Department of Computer Science

Aarhus University

Master's Thesis

Algorithms for Finding Dominators in Directed Graphs

Author:
Henrik Knakkegaard Christensen
20082178

Supervisor:
Gerth Stølting Brodal

January 2016

Abstract

A graph is a theoretical construct and by this, so are the relations of its vertices. In this thesis, we will look at different algorithms for finding the dominator relations in directed graphs. We will cover the inner workings of the different algorithms along with proof of complexity, correctness and termination. Furthermore, implementations of the algorithms will be covered and experiments of their complexities along with experiments comparing them against one another, will be performed.

Contents

1 Introduction

In 1959, Reese T. Prosser was the first to describe the dominance relationships in analysis of flow diagrams. He gave some applications to what the dominator relations could be used for [Pro59]. This included which parts of a program could be interchanged with one another and when preparing a program for a machine that admits parallel operations, which operations in the program that could be performed simultaneously. Prosser did not give an algorithm for calculating the dominator relations, but only described the relation.

Formally, the dominator relation is a relation between vertices in a directed graph. A directed graph $G = (V, E, r)$ is a theoretical construct created by $n = |V|$ vertices, $m = |E|$ edges and has a starting vertex $r \in V$. Each edge $(v, u) \in V$ connects two vertices v and u in one direction, which makes it possible to go from vertex v to vertex u through this edge. A vertex v is dominated by another vertex w, if every possible path from the starting vertex r, to the vertex v in the graph G, goes through the vertex w. The immediate dominator for a vertex v is the vertex u, which dominates v and is also dominated by all of v's dominators, except itself. Hereby u is the last vertex that every path in a graph have to go through to reach v. The immediate dominators for all vertices in a graph can be arranged into a tree structure such that for a given vertex in the tree it is dominated by all of its ancestors. In Figure 1.1, a graph G is shown to the right and to the left is the dominator tree of the graph.

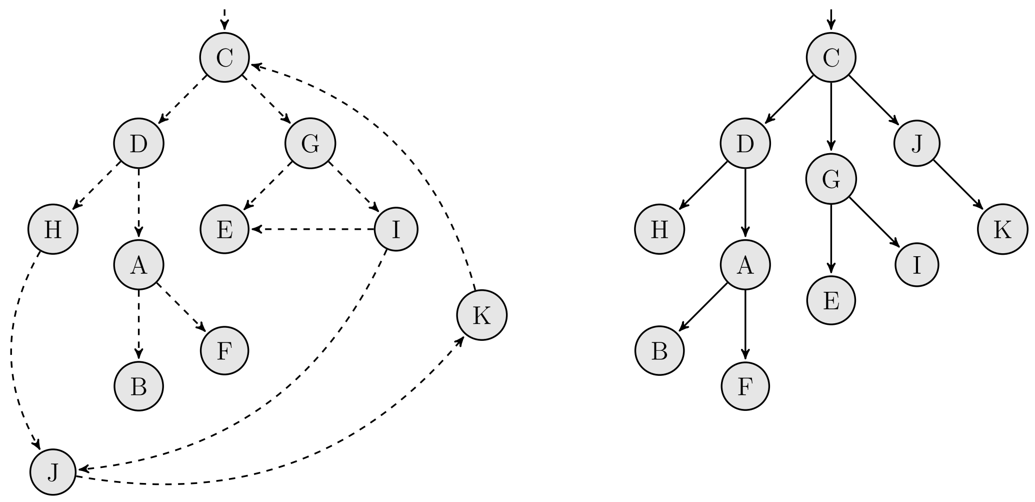


Figure 1.1: To the left, A graph with 11 vertices and 13 edges and to the right, the dominator tree for the graph.

Allen and Cocke presented an algorithm in 1972 using flow analysis that could calculate all the dominator relations in a graph [AC72]. It does this by an iterative approach where it calculates the dominator set for each vertex in the graph, depending on the dominator sets of its predecessors. If some dominator set have been changed in an iteration, the algorithm continue with another iteration over all vertices and this continues until all dominators in the graph have been found. In the same year, Aho and Ullman [AU72] described a more straightforward algorithm for finding dominator relations. By observing that a path to a given vertex can only exists if the path goes through the dominators of the vertex. They used this to formulate an algorithm that temporarily removes one vertex from the graph and then finds all vertices that can be reach from the starting vertex. All vertices that can not be reach are by this, dominated by the removed vertex. Later in 1979, Lengauer and Tarjan presented a fast algorithm for finding dominators [LT79]. This approach creates a tree structure for the graphs and uses this to find an approximated immediate dominator for each vertex, after this it uses these approximated immediate dominators alongside observation of the tree structure to find the true immediate dominators for all vertices. In 2001, Cooper, Harvey and Kennedy [CHK01] presented a simple and fast algorithm which uses the same approach as the algorithm presented by Allen and Cocke from 1972, but uses a tree structure while calculating the dominators for each vertex, insted of sets. In 2006, an algorithm which uses the same principle as the Lengauer-Tarjan algortihm was presented in [GTW06], but creates a dominator tree for a graph by using observations of the relations in the dominator tree.

This was a short history of a few algorithm that can be used to find dominators and it is these algorithms that we will use in this thesis. In Chapter 2, we will go through some structures and methods which is essential while working with dominator relations. In Chapter 3, we will go through each of the algorithms just mentioned, we will get some insight in how the they work, how they finds dominators and we will take a look at the correctness, termination and complexity of each of them. In Chapter 4 we will go through some experiments, where each algorithm have been implemented and executed.

A table of the algorithms that we are going to use, together with their complexities and output type, can be found in Table 1.1.

Section	Name	Time	Space	Output
3.1.1	Iterative, boolean vector	$O(mn^2)$	$\Theta(n^2)$	Bit matrix
3.1.2	Iterative, tree	$O(mn^2)$	$O(m+n)$	Dominator tree
3.2	Vertex removal	$O(mn)$	$O(n^2)$	List of vertices
3.2	Vertex removal	$O(mn)$	$\Theta(n^2)$	Bit matrix
3.2	Vertex removal	$O(mn)$	$O(m+n)$	Dominator tree
3.4	Lengauer-Tarjan (simple linking)	$O(m \log n)$	$O(m+n)$	Dominator tree
3.5	SEMI-NCA	$O(n^2)$	$O(m+n)$	Dominator tree

Table 1.1: Five different algorithms for calculating dominators with time complexity, space complexity and output format.

2 Structures

In this chapter, we will also go through some structures and methods which is used by the algorithms in Chapter 3. This includes what graphs and trees are, what kind we use in the relations of dominators and we will also go through some different forms of traversals on these graphs. We will achieve an understanding on how to use these structures to represent dominators.

2.1 Graphs

In graph theory, a graph is a mathematical structure used to represent a set of objects with some pairwise relations between them. Objects can contain arbitrary information such as a single value, some piece of code or the names of the employees in a company, depending of what the graph is used for.

In this chapter, we will keep it as simple as possible and just use single letters to identify each object. We call the objects in the graph for vertices and the pairwise relation between them for edges. Formally we can describe a graph $G = (V, E)$ as the collection of vertices V and edges E with the number of vertices $n = |V|$ and the number of edges $m = |E|$.

The edges are used to move around, such that it is possible to move from one vertex to another, if there is an edge connecting the two. The possible movements in a graph can be described by a path. That is, if it is possible to reach a vertex v from a vertex y by following edges from vertex to vertex any number of times, then there exists a path from y to v. We will denote such a path from y to v in the graph G as $y \overset{*}{\rightarrow}_G v$. Edges can be either undirected or directed. That is, if two vertices is connected by an undirected edge, it is possible to go from one vertex to the other and it is also possible to go back again through the same edge. If the edges in the graph are not undirected, then they are directed, which only allow movement in one direction. It is then only possible to go from one vertex to another vertex by a single edge, but not back again, through the same edge. It is then only possible to go back again, if there is another edge between the two vertices that allow movement in that direction.

A graph can also have an assigned vertex r from where all movement around the graph have to start. We call this vertex the root of the graph and we denote a graph which have such a vertex as $G = (V, E, r)$, where $r \in V$ is the root. It is necessary to have such a root defined in the graphs used in this thesis, because the dominator relation is defined as a path from this very same vertex.

In this thesis we will illustrate a graph by a set of circles which represent vertices. Lines represent undirected edges and arrows represent directed edges. If the path goes through vertices that are not drawn this is represented with a wavy arrow. In this thesis, we will use dashed edges to represent graphs and solid edges to represent trees, which will be defined in Section 2.2. In Figure 2.1, there is shown an example of a simple graph consisting of four vertices A, B, C, D. Vertex A is the root illustrated with an arrow pointing to it from nowhere. Furthermore B, C, D are connected in such a manner that there are paths to every vertex in the graph, starting from the root.

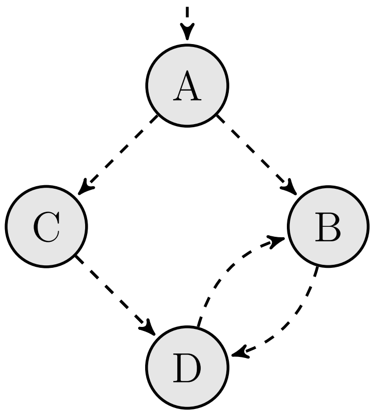


Figure 2.1: An example of a directed graph $G = (V, E, r)$ with four vertices $\{A, B, C, D\} \in V$, where the vertex A is defined as the root. The edges in the the graph contains five directed edges $\{(A, B), (A, C), (B, D), (C, D), (D, B)\} \in E$.

2.2 Trees

A tree is a subset of a graph, which means that a tree consists of the same elements, such as vertices, directed edges, undirected edges and a root. A tree can not have any cycle path in it and because of this there is a hierarchical ordering of the vertices in a tree. This means that each vertex have some relation to the other vertices in the tree. We have already defined what a root is, and to make it is easy to grasp all relations used by this thesis, we will represent all relations used in the following list, where T is the tree.

- Root: The top vertex in a tree, denoted as r.
- Path: A path in the tree is a sequence of vertices and edges connecting one vertex to another. We will denote a path from vertex y to vertex v in the tree T as $y \xrightarrow{*}_T v$.
- Depth: The depth of a vertex v is the number of edges on the path $r \xrightarrow{*}_T v$.
- Parent: Vertex p is the parent of vertex v, if the edge (p, v) exists in the tree.
- Child: Vertex v is a child of p, if the edge (p, v) exists in the tree.
- Sibling: Vertices that have the same parent.
- Ancestor: An ancestor for a vertex v is a vertex a on the path $a \xrightarrow{*}_T v$, this includes v itself.
- Descendant: Vertex v is a decendant of vertex a if a is v's ancestor. This can include the vertex itself.
- Proper ancestor: The same as ancestor but can not include the vertex itself. We will denote such a path from the proper ancestor a to the vertex v in the tree T as $a \xrightarrow{+}_T v$.
- Proper descendant: The same as decendant, but can not be the vertex itself.
- Subtree: The subtree for a vertex v is a set of vertices $E' \in E$ which are all its descendant. The vertex v is by this the root in its subtree.
- Singleton: A tree with only one vertex.
- Forest: A forest is a set of trees, where the trees in the set are not connected with each other.

2.3 Traversal Order

A traversal on a graph or tree is the process of visiting each vertex in the set. All the algorithms in Chapter 3 traverse a given graph to find the dominator relations for all vertices. They do this by visiting each vertex in the graph, and for each vertex they either do some examination or update some structure. In some algorithms, it does not matter which vertex is visited before any other of the vertices. But in most algorithms it does lower the time for finding of the dominator relations, by traversing the vertices in some specified order.

There are many ways to order the vertices for visit, but we will only go through the orderings used by the algorithms in Chapter 3. To keep it manageable, here is a list:

- RANDOM ORDER:

  The vertices are traversed in any random order.

- DEPTH-FIRST-SEARCH (DFS):

  As described in [CLRS01, Sec. 22.3], when doing the process of visiting each vertex with DFS the vertices can be numbered such that each vertex will get some unique number. The numbers indicate in which order the algorithms in Chapter 3 shall visit the vertices. The algorithms use four different ways of numbering the vertices with DFS for traversal, but common for all is that the vertices are numbered with a timestamp which indicate when it was visited and therefore all vertices gets an unique number. In Figure 2.2, the DFS-traversal of a graph is shown and two of four orderings are printed. The last two orderings are the same as the first two, but in reverse order. The four orderings created with a DFS-traversal are:

  – Pre-order

    When first visiting a non-numbered vertex with a DFS of the graph, the vertex gets a number, which is called discovery time.

  – Post-order

    The vertex gets a number when all the edges from the given vertex have been visited and numbered, this is called finishing time.

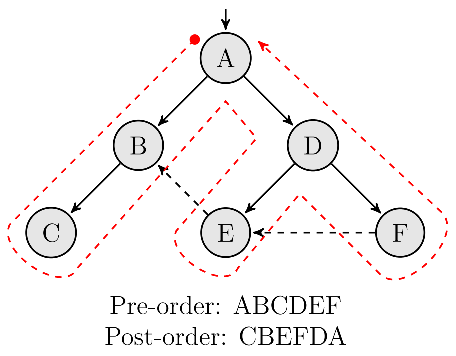
  – Reverse pre-order

    This is the reverse order of a pre-order.

  – Reverse post-order

    This is the reverse order of a post-order.

- Breadth-first-search (BFS). Level-order.: When visiting the vertices with a BFS the vertices gets a number in a pre-order fashion. This means that a vertex gets an unique number when visited An example of the BFS-traversal on a graph can be seen in Figure 2.2 where the level-order of the vertices are also printed.



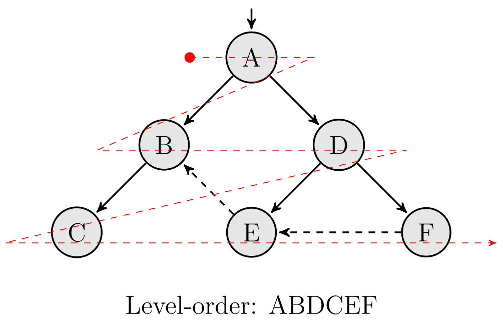


Figure 2.2: The left figure shows a DFS on a graph consisting of 6 vertices and 6 edges. All black lines are a part of the graph, but only the solid lines are a part of the tree created by the DFS. The red line indicates which order the vertices are first visited in. The pre-order and post-order ordering produced are shown underneath the graph. The figure to the right shows a BFS on a graph consisting of the same 6 vertices and 6 edges. All black lines are a part of the graph, but only the solid lines are part of the tree created by the BFS. The red line indicates which order the vertices are visited and the level-order ordering produced is shown underneath the graph.

