Studienarbeit:

Transforming Irreducible Regions of Control Flow into Reducible Regions by Optimized Node Splitting

by
Sebastian Unger

Humboldt Universität zu Berlin,
Institut für Informatik,
1998

Betreuer: Frank Müller, Ph. D.

Abstract

Modern compilers apply a number of rather complex optimizations, and the research in that area is still being pursued. Many algorithms, however, have only been implemented for reducible flow graphs.
Arbitrary programs may, however, result in irreducible flow graphs such that an optimizing compiler has to recognize and handle irreducibility. Three different methods have been used to deal with irreducible flow graphs:

1. The optimizer recognizes the affected regions and simply excludes them from certain problematic optimizations.

2. In a first pass the irreducible flow graph is converted to a reducible one by code replication. The other optimizations can then be done efficiently.

3. All affected optimizations are modified to handle irreducible loops. Unfortunately, this is not possible for all optimizations and even if it is possible, the modifications required are rather complex.

Though the second approach is simple to implement and provides a genuine solution to the problem, it is not used very often. The problem is that with all present algorithms the growth in code size can be quite large. This work describes an attempt to find an algorithm that limits the growth to the minimum necessary to resolve the irreducibility.

Problem Statement and Thesis

- Conversion of irreducible flow graphs to reducible ones by node duplication
- Introduction of a generalized structure of loops, both irreducible and reducible
- Development of a new algorithm based on properties of the minimal equivalent flow graph
- Proof of correctness, i.e. reducibility, equivalence and termination
- Proof of minimality, i.e. the new algorithm can always construct a minimal equivalent reducible flow graph

Contents

1 Introduction

In modern software design it is often stated that large software projects (such as compilers) should consist of smaller modules with a distinct and properly encapsulated functionality. In compiler design these modules are often identified as

- the scanner doing the lexical preprocessing,
- the parser, which is responsible for the syntactic analysis of the program, and
- the semantic analyzer,

forming the so called front-end of the compiler [1]. Due to the close relations between syntactic and semantic analysis the two modules are often integrated into one. The front-end emits an *internal representation* of the input, that is handed down to the back-end, consisting of

- the optimizer and
- the code generator.

In earlier compilers the functionality of the optimizer was integrated in the module of semantic analysis and/or the code-generator. This approach, however, hampered the maintainability of these modules and restricted optimizations to comparably simple algorithms. On the other hand, the separation of the semantic analysis and optimization often leads to front-ends that produce rather inefficient code and the optimizer has less information about the original structure of the program.

Nonetheless, it was only the separation of the optimizing code that made algorithms, such as peephole optimization [2] and optimizations that exploit parallelism on the instruction level (ILPO), possible. Many algorithms have been developed since then, and there are still many open issues. One issue relates to the optimization of so-called irreducible loops.

Loops contain the most frequently executed portions of programs, as often described by the 90/10 rule: 90% of the time are spend in 10% of the code [4]. As a result, loop optimizations have been the focus of many optimization techniques that have been developed. However, most loop optimizations have only been formulated for natural (also called reducible) loops with a single entry point. These techniques fail if there are any irreducible loops in the control flow graph. Therefore, most compilers check for the presence of such loops and exclude them from optimizations that cannot deal with these loops.

There are at least two other approaches to irreducible loops than complete exclusion:

- Analyzing the nesting of reducible and irreducible loops with an algorithm as described by Sreedhar et. al. [8] and then applying modified optimizations that can handle irreducible loops.

- There are algorithms which can convert any irreducible flow graph into a reducible one by duplicating some of the nodes in such a graph.

The first approach is probably the best way for new compilers. But if these extensions were to be integrated into an existing compiler many modifications would be required. These modifications can well affect other parts of the optimizer, which is a potential source for bugs. Another argument against the first approach is that some optimizations, such as ILPO, *require* reducible flow graphs.

The second approach to optimizing irreducible loops is a technique called node splitting. With this technique a single additional pass has to be build into the optimizer which converts irreducible loops to natural ones. All other optimizations can then be applied without modifications.

Node splitting is based on a certain kind of interval analysis known as T1/T2-Analysis, which was used to check for the presence of irreducible regions in a flow graph [1]. It iteratively performs two transformations on the flow graph reducing it to a simpler one. These transformations are:

T1 Remove any edge that connects a node to itself.

T2 If any node has exactly one predecessor, then replace this node and its predecessor with a single new node. All edges to the predecessor node are connected to the new node, and all edges leaving one of the original nodes will now originate from the new *abstract* node.

If these transformations are applied as long as possible the resulting graph is called the *limit graph*. As shown by M. S. Hecht [3] this limit graph is independent of the order of transformations and the nodes subject to transformations. If the final graph is trivial, i.e. it has only one node, the original flow graph was reducible. If the limit graph is non-trivial, all of its nodes either have none or *more* than one predecessor.

The idea of node splitting is to define an additional transformation T3, which is applied if neither T1 nor T2 are applicable anymore. T3 is defined as follows:

T3 Choose any node with at least two predecessors. Duplicate this node so that there is one copy for each of them. Each of the predecessors is now connected to one of the copies, and all of the outgoing edges of the original node are duplicated for each copy.

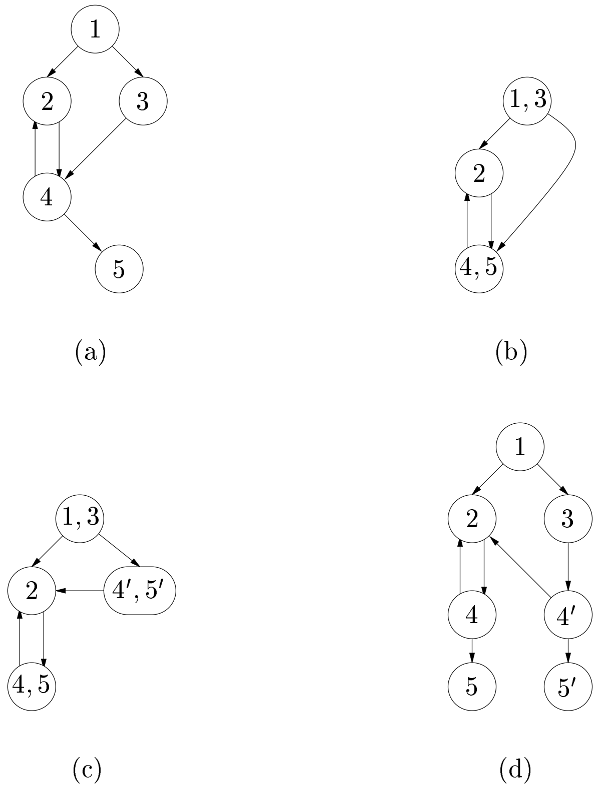


Figure 1: The process of node splitting

After the application of T3, it is possible to use T2 again on these duplicated nodes since they now have only one predecessor. If this process is repeated the resulting limit graph is always trivial.

If the above process is reversed, leaving the duplicated nodes in place, the result is a *reducible* flow graph that is equivalent to the original one. The entire process is illustrated in Figure 1.

First, the transformation T1 is applied twice, once on node 3 and once on node 5. The result is shown in Figure 1(b). Since there are now no more nodes with only a single predecessor, T1 is no longer applicable. Therefore T3 is applied to the abstract node containing nodes 4 and 5 with the result shown in Figure 1(c). Now the flow graph is reducible and the sequence T2((4′,5′)), T2((4,5)), T1((2,4,5)), T2((2,4,5)) reduces it into a single abstract node.

Reversing that process first leads to the same flow graph as in Figure 1(c). The transformation T3 that led to this graph, however, is not reversed but skipped. This means that the transformation T2(5) must now be reversed for two nodes 5 and 5′. Finally, the transformation T2(3) is reversed resulting in the final reducible flow graph of Figure 1(d).

Though this process already yields an algorithm, its performance may be poor. In Figure 1, for example, there is no need to split node 5. The resulting flow graph would still be reducible if there were just one node 5 with both 4 and $4'$ as predecessors. This algorithm is inefficient because it does not consider which nodes form the irreducible loops. In this work algorithms will be presented that exactly analyze the extent, structure and nesting of irreducible loops. Based on such an analysis a much better algorithm than that above will be constructed.

2 Formal Description of the Problem

As mentioned before, the motivation of this work is to develop an algorithm that converts an arbitrary irreducible control flow graph into an *equivalent reducible* one with the minimal possible growth in code size. This first involves the construction of an algorithm and second a prove of its correctness. It is therefore inevitable to formally specify what the algorithm should do.

First of all the algorithm operates on control flow graphs. Since this work uses only *control* flow graphs in the following they will simply be called flow graphs or just graphs. Flow graphs have been defined as follows:

Definition 2.1 (Flow graph)
A Flow graph *G is a triple (N, E, s) of nodes N, directed edges E and a single start node s, such that (N, E) is a finite directed graph and $s \in N$.*

Though this definition includes flow graphs with so-called *dead code* [1], it is assumed in the following that there is always a path from s to any node in N. The removal of dead code is a problem long solved and does not depend on any loops. It can therefore be done before the conversion of irreducible loops.

Since the semantics of a program must not be changed by any optimization, it is vital that the resulting flow graph be equivalent to the original one. Since the transformation only changes the flow graph but not the nodes itself, the definition of equivalent flow graphs can be based on the equivalence of nodes.

Definition 2.2 (Labeling)
Let $G = (N, E, s)$ be an arbitrary flow graph. A total function $l : N \to \mathbb{N}$ is then called a labeling *of G if for all $(n, m) \in E$ and $(n, m') \in E$ the following holds:*

$$l(m) = l(m') \longrightarrow m = m'$$

Note that l is not requested to be injective. Indeed, the case where l is not injective provides the necessary notion of *equivalent* nodes: All nodes that share the same label are equivalent. With that notion we can now define exactly when two flow graphs are equivalent.

Definition 2.3 (Equivalent flow graphs)
Two flow graphs G_1 and G_2 with labelings l_1 and l_2 respectively are said to be equivalent, *in short $G_1 \sim G_2$, if $l_1(s_1) = l_2(s_2)$ and if for every path $p = (p_1, \ldots, p_k)$ of G_1 there is a path $q = (q_1, \ldots, q_k)$ of G_2 such that*

$$\forall 1 \leq i \leq k : l_1(p_i) = l_2(q_i)$$

and vice versa.

In the initial flow graph an injective labeling l is chosen. But if any nodes are duplicated, their labels are kept on all copies.

Theorem 2.4
$\sim$ is an equivalence-relation.

The proof follows directly from the definition.

Besides the equivalence of the final flow graph, the requested algorithm should result in a minimum increase in code size. But the algorithm itself and the proof should only consider the nodes of flow graphs and not the instructions within nodes. Therefore an abstraction from the code size is made to a general weight function that the algorithm must minimize.

The algorithm uses an arbitrary function $\sigma : N \rightarrow \mathbb{N} \setminus \{0\}$ as the weight of any node. The weight of an entire flow graph is then computed as

$$\sigma_G = \sum_{n \in N} \sigma(n)$$

Though σ could basically be anything, its use in the above formula restricts the set of functions that make sense as σ. The execution time of a basic block for instance does not make sense since the execution time of the entire graph cannot be related to that of its basic blocks by the simple formula above.

σ will often be the number of instructions in the intermediate representation given to the optimizer. This, of course, might be different from the final code size. This approach also does not account for the change of the code size by any other optimizations that follow. Therefore, the algorithm will not always result in the minimal growth of the *final* program.

Since speed is nowadays of much more concern than size, one might want to use the overall worst case or mean execution time of the flow graph as a means to define minimality. However, since the algorithm does not change the instructions within the nodes and since the resulting flow graph is equivalent, all execution paths contain almost[1] the same instructions before and after the conversion. Of course, the instructions will not stay the same if further optimizations are applied.

