A NEW APPROACH TO FINDING DOMINATORS IN STRUCTURED CONTROL FLOW GRAPHS

Davorka Jandrlić[1]

Abstract: This paper presents a novel algorithm for identifying dominator nodes in Single-Entry Single-Exit (SESE) directed graphs, focusing specifically on the identification of dominators for a single node, a critical problem in graph analysis. Traditional algorithms compute dominators for all nodes in the graph, but our approach targets the efficient computation of dominators for a single node in linear time, offering a significant improvement in scenarios where dominator information for only one node is required. The proposed algorithm is rigorously analyzed for its time complexity, correctness, and termination. A comparative evaluation with existing comprehensive algorithms is provided, highlighting the performance and advantages of the new approach. The implementation and experimental results confirm the algorithm's efficiency and suitability for practical applications.

Key words: Graph algorithms, Single node dominator, Graph traversal, Algorithm design

1. INTRODUCTION

A directed graph is a mathematical structure that represents a set of objects, known as vertices or nodes, connected by directed edges. Each edge in a directed graph has a defined direction, originating from one node (the source) and terminating at another (the target). Directed graphs are widely used in various fields to model relationships, processes, and systems where the direction of interaction or dependency is important, such as in computer science, transportation networks, biology, and social networks. Formally, a directed graph is defined as $G = (V, E)$, where V is the set of vertices and $E \subseteq V\ x\ V$ is the set of directed edges. Furthermore, directed graphs, also known as digraphs, serve as the foundation for more advanced structures in scheduling, dependency resolution, and compiler design, and Single-Entry Single-Exit (SESE) graphs employed in control flow analysis.

In this study, we explore directed graphs in the context of identifying dominators—nodes that must be visited on every path between a given source and target. This problem plays a fundamental role in control flow analysis, compiler optimization, and network reliability, making directed graphs a critical focus of our investigation. Control flow analysis is a fundamental component of modern compiler design and has been extensively studied and documented in numerous research papers [1]-[3]. It plays a critical role in optimizing program execution by analyzing the flow of control between different blocks of code, enabling techniques such as dead code elimination, loop optimization, and efficient resource allocation. The concept of the dominator relation in graph theory was first introduced by Frances E. Allen in 1970 as part of her pioneering work on control flow analysis for optimizing compilers [4]. A node u in a directed graph is said to dominate another node v if every path from the designated entry (start) node to v must pass through u. This relation forms the foundation of dominator trees, a hierarchical structure used to efficiently represent dominance relationships within control flow graphs (CFGs). The immediate dominator of a vertex v in a directed graph is the vertex u that satisfies two key properties: u dominates v, and u is itself dominated by all other dominators of v, excluding v itself. In essence, u is the closest dominator of v along every path in the graph. Put simply, u is the last vertex that every path in the graph must pass through before reaching v.

We will now discuss some of the existing and widely used algorithms, as presented in Table 1. Additionally, we will examine the underlying methodologies, along with the advantages and disadvantages of these algorithms. The space complexity of a directed graph refers to the memory required to store the graph. It depends on the representation method used and the number of elements, specifically vertices (V) and edges (E). In graph theory, vertices represent the nodes, while edges represent the directed connections between these nodes. The space complexity is ultimately determined by how these vertices and edges are organized and stored in the computer's memory.

[1]Assoc.Prof. Davorka Jandrlić, Faculty of Mechanical Engineering, Department of mathematics, University of Belgrade, Serbia, email:djandrlic@mas.bg.ac.rs

While space complexity measures memory usage, time complexity evaluates how long the algorithm takes to execute, depending on the input size.

Table 1 – Some existing algorithms for finding dominators

Algorithm - name	Time Complexity	Space Complexity
Iterative	$O(\|E\|\|V\|^2)$	$O(\|V\| + \|E\|)$
Vertex removal	$O(\|E\|\|V\|)$	$O(\|V\|^2)$
Lengauer-Tarjan	$O(\|E\|log\|V\|)$	$O(\|V\| + \|E\|)$

1.1. Iterative algorithms

Allen and Cooke proposed an iterative algorithm in [5], which is based on the following idea: dominators of a node are determined iteratively by refining the set of possible dominators for each node. Initially, all nodes are considered dominators for every other node, except for the start node, which dominates itself. During each iteration, the algorithm eliminates non-dominators by intersecting the dominator sets of predecessor nodes. This process is repeated until no further changes occur, ensuring convergence. The algorithm effectively leverages set operations to identify dominators, making it both simple to implement and conceptually clear. However, its iterative nature can make it less efficient for very large graphs, as each iteration involves multiple set operations.

