Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger

Larry Carter [1], Jeanne Ferrante,

University of California at San Diego [2]
9500 Gilman Drive MC 0114
San Diego, CA 92093

Clark Thomborson,

University of Auckland
Auckland, NZ

Abstract

Many program analysis techniques used by compilers are applicable only to programs whose control flow graphs are *reducible*. Node-splitting is a technique that can be used to convert any control flow graph to a reducible one. However, as has been observed for various node-splitting algorithms, there can be an exponential blowup in the size of the graph.

We prove that exponential blowup is unavoidable. In particular, we show that any reducible graph that is equivalent to the complete graph on n nodes (or to related bounded-degree control flow graphs) must have at least 2^{n-1}. While this result is not a surprise, it may be relevant to the quest for finding methods of obfuscation for software protection.

Key words: compilers, computational complexity, programming languages, saftey/security in digital systems

---

1 Introduction

Control flow graphs [1] are frequently used in optimizing compilers as the basis of program analysis and optimization. Intuitively, a control flow graph

[1] Email: carter@cs.ucsd.edu
[2] Work performed while on sabbatical at Auckland University

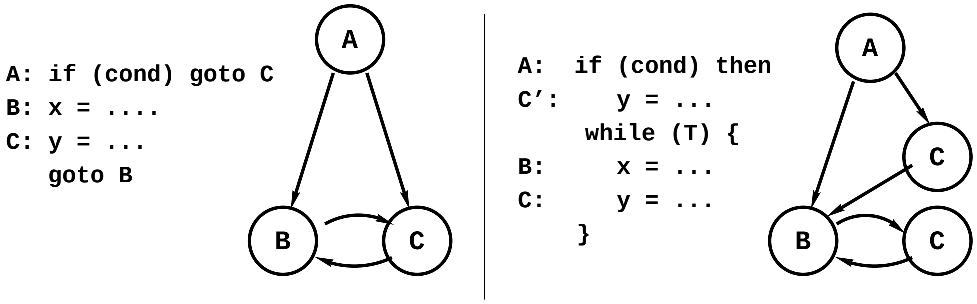

Fig. 1. (left) An "unstructured" program and the corresponding irreducible control flow graph. (right) An equivalent "structured" program and its (reducible) flow graph.

of a program is a directed graph that represents the program's potential flow of control. *Reducible* flow graphs, defined in Section 2, are a subclass of control flow graphs that include all those derived from "structured" programs, i.e., programs genenerated using a restricted set of constructs such as **if-then-else**, **while-do**, **continue** and **break**. It is well known that program analysis and optimization are considerably more efficient when restricted to reducible flow graphs [1]. Nevertheless, *irreducible* flow graphs do arise, often through the use of **go to** constructs, and must be handled by optimizing compilers.

The most commonly used technique[3] for handling irreducible flow graphs is node-splitting [1]. This technique eliminates "unstructured" program constructs by making duplicate copies of selected nodes in the control flow graph. Figure 1 gives an example.

It has often been observed in the literature that for various node splitting algorithms, there is an exponential blowup in the size of the program or flow graph.[4] However, there are different ways to choose which nodes to split, and to choose an order of splitting these nodes [5, 6]. This leaves open two questions, which we have been unable to find answered in the published literature.

- Is it possible that some (perhaps as yet undiscovered) node splitting algorithm avoids exponential blowup?
- Is it possible that for any flow graph, there is an equivalent, reducible flow graph (even one that might not be the result of a node splitting algorithm) that avoids exponential blowup?

This paper answers both questions negatively. Although this will not come as a surprise to people familiar with the field, it fills in a gap that has been considered "folklore". Until now, there was little concern about the possible

[3] Another technique adds extra predicate variables to guard statements in a loop, e.g. [3, 4]. The results of this paper do not apply to such techniques.
[4] See, for example, [1], p. 680.