[1] Except for the jump-instructions, which must be adjusted

As a result the execution time of the final flow graph (after all other optimizations have been applied) had to be used. This would imply that one has to consider the opportunities for other optimizations that are introduced by splitting a node, even before it is decided which node is to be split. This can probably not be done in polynomial time and would spoil the advantage of the entire approach, namely that it is independent of all other passes of the optimizer.

With these definitions, if G were the original flow graph and G_r the result of the algorithm, the following had to be proved:

$$G_r \text{ is reducible and } G \sim G_r \qquad \text{(correctness)}$$

$$\nexists\, G' \text{ with } G' \text{ reducible and } G' \sim G \wedge \sigma(G') < \sigma(G_r) \qquad \text{(minimality)}$$

In order to find an algorithm that satisfies these requirements a thorough understanding of the structure and properties of irreducible loops is necessary.

3 Derivation of a Formal Solution

In this section the necessary properties of irreducible loops are introduced and a new approach to the problem that utilizes these properties is presented and proved correct.

3.1 Properties of Irreducible Loops

Until some time ago, irreducible loops were thought of as a set of nodes that form a loop but none of the nodes in the loop dominates the other loop-nodes. There was no internal structure as in natural loops, which are divided into a body and an entry node that dominates nodes of the body. It was also difficult to determine which nodes were actually part of the loop and how the loops were nested since there were no back-edges as defined for natural loops. But in 1997, Janssen and Corporaal [5] found that each irreducible loop has exactly[2] one subset of at least two of its nodes that have the same immediate dominator, which in turn is *not* part of the loop. They also discovered that these sets play an important role when minimizing the number of splits. Their definition of so-called Shared External Dominator sets was:

Definition 3.1 (Loop-set)
A loop in a flow graph is a path $(n_1, \ldots, n_k)$ *where* n_1 *is an immediate successor of* n_k*. The nodes* n_i *do not have to be unique. The set of nodes contained in the loop is called a loop-set.*

[2] Correctly: exactly one maximal

Definition 3.2 (SED-set)
A Shared External Dominator set (SED-set) is a subset of a loop-set L with the properties that it has only elements that share the same immediate dominator and the immediate dominator (idom) is not part of L. A SED-set of a loop-set L is defined as:

$$\textit{SED-set}(L) = \Big\{ n_i \in L \,\Big|\, \mathrm{idom}(n_i) = e \notin L \Big\}.$$

For a definition of dominators, immediate dominators and other techniques for control flow analysis see, for instance, Muchnick [6].

Definition 3.3 (MSED-set)
A Maximal Shared External Dominator set (MSED-set) K of a loop-set L is defined as:

$$\text{SED-set } K \text{ is maximal} \iff \nexists \text{ SED-set } M, \text{ such that } K \subset M \text{ and } K, M \subseteq L.$$

For example, in the original flow graph of Figure 1, there is only one loop-set $\{2, 4\}$ and the MSED-set is the entire loop-set with 1 as external dominator.

The MSED-set is really a generalization of the single entry block in natural loops, and, in that special case, it consists of just one node. In the following the nodes of MSED-sets will be simply called the header nodes or headers.

Building on that, new generalized definitions can also be found for the bodies (an irreducible loop can have more than one), back-edges and the nesting of irreducible loops. Figure 2 illustrates this generalized structure. The domains represent the body of a natural loop. All edges from any domain back into the MSED-sets are back-edges. The node e is the immediate dominator of the header nodes. The region called e-domain will be defined and used in the next section.

The following, generalized definitions of back-edges and domains are based on MSED-sets whose definition in turn depends on the loop-set. This means, that the extension of the loop-set cannot be defined using back-edges as it is for natural loops. This is only a problem because the definition of MSED-sets does in no way require the loop-set to be maximal. However, several of the following theorems only hold if the loop-sets are SED-maximal.

Definition 3.4 (SED-maximal loop-sets)
A loop-set L is SED-maximal if there is no other loop-set L' such that $L \subset L'$ and MSED-set$(L) \subseteq$ MSED-set(L').

Based on that, domains and back-edges can be precisely defined as:

Definition 3.5 (Domains)
Let L be an irreducible SED-maximal loop-set, K be its MSED-set and h_i be the nodes of K. The domain of h_i is then defined as:

$$\mathrm{domain}(h_i) = \Big\{ n_j \in L \;\Big|\; h_i \textit{ dominates } n_j \Big\}$$

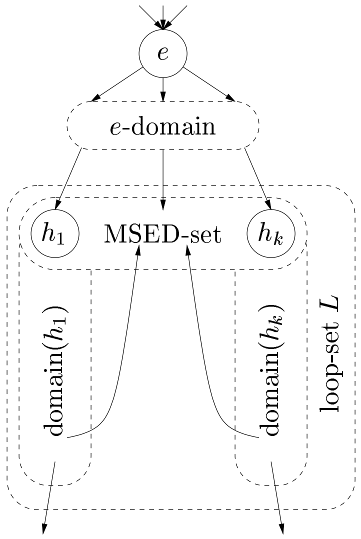


Figure 2: The structure of irreducible loops

Definition 3.6 (Back-edges)
Let L be an irreducible SED-maximal loop-set, K be its MSED-set and h_i be the nodes of K. An edge (m, n) with $m \in L$ and $n \in K$ is then called a back-edge of L.

Figure 2 already suggested several properties of irreducible loops. These will be made precise and proved now:

Theorem 3.7
The nodes of L are in K or in exactly one of its domains.

Proof: Indirect: Since L is irreducible, $s \notin L$. Let $n \in L$ be a node that is neither in K nor in any of the domains of the nodes in K.

If $\text{idom}(n) \in L$, then $\text{idom}(n)$ is also neither in K nor in any of the domains of the nodes in K because otherwise:
If $\text{idom}(n) \in K$ then $n \in \text{domain}(\text{idom}(n))$ and if $\text{idom}(n) \in \text{domain}(m)$ for some $m \in K$ then m dominates $\text{idom}(n)$ and therefore also n. But then $n \in \text{domain}(m)$. n, however, was assumed to be in no domain at all, which is a contradiction.

Since $s \notin L$ and since s is the father of all other nodes in the dominator tree, if there is such a node n, there must also be one with $\text{idom}(n) \notin L$.

In the following let e be the shared dominator of the nodes of K as defined in Def. 3.2.

If $\text{idom}(n) \notin L$, then $\text{idom}(n) \neq e$ since otherwise $n \in K$. This leaves three possible cases:

idom(n) dominates e: There is a path from n to any node in K that does not touch e, because $n \in L$ and $K \subseteq L$ and $e \notin L$. Since e dominates all nodes in K it must then dominate n. This is only possible with $e = \mathrm{idom}(n)$ or e dominates $\mathrm{idom}(n)$, which is a contradiction.

e dominates idom(n): If all paths from e to K would touch $\mathrm{idom}(n)$, then $\mathrm{idom}(n)$ would be the idom of the nodes of K. Therefore we may assume, that there is a path from e to some node in K that does not touch $\mathrm{idom}(n)$. But since $K \subseteq L$, $n \in L$ and $\mathrm{idom}(n) \notin L$, this path can be extended to n without touching $\mathrm{idom}(n)$. This is only possible if $\mathrm{idom}(n)$ dominates e, which is a contradiction.

None of the above cases: Then there is a path from s over $\mathrm{idom}(n)$ and n to any node in K without touching e. This is a contradiction to the assumption that e dominates the nodes of K.

Therefore, such a node n cannot exist!

Furthermore the domains of L are disjunct sets because the nodes of K do not dominate each other and, therefore, each domain is only dominated by one header node. □

There is another property concerning the edges of irreducible loops:

Theorem 3.8
All edges into $\mathrm{domain}(h) \setminus \{h\}$ *originate from* h.

Proof: (indirect)
Let $(m, l) \in E$ be an edge, such that $l \in \mathrm{domain}(h)$ and $h \neq m \notin \mathrm{domain}(h)$. Then one of the following would have to be true:

- h does not dominate m

  But then h does not dominate l, which is a contradiction to $l \in \mathrm{domain}(h)$.

- $m \notin L$

  But since h dominates l, it must also dominate m and therefore there is at least one path p from h to m. This means that $L \cup p$ is also a loop-set and since all nodes in p are dominated by h:
  $\text{MSED-set}\,(L \cup p) \subseteq \text{MSED-set}\,(L)$. This is not possible if L is SED-maximal.

□

This means, that first there are no edges from the outside of L into the middle of a domain, and second there are no edges between domains! All edges leaving a domain either leave the loop-set or end in the MSED-set.

Another very important property concerns the nesting of irreducible loops:

Theorem 3.9
Let L_1 and L_2 be two different, SED-maximal loop-sets, K_1, K_2 their respective MSED-sets and e_1, e_2 the external dominators. Then

- *If neither $L_1 \subset L_2$ nor $L_2 \subset L_1$ then $L_1 \cap L_2 = \emptyset$.* *(distinct loops)*
- *If $L_2 \subset L_1$ then there is a node $h \in K_1$ such that $L_2 \subset \mathrm{domain}(h)$.* *(nested loops)*

Proof: (indirect)

(i) Let $n \in L_1 \cap L_2$, $n_1 \in L_1 \setminus L_2$ and $n_2 \in L_2 \setminus L_1$. Then, because of n, $L = L_1 \cup L_2$ is a loop-set and $L \neq L_1$ and $L \neq L_2$.

If $n \in K_1$ and $n \in K_2$ then MSED-set $(L) = K_1 \cup K_2$ which means, that neither L_1 nor L_2 were SED-maximal.

If $n \notin K_1$ then by Theorem 3.7 it must be in domain(m) for some $m \in K_1$. But then either $m = idom(n) = e_2$ or m dominates e_2. In both cases m dominates all nodes in L_2 and MSED-set $(L) = K_1$. This means, that L_1 is not SED-maximal.

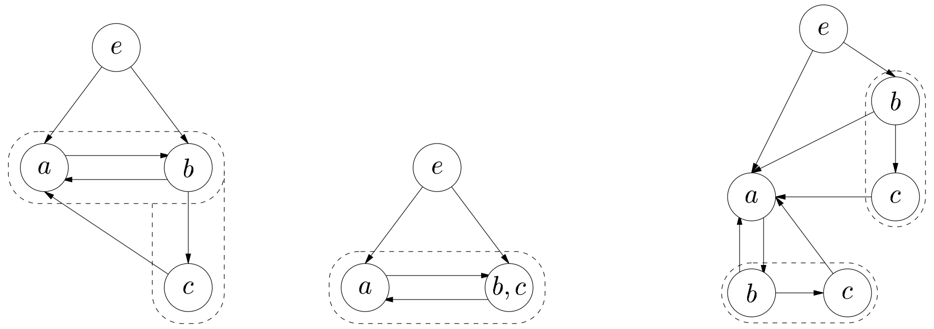
The case for $n \notin K_2$ is symmetric.

(ii) Let $n \in K_2$. If also $n \in K_1$, then the nodes of K_1 and K_2 all have the same immediate dominator and because of $L_2 \subset L_1$: $K_2 \subseteq K_1$. This is a contradiction to the assumption, that L_2 is SED-maximal.
On the other hand, if $n \notin K_1$ then by Theorem 3.7 $n \in \mathrm{domain}(m)$ for some $m \in K_1$. Because m dominates n, either m dominates $\mathrm{idom}(n) = e_2$ or $m = e_2$. In both cases m dominates all nodes of L_2 and therefore $L_2 \subset \mathrm{domain}(m)$. $\square$

3.2 Minimality and Algorithmic Transformation

How can the knowledge of the structure of irreducible loops be used in node splitting? In Section 1, a very primitive algorithm was introduced. One of its greatest deficiencies is the fact that it sometimes splits nodes that are not in any loop-set at all. An example was given in Figure 1.