In their work, Allen and Cocke used a boolean vector to represent the set of vertices that dominate a given vertex v. This set is encoded as a boolean vector of size $|V|$, where each entry w in the vector corresponds to a vertex. The value of the entry is true if vertex w dominates v; otherwise, it is false, indicating that w does not dominate v.

The space complexity of the algorithm is dominated by the $|V|$ boolean vectors, each of size $|V|$. As a result, the overall space complexity $is\ O(|V|^2)$, where $|V|$ is the number of vertices in the graph.

In each iteration, the algorithm iterates through all vertices. For each vertex, it calculates the intersection of the dominator sets of all its predecessors. Since there are $|E|$ edges (or predecessors in the graph), this results in $|E|$ intersection calculations in total. Each dominator set is represented by a boolean vector of size $|V|$, so the time complexity for calculating all the intersections in each iteration is $O(|E||V|)$.

There are modifications to this algorithm, that can reduce space complexity and improve runtime, such as the Cooper-Harvey-Kennedy algorithm [6]. However, it may not be the most efficient choice for scenarios that require dominator information for only a single node, as it computes dominators for all nodes in the graph.

1.2. Vertex removal

Perhaps the simplest approach to finding dominators is described in [7]. This method relies on the observation that if a vertex w is removed from a graph G and it was a dominator of v, then no paths will remain from the root r to v. This is because all paths from r to v must pass through w, otherwise, w would not satisfy the condition of being a dominator for v.

The algorithm processes $|V|$ vertices, and for each vertex, the graph G', refers to the graph obtained by removing a single node w from the original graph G, is traversed by following all reachable edges. Since there are $|E|$ edges in the graph, this traversal takes $O(|E||V|)$ time. Additionally, dominator relations are tracked by maintaining a linked list of visited vertices for each vertex v. Adding a new vertex to the list upon its first visit takes $O(1)$ time. Consequently, the total time complexity for calculating dominators is $O(|E||V|)$. These linked lists, which initially contain vertices that a given vertex does not dominate, can be processed further to compute new lists of vertices dominated by the given vertex. This step also maintains the $O(|E||V|)$ time complexity.

The space requirement is $O(|V|^2)$, as each vertex can potentially visit all other vertices, leading to $|V|$ linked lists, each containing up to $|V|$ vertices.

1.3. Lengauer-Tarjan

The Lengauer-Tarjan algorithm is a highly efficient algorithm for finding dominators in a flow graph [8], and is widely used in compiler optimizations. It operates in $O(|E|log|V|)$ time, where $|V|$ is the number of vertices, $|E|$ is the number of edges, it is near-linear for practical inputs. The process begins with a depth-first search traversal of the graph starting from the root node. During this traversal, each node is assigned a preorder index, and parent-child relationships are recorded. These relationships are critical for defining the structure of the graph and understanding how nodes are connected.

The algorithm then computes a "semi-dominator" for each node. The semi-dominator of a node represents the nearest dominator in terms of the Depth First Search (DFS) order, which provides a preliminary approximation of the actual dominator relationships. This step involves examining paths and relationships between nodes as identified in the DFS traversal.

To efficiently compute and update these semi-dominators, the algorithm uses a union-find data structure with path compression. This structure allows for efficient merging and querying of node sets, ensuring that operations on the graph remain computationally efficient.

Finally, the algorithm calculates the actual dominators for each node based on the computed semi-dominators. This involves iterating through the nodes in reverse DFS order and refining the dominator relationships. The output of this step is the complete set of dominator information, often represented as a dominance tree, where each node is dominated by its ancestors in the tree.

The Lengauer-Tarjan algorithm's efficiency arises from its careful combination of DFS traversal, semi-dominator approximation, and the use of union-find operations. This minimizes redundant computations and ensures that the algorithm can handle large graphs with many nodes and edges effectively. Its ability to work efficiently in practical scenarios has made it a foundational algorithm in the field of graph analysis and program optimization.

