proper descendants of i, by performing, for each node in X_i, a simple search in a graph induced by 3 tree nodes containing at most $3t$ nodes. The complete search for each tree node can thus be done in $O(t^3)$-time and since there are $O(|V|)$ tree nodes the complexity is established. By lemma 8 we can find all paths which passes "above" a tree node analogously.
2. By lemma 6 we can find all paths avoiding v and predecessors of v in the subgraph spanned by the tree nodes in the subtree rooted at i, by performing the ACJ-algorithm with w as start node. Thus the ACJ-algorithm with complexity $O(t^3)$ has to be run for each node in X_i. Since there are at most t nodes in X_i and $O(|V|)$ tree nodes the complexity is established. Again we can find all paths in the graph "above" i analogously.
3. By performing a similar search as in 1 in each tree node in a top-down fashion, we can find the nodes required.
4. By performing the ACJ-algorithm analogously to 2 in each tree node in a top-down fashion, we can find the nodes required.

□

4.5 Finding Imdom-sets

We will now present the main theorem for finding the *imdom*-sets.

Theorem 16. *$Imdom(v) = \{w|(w, v) \in E(G) \wedge w \in \bigcup_{i \in V(T)} RSTC(i, v)\}$, where $RSTC(i, v) = \emptyset$, if $v \notin X_i$.*

Proof. Let I denote the right hand side of the expression. We will prove the theorem by showing that I satisfies the three conditions in the definition of *imdom*.

1. Obvious.
2. Assume that a path from s to v exists which does not include any node in I. Let w denote the predecessor of v first reached on the path. Since the path from s to w does not include any predecessors of v, w must belong to $RSTC(j, v)$ for a tree node j to which both v and w belong, contradicting that $w \notin \bigcup_{i \in V(T)} RSTC(i, v)$.
3. Follows directly from the definition of $RSTC$. □

Theorem 17. *Given a control flow graph $G(V, E, s)$ with treewidth t and a binary tree decomposition (X, T) of G, we can find $imdom(v)$ for each node $v \in V$ in linear time.*

Proof. By theorem 15 we can compute $RSTC(i, v)$ for each pair $i \in V(T)$, $v \in V$ in $O(|V|t^4)$-time. By theorem 16 we can find the *imdom*-sets for each node v by traversing the tree and for each tree node i collect the nodes which are predecessors of v and belong to $RSTC(i, v)$ (the collection of nodes belonging to *imdom*-sets for all nodes can of course be done simultaneously). □

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. S. Alstrup, J. Clausen, and K. Jørgensen. An $O(|V| * |E|)$ algorithm for finding immediate multiple-vertex dominators. Technical Report DIKU-TR-96/4, Deparment of Computer Science, University of Copenhagen, 1996. Revised version with minor change accepted to Information Processing Letters.
3. H. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proc. 25th ACM. Symp. on theory of Comp.(STOC'93)*, pages 226–234, 1993.
4. G.R. Gao, Y.-F Lee, and V.C. Sreedhar. DJ-graphs and their application to flow-graph analyses. Technical Report 70, McGill University, School of Computer Science, ACAPS, May 1994.
5. G.R. Gao and V.C. Sreedhar. A linear time algorithm for placing ϕ-nodes. In *ACM Symp. on the Principles of Programming Languages (POPL'95)*, volume 22, pages 62–73, January 1995.
6. R. Gupta. Generalized dominators and post-dominators. In *ACM Symp. on Principles of Programming Languages (POPL'92)*, volume 19, pages 246–257, 1992.
7. R. Gupta. Generalized dominators. *Information processing letters*, 53:193–200, 1995.
8. D. Harel. A linear time algorithm for finding dominator in flow graphs and related problems. In *Proc. 17th Ann. ACM Symp. on theory of Comp. (STOC'85)*, pages 185–194, 1985.
9. M.S. Hecht and J.D. Ullman. A simple algorithm for global data flow analysis of programs. *SIAM J. Comput.*, 4:519–532, 1975.
10. B.R. Kernighan and D.M. Ritchie. *The C programming language.* Prentice–Hall, New-Jersey, 1978.
11. T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Programming Languages Systems*, 1:121–141, 1979.

12. P. Naur. Revised report on the algorithmic language algol 60. *Comm. ACM*, 1(6):1–17, 1963.
13. P.W. Purdom and E.F. Moore. Immediate predominators in a directed graph. *Comm. ACM*, 15(8):777–778, 1972.
14. R.E. Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974.
15. M. Thorup. Structured programs have small tree-width and good register allocation. Technical Report DIKU-TR-95/18 (revised version), Deparment of Computer Science, University of Copenhage, 1995.
16. N. Wirth. The programming language pascal. *Acta informatica*, 1:35–63, 1971.
17. N. Wirth. *Programming in modula-2(3rd corr.ed)*. Springer–verlag, Berlin, New York, 1985.

This article was processed using the LaTeX macro package with LLNCS style