With the knowledge about the extension of irreducible loops, this algorithm can be modified such that only nodes that are actually part of the loop are collapsed and split. Since the nesting is clear, inner loops can be handled first and the domains will always be reducible. Thus, transformations T1 and T2 will eventually collapse all domains into their respective headers and thus concentrate the irreducibility to the MSED-set alone. When all domains are collapsed, transformations T1 and T2 can no longer be applied. If T3 is now applied to one of the abstract nodes in the MSED-set, it will split not only a header node but also that



(a) (b) after T2 (c) after T3

Figure 3: A sample split

header's *entire* domain as shown in Figure 3. T3 has, in this case, chosen to split header b and its domain. Another possibility was to split node a by itself. Since two copies are made in both cases, the latter would have led to the minimal result if the weight of a is less than the weight of b and c together. On the other hand, if b had had three incoming edges (which would require an additional header node in the example) then three copies would have been made and splitting a would have been better if the weight of a were less than twice the weight of b and c.

Therefore, the order in which the nodes are split should be lightest to heaviest with respect to the weight of the abstract nodes (the headers and domains) *times* the number of incoming edges.

As the domains are collapsed into one abstract node, multiple edges from one domain to a single header node will reduce to just one edge from the abstract node to the header node. This is important because it reduces the number of copies of that node and is also true for multiple edges from the outside. In Figure 2 it has been tried to suggest by the naming that the region called e-domain (defined below) should be handled just as any other domain. That is, transformations T1 and T2 should collapse it into e, thereby reducing multiple edges from that domain to any header node into one edge. Of course, T3 should not be applied to this abstract node.

Definition 3.10 (e-domain)
Let L be an irreducible SED-maximal loop-set, K be its MSED-set and e the external dominator. That is: If e is the immediate dominator of the nodes in K, then the set e-domain *is defined as:*

$$e\text{-domain} = \left\{ n_i \in N \;\middle|\; \begin{array}{l} e \textit{ dominates } n_i, \\ n_i \notin L \textit{ and} \\ \exists \textit{ a path } p \textit{ from } n_i \textit{ into} \\ L \textit{ with } e \notin p. \end{array} \right\}$$

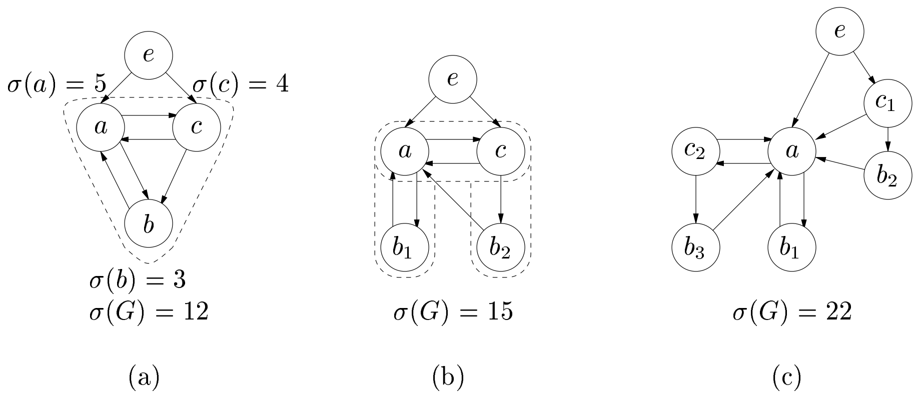


Figure 4: Splitting lightest nodes first[3]

The property of Theorem 3.8, does also hold for this e-domain:

Theorem 3.11
All edges into e-domain *originate from* e.

Proof: (indirect)
Let $(m, l) \in E$ be an edge, such that $l \in e$-domain and $e \neq m \notin e$-domain. Then one of the following would have to be true:

- e does not dominate m

  But then e does not dominate l, which is a contradiction to $l \in$ e-domain.

- $m \in L$

  But since $l \in e$-domain, there must be a path p from l into L that does not touch e. This, however, means, that e dominates all nodes in p. Thus $L \cup p$ would also be a loop-set and MSED-set $(L \cup p) \subseteq$ MSED-set (L). This is not possible if L is SED-maximal.

- There is no path from m into L that does not touch e.

  But then, there is no such path originating from l, either. This is a contradiction to $l \in e$-domain.

□

Does this algorithm always produce the minimal reducible equivalent flow graph as in the example? Unfortunately not, as is shown in Figure 4. Node b_3 could actually be avoided. Its predecessor node c_2 could as well jump to the copy b_1 as shown in Figure 5(c). Had not b but c been split first, Figure 5(c) would have

[3]Nodes with the same label have indices for better reference

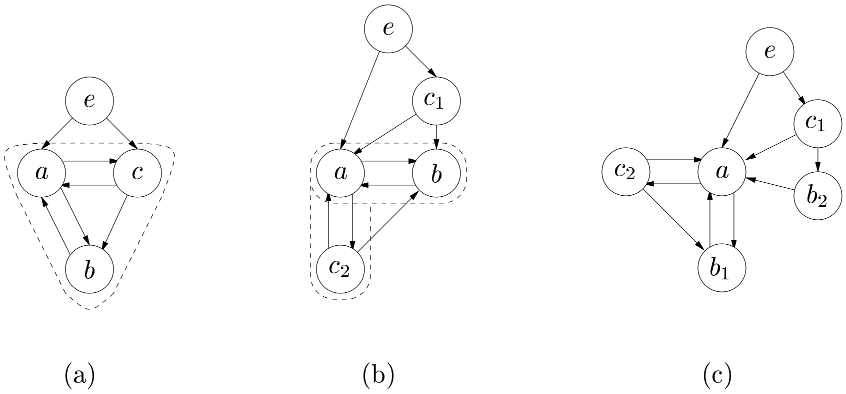


Figure 5: The best splitting sequence (c, b)

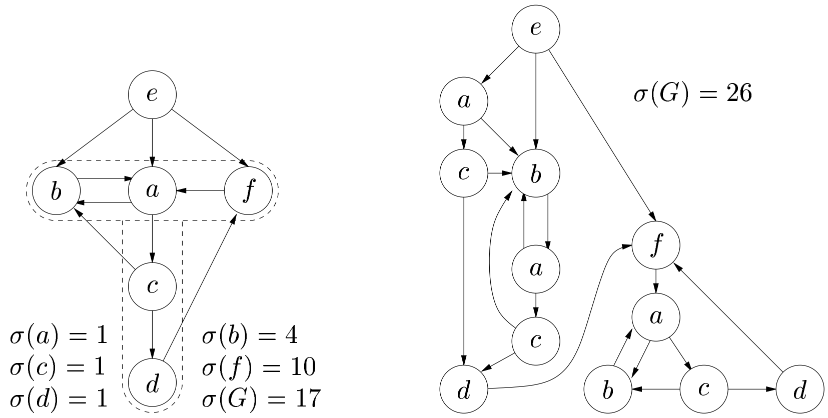
been the result. This means, that selecting the nodes to split just by their weight is not sufficient.

Another question is if there is always an order that leads to the minimum. Alas, not even that is true. Figure 6 gives a flow graph together with its minimal reducible equivalent graph where no order will split the nodes to yield a minimal graph.

The problematic node is node d. This node belonged to the domain of a. Therefore, by the algorithm above, each copy of a will get its own copy of d. But in the minimal flow graph two copies of a share the same node d. The problem is that d is only a part of the loop-set because of its edge to f. Once it has been moved out of the loop containing f, it no longer is within any loop and, therefore, it no longer is in the domain of a. This means that the domains may change in the process of splitting nodes and this cannot be handled by the simple algorithm above.

A new approach has been developed, based on the observation that all of the examples contained one of the header nodes that was not split at all.

In the previous approach, the header nodes had been selected for splitting one after another until only one remained. At that moment, the irreducibility had been resolved. However, as we have seen there were examples where no selection scheme led to a minimal result. The new approach, therefore, does not choose single nodes for splitting. Instead, it chooses a single header node (plus its domain, of course) that should be the one that is not split at all. All other nodes of the loop-set are split once. This is illustrated in Figure 7(a). Of course, the regions containing the copies of the remaining nodes of L are not yet guaranteed to be reducible. They are, however, guaranteed to be smaller than L by at least one node. Therefore, the above step can be recursively applied to these copied regions.



(a) The original flow graph

(b) The equivalent graph that is both minimal and reducible

Figure 6: T3 cannot split this graph in a minimal way.

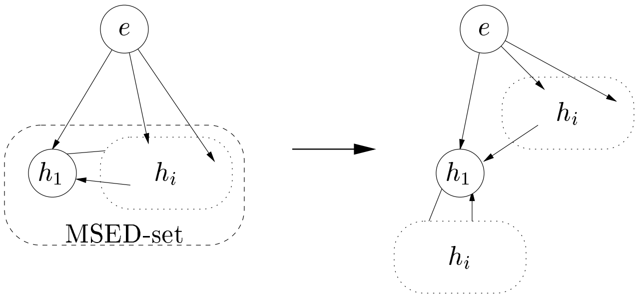
This new approach also needs a scheme for selecting the node h_1. But its advantage over the previous approach is that for each flow graph there is a selection-scheme that leads to a minimal result. All that remains is to actually find that scheme.

However, first the algorithm is defined more precisely and it is shown that it is correct, i.e. it always produces an equivalent and reducible flow graph. The following shortcut will be used often in the theorems and proofs:

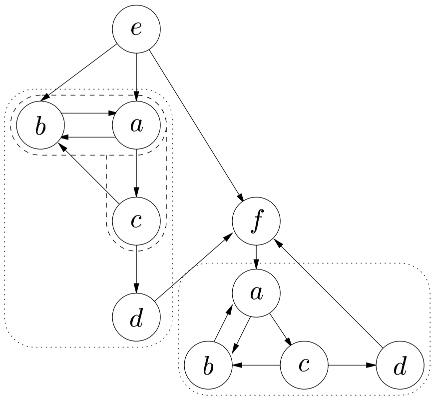
If f is a function over the nodes of any control flow graph, then $f(X)$, where X is a subset of these nodes, stands for the set $\{f(x)|x \in X\}$.

Definition 3.12 (Transformation $\mathbf{T}_r$)
Let $G = (N, E, s)$ be an arbitrary (irreducible) control flow graph, L an SED-maximal, irreducible loop-set of G, K its MSED-set, e the external dominator and h an arbitrary node from K. Then the transformation $G' = (N', E', s') = T_r(G, L, h)$ is defined as follows (with $S = (L \setminus \mathrm{domain}(h))$):

- $N' = (N \times \{1\}) \cup (S \times \{2\})$
- *$E' \subset N' \times N'$ such that the following restrictions hold:*
  - $(x, y) \in E \wedge (x, y) \notin (\mathrm{domain}(h) \times S) \iff ((x, 1), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (\mathrm{domain}(h) \times S) \iff ((x, 1), (y, 2)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times (N \setminus S)) \iff ((x, 2), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times S) \iff ((x, 2), (y, 2)) \in E'$



(a)



(b)

Figure 7: One step of the recursive algorithm

- $s' = (s, 1)$

The above transformation represents one step of the algorithm. All nodes of the loop-set L except those in the selected header node's domain are split. The new copies of these nodes are represented by the syntactical construction $S \times \{2\}$ while the old nodes are represented by $N \times \{1\}$. The labeling and weight of the new graph can be easily defined based on the labeling and weight of the original graph:

Definition 3.13 (Corresponding Labeling and Weight)
If l is an labeling of G, then the corresponding labeling *l' of G' is defined as*

$$l'((x, \cdot)) = l(x)$$

If σ is a weight-function of G, then the corresponding weight-function *σ' of G' is defined as*

$$\sigma'((x, \cdot)) = \sigma(x)$$

Here the notation $(x, \cdot)$ stands for any tuple (x, y).