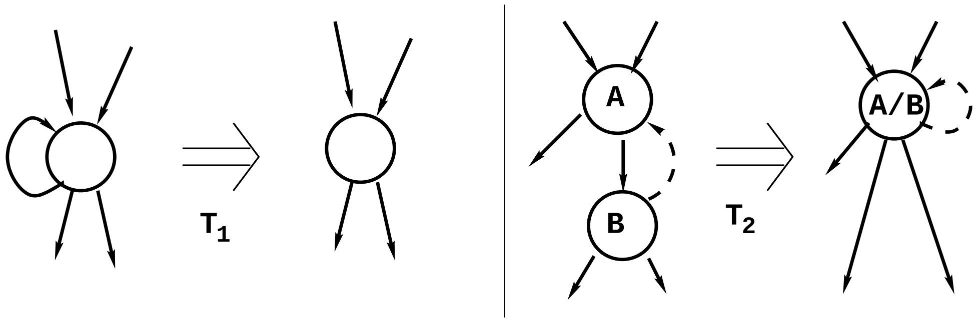

Fig. 2. Two transformations used to simplify flow graphs. $\mathbf{T}_1$ eliminates self-loops. $\mathbf{T}_2$ merges a single-entry node into its parent. If there is an arc from the child back to the parent, it becomes a self-loop. A graph G is *reducible* if repeated applications of $\mathbf{T}_1$ and $\mathbf{T}_2$ reduce G to a single node.

blowup, since it has often been observed that most programs written by humans are reducible [1]. However, we are motivated to take another look at this question by our interest in software obfuscation. In this context, it may be an advantage to purposefully generate flow graphs that are hard to analyze.

2 Reducible Flow Graphs

The term *reducible* flow graph, [5] originally defined in [2], means that the graph can be reduced to a single node by a sequence of applications of the two transformations, $\mathbf{T}_1$ and $\mathbf{T}_2$, shown in Figure 2. In the context of data flow analysis, flow graphs are annotated with information on the nodes (such as "There is an assignment to `X` and a use of `Y` in this block of code.") Many data flow algorithms work by manupulating this information appropriately as the graph is reduced via $\mathbf{T}_1$ and $\mathbf{T}_2$, producing summary information about the program (such as "The assignment to `X` in Block 3 is used in Blocks 4 and 8," and "The use of `Y` in block 3 may be uninitialized".) The details of these algorithms need not concern us; however, they motivate the following definitions:

Definition 1 *A* flow graph *is a directed graph with a distinguished node e, called the* entry*, together with a function L that assigns a label to each node of the graph. The labels need not be unique.*

Definition 2 *A finite string of labels s is* produced *by a flow graph G if $s = L(e)L(n_1)L(n_2)\ldots L(n_k)$ where $e, n_1, n_2 ..., n_k$ is a path in G beginning with the entry. $L(G)$ denotes the* language *of G, i.e. the set of strings produced*

[5] There are several equivalent definitions of reducible flow graph, see [5].

by G.[6]

Definition 3 *Two flow graphs G_1 and G_2 are* equivalent *if $L(G_1) = L(G_2)$.*

For example, the two flow graphs of Figure 1 are equivalent. In general, node splitting techniques transform an arbitrary flow graph into an equivalent reducible flow graph. We note that flow graphs and the set of strings they produce are very similar to finite automata and regular languages — the difference is that flow graphs have labelled nodes, whereas finite automata have labelled edges. In fact, the language of a flow graph G is regular,[7] but flow graphs produce more restricted languages — for instance, since every string in $L(G)$ must begin with the label of the entry node.

Before continuing with our study of flow graphs, we introduce leftists trees, which provide another way of producing a set of strings. We will then prove a lower bound on the size of any leftist tree that produces the set of all strings over a given alphabet that begin with a given symbol. This result will allow us to prove a similar result for flow graphs.

3 Leftist Trees

A *leftist tree* T is simply a rooted binary tree (i.e., each node has either zero or two children), with labels on the leaves. $|T|$ denotes the number of leaf nodes in T. We let T_l and T_r denote the left and right subtrees[8] of T. Note that T_l and T_r are either both empty or both non-empty. Define $n_l(T)$, $n_r(T)$ as follows: if T is a single node, both these denote that node. Otherwise, they are "first" (leftmost) leaf of T_l and T_r, respectively. Finally, let $s_l(T)$ and $s_r(T)$ be the labels of $n_l(T)$ and $n_r(T)$, respectively.