2.4 Dominators Relation

In a graph $G = (V, E, r)$ a dominator is the relation between two vertices. A vertex v is dominated by another vertex w, if every path in the graph from the root r to v have to go through w. Furthermore, the *immediate dominator* for a vertex v is the last of v's dominators, which every path in the graph have to go through to reach v.

The dominator relation between all vertices in a graph can be represented in a number of ways and depending on how this dominator information is going to be used, some structures for representing the relations can be more preferable than others.

2.5 Queries

An input graph for an algorithm for finding dominators is $O(m + n)$ in size and depending on the representation, the output is potentially $O(n^2)$. So a more compact structure for representing dominators is relevant. Furthermore depending on the application and how the dominator relations is going to be used, some simple queries on a graph relating to dominators might be interesting to get answered:

Query 1. Does vertex w dominate vertex v?

Query 2. Which vertices does w dominate?

Query 3. Which vertices is v dominated by?

Depending of the structures used to store the dominator relations, the query time for these three simple queries may differ. In the next section we will go through some structures and relate how these structures handle the different queries.

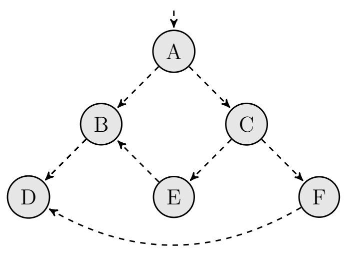
2.6 Representation of Dominators

In this thesis, we are using three different structures to represent the dominator relations for a given graph. The structures are directly derived from how the algorithms keep track of the dominator relations while finding them. We will see how the dominator relations are represented in the three structures and how space consumption and time complexity for answering the queries from Section 2.5 are affected.

Without loss of generality, we can define that all vertices dominate themselves. In some of the algorithms in Chapter 3, it helps in the implementation and description of the algorithms that vertices dominate themselves and in other algorithms it does not matter, in any case it does not change the dominator relation between two different vertices.

Boolean vectors

To represent the dominator relations using boolean vectors, each vertex in the graph will have its own vector, such that there are n vectors in total. The vectors each have n entries, one entry for each vertex in the graph. For a vertex v, the boolean value in each entry of its vector, indicate which vertices in the graph dominates it. We define that the value *true* in a given entry means that the associated vertex dominates v and that the value *false* indicates that it does not. All of these vectors can be represented in a matrix table with n^2 entries. In Figure 2.3, there is shown such a matrix table and a graph, where the matrix table indicates the dominator relations for the graph. Each row in the matrix is a boolean vector for a given vertex v. The value *true* in a entry is shown with a white box with a "t" in it and the value *false* is shown in a red box with an "f" in it. By reading a single row i, it indicates which vertices dominate vertex i and by reading a single column j, it indicates which vertices, vertex j dominates. For example, by looking at the C-row, it shows that the vertex C is only dominated by vertex A and by looking at the C-column it shows that C dominates vertex E and F.



	A	**B**	**C**	**D**	**E**	**F**
A	t	f	f	f	f	f
B	t	f	f	f	f	f
C	t	f	f	f	f	f
D	t	f	f	f	f	f
E	t	f	t	f	f	f
F	t	f	t	f	f	f

Figure 2.3: An example, how to represent the dominator relations with boolean vectors.

The space consumption for this representation is $\Theta(n^2)$ since the matrix has to hold n vectors, each with n entries. The time complexity for each query are:

Query 1. *Does vertex w dominate vertex v?* $O(1)$ time.

An entry in the matrix can be evaluated in constant time.

Query 2. *Which vertices does w dominate?* $\Theta(n)$ time.

Every entry in a column can be evaluated in $O(1)$ time and there are n entries in a column thus $\Theta(n)$ time.

Query 3. *Which vertices is v dominated by?* $\Theta(n)$ time.

Every entry in a row can be evaluated in $O(1)$ time and there are n entries in a row thus $\Theta(n)$ time.

Dominator tree

The dominator relations can also be represented in a tree structure. Where the root of the tree is the same as the root in the graph. All other vertices are linked such that their parent are their immediate dominator. The immediate dominator as introduced in Section 2.4 is a vertex u that dominates a vertex v and which is also dominated by all other of v's dominators. Such that if v has more than one dominator, its immediate dominator is the last vertex, a path from the root to v have to go through. In Figure 2.4 a graph is shown and its dominators are represented with a dominator tree. In this representation all ancestors for a vertex is its dominators and all its descendants are dominated by it.

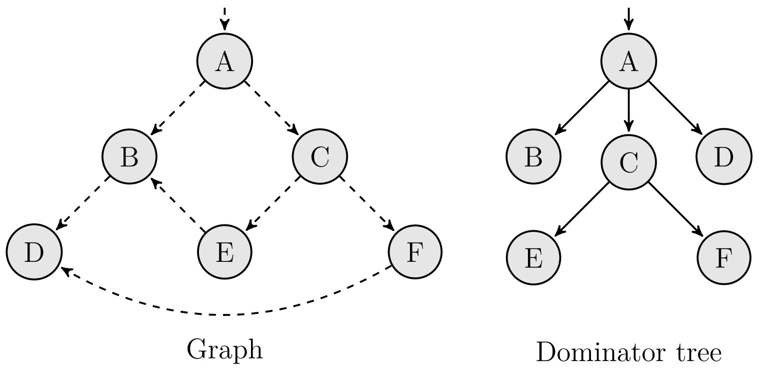


Figure 2.4: An example how to represent the dominator relation in a tree structure called dominator tree.

The space consumption for this representation is $\Theta(n)$ since all vertices are represented once and each vertex has one edge. The time complexity for each query are:

Query 1. *Does vertex w dominate vertex v?* $O(1)$ time.

By doing a depth-first-numbering of the dominator tree, such that a vertex keeps a interval of which vertices are in its subtree, this query can be answered in constant time.

Query 2. *Which vertices does w dominate?* $O(k)$ time.

It is output sensitive to return all vertices that a vertex dominates, this query can be answered by reporting all vertices in its subtree, therefore it takes time proportional to the size of the subtree, $O(k)$ where $k \neq n$

Query 3. *Which vertices is v dominated by?* $O(k)$ time.

It is output sensitive since all of v's dominators are ancestors of v in the domintor tree. The time for returning all these vertices is bound by the number of ancestors k, $k \leq n$.

List with vertices, dominators and non-dominators

By representing the dominator relation for each vertex in a link list, the implementation of algorithm in Section 3.2 can be done easily. The list for vertex v can either contain the vertices that it dominates or it can contain the set of vertices in the graph that it does not dominate. Figure 2.5 a graph and the two types of lists are shown.

We will start with the representation where the list contains the vertices that the given vertex dominators. The space consumption is worst-case $O(n^2)$ since the root vertex can have n elements in its list, another vertex can have $n-1$ vertices in its list since it can not dominate the root, a third vertex can have $n-2$ and so on, this evaluates to $O(n^2)$. The time complexity for the three queries are:

Query 1. *Does vertex w dominate vertex v?* $O(k)$ time.

It is dependent on the number of vertices w dominates, by searching for v in w's dominator list, $k \leq n$.

Query 2. *Which vertices does w dominate?* $O(k)$ time.

It is output sensitive such that to return k vertices in the list will take $O(k)$ time where $k \leq n$.

Query 3. *Which vertices is v dominated by?* $O(n^2)$ worst-case time.

We have to search through all lists for all of the vertices, which in worst-case takes $O(n^2)$ time.

For the other representation, where the list contains vertices that the given vertex does not dominate: The worst-case space consumption is $O(n^2)$ since every vertex except the root can contain all other vertices. The time complexity for the three queries are:

Query 1. *Does vertex w dominate vertex v?* $O(k)$ time.

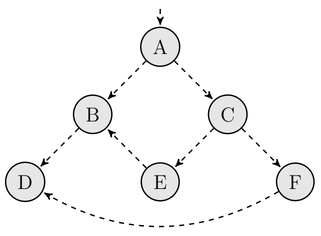
It is dependent on the number of vertices w dominate, by searching for v in w's list where $k \leq n$.

Query 2. *Which vertices does w dominate?* $\Theta(n)$ time.

We have to calculate the dominator list before we can report it, so we have to look at each vertex in the graph and look if it is in the w's list.

Query 3. *Which vertices is v dominated by?* $O(n^2)$ time.

We have to search through all lists for all of the vertices, which in worst-case takes $O(n^2)$ time.



Node	Dominated by	Not dominated by
A	**A**	BCDEF
B	**AB**	CDEF
C	**AC**	BDEF
D	**AD**	BCEF
E	**ACE**	BDF
F	**ACF**	BDE

Figure 2.5: An example of how to represent the dominator relations by a linked list containing vertices that each vertex dominates, or a linked list with vertices a vertex does not dominate.

3 Algorithms for Finding Dominator Relations

In this chapter we describe and analyse five algorithms for finding the dominator relation in graphs and a single algorithm that finds approximated immediate dominators, used by two of the other algorithms. For each algorithm, there will be a description which will contain observations and explanation of how the algorithm calculate the dominator relations in a given graph. There will also be an analysis of each algorithm containing termination, correctness and complexity. To help the reader to understand the different algorithms, each section will also contain a pseudocode implementation of the algorithms and figures with examples showing how a given algorithm finds the dominator relations in a graph.

3.1 Iterative Algorithms

The two iterative algorithms in this sections both have $O(mn^2)$ time complexity and they both use an iterative strategy to calculate dominators. They do this, by having a wrong initial solution of dominators for a given graph, then continuously iterating over all vertices and at each vertex try to change the solution so that the solution becomes closer to the right dominator relations. The algorithms continues until the right dominator relations have been found, this is when there are no changes in the solution in an entire iteration over all vertices.

3.1.1 Data-flow Equation, Boolean Vector Algorithm

In 1970, Allen defined the computation of dominance relations in terms of a data-flow equation and even gave a procedure for finding dominators in an interval. An interval is the maximal single entry subgraph where all cycles in the subgraph contains the entry vertex [All70]. She came up with The following data-flow equation:

$$\forall v \in V : dom'(v) = \{v\} \cup \left(\bigcap_{q \in pred(v)} dom'(q) \right) \tag{3.1}$$

Allen and Cocke continued in 1972 [AC72] showing an iterative algorithm for finding dominators in a directed graph using this data-flow equation.

In the control flow Equation 3.1, $dom'(v)$ is a set of vertices which dominates v, this set can be represented as a boolean vector of size n. Where entry w in the boolean vector for vertex v is *true* if vertex w dominates v. Otherwise w does not dominate v and the entry value is *false*.

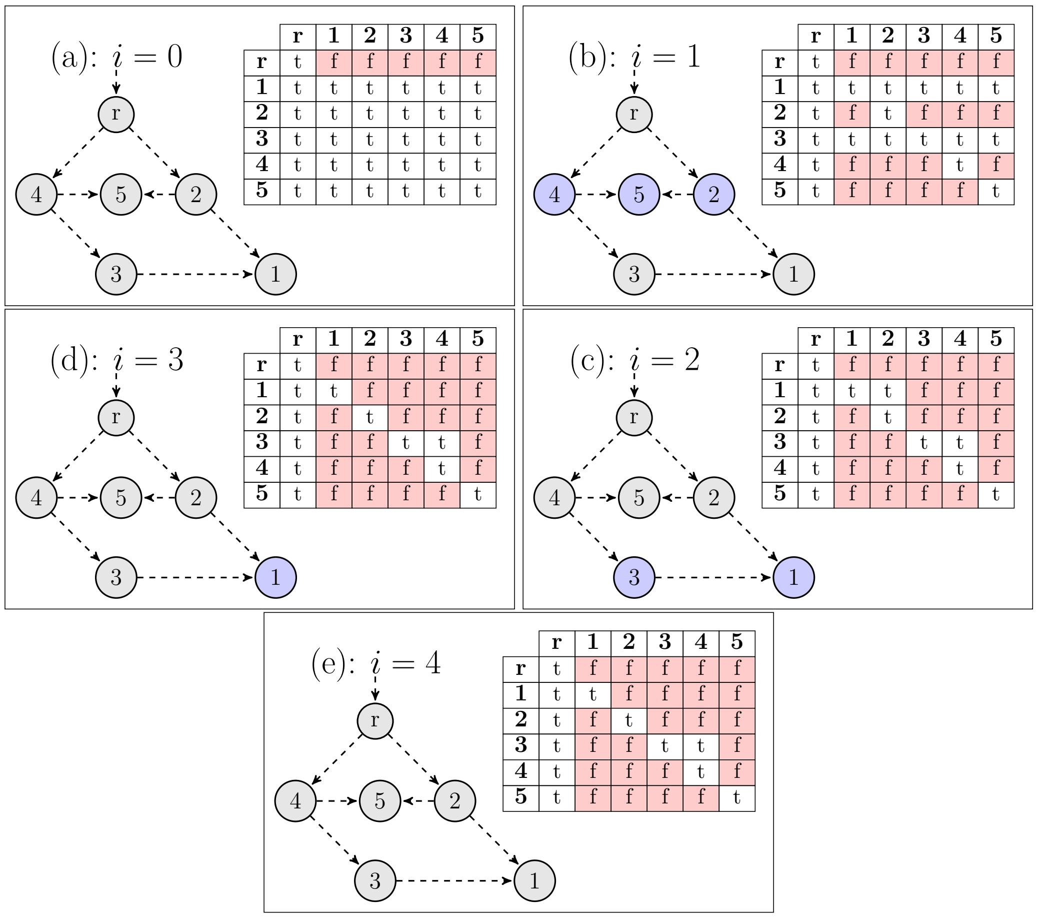
The algorithm finds the dominator relations by iterating over all vertices in the graph and applying Equation 3.1 to each of them, which results in a subset of a given vertex's dominator set.

A pseudocode of the algorithm can be found in Algorithm 1 and an example of how the algorithm calculate dominators in a graph is shown in Figure 3.1.