For the proof of correctness we assume no particular order in which T_r is applied. The algorithm just chooses an SED-maximal, irreducible loop-set L and a header node h of L and applies $\mathrm{T}_r(G, L, h)$. Then the algorithm will finally construct an equivalent reducible flow graph. Because $\sim$ is transitive, it suffices to show that the following theorem holds in order to show the partial correctness of the algorithm.

Theorem 3.14 (Partial correctness)
Let G be an arbitrary (irreducible) control flow graph, L an SED-maximal, irreducible loop-set of G, K its MSED-set, e the external dominator and h an arbitrary node from K.

Then $G \sim T_r(G, L, h)$ with respect to the labeling l and the corresponding labeling of $T_r(G, L, h)$.

Proof: $\Longrightarrow$

Let $p = (p_1, \ldots, p_n)$ be an arbitrary path of G. Then $q = (q_1, \ldots, q_n)$ with

$$q_1 = (p_1, 1)$$

$$q_{i+1} = \begin{cases} (p_{i+1}, 2) & \text{if } p_{i+1} \in S \wedge \big(q_i = (p_i, 2) \vee p_i \in \text{domain}(h)\big) \\ (p_{i+1}, 1) & \text{otherwise} \end{cases}$$

is a path of $\mathrm{T}_r(G, L, h)$ with $\forall 1 \leq i \leq n : l(p_i) = l_r(q_i)$ where l_r is the corresponding labeling of $\mathrm{T}_r(G, L, h)$.

$\Longleftarrow$

Let $q = (q_1, \ldots, q_n)$ be a path of $\mathrm{T}_r(G, L, h)$. From the construction of $\mathrm{T}_r(G, L, h)$ follows that $q_i = (p_i, k)$ with $k \in \{1, 2\}$ and if $(q_i, q_{i+1}) \in E'$, then $(p_i, p_{i+1}) \in E$. Thus, $p = (p_1, \ldots, p_n)$ is a valid path of G and by Definition 3.13 $l_r(q_i) = l(p_i)$, where l_r is again the corresponding labeling. $\square$

If the algorithm applies T_r as long as there is any irreducible loop-set, then the proof of reducibility and termination become equivalent. Therefore, to complete the proof of correctness, it is sufficient to show that T_r finally constructs a reducible flow graph.

Theorem 3.15 (Termination)
Let $G = (N, E, s)$ be an arbitrary (irreducible) control flow graph. Then, if G is repeatedly transformed by $T_r(G, L, h)$, where L is an arbitrary irreducible loop-set of G and h is a header node of L, then after a finite number of such transformations G will be reducible.

Proof: From Theorem 3.7 follows that for each irreducible loop-set L' with $L' \neq L$ one of the following is true:

- $L' \cap L = \emptyset$
  But then the nodes and edges of L' are completely unaffected by the application of T_r.
- $L' \supset L$
  But then $L \subset \mathrm{domain}(h')$ for some $h' \in$ MSED-set(L') and the changes made by T_r are restricted to $\mathrm{domain}(h')$ and do not affect MSED-set(L').
- $L' \subset L$
  Then $L' \subset \mathrm{domain}(h')$ for some $h' \in$ MSED-set(L). If $h' = h$, then L' is completely unaffected by the application of T_r. Otherwise $L' \subset S$ and thus has been split. However, since all edges *within* S have been duplicated on both copies $S \times \{1\}$ and $S \times \{2\}$, L' is present twice in $\mathrm{T}_r(G, L, h)$ and both copies still have the same number of header-nodes.

However, since $(h, 1)$ dominates all nodes in $S \times \{2\} \cup \mathrm{domain}(h) \times \{1\}$ and there is no edge from $S \times \{2\} \cup \mathrm{domain}(h) \times \{1\}$ to $S \times \{1\}$, any remaining irreducible loop-set in these regions is either a nested loop-set that was already present in G or (since all of the original header nodes still dominate their respective domains) this loop set must have at least one less header node than L.

Thus, on every application of T_r one loop-set's MSED-set becomes smaller by one node, while the MSED-sets of all other loop-sets are unaffected. Though some loop-sets are duplicated, these can only be subsets of the one that got smaller. And finally, each irreducible loop-set will become a top-level one and then can itself only become smaller. Therefore, all loop-sets will eventually have an MSED-set with only one node and thus will be reducible.$\square$

This means, that any flow graph can be converted to an equivalent and reducible one by the repeated application of T_r alone. It remains to be shown that there is always a sequence of transformations that leads to a minimal final graph.

To prove this, a sequence is constructed from an arbitrary minimal graph. This, of course, is not a constructive proof as it cannot be used to transform a flow graph into a minimal equivalent one without knowing the final graph beforehand. However, it is still important to prove the above property.

Throughout the following proof several symbols will be used without being defined over and over again. These are:

- $G = (N, E, s)$ stands for the flow graph that is being transformed. For each iteration G is the graph *before* T_r is applied.

- $G' = (N', E', s')$ is the result of the application of T_r in the current iteration.
- $G_{min} = (N_{min}, E_{min}, s_{min})$ is a minimal, reducible graph equivalent to G.
- l, l' (the corresponding labeling) and l_{min} are labelings of G, G' and G_{min} respectively.
- L is an irreducible, SED-maximal loop-set of G
- h is a header node of L.
- $S = L \setminus \mathrm{domain}(h)$
- b is a total function $b : N_{min} \to N$, such that $\forall n \in N_{min} : l_{min}(n) = l(b(n))$.
- b' is a total function $b' : N_{min} \to N'$, with $\forall n \in N_{min} : l_{min}(n) = l'(b'(n))$.

These functions b and b' will be used to map each node of G_{min} to the node of G and G', respectively, from which it is a copy.

The proof will construct a minimal sequence by choosing an L of G within some constraints. This L will then be looked up in G_{min} to find the node h for the transformation $\mathrm{T}_r(G, L, h)$. For this look-up it is necessary to define, which loop-set in G_{min} is related to L. This is done in the next definition:

Definition 3.16 (Equivalent Loop-set)
Let G, G_{min}, L, l, l_{min} and b be defined as above.

Then a loop-set L_{min} of G_{min} is called an equivalent loop-set of L, if and only if L_{min} is maximal such that

$$b(L_{min}) = L$$

Note that the L_{min} is not necessarily unique. However, if b is constructed as below, then there is always at least one equivalent loop-set.

For the initial graph, where l is an injective labeling of G, b is constructed as

$$\forall n \in N_{min} : b(n) = l^{-1}(l_{min}(n)).$$

Then, in each iteration, L is selected such that there is exactly one equivalent loop-set L_{min} of L. It will be shown later that such an L does always exist.

Since L_{min} is a loop-set of G_{min}, which is reducible, it must be a reducible loop-set itself and thus have a single entry node h'. Then, $h = b(h')$ is one of the header nodes of L and, therefore, the graph G' can be constructed by $G' = \mathrm{T}_r(G, L, h)$. The labeling and weight of G' has already been defined. The function $b' : N_{min} \to N'$ is defined as

$$\forall n \in N_{min} : b'(n) = \begin{cases} (b(n), 2) & \text{If } n \in L_{min} \wedge b(n) \in S \\ (b(n), 1) & \text{otherwise} \end{cases}$$

Thus a new iteration can be done with G', l', σ' and b' as G, l, σ and b respectively, if G' is still irreducible.

For the above algorithm to be correctly defined, the following three things have to be shown:

- There is always a loop-set L with exactly one equivalent loop-set. This is proved in Theorem 3.24.
- $b(h')$ is always a header node of L, which is proved in Theorem 3.25.
- For the finally constructed (reducible) graph G: $\sigma(G) = \sigma_{min}(G_{min})$, which is shown in Theorem 3.26.

The last point is not strictly needed for the above algorithm but it will show that the constructed graph is minimal.

To be able to prove the above items, several lemmata will be needed.

Lemma 3.17 (s is unique)
Let G, G_{min}, l, l_{min} and b be defined as above. Then

(i) $\nexists n \in N_{min} \setminus \{s_{min}\} : l_{min}(n) = l_{min}(s_{min})$ *and*

(ii) $\nexists n \in N \setminus \{s\} : l(n) = l(s)$.

Proof: (i) Since s_{min} dominates all nodes of G_{min}, any edge $(u, n) \in E_{min}$ where $l_{min}(n) = l_{min}(s_{min})$ can be replaced by the edge (u, s_{min}) without destroying the reducibility or equivalence of G_{min}. But then n has no longer any incoming edge and can be removed from G_{min} thus making G_{min} lighter. This is a contradiction to the assumption, that G_{min} is minimal. Therefore, such a node n cannot exist.

(ii) This is obviously true for the initial graph G with an injective labeling. Since s dominates all nodes in G, it cannot be in any irreducible loop and, therefore, cannot be split by the algorithm. Thus the above property holds through all iterations of the algorithm.

□

Flow graphs, which are equivalent to a graph with an injective labeling have a special property, which is much stronger than simple equivalence as defined in Definition 2.3.

Lemma 3.18
Let G, G', G_0 be flow graphs with labelings l, l' and l_0 respectively such that $G \sim G' \sim G_0$. Furthermore let l_0 be injective and let G and G' be such that the property of Lemma 3.17 holds. Then the following is true:

For all $q_0 \in N'$ and for each path $p = (p_0, \ldots, p_k)$ of G with $l'(q_0) = l(p_0)$ exists exactly one path $q = (q_0, \ldots, q_k)$ of G' with $l'(q_i) = l(p_i)$

Proof: Since G' was supposed to not contain dead code, there is a path $q' = (q'_{-j}, \ldots, q'_0)$ of G' with $q'_{-j} = s'$ and $q'_0 = q_0$. Since $G' \sim G_0$ there must be a path $\overline{q} = (\overline{q}_{-j}, \ldots, \overline{q}_0)$ of G_0 with $l_0(\overline{q}_i) = l'(q'_i)\ \forall -j \leq i \leq 0$. Since l_0 is injective $\overline{q}_{-j} = s_0$. Since $G \sim G_0$ there must be a path $\overline{p} = (\overline{p}_0, \ldots, \overline{p}_k)$ of G_0 with $l_0(\overline{p}_i) = l(p_i)\ \forall 0 \leq i \leq k$. Furthermore, since l_0 is injective there can only be one such path $\overline{p}$ and $\overline{p}_0 = \overline{q}_0$. Thus $\overline{qp} = (\overline{q}_{-j}, \ldots, \overline{q}_0, \overline{p}_1, \ldots, p_k)$ is a path of G_0 and because $G_0 \sim G'$ there must be a path $r = (r_{-j}, \ldots, r_0, \ldots, r_k)$ of G' such that $l'(r_i) = l_0((\overline{qp})_i)$ $\forall -j \leq i \leq k$. All that remains to be shown is that $r_0 = q_0$ and thus that $q = (r_0, \ldots, r_k)$ is a path as requested.

From $l_0(s_0) = l_0((\overline{qp})_{-j}) = l'(r_{-j}) = l(s')$ and from the property of Lemma 3.17 follows that $r_{-j} = s' = q'_{-j}$. From Definition 2.2 follows that $r_i = q'_i \longrightarrow r_{i+1} = q'_{i+1}$ and thus (by induction) that $r_0 = q'_0 = q_0$. From Definition 2.2 also follows that q is the only path originating from q_0 following the same labels as p.$\square$

Lemma 3.19

Let G, G_{min}, l, l_{min} and b be defined as above.
Then, for every path $q = (q_1, \ldots, q_k)$ of G_{min}, $(b(q_1), \ldots, b(q_k))$ is a path of G.