Whenever the tree T is understood from context, we may drop the "(T)" from our notation. Thus, for instance, T_l, n_l, and s_l denote the left subtree of T (or T itself if $|T| = 1$), the leftmost leaf of that subtree, and the symbol labelling that leaf.

We now give a method of associating a set of strings with a leftist tree. If $P = p_1 p_2 \ldots p_k$ is a non-empty sequence of leaves of tree T, then we say P

[6] In different contexts, we use L to mean the label of a node, the language of a flow graph, or other assignments of a set of strings to leftist trees or annotated flow graphs that will come later.

[7] If the label of each node is put on each outgoing edge, the resulting finite automaton produces $L(G)$.

[8] By a *subtree* we mean a node together with *all* of the nodes that it is an ancestor of.

is a *leftist leaf sequence* (LLS) of T if for all $i < k$, there exists a subtree T_i of T such that $p_i \in T_i$ and p_{i+1} is either $n_l(T_i)$ or $n_r(T_i)$. Intuitively, starting from p_i, you can move up the tree as far as you want, go down to either child, but then you must keep going left until you get to the leaf p_{i+1}. Given a LLS $P = p_1 p_2 \ldots p_k$ of T, we say a string of symbols $s = s_1 s_2 \ldots s_k$ is *produced* by P if for $i = 1, \ldots, k$, s_i is the label of p_i. Finally, we define the *language* of T, denoted, $L(T)$, to be the set of strings produced by LLS's in T that start with $n_l(T)$, the leftmost leaf of T.

The following three observations follow easily from the definition of leftist leaf sequence.

Lemma 1 *If P is a sequence of leaves of a subtree T' of T, then P is a LLS of T' iff it is a LLS of T.*

Lemma 2 *If $p_1 \ldots p_j$ and $p_{j+1} \ldots p_k$ are LLS's of T, and p_{j+1} is either n_l or n_r, then $p_1 \ldots p_j p_{j+1} \ldots p_k$ is a LLS of T.*

Lemma 3 *If $p_1 \ldots p_j p_{j+1} \ldots p_k$ is a LLS of T and p_{j+1} is in some subtree T' that doesn't include p_j, then p_{j+1} must be $n_l(T')$.*

We now can prove an alternate characterization of the set of strings that are produced by a tree T.

Theorem 1 *$L(T)$ can be expressed recursively by $L(T) = s_l(T)^+$ if $|T| = 1$ and $L(T) = L(T_l)\{L(T_l) \cup L(T_r)\}^*$ otherwise.*

Proof. By induction on $|T|$.

(Base case) If $|T| = 1$, i.e., if T is a single node n, then notice that if we let $T_i = T$, any sequence of one or more n's is a LLS starting with n_l. Together these LLS's produce the language $s_l(T)^+$.

(Inductive step) Assume that the theorem holds for T_l and T_r. Given any $s \in L(T_l)\{L(T_l) \cup L(T_r)\}^*$, we can find a LLS in T_l or T_r for each of the substrings that are implied by this formula. By Lemma 1, these LLS's are also LLS's of T. Since each such LLS begins with either $n_l(T_l) = n_l$ or $n_l(T_r) = n_r$, their concatenation is an LLS by Lemma 2. It is also clear that this sequence begins with n_l. Thus, s is in $L(T)$.

Conversely, suppose $s \in L(T)$. Then s is generated by a LLS P of T that begins with n_l. We can decompose P into subsequences according to when P switches from T_l to T_r or back again. By Lemma 1, these subsequences are SSL's of T_l or T_r. By Lemma 3, each such subsequence begins with n_l or n_r. By the inductive hypothesis, they produce substrings in $L(T_l)$ or $L(T_r)$. Since T begins with n_l, the first substring is in $L(T_l)$. Thus s, which is the concatenation of the substrings, is in $L(T_l)\{L(T_l) \cup L(T_r)\}^*$. QED

A simple consequence of Theorem 1 is that for any leftist tree T, $L(T)$ is *idempotent*, meaning that $L(T)^+ = L(T)$.