Detailed steps of this algorithm will not be presented here, as, despite its efficiency, it remains one of the most complex algorithms for computing dominators.

We have explained some widely used traditional comprehensive algorithms; however, while these algorithms are thorough and reliable, they may not be the most efficient choice for scenarios that require dominator information for a single specific node. Therefore, alternative methods or modifications to existing algorithms may be necessary to address this particular need. This is why we developed our own algorithm designed specifically to find the list of dominators for a given node in a graph.

2. PROPOSED ALGORITHM

2.1. SmartDominatorFinder

The proposed algorithm, SmartDominatorFinder, efficiently identifies dominators in Single-Entry Single-Exit (SESE) directed graphs. It operates in linear time, with a time complexity of $O(|V| + |E|)$ and space complexity $O(|V| + |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the graph.

2.1.1. Problem Analysis

The problem at hand involves identifying dominator nodes in a graph. By definition, dominator nodes must appear on every path from the start to the end node. However, not all nodes along an arbitrary path are dominators for all possible paths. Dominators are the nodes that form the intersection of all possible paths from the start node to the end node. Rather than focusing on the shortest path, which would reduce the set of nodes but increase time complexity, this algorithm

employs DFS to identify the first encountered path and subsequently excludes nodes that do not appear on all paths.

The key observation is that a node on the path can only be excluded as a dominator, meaning it can be avoided during traversal due to the existence of an alternative path, if there is a forward or cross edge from its ancestor to a node that is on this path, within the DFS tree rooted at the start node.

To clarify this further, let us analyze the graph presented in Figure 1, where the starting node is node 0 and the ending node is node 12.

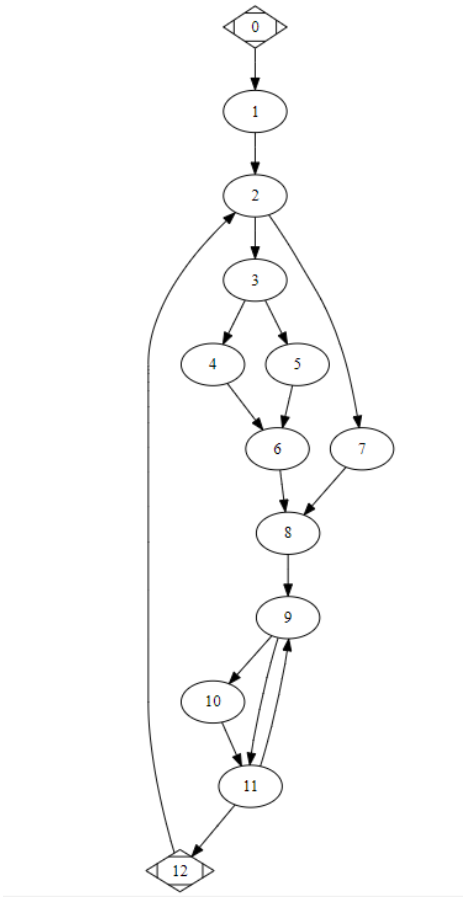

Figure 1 - Example graph - Tarjan's paper [8]