Proof: From Lemma 3.18 and Lemma 3.17 follows that there is exactly one path $p = (p_1, \ldots, p_k)$ of G such that $p_1 = b(q_1)$ and $l(p_i) = l_{min}(q_i)$. From $\forall n \in N_{min} : l(b(n)) = l_{min}(n)$ and Definition 2.2 follows that $p_i = b(q_i)$.$\square$

Lemma 3.20 (b is surjective)

Let G, G_{min}, l, l_{min} and b be defined as above.
Then b is surjective.

Proof: G was supposed to not contain dead code
$\Rightarrow \forall n \in N$ there is a path $p = (p_1, \ldots, p_k)$ of G with $p_1 = s$ and $p_k = n$. Since $G \sim G_{min}$ and Lemma 3.18 there is exactly one path $q = (q_1, \ldots, q_k)$ of G_{min} with $q_1 = s_{min}$ and $l(p_i) = l_{min}(q_i)$. Thus $b(q_1) = p_1$ and from Definition 2.2 follows that $b(q_i) = p_i$ and thus that $b(q_k) = n$$\square$

Lemma 3.21

Let G be an arbitrary control flow graph. Any node $x \in N$ is outside of all irreducible loops if and only if the following holds:

$\forall y \in N$ with x is reachable from y and y is reachable from x: $\exists z \in N$ such that z dominates both x and y and z either occurs on every path from x to y or on every path from y to x (or both).

Furthermore, if x is not in any irreducible loop, then the above nodes z are in no irreducible loop, either.

The proof uses the following definition that determines if a node is inside of an irreducible loop:

x is inside of an irreducible loop if and only if it is inside a loop-set whose MSED-set contains more than one node.

Proof: $\Longrightarrow$

Since x and y are reachable from each other, there is at least one loop-set that contains both nodes. Let L be the smallest SED-maximal loop-set with $x, y \in L$. This L is well defined because of Theorem 3.9. Since $x \in L$, L must be reducible and thus have a single entry node z with z dominates all nodes in L. If there is a path from x to y that does not touch z, then all nodes on this path are part of L (because z dominates all these nodes and L is SED-maximal). The same is true for any path from y to x that does not touch z. However, if two such paths really existed, they would form a loop-set L' that does not include z and with $L' \subset L$. This, in turn, means that there is another SED-maximal loop-set, which is strictly smaller (with respect to inclusion) than L and contains both x and y. This is a contradiction to the assumption that L is the smallest SED-maximal loop-set.

Furthermore, from Theorem 3.9 and the minimality of L follows that for each loop-set L' with $z \in L'$: $L \subseteq L'$ and thus $x \in L'$ and thus that L' is reducible. Therefore, z is not part of any irreducible loop, either.

$\Longleftarrow$ (the negation)
If x is inside an irreducible loop L it must be in domain(h) for some $h \in$ MSED-set(L). Because L is irreducible, there is a node $y \in$ MSED-set(L) with $y \neq h$. Because $x, y \in L$, x is reachable from y and y is reachable from x without leaving L. However, since $\forall z \in L, z \neq y$: z does not dominate y and since y does not dominate x the following is true:
$\forall z' \in N$ with z' dominates x and y: $z' \notin L$ and thus there are paths from x to y and from y to x that do not touch z'. $\square$

Lemma 3.22 (b is injective)
Let G, G_{min}, l, l_{min} and b be defined as above.
Then b is injective outside of irreducible loops. This means that the following is true:
$\forall m, n \in N_{min}$ with $b(m) = b(n)$ and $b(m)$ is not inside any irreducible loop: $m = n$.

Proof: Induction over the depth of $b(m)$ in the dominator-tree.

Induction-base:
From Lemma 3.17 follows directly that the above lemma is true for $m = s_{min}$.

Induction-step:
Indirect: Assume there exist nodes $m, n_0 \in N_{min}$ such that $b(m) = b(n_0)$ and $b(m)$ is not inside any irreducible loop but $m \neq n_0$. However, it is assumed, that the lemma is true for all such nodes m' and n' where $b(m')$ is an ancestor of $b(m)$ in the dominator-tree of G.

The following proof will show that, if such m and n_0 exist, then G_{min} cannot be minimal because it is possible to construct another reducible, equivalent flow graph, which is lighter with respect to σ:

Let m and n_0 be as assumed above. Then the graph G'_{min} is constructed as follows:

- $s'_{min} = s_{min}$
- $$E'_{min} = E_{min} \quad \setminus \quad \left\{ (\cdot, n) \in E_{min} \;\middle|\; \begin{array}{l} b(n) = b(m) \wedge \\ n \neq m \end{array} \right\}$$
$$\cup \quad \left\{ (n_1, m) \;\middle|\; \exists (n_1, n_2) \in E_{min} : \begin{array}{l} b(n_2) = b(m) \wedge \\ n_2 \neq m \end{array} \right\}$$
- $$N'_{min} = N_{min} \quad \setminus \quad \left\{ n \in N_{min} \;\middle|\; \begin{array}{l} \nexists \text{ a path } p = (p_1, \ldots, p_k) \text{ in} \\ E'_{min} : p_1 = s_{min} \wedge p_k = n \end{array} \right\}$$

In short, the above construction replaces all edges to any node n with $b(n) = b(m)$ except to m itself with appropriate edges to m. After that the other copies of m are dead-code and can be removed. Since the removed edges have been replaced by edges to a node with the same label and the removed nodes are just those that are no longer reachable, the new graph is equivalent to the old one, i.e. $G'_{min} \sim G_{min}$.

Since at least the node n_0 has been removed from N'_{min}, the above control flow graph is equivalent to and lighter than G_{min}. This would only be possible if it were not reducible since G_{min} was assumed to be minimal. Therefore, it is sufficient to prove that G'_{min} is reducible in order to show, that such m and n_0 cannot exist.

However, if G'_{min} is compared to G_{min}, only a few nodes and edges have been removed and some other edges have been added. But neither the removed edges nor the removed nodes may have introduced any irreducibility. Therefore, if G'_{min} were irreducible, this irreducibility would have been introduced by the *added* edges. These, however, are all targeted at m. Therefore, if G'_{min} is irreducible, m must be part of at least one of the irreducible loops.

Thus it is sufficient to show that m is outside of all irreducible loops. By Lemma 3.21 this can be shown by proving that:

$\forall y \in N$ with m is reachable from y and y is reachable from m: $\exists z \in N$ such that z dominates both m and y and z either occurs on every path from m to y or on every path from y to m (or both).

Let $y \in N'_{min}$ be such that y is reachable from m and m is reachable from y in G'_{min}. By the construction of G'_{min} these two nodes must also be reachable from each other in G_{min} and thus by Lemma 3.19 $b(m)$ is reachable from $b(y)$ and vice versa. Since $b(m)$ is not in any irreducible loop (consider the prerequisites of the lemma) and because b is surjective according to Lemma 3.20 there is a node $z \in N_{min}$ with $b(z)$ dominates $b(m)$ and $b(y)$, and $b(z)$ either occurs on every path from $b(m)$ to $b(y)$ or on every path from $b(y)$ to (m) (or both).

Assume first that $b(z) \neq b(m)$.
Then $b(z)$ is an ancestor of $b(m)$ in the dominator-tree and by Lemma 3.21 not in any irreducible loop, either. Therefore, b is injective at $b(z)$ and thus $\not\exists z' \in N_{min}$ with $b(z') = b(z)$. From this and Lemma 3.19 follows that z dominates y and all nodes $n \in N_{min}$ with $b(n) = b(m)$. Because $b(z) \neq b(m)$ and the above, $z \in N'_{min}$ and z dominates m in G'_{min}. This, however, means that the added edges to m cannot influence the dominance of z over y. Thus, z dominates both m and y in G'_{min}.

By Lemma 3.19 and the uniqueness of z, if $b(z)$ occurred on every path from $b(m)$ to $b(y)$, then z occurs on every path from m to y in G_{min} and thus on every path from m to y in G'_{min}. On the other hand, if $b(z)$ occurred on every path from $b(y)$ to $b(m)$, then, again by Lemma 3.19 and the uniqueness of z, z occurs on every path of G_{min} from y to any node n with $b(n) = b(m)$ and thus it occurs on every path of G'_{min} from y to m. Thus m is not in any irreducible loop in G'_{min} and therefore, G'_{min} is completely reducible.

On the other hand, if $b(z) = b(m)$, then $b(m)$ dominates $b(y)$. Thus, by Lemma 3.19, on every path of G_{min} from s_{min} to y a node $n \in N_{min}$ with $b(n) = b(m)$ must occur. This, in turn, means that m must occur on every path of G'_{min} from s'_{min} to y. Thus m dominates y and occurs on every path from m to y and vice versa. Therefore, m cannot be in any irreducible loop and thus G'_{min} must be completely reducible.

As was said above, this is not possible if G_{min} was truly minimal and thus such nodes m and n_0 cannot exist. $\square$

Lemma 3.23 (Existence of equivalent loop-sets)
Let G, G_{min}, l, l_{min} and b be defined as above.
Then for each loop-set L of G, there is an equivalent loop-set L_{min} of G_{min}.

Proof: Since G was supposed to not contain dead code and since L is a loop-set there is at least one infinitely long path $p = (p_1, p_2, \ldots)$ of G such that $p_1 = s$ and $\exists k \forall j > k : p_j \in L$ and $\forall k \forall n \in L \exists j > k : p_j = n$. From Lemma 3.18 follows that there is a path $q = (q_1, q_2, \ldots)$ of G_{min} such that $q_1 = s_{min}$ and $l(p_i) = l_{min}(q_i)$. Since G_{min} is finite there must be a loop-set L_{min} such that $\exists k' \forall j > k' : q_j \in L_{min}$ and $\forall k \forall n \in L_{min} \exists j > k : q_j = n$. From Lemma 3.19 follows that $b(q) = (b(q_1), b(q_2), \ldots)$ is a path of G following the same labels as p. Since, furthermore, $b(q_1) = s = p_1$ follows from Definition 2.2 that $b(q) = p$ and thus that $b(L_{min}) = L$.□

With the above lemmata the remaining properties as listed on page 22 can be proven.

Theorem 3.24 (Uniqueness of equivalent loop-sets)
Let G, G_{min}, l, l_{min} and b be defined as above.
Then, there is always at least one loop-set L of G, which has exactly one equivalent loop-set L_{min}.

Proof: Let L be a loop-set of G such that the external dominator e of L is not in any irreducible loop. Since s is never in any irreducible loop and because of Theorem 3.9 and because G is finite, such a loop-set must always exist. By Lemma 3.23, L must have at least one equivalent loop-set. Thus it is enough to show that L has at most one equivalent loop-set.

This is shown indirectly: Assume there are two different equivalent loop-sets $L_{min,1}$ and $L_{min,2}$. Because equivalent loop-sets are maximal, either on every path from $L_{min,1}$ to $L_{min,2}$ or on every path from $L_{min,2}$ to $L_{min,1}$ a node n with $b(n) \notin L$ must occur. Without restricting generality it is assumed that it occurs on every path from $L_{min,1}$ to $L_{min,2}$. Let m be the header-node of $L_{min,1}$. Then another flow graph G'_{min} can be constructed as in the proof for Lemma 3.22 except that only edges to nodes n with $b(n) = b(m)$ and $n \in L_{min,2}$ are replaced. Then, for the same reasons, $G'_{min} \sim G_{min}$ and since $b(L_{min,2}) = L$ at least one such node n has been removed. Therefore, it is again enough to show that G'_{min} is reducible in order to prove that such two equivalent loop-sets cannot exist.

Since only edges to m have been added, the above assumption about paths from $L_{min,1}$ to $L_{min,2}$ is still true in G'_{min}.