Algorithm 1 Iterative Algorithm using boolean vectors

```
Input: Graph $G = (V, E, r)$
Output: Dominator set for each vertex.
1: $dom'(r) \leftarrow \{r\}$
2: for $v \in V - \{r\}$ do
3:     $dom'(v) \leftarrow V$
4: end for
5: while changes occured in any $dom'(v)$ do
6:     for $v \in V - \{r\}$ do
7:         $dom'(v) \leftarrow \{v\} \cup \left( \bigcap_{q \in pred(v)} dom'(q) \right)$
8:     end for
9: end while
```

Input: A directed graph $G = (V, E, r)$.
Output: All the dominators for each vertex in G. The structure is boolean mvectors.
Method: We start by initialize the data structure $dom'(r) = \{r\}$ and all other vertices to $dom'(v) = V$. Then start the search for dominators by iterating over all vertices in the graph and applying the data-flow equation (3.1) until no dominator sets changes [ASU86, AC72].



(a)

	r	1	2	3	4	5
r	t	f	f	f	f	f
1	t	t	t	t	t	t
2	t	t	t	t	t	t
3	t	t	t	t	t	t
4	t	t	t	t	t	t
5	t	t	t	t	t	t

(b)

	r	1	2	3	4	5
r	t	f	f	f	f	f
1	t	t	t	t	t	t
2	t	f	t	f	f	f
3	t	t	t	t	t	t
4	t	f	f	f	t	f
5	t	f	f	f	f	t

(d)

	r	1	2	3	4	5
r	t	f	f	f	f	f
1	t	t	f	f	f	f
2	t	f	t	f	f	f
3	t	f	f	t	t	f
4	t	f	f	f	t	f
5	t	f	f	f	f	t

(c)

	r	1	2	3	4	5
r	t	f	f	f	f	f
1	t	t	t	f	f	f
2	t	f	t	f	f	f
3	t	f	f	t	t	f
4	t	f	f	f	t	f
5	t	f	f	f	f	t

(e)

	r	1	2	3	4	5
r	t	f	f	f	f	f
1	t	t	f	f	f	f
2	t	f	t	f	f	f
3	t	f	f	t	t	f
4	t	f	f	f	t	f
5	t	f	f	f	f	t

Figure 3.1: An example of how the interative algorithm with boolean vectors calculate the dominator relations. This example does not visit the vertices in an optimized order. In each subfigure (a,b,c,d,e), there are an iteration counter i, a boolean vector for each vertex, ordered in a matrix, where entries are colored red if the value is *false*, otherwise the entry is *true* and colored white. In the graph, the vertices are numbered in the order that they are visited, in each iteration of the algorithm. The vertices are colored blue if their dominator boolean vector changes. In (a), the initialization step is performed where $dom'(v) = V$ for all $v \in V$ except $dom'(r)$ which is initialized to $\{r\}$. In the first iteration (b), vertex 1 and 3 do not inherit any changes from their predecessors, but 2 and 4 inherit from r and then 5 inherits from 2 and 4. In the second iteration (c), vertex 1 only inherits changes from 2 and 3 inherits changes from 4. In iteration (d), vertex 1 now inherits changes from 3. In the last iteration (e), there are no changes and the algorithm terminates.

Termination: In each iteration when calculating a new set of dominators for a given vertex v, the calculation involves the intersection of v's predecessors boolean vertices together with a union of the vertex itself. This calculation always results in a subset of the boolean vector for any vertex. Since the boolean vectors cannot continue getting smaller indefinitely, the algorithm will eventually terminate [ASU86, sec. 10.9].

Correctness: We started by initializing dominators for all the vertices to V, except the root which we initialized to itself, $dom'(r) = \{r\}$. Invariant: At the beginning of each iteration dom' is a superset of dom. In each iteration the data-flow equation allows the removal of a vertex x from the dominator set of a given vertex v, but only if x has been removed in a dominator set of at least one of v's predecessors. This means that we can find a path $r \overset{*}{\rightarrow}_G q \mid q \in pred(v)$ which does not contain x. Thereby we have a path $r \overset{*}{\rightarrow}_G v$ without x and it is correct to remove x from the set $dom(v)$.

It is not possible to add any vertex to any dominator sets due to the initialization and data-flow equation, which always calculate a subset of any given dominator set [ASU86]. By this we cannot add any vertices to any dominator sets and only remove vertices from any set, if the data-flow equation allows it.

To see that the algorithm will not terminate before all dominators have been correctly calculated. Assume that somewhere in the execution of the algorithm, a given vertex v have a vertex x in its dominator set, but x is not a dominator for v. In that iteration, the algorithm will iterate over all vertices in V as it does in every iteration. It will find an ancestor a of v in a path from $r \overset{*}{\rightarrow}_G a \overset{+}{\rightarrow}_G v$ without x, otherwise x would be dominating v. All of a's successors will then remove x from their dominator sets in this iteration, this will result in an update of these dominator set and the algorithm will take another iteration. This will continue until x is removed from v's dominator set. There by, all *false* dominators will be removed in each dominator set before the algorithm terminates and thus the algorithm gives the correct output.

Complexity: In each iteration the algorithm iterates through all vertices, then for each vertex it calculates the intersection of all its predecessors dominator sets. This results in a total of m intersection calculations in total, since there are m predecessors in the graph in total. Each dominator set is a boolean vector of size n, thus the total time for calculation all the intersection in each iteration is $O(mn)$ time. We do this until the algorithm terminates, which is when any vertex x, which is not a dominator for any vertex v, have be removed from v's dominator set. In the graph, a path $r \overset{*}{\rightarrow}_G v$ which only passes through any given vertex once, can be no longer than $n-1$. A vertex v

with x in its dominator set, means that there is a path $r \xrightarrow{*}_G a \xrightarrow{+}_G v$ without x and since the maximal length to v from r is $n-1$, x will be removed in at most $n-1$ iterations. By this we have the total complexity for calculating dominator relation to $O(mn^2)$ time.

The space consumption of the algorithm is dominated by the n boolean vectors of size n, thereby resulting in $\Theta(n^2)$.

3.1.2 Dominator-tree Algorithm

Another iterative algorithm for finding dominators in graphs, was presented in 2001 [CHK01]. The algorithm uses the same strategy as the iterative boolean algorithm in Section 3.1.1, but the efficiency of the algorithm is improved by saving memory in the way the dominator relations are represented, this also result in some time saving. The observation they use is that the dominator set $dom(v)$ for a given vertex v can be represented as a list of immediate dominators $idom(v)$:

$$dom(v) = \{v\} \cup idom(v) \cup idom(idom...(v))$$

They saw the link between this relationship and the dominator tree data structure, where each vertex in the dominator tree have its immediate dominator as parent. They also present a detailed pseudocode implementation, but the explanation in [GTW06] is easier to understand, and it is also this explanation, which is incorporated in the pseudocode that can be found in Algorithm 2.

Another observation used in the algorithm is that when any directed spanning tree of a graph rooted at r is created, any vertex in the graph will have its immediate dominator as an ancestor in the spanning tree, since every path in the graph to a vertex have to go through its dominators.

They used this observation to make an iterative algorithm that creates a directed spanning tree from an input graph. Afterwards, it applies updates that changes edges in the spanning tree, which continues in an iterative fashion until the spanning tree becomes the dominator tree of the graph. At this point, the algorithm terminates, returning the resulting dominator tree.

The most straightforward way to implement this, is to iterate over all vertices until a vertex v is found where the edge to this vertex in the spanning tree can be updated, such that the spanning tree is getting closer to be the dominator tree. An example of how the algorithm finds the dominators can be found in Figure 3.2 and the psedocode can be seen in Algorithm 2.

An edge to a vertex can be updated when some of its ancestors can be shown not to be its dominator. We can be sure that these vertices are not the dominator of v by calculating the nearest common ancestor in the spanning tree between v and one of its predecessors from the graph. The predecessor q is also located somewhere in the spanning tree and by doing this, we find a vertex a which dominates v and q in the spanning tree. Thereby, all the vertices x on the path $a \overset{+}{\rightarrow}_T x \overset{+}{\rightarrow}_T v$ can not dominate v in G since there is a path $x \notin (a \overset{*}{\rightarrow}_G q \overset{+}{\rightarrow}_G v)$.

Algorithm 2 Iterative Algorithm, Tree

```
Input: Graph $G = (V, E, r)$
Output: Dominator tree
1: Create any tree $T$ as subgraph of $G$ rooted at $r$
2: while changes occured in $T$ do
3:     for $v \in V$ do
4:         for $u \in pred_G(v)$ do
5:             if $u \neq parent_T(v)$ and $parent_T(v) \neq \text{NCA}_T(u, parent_T(v))$
6:                 $parent_T(v) \leftarrow \text{NCA}_T(u, parent_T(v))$
7:         end for
8:     end for
9: end while
```

Input: A directed graph $G = (V, E, r)$.
Output: Dominator tree.
Method: Start by creating any directed spanning tree T rooted at r, which is a sub graph of G. Then iterate over all vertices and find a vertex v that has a predecessor q which is not its parent p in T. Calculate the nearest common ancestor a for p and q in T and change the edge (p, v) to (a, v), continue doing so until no edges changes in an entire iteration, then terminate.
Termination: The sum of depth is strictly decreasing. Every time we make an update in T, we change an edge for a vertex from its parent, to one of its ancestors. Thereby we reduce the depth of the vertex in the tree and since we cannot increase the depth and we can not have a negative depth, the algorithm will terminate.

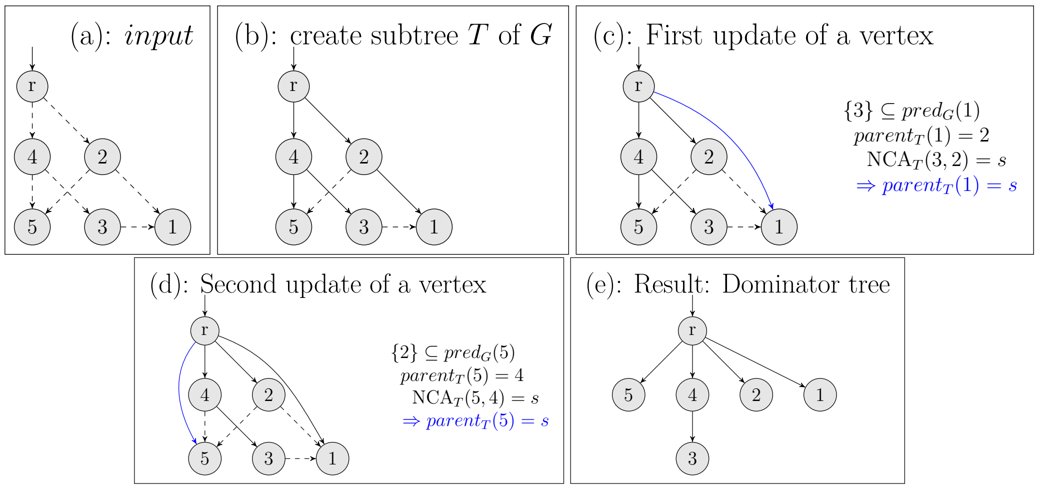

(a): input
(b): create subtree T of G
(c): First update of a vertex
$\{3\} \subseteq pred_G(1)$
$parent_T(1) = 2$
$\mathrm{NCA}_T(3,2) = s$
$\Rightarrow parent_T(1) = s$
(d): Second update of a vertex
$\{2\} \subseteq pred_G(5)$
$parent_T(5) = 4$
$\mathrm{NCA}_T(5,4) = s$
$\Rightarrow parent_T(5) = s$
(e): Result: Dominator tree
r
1
2
3
4
5

Figure 3.2: An example of how the iterative algorithm using a tree, calculates dominator relations. The graph in subfigure (a) is the input graph. In (b) a tree rooted at r, which is a subgraph of G, all dashed edges in the graphs in subfigure (a-d) represents the edges of the input graph G, which is not in T, and all solid edges represents the tree T. Every graph/tree shown in each subfigure is the result after the given step described for each subfigure. (c) Shows the first iteration of the algorithm, where vertex 1 have been found to satisfy the conditions described in Section 3.1.2 and that can be seen in line 5 of Algorithm 2. The conditions have also been written in black to the right of the tree. The blue arrow and text is the update that has been performed. In (d) a second vertex is being updated. In subfigure (e) there are no vertices satifing the conditions and the algorithm terminates, returning the dominator relation in the form of a dominator tree.

Correctness: We use the observation that the immediate dominator for each vertex $v \neq r$ is in the path $r \xrightarrow{+}_T v$, where T is the tree initialized as a directed spanning tree. In each iteration, we update at least one edge in T, where the updated edge is the edge from a parent p to a vertex v, and is updated such that it points from an ancestor of p, instead of p, but still points to v. The ancestor a is found by doing a nearest-common-ancestor search with p and one predecessors $q \in pred_G(v)$, which is also located somewhere in T. This search yields a vertex a, which dominates both v and q in T. By this, the vertices that are proper decendens of q and proper ancestors of v, can not dominate v and the edge update is correct.

When the algorithm terminates the tree T returned is the dominator tree. In every iteration of the algorithm, if T is not the dominator tree, then there is at least one vertex, which does not have its immediate dominator as a parent in T. The algorithm is bound to find one edge to update in an iteration. This is due to a given vertex v which does not have its immediate dominator as parent in T, will have a path $r \xrightarrow{+}_G v$ through one of its predecessors without p, otherwise p would be a dominator to v. All predecessors can not be dominated by p, this would mean that there are no path around p to v and that p is v's immediate dominator. By applying this statement on v and its predecessor, and its predecessor, and so on, we will eventually find a vertex which have a predecessor, which is not dominated by p in T and an edge can be updated. An illustration of this can be seen in Figure 3.3.

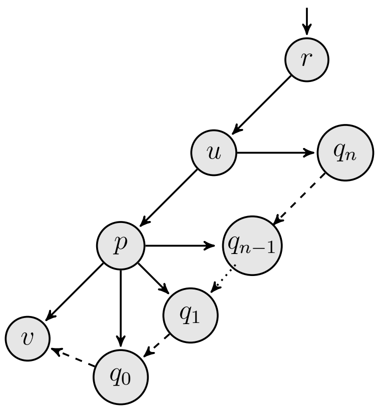


Figure 3.3: Illustration that a vertex v, which is not dominated by its parent p in a spanning tree, will have a path from the root not containing p.

