Dominators in Graphs

Ming-Ho Yee

May 8, 2020

- This talk will define dominators, briefly discuss some applications and the history, and then focus on algorithms for computing dominators
- The focus will be on the Lengauer-Tarjan algorithm, which I implemented for RIR

Definitions

d dominates n $\quad d \text{ dom } n, \quad d \in \text{Dom}(n)$

- If every path from s_0 (start node) to n contains d.
- Every node dominates itself

d strictly dominates n

- If d dominates n and $d \neq n$

d immediately dominates n $\quad d = \text{idom}(n)$

- If d strictly dominates n and every other dominator of n dominates d

Dominator tree

- Every child is immediately dominated by its parent
- Root node is the start node
- Ancestor a of node n → $a \text{ dom } n$

- We're talking about control-flow graphs for function
  - So these are directed graphs, and every function has a start node
- A node n may have multiple dominators, but only one immediate dominator
- The dominator tree directly encodes immediate dominance
  - But (non-immediate) dominance can be read from the tree

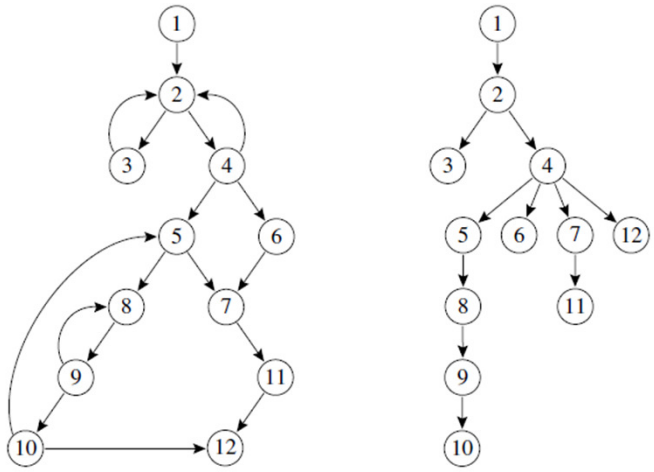


FIGURE 18.3. (a) A flow graph; (b) its dominator tree.

[Appel and Palsberg, 2004]

- Let's look at some examples
- Node 3: every path from the start (1) contains 1, 2, 3
- Node 7: many ways to get there, thanks to branches
  - 4 -> 5 -> 7 or 4 -> 6 -> 7
  - Also loops before 4, but those are optional
  - So we see that 5 and 6 do not dominate 7, but 4 does
  - And 4 is the immediate dominator

Applications

- Static Single Assignment
  - Compute the *dominance frontier* first
  - Use the frontier to insert phi nodes
- Finding loops
- Hoisting instructions

- And many other optimizations…

- Probably the most important application is SSA
  - This is an intermediate representation where each variable is assigned only once
  - Use phi nodes to represent merges of values at control-flow merge points
  - SSA itself has many applications for optimizations, because we know that every variable is defined before used, and written to only once
- Dominance frontier of a node is where that node's dominance "ends"
  - It is where there is a merge of control-flow and data-flow , i.e. where phi nodes need to be inserted
- We use dominance for many other optimizations in RIR
  - Finding loops: node d dominates n, but there is an edge from n to d
  - Hosting instructions: move instructions up to an earlier node

Anecdotal Performance in Ř

	Before		After	
	Total	*Average*	*Total*	*Average*
Benchmark (5 iterations)	89,205 ms	17,841 ms	82,425 ms	16,485 ms
Constructing the dominator tree	4,988 ms	80 µs	90 ms	1.5 µs
a dom? b	42 ms	46.1 ns	169 ms	182 ns
a idom? b	186 ms	24 ns	153 ms	19 ns

- The data for this table is very rough, but it was enough to see problem areas and improvements
- This whole adventure started when I was looking at a benchmark that we were very slow on
- Ran a profiler, saw a lot of time spent constructing the dominator tree
  - Even though we don't count warmup in our benchmarks, this still seemed concerning
  - But this table includes warmup
- There are 5 iterations of the benchmark, but each iteration compiles hundreds of functions
  - Some functions are much larger than others, e.g. 200, 300, or even 700 nodes
  - And dominators are computed multiple times
- Made the mistake of looking at total time, rather than average time
  - I thought the implementation was inefficient, but everything I tried made it worse, it's hard to speed up something that takes 80 us
    - Can't outsmart the STL
  - Turns out we needed a better algorithm
- The old algorithm explicitly stored the dominance relation, but now we read it from the dominator tree