As in the proof of Lemma 3.22, if G'_{min} is irreducible, this irreducibility could only have been introduced by the added edges to m. Thus m would have to be in at least one irreducible loop-set. It is therefore enough to show that

$\forall y \in N'_{min}$ with m and y are reachable from each other: $\exists z \in N'_{min}$ such that z dominates y and m and z either occurs on every path from m to y or on every path from y to m.

Let y be an arbitrary node in G'_{min} such that y and m are reachable from each other. If $y \in L_{min,1}$, then m dominates y and thus $z = m$ is such a node. Therefore, $y \notin L_{min,1}$. Then on every path $p = (m, \ldots, y, \ldots, m)$ a node n with $b(n) \notin L$ must occur, since $L_{min,1}$ is maximal. Thus there is an edge (p_i, p_{i+1}) in p such that $b(p_i) \in L$ and $b(p_{i+1}) \notin L$. From Theorem 3.11 follows that $b(p_{i+1}) \notin e$-domain. This means that $\exists j > i : b(p_j) = e$. However, since e is not in any irreducible loop, there is only one node e_{min} with $b(e_{min}) = e$ and thus e_{min} occurs on every path $p = (m, \ldots, y, \ldots, m)$.

This, however, means that for each loop-set L' of G'_{min} with $m, y \in L'$: $e_{min} \in L'$. Let L'_0 be the smallest such loop-set with respect to inclusion. This is well defined because of Theorem 3.9. From Lemma 3.19 follows, that $b(L'_0)$ is a loop-set and $e \in b(L'_0)$. Thus $b(L'_0)$ is reducible and has a single header-node. Since b is surjective, there is a node $z \in N_{min}$ such that $b(z)$ is that header-node and thus dominates all nodes in $b(L'_0)$. Furthermore, because $b(z) \notin L$, $z \in N'_{min}$. Because of Theorem 3.9 $b(z)$ is not in any irreducible loop and thus z is unique. This, however, means that z dominates all nodes in L'_0 (including y and m) and thus that L'_0 is reducible and z is its single entry-node. Then, however, z must either occur on every path from m to y or on every path from y to m because L'_0 was the smallest loop-set containing y and m.

Thus m cannot be in any irreducible loop and G'_{min} is completely reducible. This is not possible if G_{min} is minimal as assumed and thus L cannot have two (or more) equivalent loop-sets.□

Theorem 3.25 ($b(h')$ is a header node)
Let G, G_{min}, l, l_{min} and b be defined as above.
Let L be a SED-maximal loop-set of G, e its external dominator and e not in any irreducible loop-set. Let L_{min} be the equivalent loop-set of L and h' the single entry-node of L_{min}.

Then $b(h') \in$ MSED-set(L).

Proof: Indirect.

Let L be such a loop-set, L_{min} its equivalent loop-set and h' the entry node of L_{min} with $b(h') \notin$ MSED-set(L). Then by Theorem 3.7 $b(h') \in$ domain(h) for some $h \in$ MSED-set(L). Because L_{min} is an equivalent loop-set, there is a node $m \in L_{min}$ with $b(m) = h$. On the other hand, since h dominates $b(h')$ in G, on every path from s_{min} to h' a node n with $b(n) = h$ must occur. Furthermore, because h' is the entry node of

L_{min}, $n \notin L_{min}$ and thus $n \neq m$. Thus, if G'_{min} is constructed as in the proof for Lemma 3.22, then $G'_{min} \sim G_{min}$, for the same reasons as given there, and $\sigma(G'_{min}) < \sigma(G_{min})$ since at least n has been removed. Thus it is sufficient to show that G'_{min} is reducible in order to prove that h' must be a header node of L.

If G'_{min} was reducible, m had to be in at least one irreducible loop, since only edges to m have been added. Thus, all that remains to be shown is that:

$\forall y \in N'_{min}$ with m and y are reachable from each other: $\exists z \in N'_{min}$ such that z dominates y and m, and z either occurs on every path from m to y or on every path from y to m.

Let y be such a node.
Since $h = b(m)$ dominates $b(h')$ in G and since h' dominates all nodes in L_{min}, on every path of G_{min} from s_{min} to any node in L_{min} at least one node n with $b(n) = h$ must occur. Since all added edges are edges to m and $b(m) = h$, this must still be true for G'_{min}. Thus: On every path of G'_{min} from s'_{min} to any node in $L_{min} \cap N'_{min}$ at least one node n with $b(n) = h$ must occur. However, by the construction of G'_{min}, m is the *only* node in N'_{min} with $b(m) = h$ and thus m dominates all nodes of $L_{min} \cap N'_{min}$. Therefore, if $y \in L_{min}$, then $z = m$ fulfills the above requirements.

Thus, let $y \notin L_{min}$.
Since $b(y) \in L$ and L_{min} is the only equivalent loop-set of L, there must be a path q of G_{min} from y into L_{min} such that $b(q_i) \in L$ for all i. However, since $y \notin L_{min}$, this means that on every path of G_{min} from any node in L_{min} to y, a node n with $b(n) \notin L$ must occur. In particular, this is true for m and since only edges to m have been added in G'_{min} it is also true that: On every path $p = (m, \ldots, y)$ of G'_{min} a node n with $b(n) \notin L$ must occur. Since the same applies to every path $p = (m, \ldots, y, \ldots, m)$ the arguments of the proof of Theorem 3.24 apply here as well and there is a node z, which fulfills the above requirements.

Therefore, m cannot be in any irreducible loop and G'_{min} must be completely reducible. Since this is a contradiction to G_{min} minimal, h' must be a header node of L□

Theorem 3.26 (The constructed graph is minimal)
Let G, G_{min}, l, l_{min} and b be defined as above. Furthermore, let G be the final graph, namely let G be reducible.
Then $\sigma(G) = \sigma_{min}(G_{min})$.

Proof: By Lemma 3.20 b is surjective.
Since G is completely reducible, b is completely injective by Lemma 3.22.
Thus b is bijective and thus $\sigma(G) = \sigma_{min}(G_{min})$.□

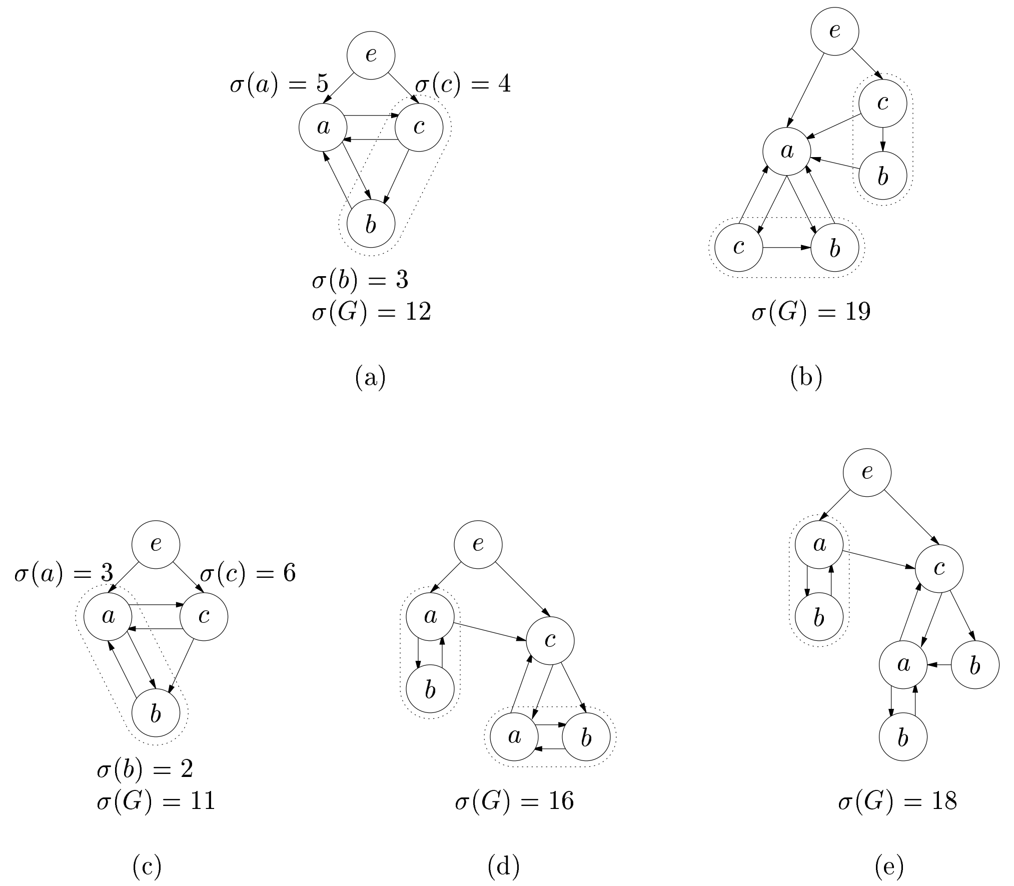


Figure 8: Two possible traces of the recursive alg.

The previous theorems show that there is always a sequence that constructs a minimal, equivalent and reducible flow graph for any control flow graph G. The proof, however, already used such a minimal graph for the selection of the header node, on which the transformation T_r is applied and, therefore, does not provide a usable method for choosing the nodes. The Theorems 3.14 and 3.15, on the other hand, were proven independently of any order, and, therefore, if another selection scheme is used, the algorithm remains correct, though possibly not minimal.

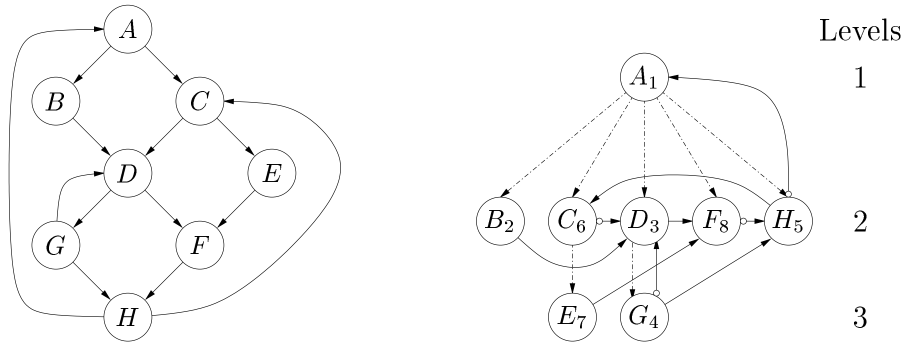
In the prove above, it was necessary to handle the *outer* loops first. This, however, is not really necessary. It is possible to give a similar (but even more technical and incomprehensible) proof that does not depend on any order in which the irreducible loop-sets L are chosen.

Figure 8 shows the steps of the algorithm for two different selections of the h_i. Which of the resulting flow graphs is minimal depends on the weight of the nodes. The following table lists different weights and the corresponding minimal graph.

Weight of			Minimal		
a	b	c	sequence	graph	weight
5	3	4	a	8(b)	19
4	3	5	a	8(b)	20
3	2	6	c, a	8(e)	18

As can be seen from this table, the weight of the nodes alone is not sufficient to decide which node has to become h_i in each step. The resulting number of copies for other nodes has to be considered as well. Though it might be possible to deduce that number from the structure of the MSED-set without actually constructing each possible final graph, such a method could not be found.

Still, the weight of the nodes is known from the beginning and can be used as a heuristic for the selection of the h_i. The algorithm given in the next section uses such a heuristic by choosing the header h_i such that $\sigma(\text{domain}(h_i))$ is maximal for all possible h_i. This, however, requires that the final weight of the domains is known and thus that the inner (nested) loops are converted first.



(a) Flow graph

(b) DJ graph with dfs-numbers and `sp_back` edges marked

Figure 9: An example of a flow graph and its DJ graph

4 Operational Description of the Solution

In this section, the algorithm is presented in a more operational form. First, a solution is given for the problem of finding SED-maximal loop-sets and MSED-sets. Then the algorithmic idea of the last section is presented in a pseudo-code language that can be easily translated to a real life programming language.