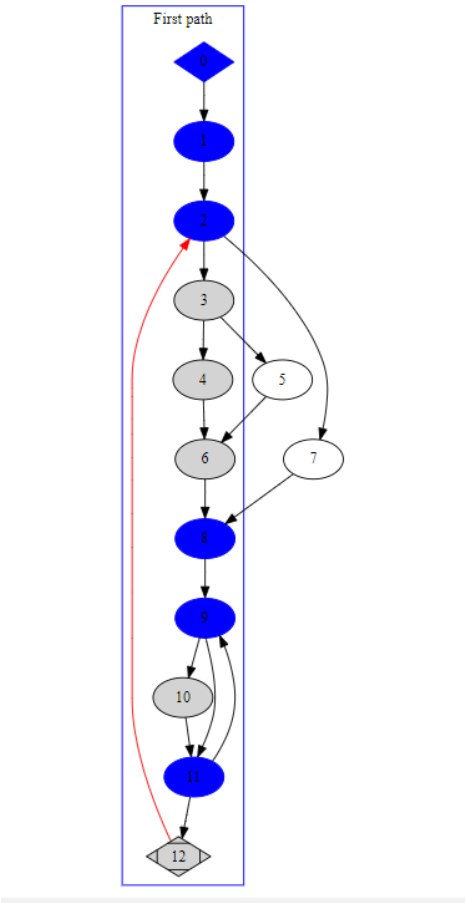

Figure 2 - Discovered path during DFS traversal

During the DFS traversal, we obtain a single path, the first discovered path, which we represent vertically and enclosed in a box, as illustrated in Figure 2. However, we also display the existing edges in the graph to identify forward, back, and cross edges. This allows us to analyze the situations where these edges exist and their impact on the dominator identification process. Let us introduce the term "deepest", refering to child node, which plays a key role in the explanation of the algorithm. A "deepest" node refers to a successor of the observed node that satisfies two conditions: it is the first successor of the observed node that belongs to the path, if such a successor exists, and among all such successors, it is the last one encountered within the path. While there may be multiple successors of a node, we only consider the ones on the path and the "deepest" one is the last node reached within that path.

In the example graph shown in Figure 1, the deepest node for node 1 is node 2. However, for node 2, the deepest node is node 8. This is because there is a path $2 \rightarrow 7 \rightarrow 8$, where node 7 is not part of the discovered path, but node 8 is. Thus, node 2 has two possible paths: one leading to node 3 (which is on the path) and another leading to node 8 (which is also on the path). We only consider successors

that are part of the path, and among them, the deepest node is node 8, as it is the last node reached on the path.

Now, we can formulate the conditions that hold for identifying dominators:

1. **Forward Edge Condition**: If a forward edge exists from a node v on the path to any of its descendants w (also on the path), then all nodes between v and w should be skipped. This is because an alternative path bypassing these nodes exists. In Figure 2, an example of such a forward edge is the edge 9→11. This indicates that a path avoiding node 10 exists, making it unnecessary to include node 10 in dominator set.
2. **Deepest Node Condition**: For any node v on the path, if v is the deepest node of its direct predecessor u on the path (i.e., $deepest(u) = v$), then v is a dominator.
3. **Cross Edge Condition**: If a cross edge exists between a node v on the path and an ancestor u of v , then all nodes between the first ancestor of u, which is on the path, and the deepest child of u should be skipped. In Figure 2, edges 7→8 and 5→6 are cross edges connecting a node to an ancestor on the path (specifically, relations 2→7→8 and 3→5→6). In such cases, nodes between the ancestor on the path and the deepest node should be skipped. Note: The deepest node for node "7" is "8" with ancestor node "2", and the deepest node for node "5" is "6" with ancestor node "3".
4. **Back Edges**: Back edges should be disregarded during the traversal, as they do not contribute to the identification of dominators. These edges create cycles and do not provide additional information relevant to determining dominator relationships.

Now, we can conclude that a preorder numeration of nodes is necessary, as it corresponds to the entry time of each node. This enables us to efficiently identify the deepest node when needed. We can achieve this by performing the DFS traversal. The algorithm steps are as follows:

- Perform a DFS traversal from the start node to the target node to identify the first path, which is stored as *path*. The potential dominator nodes are contained within this *path*.
- Conduct a second DFS traversal to calculate the entry time and the deepest node for each node in the *path*. Specifically, for each node v on the *path*, determine its deepest child, denoted as $deepest(v)$.
- Traverse the *path* and verify whether each node is a dominator by considering the following two conditions:
  - If the deepest node of a predecessor of the observed node is the observed node itself, then the observed node is a dominator.
  - If the deepest node of an immediate predecessor is a successor of the current node (not necessarily immediate, but within the path), indicating the existence of a bypassing path, then all nodes up to the deepest node of the predecessor should be skipped.

To conclude, once the path is discovered, if the edge $v \rightarrow u$ is part of the path, it suffices to check if $deepest[v] = u$, making u a dominator. Otherwise, skip the check until $deepest[v]$.

2.1.2. Time Complexity

The overall time complexity of the algorithm is determined by the two depth-first search (DFS) traversals, which have a combined complexity of $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Additionally, there is a single traversal through the discovered path, as outlined in **Pseudocode 1**, line 10 to 24. To ensure that additional operations, such as calculating the deepest nodes, do not increase the overall complexity, we provide a detailed analysis.