Complexity: An spanning tree can be computed in $O(m)$ time. Nearest-common-ancestor computation takes $O(n)$ with a naively approach. In each iteration all vertices are visited and for all of their predecessors, $O(m)$ nearest-common-ancestors are computed. The number of iterations the algorithm needs to do is $O(n)$, since the maximal length to the root for any vertex is $n - 1$. This results in the total computation time $O(mn^2)$. The space consumption is dominated by the graph $O(m + n)$.

3.1.3 More Complexity for Iterative Algorithms

Although the two iterative algorithms have $O(mn^2)$ time complexity some types of graphs and some traversal order of the vertices can speed up the calculations for finding dominators. Kam and Ullman [KU76] shows when processing the vertices in reverse post-order by DFS, certain data-flow equations can be solved in at most d(G)+3 passes on an arbitrary instance, these data-flow equations also include the dataflow Equation 3.1 [GTW06].

The iterative algorithms will terminate after a single iteration, if the input graph is acyclic and the vertices is traversed in reverse postorder. This is because the vertices in a reverse post-order are topological sorted, such that each vertex is traversed before any of its successors [GTW06].

3.2 Vertex-removal Algorithm

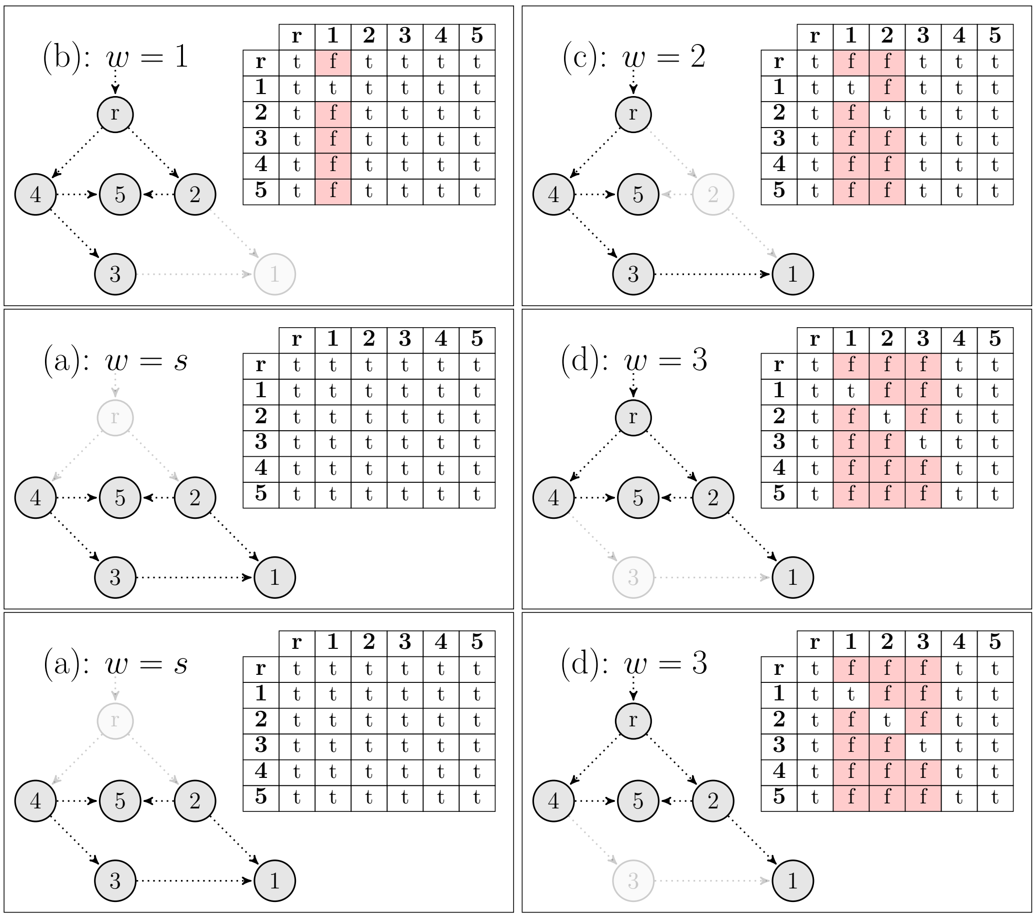
A more straightforward way to find dominators is described in [AU72, p. 916]. It uses the observation that if a vertex w is removed from a graph G and it was a dominator to v, then there are no longer any paths from r to v in the graph, since all path have to go through w, otherwise w could not be a dominator to v.

A pseudo implementation of the algorithm can be found in Algorithm 3 and an example of how the algorithm calculates dominators on a graph can be found in Figure 3.4.

Algorithm 3 Removal of vertices

```
1: for $w \in V$ do
2:     Traverse $V'$, $V' \leftarrow V - \{w\}$
3:     $w$ dominates vertices not visited.
4: end for
```

Input: A directed graph $G = (V, E, r)$.
Output: Dominator relations in some structure depending on the implementation, which changes the space requirement slightly.
Method: For each vertex w, traverse the graph for $V' = V - \{w\}$ starting from the root r, w dominates all vertices not visited in the traversal.
Termination: The algorithm traverse the graph n times, once for each vertex. In each traversal we follow each reachable edge once and since there are a finite number of edges the algorithm terminates.
Correctness: When removing a vertex w in the graph V', if there are no path from r to a vertex v in a traversal of V', but there are in V. This means w is a vertex on all paths from r to v in G and by this w is a dominator to v. Also every other vertices which are possible to visit in V' means that there is a path from r to v, which does not containing w, thereby w is not a dominator to those vertices.
Complexity: There are n vertices and for each vertex we traverse the graph V' by following each edge that is reachable. Since there are m edges all of this takes $O(mn)$ time. But we still have to keep track of the dominator relations, this can be done by maintaining a linked list of visited vertices



(b): $w = 1$

	r	1	2	3	4	5
r	t	f	t	t	t	t
1	t	t	t	t	t	t
2	t	f	t	t	t	t
3	t	f	t	t	t	t
4	t	f	t	t	t	t
5	t	f	t	t	t	t

(c): $w = 2$

	r	1	2	3	4	5
r	t	f	f	t	t	t
1	t	t	f	t	t	t
2	t	f	t	t	t	t
3	t	f	f	t	t	t
4	t	f	f	t	t	t
5	t	f	f	t	t	t

(a): $w = s$

	r	1	2	3	4	5
r	t	t	t	t	t	t
1	t	t	t	t	t	t
2	t	t	t	t	t	t
3	t	t	t	t	t	t
4	t	t	t	t	t	t
5	t	t	t	t	t	t

(d): $w = 3$

	r	1	2	3	4	5
r	t	f	f	f	t	t
1	t	t	f	f	t	t
2	t	f	t	f	t	t
3	t	f	f	t	t	t
4	t	f	f	f	t	t
5	t	f	f	f	t	t

(a): $w = s$

	r	1	2	3	4	5
r	t	t	t	t	t	t
1	t	t	t	t	t	t
2	t	t	t	t	t	t
3	t	t	t	t	t	t
4	t	t	t	t	t	t
5	t	t	t	t	t	t

(d): $w = 3$

	r	1	2	3	4	5
r	t	f	f	f	t	t
1	t	t	f	f	t	t
2	t	f	t	f	t	t
3	t	f	f	t	t	t
4	t	f	f	f	t	t
5	t	f	f	f	t	t

Figure 3.4: An example of how the vertex removal algorithm calculates the dominator relations. In each subfigure (a,b,c,d,e,f) there are a variable w which shows which vertex is being removed from the graph. There is a graph V' where the removed vertex w, is transparent. And there is a boolean matrix of all the dominator relations. The boolean matrix have been chosen to make a more simple example. In subfigure (a) the start vertex s have been removed from the graph making it impossible to visit any vertices, thereby s dominates all vertices. In subfigures (b,c,d and f) vertices (1,2,3 and 5) are respectively removed from the graph and in each traversal that given vertex is the only one not visited, thereby it is the only vertex dominated. In subfigure (e) vertex 4 is removed, making it impossible to visit vertices 3 and 4 in a traversal and by this vertex 4 dominates vertex 3 and itself.

for each vertex v. Maintaining such a list can be done by adding a new vertex to the list when first visited, in $O(1)$ time, resulting in $O(mn)$ time for calculating dominatores, where the resulting lists contain vertices which the given vertex does not dominate. By searching these lists it is possible to calculate new lists, containing which vertices it dominator, with the same time complexity.

By keeping the linked lists in memory the space requirement is $O(n^2)$ since each vertex can visit all others.

It is also possible to calculate a boolean matrix of these n linked list with the same time complexity by filling out a bit matrix instead of pushing to a linked list, this uses $\Theta(n^2)$ space.

Furthermore it is also possible to construct a dominator tree D while calculating dominators in the same time complexity. This can be done by creating n singletons and after each traversal, calculate the vertices $v \in V$ that vertex w dominates. For each vertex w dominates, link v's singleton to w's singleton, if v's singleton already is linked into a tree, this means that the root of the tree w' also dominates v. To figure out if w dominates w' or reversed, ascend the tree until w' is found or until the given vertex is not in w dominator set. Then link w to the forest accordingly. By this all vertices ascended are the vertices w dominates and these vertices are not needed to be linked/ascended again by w. Thus this takes $O(n)$ time for the total ascending for a vertex w. All of this are still dominated by $O(m)$ edges that the algorithm visited in its traversal.

The space consumption is dominated by $O(m+n)$ since we can delete the linked list after each traversal, and then only have to keep track of n vertices which only have one pointer each.

3.3 Semidominators

Semidominators are presented in [LT79], a semidominator for a vertex v in a graph G is an ancestor of v in a DFS tree T of G and is used as an approximation of the immediate dominator of v. Semidominators are used by the algorithms in Section 3.4 and 3.5 to calculate the dominator relations in G. After creating a DFS tree T on a graph and number all vertices in preorder, we can more formally define what a semidominator is.

The semidominator for a vertex v denoted $sdom(v)$ is a semidominator path that has the smallest starting vertex and ends in v. A semidominator path is a path $P = (v_0, v_1, \ldots, v_{k-1}, v_k)$ in T where $v_i > v_k$ for $0 < i < k$ and by this the semidominator for v is:

$$\begin{aligned} sdom(v) = \min\{v_0 \mid &\text{with a path } v_0, v_1, \ldots, v_k = v \\ &\text{where } v_i > v \text{ for } 0 < i < k\} \end{aligned}$$

Lemma 3.3.1 The semidominator for vertex v is an proper ancestor to v in the DFS tree T: $sdom(v) \overset{+}{\rightarrow}_T v$.

By creating a DFS tree, all vertices smaller than v are ancestors to v or else they do not have a path to v. Such a path from one of those vertices would result in v being a child of that vertex when constructing the DFS tree. If the semidominator for vertex v is a predecessor, this predecessor $q = v_0$ has to be smaller than v. If the semidominator is not a predecessor, the semidominator path to the semidominator, contains vertices larger than v and will have a starting vertex v_0 smaller than v. Any vertex larger than v in such a semidominator path can add its parent to the path and so on, until a vertex smaller than v is found. By this the semidominator for v has to be an ancestor of v in the DFS tree.

Calculating semidominators

Input: A directed graph $G = (V, E, r)$.
Output: Semidominator for each vertex in the graph G.
Method: There are a couple of ways to calculate semidominators. One method could be to look at all possible semidominator paths for each vertex v. By looking at v's predecessors, if the predecessor has a smaller number than v this is a semidominator path which also could be the semidominator for v. Otherwise the predecessor is larger than v and the predecessor is bound to have some predecessors of its own, recurrently traverse the graph by all predecessors and by this find the semidominator path which have the smallest starting vertex, this approach has lots of redundancy and is not recommended.

A simple and faster way to calculate semidominators for all vertices, is to calculate the semidominators one at a time in reverse preorder, such that when calculating the semidominator for v, the semidominators for all v' where $v' > v$ have been calculated.

By doing this there are two ways to find semidominator paths for v.

1) Each vertex v have at least one predecessor q, with a lower number than itself, therefore it is easy to find a semidominator path $(q = v_0, v)$ with the smallest value of all predecessors.

2) The trick by calculating each semidominator by reverse pre-order, is that any vertex larger than v, including its predecessors $q > v$, already have found their semidominators. The semidominator path for these predecessors is $(v_0, v_1, ..., v_k = q)$ and since we know $q > v$ and $v_i > q$ for $0 < i < k$, the vertex v can easily be concatenate with the semidominator path for q's semidominator to get a semidominator path $(v_0, v_1, ..., q, v)$ for vertex v. We still need to search all vertices $v' > v$, which have a path to v that satisfies the

definition of semidominator paths. This means the path need to go through one of v's predecessors q and by this we only have to look at vertices that have a path to q. Any vertex v' that has such a path, has to be an ancestor to q and by this we only have to look at the semidominators of the vertices that are ancestors of q.

The semidominator for v, is the semidominator path with the smallest starting vertex found in one of the two methods.

By keeping track of parent and paths, the two methods can be evaluated in the same manner. The way [LT79] do this is to create a new tree identical to the DFS, one vertex at a time in the reverse preorder. They create a singleton for each vertex and initialize the semidominator for those vertices to them self. After the semidominator for a given vertex has been calculated, the singleton for the vertex is linked into a forest, by linking it to its parent vertex from the DFS tree. This way all predecessors that are smaller than v are still singletons and all vertices larger are linked in the forest. To find semidominator paths for each predecessor, an $eval(z)$ method is defined to return the vertex z if z is the root of the tree in a forest. Otherwise it returns the vertex with has the smallest semidominator of the vertices that are ancestors of z and proper descendant of the root in the tree z is in. The method will return q if $q < v$ since q is a singleton. If $q > v$, q's semidominator is compared to all of its ancestors semidominators in the forest and the vertex with the smallest semidominator, which are not the root of the tree is returned. They furthermore optimize the $eval(x)$ method by compressing the trees in the forest. Such that when evaluating a vertex v, all of v's ancestors will from their respective positions, find and keep a pointer to the vertex with the smallest semidominator in the tree. After this, all the vertices on the ancestor path changes their parent relation to the the root of the tree. Because of this next time v or any other vertices that have changed their parent relation in the tree, are being evaluated, they do not need to go through the same ancestors once again to the root and the evaluation is therefor faster.