4.1 Dominator Jump Graphs

Sreedhar et. al. [8] present an algorithm that detects irreducible loops and their nesting using a structure called Dominator Jump Graphs (DJ-Graphs). Since these graphs, together with a modified version of the algorithm, can be used to implement the method given in the last section they will be introduced here.

A DJ-graph is a directed graph whose nodes are those of the control flow graph. Its edges, however, are the edges of the dominator tree plus the back- and cross-edges of the control flow graph. Back-edges are, as usual, the edges where the target dominates the source. Cross-edges are those edges of the original control flow graph where the target and the source are not related with respect to dominance. Figure 9 gives the same example as given by Sreedhar [8].

To detect irreducible loops, a depth first search over the DJ-graph is used to build a spanning tree. In this spanning tree all edges from a node n to an ancestor node of n are marked as `sp_back` edges. The algorithm now proceeds through the DJ-Graph from the bottom upwards considering one level at a time. If any edges are found that are cross-edges as well as marked `sp_back`, this indicates the presence of an irreducible loop at that level. If such an edge is found all strongly connected components (SCC) on that level and any higher numbered one are

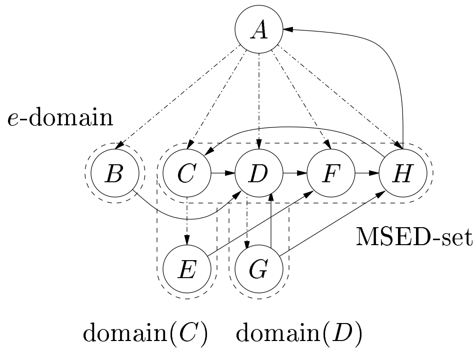


Figure 10: The DJ graph and its relation to the structure of IR-loops

constructed. Each such SCC, that is not trivial, is then a possibly irreducible loop. In Figure 9(b) there are four edges marked `sp_back`. However, only two of them are also cross-edges and thereby indicate the presence of an irreducible loop.

The algorithm given by Sreedhar [8] collapses every loop found into a single abstract node and proceeds with the next level. That way it detects all loops as well as their nesting. But not only the nesting of irreducible loops can be detected using DJ-graphs. All the structures as defined in Section 3.1 can be directly retrieved from DJ-graphs.

As was already said, each SCC is one loop-set. Furthermore, these SCCs are always SED-maximal loop-sets. All nodes in the SCC that are at the highest level make up the MSED-set of the loop. The other nodes of the SCC make up the domains. Figure 10 shows the relation between DJ-graphs and the structure of irreducible loops.

Since the nodes of an MSED-set share the same immediate dominator they are all children of the same node in the dominator tree. The nodes E and G, for example, could not be in the same MSED-set. Therefore it is not necessary to handle one complete level at a time. Instead, it is sufficient if all levels below a given node are handled before this node and its siblings are considered. This can be done by traversing the graph in a depth-first order. Another advantage of that approach is that the recursive handling of copied regions can be easily implemented. However, if the algorithm is called recursively it does not need to perform an entire depth-first search again. It is sufficient to stay within the SCC that was just split.

Another minor modification replaces the depth-first search over the DJ-Graph by a depth-first search over the control flow graph to mark `sp_back` edges. However, the two searches are equivalent.

The algorithm to find all SCCs that was used by Sreedhar and Lee [8] can be kept almost unmodified. It is a rather simple but effective one. First, a depth-first search over the control flow graph is done, starting with the nodes that contain the MSED-set for the irreducible loop indicated by the `sp_back` edges. The nodes are numbered in a post-traversal order. In a second pass, a depth-first search on the *inverse* graph is done, always starting at the highest numbered node from pass one that was not yet visited. Each sub-tree found then constitutes an SCC.

4.2 Notation

This section gives an introduction to the pseudo code language, the data types and external functions used by the operational description of the algorithm.

The notation utilizes three simple or basic types: `bool`, `integer` and `void`. The last type is specified as the return type of functions that do not actually return anything. Besides these, a complex list type can be build with the construct `list of <type>` and manipulated with several of the functions below. The last type is the type used for a node. It is some sort of structure whose details are hidden behind the type `node`. However, the code assumes that at least the following members exist in the structure:

Members of the structure `node`		
Type	Member name	Description
`list of node`	`preds`	These two implement the control flow graph,
`list of node`	`succs`	
`node`	`idom`	while these are used for the dominator tree.
`list of node`	`succs_dom`	
`integer`	`level`	The depth in the dominator tree as shown in Figure 9(b).
`integer`	`weight`	This is calculated whenever a irreducible loop is found. For the header nodes it then contains the weight of the entire domain.
`node`	`copy`	If a region is split first all nodes in that region will get a copy pointed to by this member. After that the links are adjusted.
`node`	`header`	For every node in the loop currently handled this points to the corresponding header node. The header node points to itself.
`bool`	`done`	Used in various routines that traverse the different graphs.
`bool`	`active`	The algorithm to detect irreducible loops in a DJ-graph should build a spanning tree. This field is used during the depth first search to determine whether a successor node is an ancestor of the current node. That way, the spanning tree does not have to be constructed in truth.

Members of the structure `node` (continued)		
Type	Member name	Description
`<any>`	`sp_back_data`	There are three functions that manipulate this field. The code does not make any assumptions about how these are implemented with the exception that the data of marked edges is somehow saved in the source nodes of the edges.

Note that the type `node` and any list types are reference types. That means that all arguments to functions of these types are assumed to be "call by reference"-arguments. If two variables of a reference type are compared, this yields only true if they refer to exactly the same object.

The only construct of the language that may not be straight forward is:

```
foreach (<var> in <list-var>) { <statements> }
```

which assigns the elements of `list-var` to `var` in the reverse order they were added and executes `statements` for each element.

In the boolean expressions of `if`-statements the normal C-operators are used: `==` for equality, `&&` for logical *and*, `||` for logical *or* and `!` for logical *not*.

Besides these structures of the language itself, the algorithm uses several functions and two global symbols that are not defined. These functions either depend on the target system and language or implement algorithms that are well known and that do not need to be modified in any way. Nonetheless, they are listed below together with a short operational description. The two global symbols are `NULL`, which can be used for any reference type and stands for a nonexistent object, and `start`, which is the node s of the control flow graph.

Functions that manipulate lists		
Function name	Arguments	Description
`is_empty`	`list`	The name should be self-descriptive.
`new_list`	`type`	Returns a reference to a new, empty `list of type` object.
`add_list`	`type, list of type`	Adds an element to the head of the list.
`rmv_list`	`type, list of type`	Removes an element from the list.
`in_list`	`type, list of type`	Checks if an element is in the list.
`size_list`	`list of type`	Returns the number of elements in the list. However, this is called only once to check if there are one or more elements. Maybe this test can be implemented more efficiently.

Functions that mark edges		
Function name	Arguments	Description
remove_marks	node	This function removes all previous marks from the given node's outgoing edges.
mark_sp_back	node, node	Marks the edge from the first argument to the second as an sp_back-edge.
is_sp_back	node, node	Returns true if the given edge has been marked.

Other functions		
Function name	Arguments	Description
copy	node	Returns a reference to a newly created node-object, that is preset with all informations from the argument node.
sigma	node	Returns the integer-weight of the given node.
dom	node, node	Returns true if the first argument dominates the second.
remove_dead_code	void	A procedure that removes all nodes from the control flow graph, that cannot be reached from start (s).
build_dominator_tree	node	Though this function takes an argument, it is for now always called with the start-node. It is, however, not necessary to rebuild the entire dominator-tree if just a small region has changed. The algorithm could also give a node as an argument such that only the dominator-tree below that node would have to be rebuild. This could save much time.[4]

4.3 Annotated Pseudo-Code

Before the algorithm is actually called, some data structures have to be initialized and dead code has to be removed:

```
remove_dead_code(start);
build_dominator_tree(start);
set_level(start,1);
mark_undone(start);
search_sp_back(start);
```

[4] Sreedhar, Gao and Lee [7] have given an algorithm for incremental updates to the dominator tree.

```
    mark_undone(start);

    splt_loops(start, new_list(node));
```

The first argument to splt_loops is the node that dominates all nodes that have yet to be done. The second argument is a list of nodes that, if it is not empty, defines a region that should not be left since all nodes outside have already been done and have not changed. The function returns true if the given node has any edges that indicate an irreducible loop at its level.

```
  bool splt_loops(node top_node, list of node set) {
    bool cross;
    node child;
    node pred;

    cross = false;
    foreach (child in top_node.succs_dom) {
      if (is_empty(set) || in_list(child, set)) {
         if (splt_loops(child, set)) {
            cross = true;
         }
      }
    }

    if (cross) handle_ir_children(top_node, set);
```

First all levels below the given node are handled as long as this does not leave the current region of interest. If any of these calls return true, then a child of top_node has detected an irreducible loop on the level just below top_node. This is handled after all children so that the domains in that loop are already reducible.

```
    foreach (pred in top_node.preds) {
      if (is_sp_back(pred, top_node) &&
          !dom(top_node, pred)) {
        return true;
      }
    }
    return false;
  }
```

After the children of the current top_node have been handled completely, it is checked whether there is any edge that indicates an irreducible loop on the level of top_node itself and the result is returned.

The function handle_ir_children is called with the external dominator node as an argument. It has to find all SED-maximal loop-sets and then split the irreducible ones one after another.

```
void handle_ir_children(node top_node, list of node set) {
  node child;
  node tmp;
  list of node dfslist;
  list of node scc;
  list of list of node scclist;

  dfslist = new_list(node);
  scclist = new_list(list of node);

  foreach (child in top_node.succs_dom) {
    if ((!child.done) &&
        (is_empty(set) || in_list(child, set))) {
      SCC1(dfslist, child, set, top_node.level);
    }
  }
```

SCC1 implements the first pass of the SCC-algorithm. But instead of numbering the nodes they are collected in the list dfslist.

```
  foreach (tmp in dfslist) {
    if (tmp.done) {
      scc = new_list(node);
      SCC2(scc, tmp, top_node.level);
      add_list(scc, scclist);
    }
  }
```

Starting with the "highest numbered" node SCC2 is called, which implements the second pass of the SCC-algorithm. Each new call to SCC2 starts a new tree and, therefore, a new SCC. These SCCs are all collected in the list scclist.

```
  foreach (scc in scclist) {
    if (size_list(scc) > 1) {
      handle_scc(top_node, scc);
    }
  }
}
```

After all SCCs have been found, they are now converted by `handle_scc` one by one.

```
void handle_scc(node top_node, list of node scc) {
  node tmp;
  list of node msed;
  list of node tops;

  msed = new_list(node);

  foreach (tmp in scc) {
    if (tmp.level == top_node.level + 1) {
      GetWeight(tmp, tmp, scc);
      add_list(tmp, msed);
    }
  }

  if (size_list(msed) <= 1) return;
```

`handle_scc` first determines, which nodes of the SCC are in the MSED-set by comparing the level of the nodes with that of the external dominator. For each node in the MSED-set the function `GetWeight` is called, which sums up the weight of all nodes in its domain and sets the header links. If the MSED-set consists of just one node, the SCC is a reducible loop and does not need any further processing.

```
  tmp = ChooseNode(msed);
  SplitSCC(tmp, scc);
```

Otherwise `ChooseNode` is called, which applies the heuristic and chooses the heaviest node. `SplitSCC` then splits all nodes in the SCC except the chosen node and its domain, rearranges the control flow graph and changes the dominator information such that the copied regions are independent subtrees in the dominator tree.

```
  build_dominator_tree(start);
  set_level(start,1);
  mark_undone(start);
  search_sp_back(start);
  mark_undone(start);
```

Since the changes made by `SplitSCC` may have made the dominator tree invalid, it is recomputed. However, to rebuild the entire tree is rather inefficient since only the nodes below `top_node` have changed. A more elaborate algorithm would use `top_node` as the argument and `build_dominator_tree` would use the old information to rebuild just the subtree below it. The same is also true for the remaining re-initializations.

```
    tops = new_list(node);
    find_top_nodes(scc, tops);
    foreach (tmp in tops) {
      splt_loops(tmp, scc);
    }
  }
```

After all structures have been rebuild, `find_top_nodes` is used to collect the nodes that may have become external dominators. For these, `splt_loops` is called recursively to resolve any remaining irreducibility. However, such irreducible regions will always be subsets of the SCC that was just split. Thus the current SCC is given as the second argument such that `splt_loops` will never recurse out of it.

```
  void SplitSCC(node hdrnode, list of node scc) {
    node tmp;
    node tmp1;

    foreach (tmp in scc) {
      if (tmp.header != hdrnode) {
        tmp.copy = copy(tmp);
      }
    }
```

`SplitSCC` first makes a copy for every node that is to be split. The component `header` has been set by the previous call to `GetWeight` and points to the header of the domain the node belongs to. That way it is simple to find out which nodes are in the domain of the node that should not be split.