We need a few more definitions:

Definition 4 *We say that "a string t is a suffix of $L(T)$" if $\exists s \in U^*$ such that $st \in L(T)$.*

Definition 5 *Given an alphabet $\Sigma \subseteq U$, we say that "T is Σ-complete" if $\forall t \in \Sigma^*$, t is a suffix of $L(T)$.*

In our examples, the structure of a leftist tree T will be presented using parentheses recursively. We write $T = s$ if T is a single node labeled by s, and $T = (T_l T_r)$ if $|T| > 1$. For example, $T = (a(bc))$ is a tree whose left subtree is a and whose right subtree is (bc).

Example 1. Let $T = (ab)$. Then $L(T) = a(a+b)^*$, thus T is Σ-complete for $\Sigma = \{a, b\}$.

Example 2. Let $T = (a(bc))$. Then $L(T) = a(a^* + b(b+c)^*)^*$. The string ac is not a suffix of $L(T)$, so T is not complete for $\Sigma = \{a, b, c\}$. However T is both $\{a, b\}$-complete and $\{b, c\}$-complete.

Example 3. $T = ((ab)(cb))$ is $\{a, b, c\}$-complete.

Lemma 4 *A string t is a suffix of $L(T)$ iff s is produced by a LLS of T (with no constraint on the first node of the LLS).*

The only non-trivial part of the proof is to show that you can construct a LLS from $n_l(T)$ to the first symbol in the string. This can be done by use leftmost leaves of the trees rooted at the nodes that form a path from the root of T to the leaf correspond the first symbol of t.

The next two lemmas provide insight into the structure of Σ-complete trees.

Lemma 5 *Given an alphabet Σ, if T is a Σ-complete tree of minimal size, then its right subtree T_r is $(\Sigma - s_l)$-complete.*

Proof. By contradiction. Assume T_r is not complete over the reduced alphabet $(\Sigma - s_l)$. Then there exists a string $u \in (\Sigma - s_l)^+$ such that u is not a suffix of $L(T_r)$. Because T is of minimal size, T_l is not Σ-complete, so there exists a shortest string t in Σ^+ that is not a suffix of T_l. Let j and k be the lengths of t and u. Because T is Σ-complete, we know that tu is a suffix of $L(T)$. Denote tu by the string $s = s_1 s_2 \ldots s_{j+k}$. By Lemma 4, there must be a LLS $p_1 p_2 \ldots p_j p_{j+1} \ldots p_{j+k}$ of T that produces s.

We will derive a contradiction by considering where p_j, the leaf corresponding to the last symbol of t, is. First, suppose that $p_j \in T_r$. Note that if $i > j$, $s_i \neq s_l$, since s_i is a symbol in u and u contains no s_l's. Thus, applying Lemma 3 inductively tells us that for all $i > j$, p_i must be a leaf of T_r. But this would mean that t is a suffix of $L(T_r)$, which contradicts how we chose t.

On the other hand, suppose that $p_j \in T_l$. By construction, $p_1 \ldots p_j$ is *not* a LLS of T_l (since s is not a suffix of T_l), so there must be some nodes of $p_1 \ldots p_{j-1}$ in T_r. Choose m to be the largest integer $m < j$ such that $p_m \in T_r$. This ensures that $p_{m+1}p_{m+2} \ldots p_j$ is a LLS of T_l. Furthermore, since $p_m \in T_r$ and $p_{m+1} \in T_l$, we must have $s_{m+1} = s_l$ by Lemma 3. Now we use the fact that t is a *shortest* non-suffix of T_l to conclude that there must be some LLS $p'_1p'_2 \ldots p'_m$ in T_l that produces $t_1t_2 \ldots t_m$. Now consider the sequence $p'_1 \ldots p'_m p_{m+1} \ldots p_j$. By construction, it is entirely in T_l and it produces t. It also is a LLS by Lemma 2. Thus, we conclude that t is produced by a LLS in T_l, which contradicts the fact that it is *not* a suffix of T_l. QED

Lemma 6 *Given an alphabet Σ, if T is a Σ-complete tree of minimal size, then its left subtree T_l is $(\Sigma - s_r)$-complete.*

Proof. The proof is the same as for Lemma 5, with the roles of T_l and T_r reversed. The abbreviated proof follows.