Pseudo code of this algorithm can be seen in Algorithm 4 and an example can be seen in Figure 3.5.

Termination: For each vertex v the algorithm looks at v's predecessors q, if $q < v$ we notate this and continues. Otherwise if $q > v$ we concatenate its semidominator path with v and save this, then recursively traverse up the tree in the forest, doing the same for every vertex that are proper ancestors of q in the tree. The tree can not have any loops so we can only traverse each vertex once. By this the two method both terminates and will do this for every vertex in the graph.

Correctness: The semidominaters are calculated one vertex at a time in

Algorithm 4 Semidominators

```
1: Create a DFS tree $T$.
2: set $semi(v) = v \mid v \in V$
3: for $v \in V - \{r\}$ in reverse preorder by the DFS do
4:     for $q \in pred_G(w)$ do
5:         $z \leftarrow \text{eval}(q)$
6:         if $semi(z) < semi(v)$ then $semi(v) \leftarrow semi(z)$
7:     end for
8:     Link $w$ and $parent_T(v)$
9: end for
```

reverse preorder, this means the first vertex will not have any predecessors larger than itself and it is easy to see that it will find its semidominator between its predecessors. For all other vertices every vertex larger have found their semidominator. By Lemma 3.3.1 the semidominator for v is an ancestor in the DFS tree and if there is a predecessor q with a path $sdom(v), v_1, ..., q, v$ then $sdom(v)$ is also a ancestor of q. By this it is possible to traverse the path for every predecessor of v and find the semidominator for v. This can be done for every vertex and thus the semidominator for every vertex will be calculated.
Complexity: For each vertex, we search all predecessors for a semidominator path, there are m predecessors in total for all the vertices in the graph. In each search we traverse up the path in a tree while compressing the paths, by thisa forest containing trees which are balanced will give less compression for all other vertices in the tree. Since the height of a balanced tree is $O(\log n)$ the time complexity for calculating semidominators are $O(m \log n)$.

3.4 Lengauer-Tarjan

The Lengauer-Tarjan algorithm [LT79], builds on the depth-first-search observation, that after calculating a DFS tree from a graph, each vertex in the tree has its immediate dominator as an ancestor. It uses this along with the definition of semidominators, which makes immediate dominators estimation for all vertices. By using a simple implementation for finding semidominators, it is possible to find immediate dominators for all vertices in $O(m \log n)$ time.

Lemma 3.4.1 The immediate dominator for v is an ancestor to v's semidominator: $idom(v) \xrightarrow{*}_T sdom(v)$.

Any vertex v has its immediate dominator as a proper ancestor in the DFS tree and all of these vertices have a smaller number than v. According to Lemma 3.3.1 the semidominator of v is also an ancestor to v, where the vertices in the semidominator path are all larger than v except the first and last vertex (v_0 and v). Therefor the immediate dominator cannot be on the path from $sdom(v) \xrightarrow{+} v$, by this the immediate dominator for v is an ancestor to its semidominator.

Lemma 3.4.2 The immediate dominator for v is its semidominator, if there

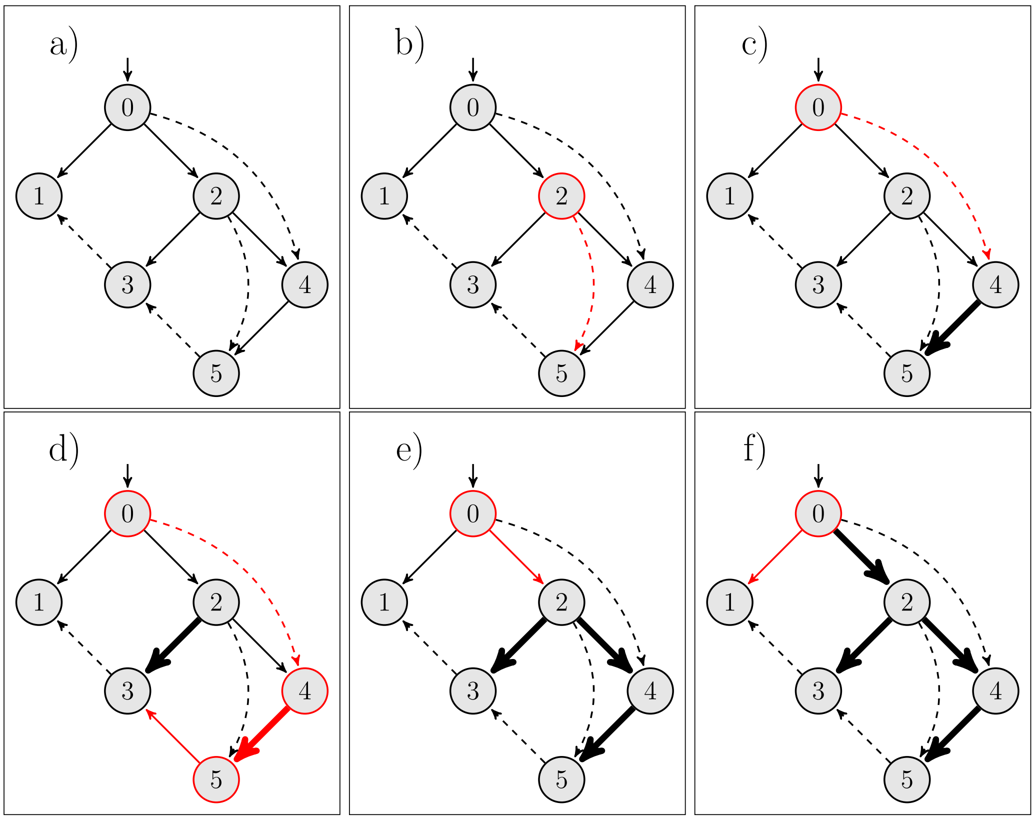


Figure 3.5: Example how to calculate semidominators, red paths is the semidominator path to the semidominator for a given vertex. The bold edges represent the edges between singletons in the forest. a) The input graph. b) The semidominator for the vertex 5 is being found by its smallest predecessor. c) The semidominator is found for vertex 4 by smallest predecessor. d) Vertex 3 finds its semidominator by following the tree in the forest by a predecessors larger than itself. e) Vertex 2 finds it semidominator by smallest predecessor. The semidominator for vertex 1 is finally found by smallest predecessor.

is no vertex a on the path $sdom(v) \xrightarrow{+}_T a \xrightarrow{*}_T v$ which have a smaller semidominator than $sdom(v)$.

If there is a vertex a on the path $sdom(v) \xrightarrow{+}_T a \xrightarrow{*}_T v$ with a semidominator $sdom(a) < sdom(v)$, then there exists a path from $sdom(a) \xrightarrow{*}_T a$ not containing $sdom(v)$, which means $sdom(v) \neq idom(v)$. But if there does not exists such a vertex, which has a smaller semidominator than $sdom(v)$, then there does not exists a path from $r \xrightarrow{*}_T v$ without $sdom(v)$ and by Lemma 3.4.1, $idom(v) = sdom(v)$.

Figure 3.6 contains an illustration of this.

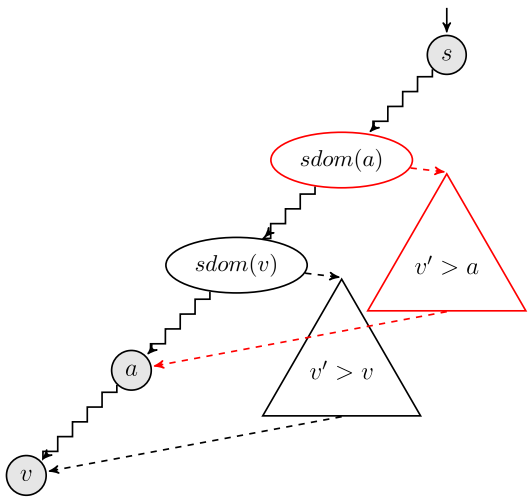


Figure 3.6: An illustration of Lemma 3.4.2. If there does not exists any vertex a, on the path $sdom(v) \xrightarrow{+}_T a \xrightarrow{*}_T v$ which have a semidominator $sdom(a) < sdom(v)$, then there are no path $r \xrightarrow{*}_T v$ without $sdom(v)$.

Lemma 3.4.3 The immediate dominator for v, is the same immediate dominator as it is for a vertex a if it exists, where a is the vertex with the smallest semidominator on the path $sdom(v) \xrightarrow{+}_T a \xrightarrow{*}_T v$ and where $sdom(a) < sdom(v)$.

By choosing the vertex with the smallest semidominator on the path $sdom(v) \xrightarrow{+} a \xrightarrow{*}_T v$ we will find $sdom(a)$ which is a proper ancestor to $sdom(v)$ and which has a path $sdom(a) \xrightarrow{+}_T a \xrightarrow{*}_T v$ not containing $sdom(v)$. Any other vertex b on the path $sdom(v) \xrightarrow{+}_T b \xrightarrow{*}_T v$, which has a semidominator $sdom(a) < sdom(b) \leq sdom(v)$, can not have a immediate dominator which is smaller than a's immediate dominator since this would mean that there exists a path $idom(b) \xrightarrow{+}_T b \xrightarrow{*}_T a$ without $idom(a)$, which is a contradiction. By this v's immediate dominator can not be larger than a's immediate dominator and it can not be lower than $sdom(a)$. Thus $idom(v) = idom(a)$.

Figure 3.7 contains an illustration of this.

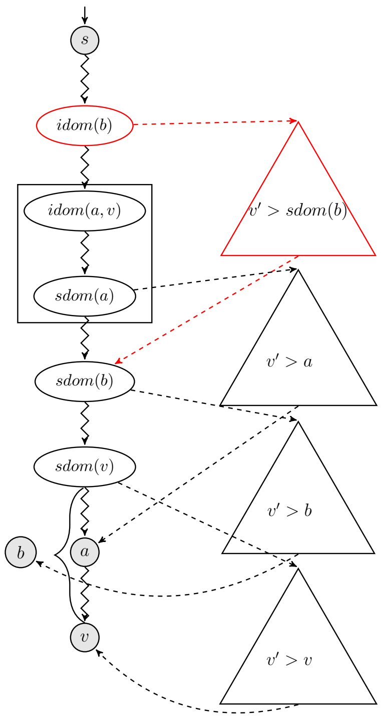


Figure 3.7: An illustration of Lemma 3.4.3.

Input: A directed graph $G = (V, E, r)$.
Output: Immediate dominator relations.
Method: First create a DFS tree and number all vertices in preorder, then calculate semidominators for all vertices in reverse preorder and link them accordently one at a time. After the semidominator for vertex v has been calculated, add v to a bucket associated with its semidominator. Then pop elements from the bucket associated with v's parent and evaluate these vertices in regard to Lemma 3.4.2. Finally iterate over all vertices in preorder and appaly Lemma 3.4.3 to those vertices where $idom(v) \neq sdom(v)$.

A pseudo code implementation can be found in Algorithm 5 and an example on how the algorithm calculates the immediate dominators can be seen in Figure 3.8.

Algorithm 5 Lengauer-Tarjan

```
1: Create a DFS tree $T$.
2: for $w \in V - \{r\}$ in reverse preorder by the DFS do
3:     Calculate semidominator for $w$
4:     Add $w$ to bucket of $semi(w)$
5:     while bucket of $parent(w)$ is not empty do
6:         $v \leftarrow$ pop one element from the bucket
7:         $u \leftarrow eval(v)$
8:         if $semi(u) < semi(v)$ then
9:             $idom(v) \leftarrow u$
10:        else
11:            $idom(v) \leftarrow semi(v)$
12:        end if
13:    end while
14: end for
15: for $w \in V - \{r\}$ in preorder by the DFS do
16:    if $idom(w) \neq semi(w)$ then
17:        $idom(w) \leftarrow idom(idom(w))$
18:    end if
19: end for
```

Termination: The algorithm terminates since we already know that finding semidominators for each vertex does terminate. Traversing up the tree for each vertex to find smaller semidominaters on the path $sdom(v) \xrightarrow{+}_T v$ in Lemma 3.4.2 also terminates since there can not be any cycles on such a path and the path is at most $O(n)$. Finally going through all vertices for applying Lemma 3.4.3 also terminates since we just iterate over all n vertices.
Correctness: The calculation of semidominators in Section 3.3 are correct and by applaying Lemma 3.4.2 and 3.4.3 we will find the immediate dominator for each vertex in the graph.
Complexity: Calculating semidominators for each vertex takes time $O(m \log n)$ applaying Lemma 3.4.2 afterwards is dominated by the semidominator calculation and applaying Lemma 3.4.3 at last can be done in linear time, which is also dominated by the semidominator calculation. Thus the time complexity for calculate dominator relations are $O(m \log n)$. The space consumption is dominated by the graph $O(m + n)$.