Note that `copy` is supposed to copy at least all of the information in the members listed in the last section.

```
    foreach (tmp in scc) {
      if (tmp.header != hdrnode) {
        if (tmp.idom.copy == NULL) {
          add_list(tmp.copy, tmp.idom.succs_dom);
        } else {
          rmv_list(tmp, tmp.idom.copy.succs_dom);
          add_list(tmp.copy, tmp.idom.copy.succs_dom);
```

```
    tmp.copy.idom = tmp.idom.copy;
  }
  foreach (tmp1 in tmp.succs_dom) {
    if (tmp1.copy == NULL) {
      rmv_list(tmp1, tmp.copy.succs_dom);
    }
  }
```

After the copies are made several steps are done on each node that got a copy. First, the dominator information is adjusted: If the node had an immediate dominator outside the copied region, then the copy has the same dominator and thus becomes another successor in the dominator tree. If the node's dominator has itself got a copy, then the dominance relation between the copies is made the same as between the originals. As a last step regarding the dominators the successors are updated. Successors within the copied regions have already been corrected by the previous step. But if any node has a successor outside the copied region, its copy has got that successor as well, thereby destroying the tree structure. This is corrected by letting the outside node be a child of the original only and removing it from the list `succs_dom` of the copy. This does not reflect the true dominance relations but retains the tree structure on which an incremental `build_dominator_tree` might depend.

```
  foreach (tmp1 in tmp.succs) {
    if (tmp1.copy != NULL) {
      rmv_list(tmp1, tmp.copy.succs);
      add_list(tmp1.copy, tmp.copy.succs);
      rmv_list(tmp, tmp1.copy.preds);
      add_list(tmp.copy, tmp1.copy.preds);
    } else {
      add_list(tmp.copy, tmp1.preds);
    }
  }
```

Once the dominator information has been corrected, the successor links are adjusted in a similar way. Links between nodes inside the copied regions are just replicated on the copies. If a successor is outside the copied region it has got just one more predecessor.

```
  foreach (tmp1 in tmp.preds) {
    if (tmp1.copy == NULL) {
      if (!in_list(tmp1, scc)) {
        rmv_list(tmp1, tmp.copy.preds);
      } else {
```

```
              rmv_list(tmp1, tmp.preds);
              rmv_list(tmp, tmp1.succs);
              add_list(tmp.copy, tmp1.succs);
            }
          }
        }
      }
    }
```

The last step adjusts the predecessor links. Again, the predecessors inside the split region have already been corrected. What has to be done with a predecessor outside depends on whether it is in the domain of h_1 (`hdrnode`) or not. In the first case, it remains a predecessor of the original and has to be removed from the `preds`-list of the copy. In the second case, it becomes a predecessor of the copy alone.

```
    foreach (tmp in scc) {
      add_list(tmp.copy, scc);
      tmp.copy = NULL;
    }
  }
```

After all connections between the nodes have been updated, the new nodes become part of the SCC and the `copy` members are reset so that subsequent invocations of `SplitSCC` do not get confused. Note that the list used in the `foreach`-statement is changed in the body of that statement. It is assumed, that the `foreach`-statement is executed for the original members of the list only and not for the newly added ones.

```
  void SCC1(list of node dfslist, node cnode,
            list of node set, integer level) {
    node child;

    cnode.done = true;
    foreach (child in cnode.succs) {
      if (!child.done && child.level > level &&
          (is_empty(set) || in_list(child, set))) {
        SCC1(dfslist, child, set, level);
      }
    }
    add_list(cnode, dfslist);
  }
```

`SCC1` traverses the control flow graph in a depth-first search collecting the nodes in the list `dfslist`. The only things special in that search are that it never traverses to a level higher (or lower numbered) than the level given, which was the level of the external dominator, and that it never leaves the region defined by `set`.

```
void SCC2(list of node scc, node cnode, integer level) {
  node pred;

  cnode.done = false;
  foreach (pred in cnode.preds) {
    if (pred.done && pred.level > level) {
      SCC2(scc, pred, level);
    }
  }
  add_list(cnode, scc);
}
```

To avoid at least one call to `mark_undone`, `SCC2` uses the member `done` in the inverse meaning. A node with `done` being `false` has actually been done or was outside of `set` in the call to `SCC1`. That way `SCC2` also stays within that region.

```
void find_top_nodes(list of node scc, list of node tops) {
  node tmp;
  node top;

  foreach (tmp in scc) {
    top = tmp.idom;
    while (in_list(top, scc)) {
      top = top.idom;
    }
    if (!in_list(top, tops)) {
      add_list(top, tops);
    }
  }
}
```

The split SCC may have fallen into several pieces dominated by different nodes. Since all of these may still be irreducible, `splt_nodes` must be called for each node that dominates such a piece. Therefore, `find_top_nodes` collects all of these in the list `tops`.

`ChooseNode` implements the heuristic and returns the heaviest node of the given MSED-set.

```
node ChooseNode(list of node msed) {
  integer MaxWeight;
  node MaxNode;
  node tmp;

  MaxWeight = 0;
  foreach (tmp in msed) {
    if (tmp.weight > MaxWeight) {
      MaxWeight = tmp.weight;
      MaxNode = tmp;
    }
  }

  return MaxNode;
}
```

`GetWeight` sums up the weight of the nodes in each domain and sets the header node's `weight`-member.

```
void GetWeight(node tmp, node hdnode, list of node scc) {
  node child;

  tmp.weight = sigma(tmp);
  foreach (child in tmp.succs_dom) {
    if (in_list(child, scc)) {
      GetWeight(child, hdnode, scc);
      tmp.weight = tmp.weight + child.weight;
    }
  }
  tmp.header = hdnode;
}
```

`search_sp_back` marks all `sp_back`-edges as such.

```
void search_sp_back(node cnode) {
  node child;

  cnode.done = true;
  cnode.active = true;
  remove_marks(cnode);

  foreach (child in cnode.succs) {
    if (child.active) {
```

```
        mark_sp_back(cnode, child);
      } else {
        if (!child.done) {
          search_sp_back(child);
        }
      }
    }

    cnode.active = false;
  }
```

`set_level` is used to calculate the depth of each node in the dominator-tree.

```
  void set_level(node cnode, integer level) {
    node child;

    cnode.level = level;
    foreach (child in cnode.succs_dom) {
      set_level(child, level + 1);
    }
  }
```

`mark_undone` resets the `done`-flag used by various other functions. The `active`-flag may as well be reset just once in `main` before the first call to `search_sp_back`. The algorithm then manages to reset this flag on the way.

```
  void mark_undone(node cnode) {
    node child;

    cnode.done = false;
    cnode.active = false;
    foreach (child in cnode.succs_dom) {
      mark_undone(child);
    }
  }
```

5 Related Work

Hecht [3] presents an algorithm to convert irreducible flow graphs to reducible ones. Though this algorithm was quiet inefficient with respect to the resulting flow graphs size, the accompanying theorems and proofs are still important.

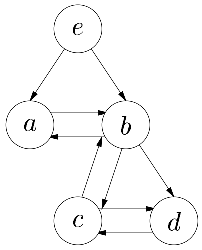


Figure 11: An example with an RC-node. (Node b)

An algorithm, which can analyze the extension and nesting of irreducible loops, is presented by Sreedhar et. al. [8]. The DJ-graphs used by this algorithm can also be used to identify the remaining structures of irreducible loops, such as domains, as defined in this work.

DJ-Graphs are also used as a representation to adjust common optimization algorithms to operate on them and thereby recognize and handle irreducible loops.

The notion of MSED-sets is introduced by Janssen and Corporaal [5], and a node-splitting algorithm, called "Controlled Node Splitting" is presented that tries to minimize the number of splits. Basically, this algorithm is the same as introduced in Section 1. The only difference is that it restricts the set of nodes that are candidates for splitting. The restrictions introduced are:

1. Only nodes that are elements of an SED-set are candidates for splitting.

2. Nodes that are elements of RC are not candidates for splitting.

The set RC is defined as the set of nodes of SED-sets that dominates other SED-sets and that are reachable from them.

By introducing these restrictions, the growth in code size dropped to almost one tenth. Compared to this work, the second restriction means that a node must not be split, while its domain is itself irreducible. Consider Figure 11. Since the nesting is not known to the algorithm, it might split node a as the only possibility in the MSED-set $\{a, b\}$, though node b could have been split once c or d had been split and reduced into it. If a is very heavy compared to b, c and d, this might increase the code size and would have been avoided by always reducing the domains first. In that way, the algorithm would never come upon an RC node and would have the freedom to choose b if it is lighter than a, even though it contains c and d. However, this would always result in splits of entire domains, which leads to the problems shown in Section 3.2. Another problem has already been shown in the example of Section 1. The transformations T1 and T2 might reduce parts of the control flow graph into an MSED-node that are not even part of the loop and, therefore, do not need to be split. This problem has not been solved by the above restrictions.

6 Conclusion

In this work the problem of converting irreducible flow graphs into equivalent reducible ones with the minimal possible growth in code size has been analyzed. It has been shown that the traditional approach to strictly split one node at a time cannot always lead to the minimal result, even if the best splitting sequence is used. By analyzing the structure of irreducible loops, it has been found that they always consist of a number of regions called domains and that nested loops are always contained in one of these domains. A new approach has been developed that always *keeps* one of these domains and splits the nodes of the others. This approach always results in the minimal flow graph under the assumption that always the right domain is chosen to be left untouched. Though no algorithm for determining the right domain has been found, a heuristic can be used, which suggests considerable improvements over previous approaches. In the future, further research should be carried out to either find a way to determine the right domain to be left unmodified, which would make this algorithm indeed minimal, or to show that this cannot be done in polynomial time.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilerbau – Teil 2*, volume 2. Addison-Wesley, 1988.

[2] J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimizations. In *SIGPLAN '84 Symposium on Compiler Construction*, pages 111–116, June 1984.

[3] M. S. Hecht. *Flow Analysis of Computer Programs.* Programming Languages Series. Elsevier North-Holland, Amsterdam, 1977.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[5] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Programming Languages and Systems*, 19(6):1031–1052, November 1997.

[6] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, July 1997.

[7] V. Sreedhar, G. Gao, and Y. Lee. An efficient incremental algorithm for maintaining dominator trees and its application to ϕ-nodes update. Technical report, McGill University, July 1994.

[8] V. Sreedhar, G. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Trans. Programming Languages and Systems*, 18(6):649–658, November 1996.