History

- 1959: Definition of *dominance* [Prosser 1959]
- 1969: First sketch of algorithm [Lowry and Medlock, 1969]
  - Complexity is at least quadratic
- 1970: Data-flow equations [Allen, 1970]
- 1972: Iterative data-flow algorithm [Allen and Cocke, 1972]

… more quadratic algorithms …

- 1979: Almost linear complexity [Lengauer and Tarjan, 1979]

… more almost linear algorithms …

- This is actually a very old problem in computer science
- The 1972 algorithm presumably resembles the ones we see in textbooks today, but I couldn't find the paper
- There were many other quadratic algorithms, until 1979
- After that, there was a focus on almost linear algorithms
  - But in practice, these were often too slow on "real" graphs, and only profitable on extremely large graphs

Data-flow Equations

Let $Dom(n)$ be the set of nodes that dominate n. Then:

$$\mathrm{Dom}(s_0) = \{s_0\}$$

$$\mathrm{Dom}(n) = \{n\} \cup \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right)$$

Iterate until you reach a fixed point.
$O(n^2)$ time complexity, and very slow in practice

- We can define dominators as a classic dataflow problem
  - Computing sets of nodes, base case is that the start node dominates itself
  - Forward analysis, intersect the dominators of your predecessors
  - And every node dominates itself
- RIR used a variant of this
  - It seemed that the merge (set intersection) was the most expensive part

Another Algorithm [Aho and Ullman, 1972]

For each node $v \neq s_0$:

- Remove v from the graph
- Consider the set of nodes S that are now unreachable
- Then $\mathrm{Dom}(v) = S$

$O(n^2)$ time complexity

- Want to briefly show this completely different algorithm
- It works by removing nodes one at a time
  - All nodes now unreachable must be dominated by the node that was removed!

Iterative Algorithm, revisited

- Set intersection is the bottleneck
- Idea: use a consistent ordering for sets
  - Perform intersection by walking through both sets in order, with pairwise comparisons
  - Order encodes a path through the dominator tree
- Very simple implementation [Cooper, Harvey, and Kennedy, 2006]
- But slower in practice than the Lengauer-Tarjan algorithm [Georgiadis, Tarjan, and Werneck, 2006]

- In 2006, Cooper, Harvey, and Kennedy made the following observations
  - Set intersection is the bottleneck
  - But this could be sped up (and done with efficient space requirements) if the sets had a consistent ordering
  - So intersection is just a walk through both sets in order, comparing elements
  - Ordering is the path through the dominator tree
- Implementation is very simple
  - Original paper claims it is faster than the LT algorithm
  - But a later paper says that wasn't the case, there were issues in Cooper, Harvey, and Kennedy's implementation

Lengauer-Tarjan Algorithm

- Simple version: $O(m \log n)$
- Sophisticated version: $O\big(m\ \alpha(m,n)\big)$
  - Where $\alpha(m,n)$ is the inverse of the Ackerman function

The simple Lengauer-Tarjan algorithm is faster in practice, and less sensitive to pathological graphs.

This explanation is adapted from Appel and Palsberg [2004].

- A bit of handwaving here: m = #edges, n = #nodes
  - Some presentations will use a value N = m + n
  - In the worst case, we know that m = O(n^2) but it's probably not that bad in practice
- The paper has a simple version that is O(m log n) and a sophisticated version whose complexity involves the inverse Ackerman function
- In the original paper, the sophisticated version was faster
  - In a later paper, this was only true for extremely large graphs
  - So it seems the simple version is "fast enough" in practice
- The rest of this talk is about the Lengauer-Tarjan algorithm
  - The details are a bit involved
  - And even the implementation is a bit tricky
  - But transcribing the pseudocode was straightforward

Depth-First Spanning Tree

- Use DFS to compute a spanning tree
  - Assign a *dfnum* to each node

a is an *ancestor* of b
- If $a = b$ or there is a path from a to b in the spanning tree
- *i.e.* $\mathrm{dfnum}(a) \leq \mathrm{dfnum}(b)$

a is a *proper ancestor* of b
- If a is an ancestor of b and $a \neq b$
- *i.e.* $\mathrm{dfnum}(a) < \mathrm{dfnum}(b)$