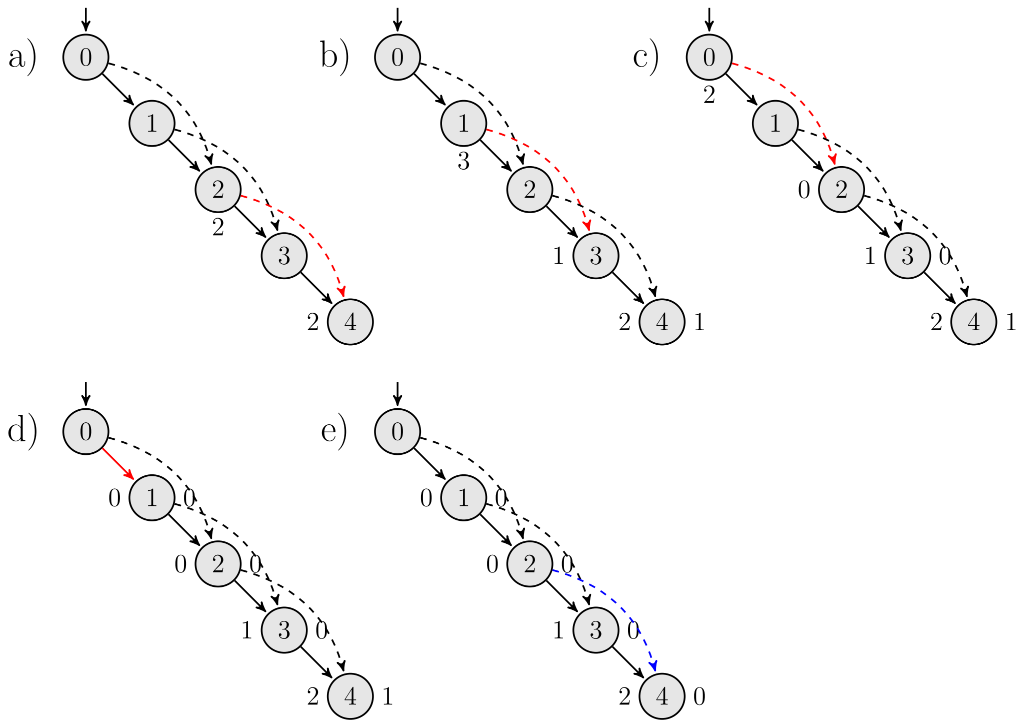


Figure 3.8: Example of how Lengauer-Tarjan algorithm calculates the dominator relation. The semidominator for each vertex is shown to the left of it, its immediate dominator is shown to the right and its bucket is shown below. a) The largest vertex, vertex 4 finds its semidominater $sdom(4) = 2$ and adds itself its semidominators bucket $bucket(2).add(4)$. b) Vertex 3 finds its semidominator and adds itself to $bucket(1)$, after this it empties its parents bucket and finds an approximated immediate dominators for the vertices $\{4\}$, such that $idom(4) = 1$. c) Vertex 2 does the same as in "b)" but with $bucket(0).add(2)$ and $idom(3) = 0$. d) Vertex 1 does the same but with $bucket(0).add(1)$, then it empties this same bucket and finds immediate dominators for $\{1, 2\}$, such that $idom(1) = idom(2) = 0$. d) Finally the semidominator is compared with the immediate dominator for each vertex where vertex 4 is the only that differs, this results in an update of its immediate dominator as $idom(4) = idom(idom(2)) = 0$.

3.5 SEMI-NCA

The SEMI-NCA algorithm was as introduced in [GTW06], it uses semidominaters just as the Lengauer-Tarjan algorithm in Section 3.4, but finds the immediate dominators in a different way. After calculating the semidominator for each vertex, the dominator tree D is being build by traversing the vertices in pre-order and inserting them into D, in the right location. A pseudo code implementation of the algorithm can be found in Algorithm 6 and a figure containing an example of how the algorithm calculates the immediate dominator for all vertices, can be seen in Figure 3.9.

Lemma 3.5.1 For any vertex $v \neq s$, $idom(v)$ is the nearest common ancestor in D for $sdom(v)$ and $parent_T(v)$:

$$idom(v) = NCA_D(parent_T(v), sdom(w))$$

By the DFS observation we know that $idom(v) \leq sdom(v) \leq parent_T(v)$ such that $sdom(v)$ and $parent_T(v)$ both are dominated by $idom(v)$.

If $idom(v) = sdom(v)$ clearly $parent_T(v)$ is dominated by $sdom(v)$ and that a nearest common ancestor search in D for $parent_T(v)$ and $sdom(v)$ would return $idom(v)$. If $idom(v) \neq sdom(v)$ then by Lemma 3.4.2, $parent_T(v)$ can not be dominated by $sdom(v)$, since there have to exists a path not containing $sdom(v)$ to v, which goes through a vertex a on the path $sdom(v) \overset{+}{\rightarrow}_T a \overset{*}{\rightarrow}_T v$. A nearest common ancestor search in D will return a vertex u that dominates $sdom(v)$ and $parent_T(v)$. $idom(v)$ can not be on the path $r \overset{*}{\rightarrow}_D idom(v) \overset{+}{\rightarrow}_D u$ this would indicate that there is a path to v without $sdom(v)$ or $parent_T(v)$. $idom(v)$ can not be on the paths $x \overset{+}{\rightarrow}_D idom(v) \overset{*}{\rightarrow}_D sdom(v)$ or $u \overset{+}{\rightarrow}_D idom(v) \overset{*}{\rightarrow}_D parent_T(v)$ since it would not dominate $sdom(v)$ or $parent_T(v)$, by this $idom(v) = u$.

Lemma 3.5.2 Any vertex y in a dominator tree D, on the path $idom(v) \overset{*}{\rightarrow}_D y \overset{+}{\rightarrow}_D parent_T(v)$ is larger or equal to $sdom(v)$

All vertices on the path $r \overset{*}{\rightarrow}_D idom(v)$ dominates $sdom(v)$ and $parent_T(v)$. Any vertex z on the path $idom(v) \overset{+}{\rightarrow}_T z \overset{*}{\rightarrow}_T sdom(v)$ which dominate $sdom(v)$ also dominates $parent_T(v)$. Any vertex a on the path $sdom(v) \overset{*}{\rightarrow}_T a \overset{*}{\rightarrow}_T p_T(v)$ can not dominate $sdom(v)$ but can dominate $parent_T(v)$. By this any vertex y on the path $idom(v) \overset{*}{\rightarrow}_D y \overset{*}{\rightarrow}_D p_T(v)$ have to be larger than $sdom(v)$.

Input: A directed graph $G = (V, E, r)$.
Output: Dominator tree.

Method: First create a DFS tree T and number all vertices in preorder, then calculate all the semidominator for all vertices. Create a singleton D with the root vertex of the graph. Then for each vertex v, find the nearest common ancestor between $sdom(v)$ and $parent_T(v)$ by ascending the path $r \overset{*}{\dashrightarrow}_D parent_T(v)$ and report the first vertex x with $x \leq sdom(v)$. Add v to the dominator tree D with a as its parent. After all vertices have been traversed return the dominator tree D.
Termination: By Section 3.3 the calculation of semidominators terminates. Finding nearest common ancestors between $sdom(v)$ and $parent_T(v)$ in D for each vertex, will also terminate since the ascending path for $r \overset{\cdot}{\dashrightarrow} p_T(v)$ has a finite length.
Correctness: By Lemma 3.5.1 the immediate dominator for a vertex v is the nearest common ancestor for $sdom(v)$ and $parent_T(v)$, by Lemma 3.5.2 this vertex can be found by ascending the path $r \overset{\cdot}{\dashrightarrow}_D parent_T(v)$, by this the immediate dominator for each vertex is going to be found.
Complexity: Calculating semidominators is done in $O(m \log(n))$ time, this is dominated by the time ascending $r \overset{*}{\dashrightarrow}_D parent_T(v)$ for each vertex which yields a total time of $O(n^2)$. The space consumption is dominated by the graph G which is $O(m+n)$.

Algorithm 6 Semi-NCA

```
1: Create a DFS tree $T$.
2: Calculate semidominator for $w$
3: Create a tree $D$ and initialize it with $r$ as the root.
4: for $w \in V - \{r\}$ in preorder by the DFS do
5:     Ascend the path $r \overset{*}{\dashrightarrow}_D parent_T(v)$ and find the deepest vertex which
       number is smaller than or equal to $sdom(v)$. set this vertex as parent for
       $v$ in $D$.
6: end for
```

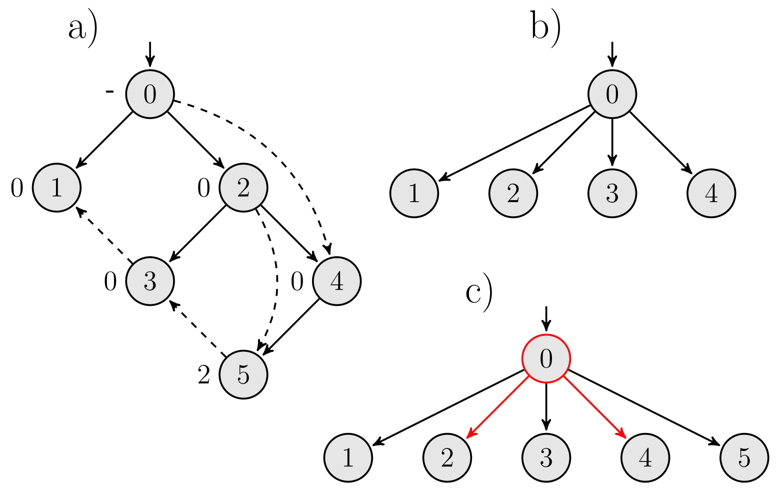


Figure 3.9: Example how SEMI-NCA calculates dominators in a graph. a) Shows the input graph where the semidominators have been calculated and is displayed to the left for each vertex. b) Contain the dominator tree D after vertex $1, 2, 3$ and 4 have been found trivially, since their semidominators are the root. c) Contain the dominator tree after the last vertex have been inserted, the parent$_D$ for vertex 5 have been found by a nearest common ancestor search, between its semidominator and its parent$_T$ in D, which is shown by red edges.

4 Experiments

In this chapter we will take a look at the execution of the five algorithms from Chapter 3. The algorithms have been implemented in `c++` and compiled on Ubuntu 14.04 with `g++` version 4.8.4 with -O2 flag. All test ran on an intel-i7-3632QM 2.2GHz, which has 32kb L1 cache, 256kb L2 cache and 6mb L3 cache.

We will see how the algorithms performs when they encounter their worst-case inputs and if their theoretical complexities holds. To do this, we will first look a how their worst-case input graphs are constructed, which is a part of Section 4.2. In Section 4.3 we will then look at the results of the worst-case tests. Finally we will in Section 4.4 see how each algorithm compares to the others by running them all on the different graph families from Section 4.2.

In each run time experiment, we will choose the lowest running time from a number of runs, instead of average as it is often done. This makes sense since comparing deterministic algorithms, external factors can only increase the running time [CHK01, p. 10f].

4.1 Implementation

To better understand the run time of the different algorithms some implementation details might be in order.

The graph. I have implemented a single graph such that all algorithms get the same type of input. Each algorithm then creates a secondary structures that it needs. This could be a structure for holding the dominator relations, such as a matrix or it could be a structure needed to calculate the order the vertices shall be visited in. The graph have been implemented such that it has an array containing all vertices in the graph. Each vertex has two unsigned integers to keep track of the number of predecessors and successors. The vertices then keep track of their edges by two linked list one for predecessors and one for successors. This way it does not matter if an algorithm needs predecessors or successors and it does not need to calculate them separately. By this each vertex uses 20bytes and each edge uses 8bytes.

Iterative boolean matrix. The iterative boolean algorithm has a secondary structure for storing the dominator relation between each vertex and to calculate a traversal ordering. All in all it uses $(n^2 + 5n)$bytes extra for all vertices, where n is the number of vertices.

Iterative tree. The iterative tree algorithm has a secondary structure for the spanning tree. It contains a parent pointer, a depth value and a boolean. Furthermore it keeps track of this secondary structure using an array. By this it uses 18bytes extra pr. vertex. I have implemented two types of ordering and from here on out we will refer to the iterative tree algorithm that uses a reverse post-order as iDFS and we will refer to the iterative tree algorithm that uses a BFS as iBFS.

Vertex removal. I have implemented the vertex removal algorithm such that it uses a matrix to keep track of the dominator relations. This results in a total of $(n^2 + 8n)$bytes extra for all vertices, n is the number of vertices.

Simple Lengauer-Tarjan In my implementation the semidominator calculation for each vertex uses 25bytes for each vertex plus $O(n)$ in the buckets, furthermore with secondary structures the entire representation is 30bytes for each vertex.

SEMI-NCA. The SEMI-NCA algorithm have been implemented such that it uses six arrays to keep track of semidominators, dominator tree, best semidominators, DFS-tree, graph-to-dfs-value and dfs-value-to-graph. In total 24bytes bytes extra for each vertex.

4.2 Graph Families

We will go through different families of graphs which evokes the worst-case behavior on the five algorithms used to calculating dominators in Chapter 3. Together with graph families that favors some algorithm over the others. The construction of `itworst(k)`, `sltworst(k)`, `idfsquad(k)`, `idfsquad(k)` and `sncaworst(k)` graph families are from [GTW06] and the fifth graph family is my own. An example of the graph families can be found in Figure 4.1.

Iterative. `itworst(k)` is the graph family with worst-case input for the iterative algorithms. It is constructed by $|V_k| = 4k + 1$ vertices and $|E_k| = k^2 + 5k$ edges. The vertices are $\{r\}$ and $\{w_i, x_i, y_i, z_i \mid 1 \leq i \leq k\}$.

The edges is the union of:

$$\{(r, w_1), (r, x_1), (r, z_k\}, \{w_i, w_{i+1}), (x_i, x_{i+1}), (y_i, y_{i+1}), (z_i, z_{i+1}) \mid 1 \leq i \leq k\},$$

$$\{(z_i, z_{i-1}) \mid 1 < i \leq k\}, \{(x_m y_1), (y_k, z_1)\}, \text{ and } \{(y_1, w_j) \mid 1 \leq i, k \leq k\}$$

Simple Lengauer-Tarjan. `sltworst(k)` is the graph family with worst-case input for the Lengauer-Tarjan with a simple linking and evaluation implementation. It is constructed by $|V_k| = k$ vertices and $|E_k| = 2k - 2$ edges. The edges is in the graph are constructed of to sets, the first one is of $k - 1$ edges as $(x_i, x_{i+1}), 1 \leq i < k$. The Second set is also of $k - 1$ edges (x_i, x_j) edges where $j < i$, with property that x_i will be at the maximum depth in a tree rooted at x_j after x_j is linked.