The subroutine preorderEnumerate is a slight modification of the standard DFS traversal, augmented with additional calculations for determining the deepest reachable nodes. These additional operations include constant-time checks (e.g., verifying whether a node belongs to the path) and constant-time updates or retrievals from a map structure. These steps are executed during each traversal of an edge or vertex and thus do not exceed $O(1)$ per operation.

As shown in Lines 7 to 19 of **Pseudocode 2**, these additional operations are bounded by the number of edges and vertices, ensuring the overall complexity remains $O(|V|+|E|)$. Therefore, the algorithm maintains linear time complexity despite the added calculations.

2.1.3. Space Complexity

The graph is represented using a Map<String, Node> where each key is a node's unique identifier (a string) and each value is a Node object. Each Node contains its id and integer fields (orderIndex), and a LinkedHashSet<Edge> for storing outgoing edges. The outgoing edges are represented as Edge objects, which contain references to their target nodes. The space complexity of this representation is determined by the nodes and edges. Each node contributes $O(1)$ space for its id, integer fields, and references, resulting in $O(|V|)$ for all nodes. The outgoing edges are stored in a LinkedHashSet, which uses $O(1)$ space per edge, giving $O(|E|)$ for all edges. The Map holding the nodes requires $O(|V|)$ space for keys and references. Therefore, the total space complexity of the graph representation is $O(|V|+|E|)$, which is typical for an adjacency list implementation.

2.1.4. Termination

The termination of the algorithm is guaranteed due to the following reasons: First, the algorithm begins by performing a DFS traversal to identify a path from the source to the target node. This step is guaranteed to terminate because the graph is finite, and the traversal operates on a well-defined set of edges.

After identifying the path, the algorithm processes this path, which has a maximum length of $|V|$, the number of vertices in the graph. During this traversal, for each node on the path, the algorithm either proceeds sequentially to the next node or skips directly to the "deepest" node, as determined by the stored preorder indices. This ensures that the loop progresses and does not revisit nodes, thereby avoiding any possibility of an infinite loop.

Additionally, the algorithm includes explicit handling of edge cases. If the source is equal to the target, or if the target is unreachable, the algorithm terminates early by returning an empty result. These safeguards, combined with the bounded nature of the path traversal, guarantees that the algorithm completes its execution in a finite number of steps.

2.1.5. Experimental validation

To validate the correctness and performance of the proposed algorithm, we ran it against the comprehensive test suite provided in [8]-[11]. This suite includes a wide range of test cases sourced from published papers, covering various graph structures and edge cases. The proposed algorithm successfully passed all the tests, confirming its correctness across all scenarios. Additionally, the code can be executed and tested against existing algorithms, ensuring compatibility and correctness in comparison to established algorithms. This allows for direct comparison and further validation of the algorithm's performance and results against trusted references.

Tests were originally collected for dominator trees and finding all dominators for all nodes. These tests were adapted here as follows: for each node in the graph, it was treated as the target node, and the list of its dominators was calculated. From this list, we extracted the last dominator, known as the immediate dominator, and compared it with the results of the given tests. By tracing the immediate dominators backward, we generated the full list of dominators, as presented in Table 2.

Table 2 - Applied tests

Test	Dominators	Target node
Tarjan test[8]	[0, 3, 7]	12
Muchnick[9a]	[0]	7
Muchnick[9b]	[0, 1]	6
Cytron[10]	[0]	13

Cytron[10]	[0, 1, 2]	8
Appel[11a]	[0, 1, 2]	6
Appel[11b]	[0, 1, 6]	9
Appel[11c]	[0, 1]	7

Algorithm SmartDominatorFinder

Input: source, target, graph
Output: Dominator list or empty list if there is no dominators

```
result = empty list
if source equals target:
    return result                    Nothing to do if source equals target
path = empty list
if not findFirstPreorderPath(source, target, empty set, path):
    return result                    Target is unreachable
add source to path
deepest = empty map                  Initialize map to store the deepest node for each node
pathAsSet = set created from path    for O(1) search time
preorderEnumerate(source, empty set, pathAsSet, deepest, target)  First DFS traversal to get preorder enumeration and deepest nodes
    add source to result             Add the source node as a dominator
for i = 1 to length of path - 1:     Traverse the path to identify dominator nodes
    curr = path[i]
    if curr equals target:
        break
    pred = path[i - 1]
    if deepest[pred] equals curr:
        add curr to result
    else:
        if deepest[pred] > deepest[curr]:        According to entry time
            while i < length of path and path[i] is not equal deepest[pred]:   Skip nodes up to the deepest node of the predecessor
                increment i
return result
```