Note: Can test ancestor relation by comparing *dfnum*s

- First, we need to look at depth-first spanning trees
  - Traverse the CFG using DFS and number nodes, this constructs a spanning tree
  - Nodes are numbered in order of encounter
  - Edges in the CFG can be classified as “tree edges” or “non-tree edges”
- Note that these are not “if and only if”
  - If you know a node is the ancestor or descendent of another node, can compare dfnums
  - But dfnums are meaningless if two nodes are unrelated

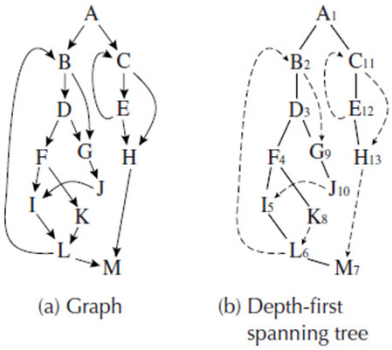


FIGURE 19.8. A control-flow graph and trees derived from it. The numeric labels in part (b) are the *dfnum*s of the nodes.

[Appel and Palsberg, 2004]

- Solid edges are tree edges, dashed edges are non-tree edges
- DFS means when we start at A, and see B and C, we visit all of B's subtree before visiting C
- Let's look at ancestors
  - B is an ancestor of M, because B is 2 and M is 7
  - C is not an ancestor of M
    - There is a path in the original CFG, but not the depth-first tree
- May have encountered some terminology, but it's not needed for dominators
  - Forward edge: non-tree edge from a node to its descendent, e.g. C -> M
  - Back edge: non-tree edge from a node to its ancestor, e.g. E -> C
  - Cross edge: non-tree edge to a node with no ancestor relation, e.g. H -> M

Dominators and *dfnum*s

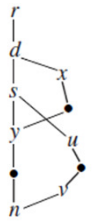
- If $\text{idom}(n) = d$, then d must be an ancestor of n
  - *i.e.*, $\text{dfnum}(d) < \text{dfnum}(n)$
- Therefore: ancestors of n are idom candidates!
- If an ancestor x does not dominate n, there must be some "detour" starting above x.
  - Nodes on the detour are not ancestors of n
  - *i.e.* their *dfnum*s must be greater than n's

- Why is this important for dominators?
  - Immediate dominator of n is an ancestor of n!
- This gives a very rough guess for idom candidates, but we can do better
- What about the cases where an ancestor x is not a dominator?
  - There must be some detour that bypasses x
  - Look at the node where the detour starts

Semidominators

s semidominates n $s = \text{semi}(n)$

- If s is the highest ancestor with a path to n, using non-ancestor nodes
  - Highest ancestor → smallest *dfnum*
    - $\text{dfnum}(s) < \text{dfnum}(n)$
  - Path $p = s, u_1, \ldots, u_k, n$ using non-ancestor nodes
    - $\text{dfnum}(u_i) > \text{dfnum}(n)$



$\text{semi}(n)$ is a candidate for $\text{idom}(n)$

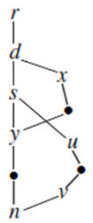
- Often, $\text{semi}(n) = \text{idom}(n)$
- An exception: $\text{semi}(n)$ itself is bypassed

- Let's formalize that idea
  - For now, ignore the d -> x -> y path in the image
- We want to find s, the semidominator of n
  - s is the highest ancestor of n, with a detour path to n using non-ancestor nodes
    - s and n are ancestors of n, but the nodes in between cannot be ancestor nodes
    - Note that such a path may simply be the edge from s to n; that counts
- Intuitively, what does this mean?
  - There is a path from s to n using only tree edges
  - There is a "detour" path that contains non-tree edges
  - s is the highest ancestor of n with this property
- If there is a path from the start to s, there are two branches to n
  - s is possibly the immediate dominator of n
  - But an exception is if s itself is bypassed, e.g. d -> x -> y

Semidominator Theorem

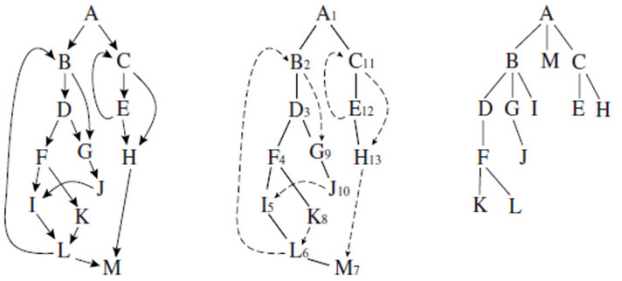
Consider all predecessors v of n in the CFG. Then:

- If v is a proper ancestor of n $\quad \text{dfnum}(v) < \text{dfnum}(n)$
  - v is a candidate for $\text{semi}(n)$
- If v is a non-ancestor of n $\quad \text{dfnum}(v) > \text{dfnum}(n)$
  - For each u that is an ancestor of v, (and not an ancestor of n) $\text{semi}(u)$ is a candidate for $\text{semi}(n)$



$\text{semi}(n)$ is the candidate with the lowest *dfnum*

- How do we compute semidominators? We use the Semidominator Theorem.
- We start by looking at the predecessors of n in the CFG
  - If the pred is an ancestor, that's a candidate for semidominator. Recall that preds can be semidominators, because the "bypassing path" can be empty
  - If the pred is a non-ancestor, it's more complicated
    - Consider all the ancestors of the pred that aren't ancestors of n, i.e. nodes on the "bypassing path"
    - The semidominators of those nodes are candidates for semi(n)
    - Note that a semidominator can be a predecessor or the "start of a bypassing path," e.g. a non-tree path from u to v
- Take the one with the lowest dfnum, i.e. the highest node up the tree
- This may be clearer with examples



(a) Graph (b) Depth-first spanning tree (c) Semidominator tree

FIGURE 19.8. A control-flow graph and trees derived from it. The numeric labels in part (b) are the *dfnum*s of the nodes.

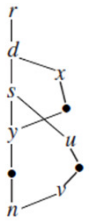
[Appel and Palsberg, 2004]

- Let's look at a few examples
- E has only one predecessor, C
  - C is an ancestor of E, so it is a candidate for semi(E)
  - It is the only one, so it is the semidominator
- H has two predecessors: E and C
  - C and E are both ancestors, so they are both candidates
  - But C has the lower dfnum, so it is the semidominator
- L has two predecessors: I and K
  - I is an ancestor, so it is a candidate semidominator
  - K is a non-ancestor, so look at all of K's ancestors (including K) that are not ancestors of L
    - This is only K
    - Look at K's semidominator, F, which is a candidate semidominator for L
  - F has a smaller dfnum than I, so F is the semidominator
- M has two predecessors: L and H
  - L is an ancestor, so it is a semidominator candidate
  - H is a non-ancestor, so we look at its ancestors: H, E, C
    - Their semidominators are C, C, and A
  - A has the smallest dfnum, so it is the semidominator

Dominator Theorem

Consider the spanning tree path from $s = \text{semi}(n)$ to n.

Let y be the node with smallest numbered semidominator, *i.e*. minimum $\text{dfnum}(\text{semi}(y))$.



Then:

$$\text{idom}(n) = \begin{cases} \text{semi}(n) & \text{if } \text{semi}(y) = \text{semi}(n) \\ \text{idom}(y) & \text{if } \text{semi}(y) \neq \text{semi}(n) \end{cases}$$

- Now that we have semidominators, we can compute dominators
- Suppose we have computed s, the semidominator for n
  - Consider the path from s to n, using tree edges, skipping s but including n
  - Look at the semidominators of each node of that path
  - Let y be the node whose semidominator has the smallest dfnum
    - i.e. y is the node with the "highest" semidominator
- Consider the diagram and pretend the path d -> x -> y doesn't exist
  - s is the semidominator of n, so we look at the s -> y -> n path
  - y's semidominator is s, and that is the highest semidominator
  - semi(y) = semi(n) so the semidominator is the immediate dominator
- Now suppose there is a d -> x -> y path that bypasses s
  - semi(n) = s, but semi(y) = d
  - Then idom(y) = idom(n)

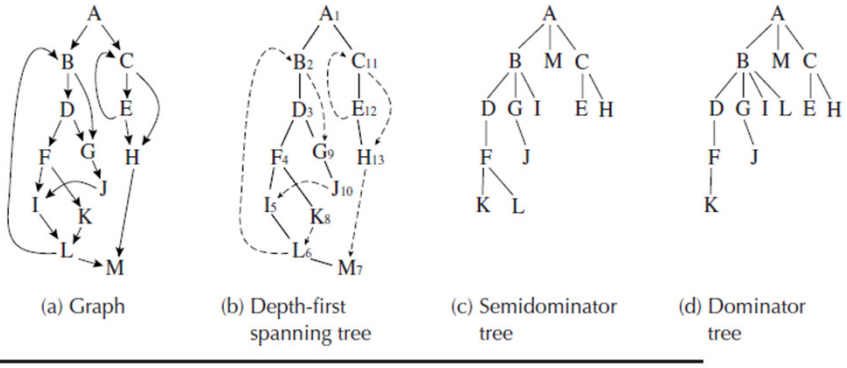


FIGURE 19.8. A control-flow graph and trees derived from it. The numeric labels in part (b) are the *dfnum*s of the nodes.