IDFS. `idfsquad(k)` denotes the family which favors IBFS over IDFS. The family is constructed of $|V_k| = 3k + 1$ vertices and $|E_k| = 5k$ edges. The vertices are $\{r\}$ and $\{x_i, y_i, z_i \mid 1 \leq i \leq k\}$. The set of edges is the union of:

$$\{(r, x_1), (r, z_1)\}, \{(x_i, x_{i+1}), (y_i, z_{i+1}) \mid 1 \leq i < k\}$$

$$\text{and } \{(x_i, y_i), (y_i, z_i), (z_i, y_i) \mid 1 \leq i \; k\}$$

IBFS. `idfsquad(k)` is the graph family which favors IDFS over IBFS and is constructed by $|V_k| = k + 4$ vertices and $|E_k| = 2k + 3$ edges. The vertices are $\{r, w, y, z\}$ and $\{x_i \mid 1 \leq i \leq k\}$ and the edges is the union of

$$\{(r, w), (r, y), (y, z), (x_k)\} \text{ and}$$

$$\{(w, x_i) \mid 1 \leq i \; k\} \text{ and } \{(x_i, x_{i-1}) \mid 1 < i \leq k\}$$

SEMI-NCA. `sncaworst(k)` is the graph family with worst-case input for the SEMI-NCA algorithm. The graph is constructed of $|V_k| = 2k + 1$ vertices and $|E_k| = 3k$ edges. The vertices are $\{r, x_i, y_i \mid 1 \leq i \leq k\}$ Where the edges is the union of:

$$\{(r, x_1)\}, \{(x_i, x_{i+1}) \mid 1 \leq 1 < k\} \text{ and } \{(r, y_i), (x_k, y_i) \mid 1 \leq i \leq k\}$$

Vertex removal. `vrworst(k)` is the worst case input to the vertex removal algorithm. The graph is constructed of $|V_k| = k$ vertices and $|E_k| = k(k-1)$ edges. The vertices are $\{r = x_1\}$ and $\{x_i \mid 1 \leq i \leq k\}$ and the edges are:

$$\{(x_i, x_j) \mid 1 \leq i, j \leq k \text{ where } i \neq j\}$$

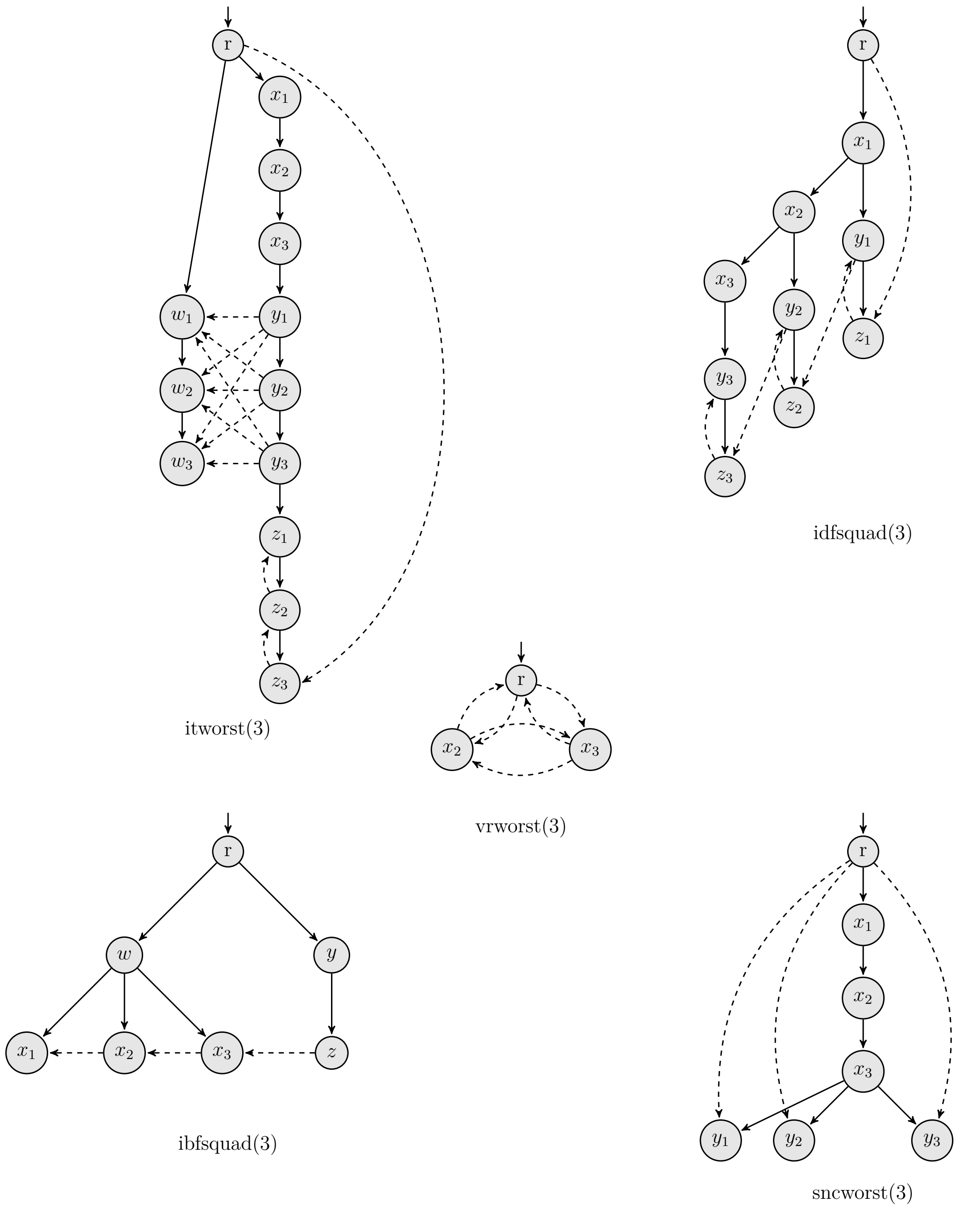


Figure 4.1: `itworst(k)`, `idfsquad(k)`, `idfsquad(k)` and `sncaworst(k)` are from [GTW06]. The solid edges are the DFS tree used by the individual algorithms.

4.3 Worst-case Testing

For each test the algorithm has been given its worst-case graph as input. The resulting execution time along with variable regulating the size complexity of the graph, have been plotted for each algorithm. Such that the y-axis is the execution time divided by the theoretical complexity of the execution. For example, a worst-case graph with $|V_k| = 4k + 1$ and $|E_k| = k^2 + 5k$ given to a algorithm with $O(mn^2)$ complexity, will have a theoretical execution time of $\Theta(k^4)$. All algorithms in each test have been executed 10 times and the minimum execution time for these runs have been plotted.

Vertex removal.
The vertex removal algorithm from Section 3 has been given the `vrworst(`k`)` graph family, which means the theoretical execution takes $\Theta(k^3)$. In Figure 4.2 the plot of this execution is shown. From 0 to around 50 the algorithm terminates very fast, this could be due to the L1 cashe. From there the graph is roughly constant until a jump at around 250 which corresponds very well with the L2 cashe, since algorithm at this point would be using about 350kb. From 500 and forward the graph are roughly constant again. Thus showing the the worst-case time.

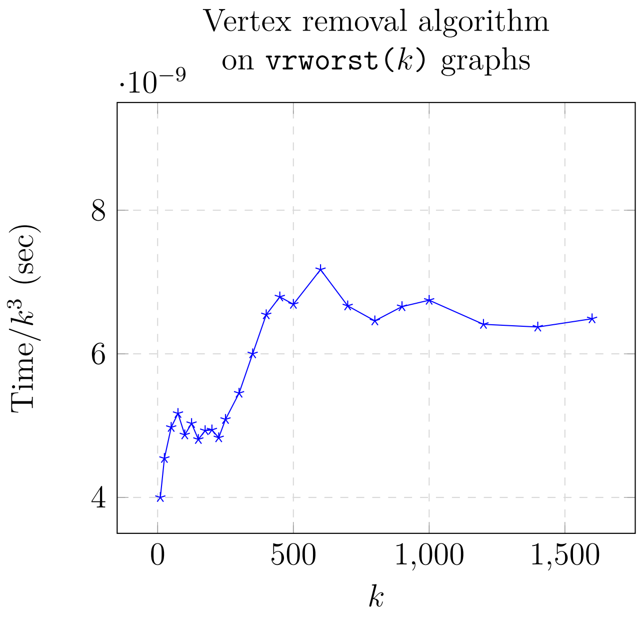


Figure 4.2: Run time experiment for vertex removal with `vrworst(`k`)` graphs as input.

Iterativ, Boolean vector.

The iterative boolean algorithm from Section 3.2 has been given the `itworst(`k`)` graph family, which means the theoretical execution takes $\Theta(k^4)$. In Figure 4.3 the plot of this execution is shown. It can be seen that the execution time starts very high, then drops and continues in a constant fashion. This drop could be explained by the allocation of the boolean vectors, which has a higher correspondence for each vertex when k is low than when it is higher, due to the $k^2 - 1$ edges vs k vertices. The graph continues her after in a constant fashion. Thus showing the worst-case time.

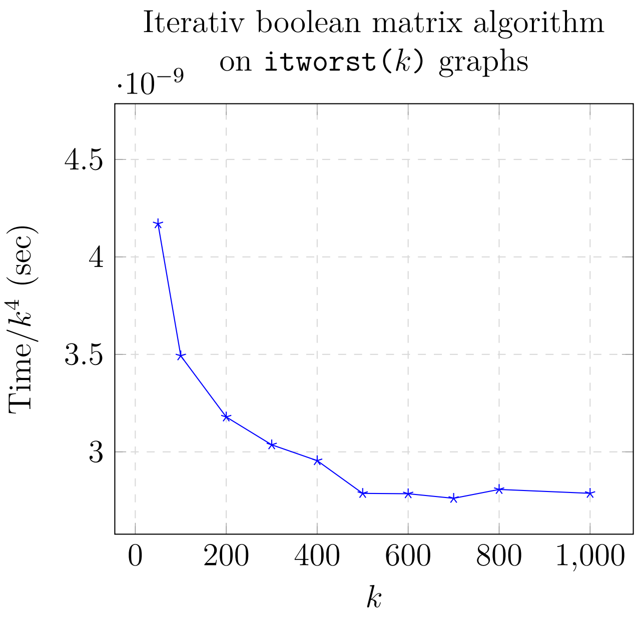


Figure 4.3: Run time experiment for iterative boolean matrix algorithm with `itworst(`k`)` graphs as input.

IDFS

The iterative algorithm from Section 3.1.2. The algorithm has been given the `itworst(`k`)` graph family, which means the theoretical execution takes $\Theta(k^4)$. In Figure 4.4 the plot of this execution is shown. Where it can be seen that the execution time starts very low and increases at around $k = 50$ to an almost constant function. The jump from 0 to 50 in the figure, could be explained by the L1 cache since $k = 50$ uses about 66kb. After this is continues with a small increase and when $k = 100$ is uses about 243kb, which could explain the small slope in the graph from 100 to 300. After 300 the algorithm on this graph took way to long to further experiment with. Due to the, very small increasement, the graph indicates a worst-case time.

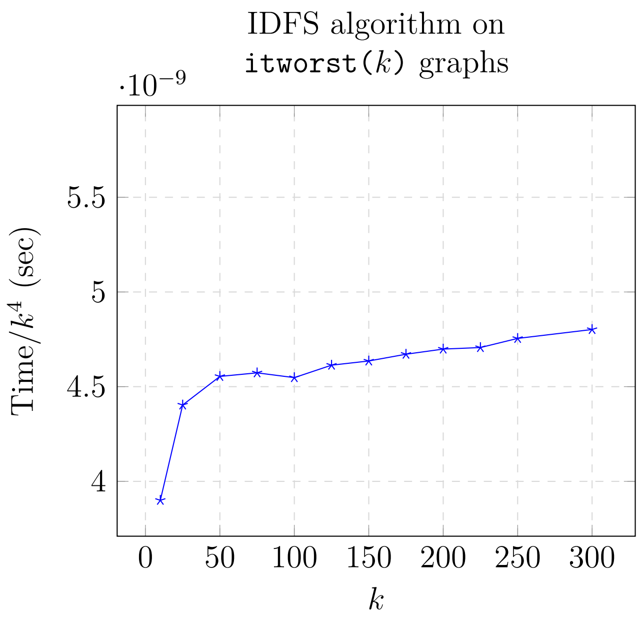


Figure 4.4: Run time experiment for iterative tree matrix algorithm using reverse post order, with the `itworst(`k`)` graphs as input.

IBFS
Same patterns as the IDFS. thus the graph indicates a worst-case time.
SEMI-NCA
The SEMI-NCA algorithm from Section 3.5, has been given the `scaworst(`k`)` graph family, which means the theoretical execution takes $\Theta(k^2)$. In Figure 4.6 the plot of this execution is shown.

I have no good explanation of the almost vertical slope in the start of the graph, I have tried to run the experiment a couple of times, but it always starts like this. But from 5000 and forward the graph seems constant. Thus indicating a worst-case time.
Lengaur-Tarjan
The Lengauer-Tarjan algorithm from Section 3.5, has been given the `sltworst(`k`)` graph family, which means the theoretical execution takes $\Theta(k \log k)$. In Figure 4.7 the plot of this execution is shown.

Just as the SEMI-NCA algorithm this graph start out with a steep slope that I can not explain. At around $k = 0.08\dot{1}0^5$, which is about 500kb, the graph starts to increase and at $k = 0.32\dot{1}0^5$ which corresponds to about 2mb it starts to be a constant graph again. This jump could be due to the L2 cashe and maybe cashe misses, which I have not measured. Besides of this, the graph indicating a worst-case time.

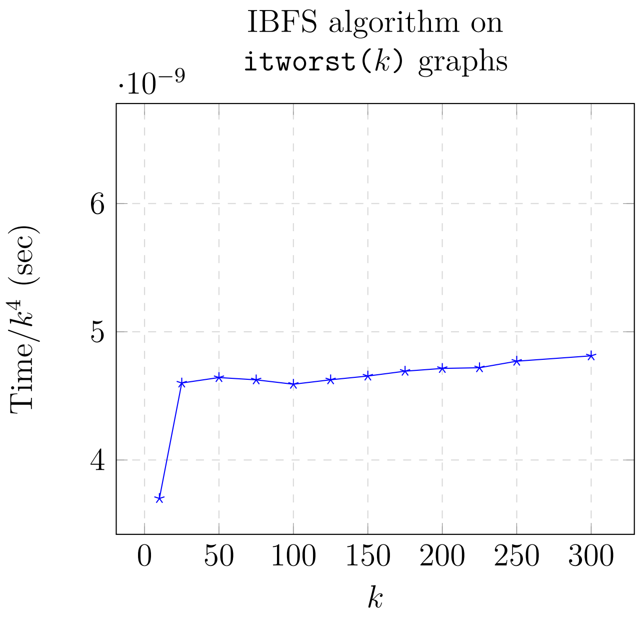


Figure 4.5: Run time experiment for iterative tree matrix algorithm using lever-ordering, with the `itworst(`k`)` graphs as input.