Pseudocode 1- Main steps

preorderEnumerate - Node Numeration and Deepest Node Identification

Input: source – source node, **visited** – empty set for storing visited nodes**, pathAsSet** – discovered path as a set, **deepest** – currently empty map for storing the deepest reachable node for each node, **target** – target node.
Output: Preorder indices assigned to all nodes, deepest map updated with the deepest reachable node for each node

1: Assign preorder index to observed **node** as the size of **visited** + 1.
2: Add **node** to **visited**
3: **for each** successor **succ** of **node**:
4: **if succ** is not in **visited**, recursively call:
5: PreorderEnumerate(succ, visited, path, deepest, targetNode)
6: Get the deepest node associated with **node**, denoted as **deepestOfNode**.
7: **if succ** is part of the **path**:
8: **if deepestOfNode** is null:
9: Set **deepestOfNode** to **succ**
10: **else:**
11: Set **deepestOfNode** to the node with the greater entrytime between **deepestOfNode** and **succ**
12**: else** *succ is not part of the path*:
13: Get the deepest node associated with **succ**, denoted as **deepestOfSucc**
14: i**f deepestOfSucc** is null: Continue to the next successor.
15: **else:**
16: **if deepestOfNode** is null:
17: Set **deepestOfNode** to **deepestOfSucc**.
18: **else**:
19: Set **deepestOfNode** to the node with the greater entry time between **deepestOfNode** and **deepestOfSucc**

Pseudocode 2 - preorderEnumerate steps

3. CONCLUSION

In this paper, we described, implemented, and tested a novel algorithm for efficiently finding all dominators of a specified target node in a directed graph. The proposed algorithm addresses the limitations of traditional approaches, which are designed to calculate dominators for all nodes, by focusing solely on the dominator relationships relevant to a single target. This specificity results in significant improvements in efficiency, making the algorithm particularly well-suited for applications where dominator information is required for individual nodes.

The algorithm was rigorously tested against a comprehensive set of benchmarks collected from existing published works. The results demonstrate its correctness and reliability, as it successfully passed all tests. By addressing a specific need in dominator analysis, this work provides a valuable contribution to the field of graph algorithms, with potential implications for control flow analysis, program optimization, and other related domains.

4. REFERENCES

[1] Lowry, E. S. & Medlock, C. W., *Object Code Optimization*, CACM, pp. 13-22, Jan. 1969.
[2] Busam, V. A. & Englund, D. E., *Optimization of Expressions in Fortran*, Comm. ACM., pp. 666-674, Dec. 1969.
[3] Mendicino, S. F. et al., *The LPLTRAN Compiler*, CACM, pp. 747-755, Nov. 1969.
[4] Allen, F. E., *Control flow analysis*. In Proceedings of a Symposium on Compiler Optimization, pp. 1-19, 1970.
[5] Allen, F. E., & Cocke, J., *A program data flow analysis procedure*. Communications of the ACM, 19(3), 137–147. DOI: 10.1145/361607.361619, 1972.
[6] Cooper, K. D., Harvey, T. J., & Kennedy, K., *A simple, fast dominance algorithm*. Software: Practice and Experience, 4, 110, 2001.
[7] Aho, V. A., & Ullman, D. J., *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., [p. 916], 1972.

[8] Lengauer, T., & Tarjan, E. R., *A fast algorithm for finding dominators in a flowgraph.* ACM Transactions on Programming Languages and Systems (TOPLAS), 1(1):121–141, [pp. 121-131], 1979.
[9] Muchnick, S. S. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, San Francisco, CA, p253. figure 8.18 and p256. figure 8.21, 1997.
[10] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. & Zadeck, F. K., *Efficiently computing static single assignment form and the control dependence graph*. ACM Transactions on Programming Languages and Systems, 13(4):451–490, fig.9, 1991.
[11] Appel & Palsberg, J., *Efficient Computation of the Dominator Tree.* In Modern Compiler Implementation in Java, 2nd ed., a) p441. figure 19.4, b) p449. figure19_8_1 and c) p449. figure19_8_3, 2004.