[Appel and Palsberg, 2004]

- Let's look at some examples
- I's semidominator is B
  - Consider the tree path from B to I, the nodes are I, F, and D
  - Their semidominators are B, D, B
  - Nodes with the highest semidominator are I and D, with semidominator B
  - This is the semidominator of I, so it is the immediate dominator of I
- L's semidominator is F
  - Its tree path contains I, L
  - Their semidominators are B, F
  - Highest semidominator is B
  - So I's dominator is L's dominator, which is B

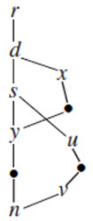
Lengauer-Tarjan Algorithm

1. Perform DFS to number nodes and create the depth-first spanning tree
2. For each node n (in decreasing *dfnum* order):
   - Use the Semidominator Theorem to compute $\mathrm{semi}(n)$
   - Insert n into the spanning forest
3. Implicitly define the idom by applying the first clause of the Dominator Theorem
4. For each node n (in increasing *dfnum* order):
   - Explicitly define the idom by applying the second clause of the Dominator Theorem

- Those are the ideas behind the LT algorithm
  - Still a bit hard for me to get the intuition, maybe looking at the proofs would help
  - Now the trick is how to compute these, and the algorithm is also pretty complicated
- Step 1 is easy: perform DFS to number the nodes
- Step 2: consider nodes in descending dfnum order, i.e. bottom up
  - Look at predecessors and compute semidominators
  - Maintain a spanning forest representing the CFG; here we insert n into the forest
- Step 3: apply first clause of dominator theorem
  - If semidominators are equal, then that is the dominator
  - We can't apply the second clause, because we don't yet know the dominators
- Step 4: iterate in increasing dfnum order now, top down
  - Now that we know dominators, we can finish applying the dominator theorem

Spanning Forest

- Build a spanning forest as the CFG is traversed
  - When n is processed, only non-ancestors of n are in the forest
- `link(p, n)`
  - Add the edge (n, p) to the spanning forest
- `ancestorWithLowestSemi(v)`
  - Search upwards in the forest, starting from v
  - Find the ancestor of v whose semidominator has the lowest *dfnum*



- The spanning forest keeps track of the nodes we have processed
  - I.e. when n is processed, only its non-ancestors are in the forest
- E.g. when n is processed, u and v are already in the forest
  - This is because we iterate in decreasing dfnum order
  - u and v were encountered before n, so they have higher dfnums
  - So when we encounter n, only higher numbered nodes are in the forest
- API allows us to add an edge to the spanning forest
- Or we can query for the ancestor with semidominator with lowest dfnum
  - i.e. ancestor with “highest up the tree” semidominator
  - ancestorWithLowestSemi(v) means we would search v and u, but not s, d, or r

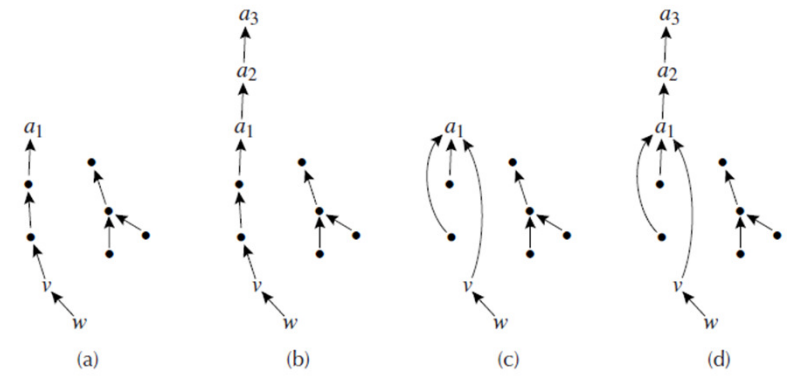


FIGURE 19.11. Path compression. (a) *Ancestor* links in a spanning tree; AncestorWithLowestSemi(v) traverses three links. (b) New nodes a_2, a_3 are linked into the tree. Now AncestorWithLowestSemi(w) would traverse 6 links. (c) AncestorWithLowestSemi(v) with path compression redirects ancestor links, but *best*[v] remembers the best intervening node on the compressed path between v and a_1. (d) Now, after a_2 and a_3 are linked, AncestorWithLowestSemi(w) traverses only 4 links.