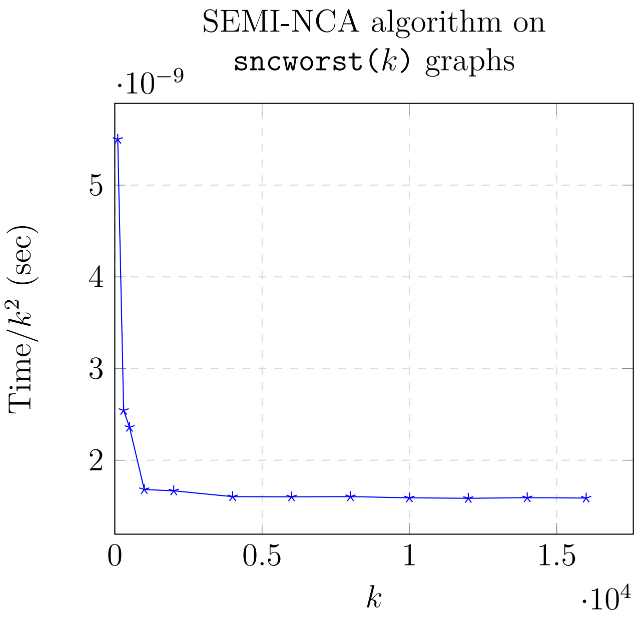


Figure 4.6: Run time experiment for SEMI-NCA algorithm, with the `scaworst(`k`)` graphs as input.

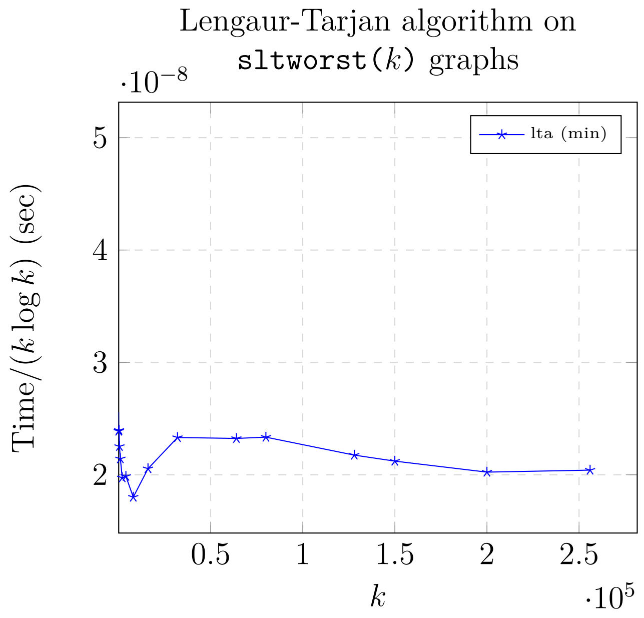


Figure 4.7: Caption

4.4 Comparison of the Algorithms

The complexity alone does not always give the full picture of how algorithms compare in practice. I have therefore executed all the algorithms on `itworst(k)`, `sltworst(k)`, `idfsquad(k)`, `idfsquad(k)`, `sncaworst(k)`, `vrworst(k)` and plotted the resulting execution time for each algorithm. By doing this, it is possible to see how the algorithms perform with respect to each other, on these types of graphs. We are going to see that some algorithms finds the dominator relations faster than all the other algorithms in all of these graph families. Furthermore we are also going to see that some algorithms, depending on the graph families given as input, dominates each other.

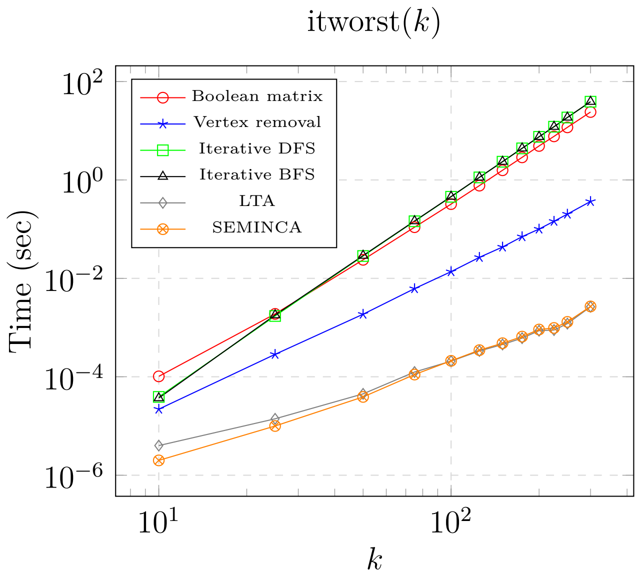


Figure 4.8: Comparson of the six algorithms on `itworst(k)`.

itworst(k). When comparing the six algorithm on `itworst(k)` graph family, which can be seen in Figure 4.8. It can be see that some of the algorithms are faster than others, in fact all the iterative algorithm are clustered together and takes more time than any of the other, which makes sense since this is the worst-case graph for iterative algorithm. Both algorithm that uses semidominators are also clustered together and runs much faster than any other algorithms. The Vertex removal algorithms is laying between the other two types of algorithms.

dfsquad(k). `dfsquad(k)` is a graph family that should favor IBFS over

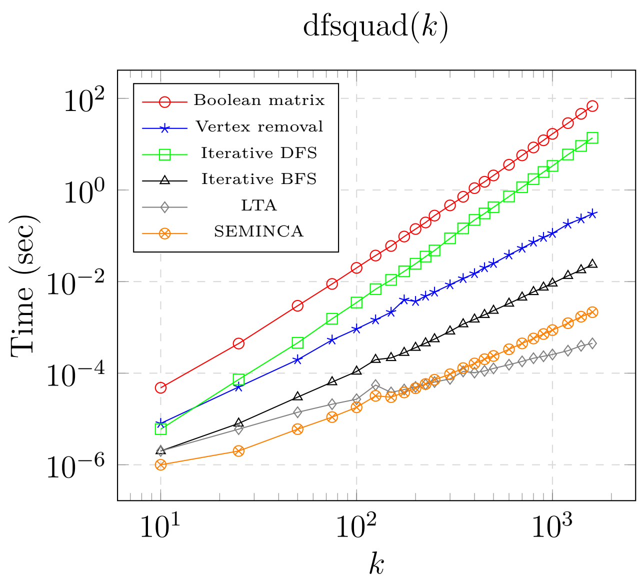


Figure 4.9: Comparson of the six algorithms on `dfsquad(k)`.

IDFS and as it can be seen in Figure 4.9 it does so. This is due to the reverse post-order the IDFS have to go through, where the vertices have been placed in such a manner that it has to make $(k + 1)$ iterations over the graphs to find the dominator relations [GTW06]. IBFS on the other hand, calculates the dominators in the first iteration.

bfsquad(k). `bfsquad(k)` is a graph family that should favor IDFS over IBFS and as it can be seen in Figure 4.10 it does so. This is due to the level-order the IBFS have to go through, where the vertices have been placed in such a manner that it has to make $O(k)$ iterations over the graphs to find the dominator relations [GTW06]. IDFS on the other hand, finds the dominators in the first iteration because of the reverse post-order.

vrworst(k). `vrworst(k)` is a full connected graph, where every vertex have to visit all other vertices in the graph in on way or another, no matter which algorithm is used. In Figure 4.11 it can be seen that the IBFS is now the fastest algorithm along with the two algorithms using semidominators.

sncaworst(k). In Figure 4.12 the comparison for the six algorithms on the `sncaworst(`k`)` graph family is shown. As it can be seen the Lengaur-Tarjan algorithm calculates the dominator relations much faster than any of the other, but the two semidominator algorithms are still the once fastest compared to all the other types.

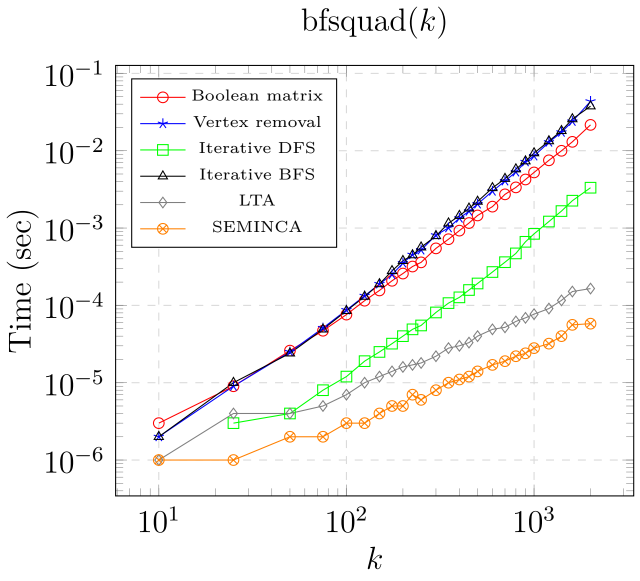


Figure 4.10: Comparson of the six algorithms on `bfsquad(k)`.

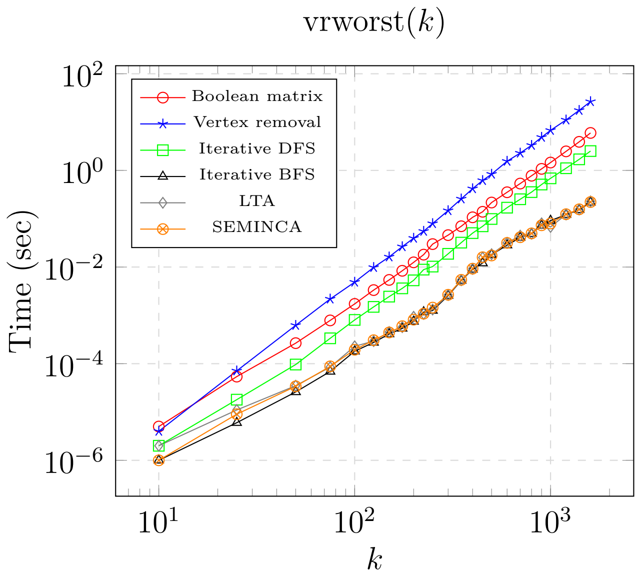


Figure 4.11: Comparson of the six algorithms on `vrworst(k)`.

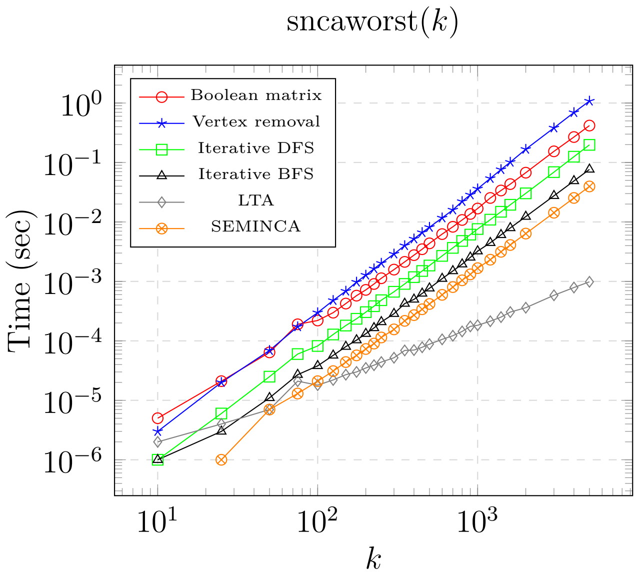


Figure 4.12: Comparson of the six algorithms on `sncaworst(k)`.

Result

Depending on the type of graphs that the algorithms get as input, all the algorithms can find the dominator relation faster than some of the other algorithms. In all the comparison, the algorithms using semidominators is always the fastest. Even though they are the fastest, one is not always faster than the other, as it can be seen in Figure 4.10 and Figure 4.12. Furthermore we saw in Figure 4.9 and Figure 4.10 that IBFS and IDFS are not always faster than the other one either.

5 Conclusion

In this thesis we have looked at the dominator relations between vertices in directed graphs and how to find them using five different algorithms. We have covered the basics about graphs, trees and traversal. This was used to get an understanding of dominator relations and how to represent all of these relations in different data structures.

We have for each algorithm covered complexity, termination and correctness, along with insight and examples on each algorithm inner workings.

All the algorithms have been implemented and different tests have been conducted. The worst-case testing gave an indication that each algorithm performed as expected. It was seen that the semidominator algorithms outperform all others, when testing all the algorithms against each other, on the graph families that we had available.

A Appendix

A.1 Notation

This appendix contain a list of the symbols used in Chapter 1, 2 and 3 used to describe structures and relations.

Symbol	Meaning
G	a graph $G = (V, E)$ or $G = (V, E, r)$.
T	a tree.
D	a dominator tree.
V	the set of vertices in a graph.
E	the set of edges in a graph or a tree.
r	starting vertex in a graph G or the root in a tree T.
n or $\|V\|$	the number of vertices in a set V.
m or $\|E\|$	the number of edges in a set E.
v	v is a vertex in a graph or tree.
w	w is a vertex that dominates v.
u	u is a vertex that is the immediate dominator for v.
x	x is a vertex that is not a dominator for v.
p	p is a parent in a tree.
q	q is a predecessor in a graph.
a	a is an ancestor in a tree.
$dom(v)$	$dom(v)$ is the set of vertices that dominates v
$idom(v)$	$idom(v)$ is the immediate dominator for v
$sdom(v)$	$sdom(v)$ is the semidominator for v
$v \overset{\cdot}{\rightarrow}_G y$	a path from v to y in G
$v \overset{+}{\rightarrow}_G y$	a path from v to y in G where $v \neq y$

Bibliography

[AC72] Frances E. Allen and John Cocke. Graph-theoretic constructs for program control flow analysis, 1972. [pp. 1-22].

[All70] Frances E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970. [pp. 1-11].

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques.* Addison wesley, 1986. [pp. 670-671].

[AU72] Alfred V. Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling.* Prentice-Hall, Inc., 1972. [p. 916].

[CHK01] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001. [pp. 1-8].

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms (second edition), 2001. [sec. 22.3].

[GTW06] Loukas Georgiadis, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Finding dominators in practice. *J. Graph Algorithms Appl.*, 10(1):69–94, 2006. [pp. 69-81].

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976. [p. 159].

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979. [pp. 121-131].

[Pro59] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959. [p. 136].