[Appel and Palsberg, 2004]

- If the spanning forest is implemented naively, queries are O(n)
  - Overall algorithm complexity would be O(n^2)
  - Start with (a)
    - If we query on w, we follow the path all the way up to a1
    - Suppose we add more nodes to the end of a1, giving us (b)
    - If we query w again, we follow the entire path again to get to a1, before traversing the new links
- We want to be smart, use “path compression” so queries are O(log n)
- Idea behind path compression
  - Every time we query, we update the links so that the path becomes shorter
  - We query from w, and once we reach the end (a1), we set each node to point to a1
  - This is image (c)
  - Then, when we add a2 and a3, the query goes from w to v, and then jumps straight to a1

```
Dominators() =
  N ← 0;  ∀n. bucket[n] ← {}
  ∀n. dfnum[n] ← 0, semi[n] ← ancestor[n] ← idom[n] ← samedom[n] ← none
  DFS(none, r)
  for i ← N − 1 downto 1
    n ← vertex[i];   p ← parent[n];   s ← p
    for each predecessor v of n
      if dfnum[v] ≤ dfnum[n]
        s′ ← v
      else s′ ← semi[AncestorWithLowestSemi(v)]
      if dfnum[s′] < dfnum[s]
        s ← s′
    semi[n] ← s
    bucket[s] ← bucket[s] ∪ {n}
    Link(p, n)
    for each v in bucket[p]
      y ← AncestorWithLowestSemi(v)
      if semi[y] = semi[v]
        idom[v] ← p
      else samedom[v] ← y
    bucket[p] ← {}
  for i ← 1 to N − 1
    n ← vertex[i]
    if samedom[n] ≠ none
      idom[n] ← idom[samedom[n]]

DFS(node p, node n) =
  if dfnum[n] = 0
    dfnum[n] ← N;  vertex[N] ← n;  parent[n] ← p
    N ← N + 1
    for each successor w of n
      DFS(n, w)

AncestorWithLowestSemi(node v) =
  a ← ancestor[v]
  if ancestor[a] ≠ none
    b ← AncestorWithLowestSemi(a)
    ancestor[v] ← ancestor[a]
    if dfnum[semi[b]] <
        dfnum[semi[best[v]]]
      best[v] ← b
  return best[v]

Link(node p, node n) =
  ancestor[n] ← p;   best[n] ← n
```

[Appel and Palsberg, 2004]

- This is the pseudocode
  - It's succinct and each individual statement isn't very complicated
- But there are a lot of subtleties
  - When we compute semi(n), we have this bucket for temporary storage
  - We can't proceed until we've added n to the spanning forest
    - This is what Link(p, n) is for
  - But then we iterate over bucket[p] instead of bucket[s]
- And then the ancestorWithLowestSemi implementation is a bit complicated

Implementation in Ř

Straightforward translation from pseudocode to C++

- About 100 LOC, without comments

https://github.com/reactorlabs/rir/blob/b265f9e/rir/src/compiler/util/cfg.cpp#L52

- However, it was a very straightforward translation to C++
- Only 100 LOC, without comments
- You can find the annotated implementation on GitHub

References

A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, 1972.

F. E. Allen. Control flow analyses. In *SIGPLAN Notices, 1970.*
http://www.cs.columbia.edu/~suman/secure_sw_devel/p1-allen.pdf

F. E. Allen and J. Cocke. Graph-theoretic constructs for program flow analysis. *Technical Report RC 3923 (17789)*, IBM T. J. Watson Research Center, 1972.

A. Appel and J. Palsberg. Efficient Computation of the Dominator Tree. In *Modern Compiler Implementation in Java*, 2nd ed., 2004.

K. D. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. *Rice Computer Science TR-06-33870*, 2006.
https://www.cs.rice.edu/~keith/EMBED/dom.pdf

L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. In *J. Graph Algorithms Appl.*, 2006.
http://doi.org/10.7155/jgaa.00119

T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM TOPLAS*, 1979.
https://doi.org/10.1145/357062.357071

E. S. Lowry and C. W. Medlock. Object code optimization. In *Comm. ACM*, 1969.
https://doi.org/10.1145/362835.362838

R. Prosser. Applications of Boolean matrices to the analysis of flow diagrams. In *Proc. Eastern Joint Computer Conf.*, 1959.
https://doi.org/10.1145/1460299.1460314