Suppose there were a string $u \in (\Sigma - s_r)^+$ of length j that is not a suffix of $L(T_l)$. T_r is not Σ-complete, so there is a shortest string $t \in \Sigma^+$ that is not a suffix of T_l. Let $p_1p_2 \ldots p_jp_{j+1} \ldots p_{j+k}$ be a LLS of T that produces tu. p_j must be in T_r; otherwise, the remainder of the leaf sequence would be stuck in T_l and produce u. We find $m < j$ such that $p_{m+1} \ldots p_j$ is a LLS of T_r. The initial substring $p_1 \ldots p_m$, being shorter than t, has a LLS in T_r. Extending this LLS with $p_{m+1} \ldots p_j$ gives a LLS in T_r that produces t — a contradiction. QED

Lemmas 5 and 6 enable us to prove a lower bound on the size Σ-complete trees.

Theorem 2 *Let S_n be the the minimal number of leaves in a Σ-complete tree with $|\Sigma| = n$. Then $S_n \geq 2^{n-1}$.*

Proof. By induction on n. The induction basis $n = 1$ is established by noting that $T = a$ is the minimal-sized tree to generate the 1-symbol complete language $L(T) = a^+$. The inductive step $S_n \geq 2S_{n-1}$ is established by considering the subtrees T_l and T_r for a Σ-complete tree T of minimal size with $|\Sigma| = n$. By Lemma 6, the subtree T_l is $(\Sigma_T - s_r)$ complete, so by the inductive hypothesis it has at least 2^{n-2} leaves. By Lemma 5, the subtree T_r is $(\Sigma - s_l)$ complete, so by induction it has at least 2^{n-2} leaves. This gives us the desired inequality. QED

Theorem 3 $S_n = 2^{n-1}$.

Proof. A recursive construction will match the bound of Theorem 2. Our basis, for alphabet $\Sigma = \{x_1\}$, is the one-node tree $T = x_1$. We build a tree T with 2^{n-1} leaves for alphabet $\Sigma = \{x_1, x_2, ..., x_n\}$ from a left subtree of size S_{n-1} with $L(T_l) = x_1(\Sigma - x_n)^*$ and a right subtree also of size S_{n-1} with $L(T_r) = x_n(\Sigma - x_1)^*$. The resulting language $L(T)$ is $L(T_l)\{L(T_l) \cup L(T_r)\}^*$ $= x_1(\Sigma - x_n)^*\{x_1(\Sigma - x_n)^* \cup x_n(\Sigma - x_1)^*\}^* = x_1(\Sigma)^*$. Thus, T is Σ-complete and has 2^{n-1} leaves. QED.

4 The Complete Graph Expands Exponentially

Given a finite alphabet Σ and $e \epsilon \Sigma$, let $K_{\Sigma,e}$ be the flow graph comprising the complete directed graph with entry node e whose nodes are the elements of Σ. Let $L(K_{\Sigma,e})$ be the language produced by K_n. It's not hard to see that $L(K_{\Sigma,e})$ is the reqular language $e\Sigma^*$, that is, the set of all strings over Σ that begin with "e". The main result of this paper is the following theorem:

Theorem 4 *If R is a reducible flow graph that is equivalent to $K_{\Sigma,e}$ (i.e. that is, if $L(R) = e\Sigma^*$), then R must have at least 2^{n-1} nodes, where n is the cardinality of Σ.*

We first show how to associate a leftist tree T with any reducible flow graph R. Each leaf of T will correspond to a node of R, and $L(T)$ will be a superset of $L(R)$. We do this by annotating each node n of R with a leftist tree T_n. As we reduce R using the transformations $\mathbf{T}_1$ and $\mathbf{T}_2$, we will build up the trees that annotate the nodes. By the time R has been reduced to a single node, its associated tree will be the desired leftist tree. To each intermediate annotated flow graph there corresponds a language in the natural way — it is the set of strings of the form $s_1 s_2 ... s_k$, where $s_i \epsilon L(T_{n_i})$ and $n_1, n_2, ...$ is a path through the flow graph.

Over the course of the construction, the following invariants will hold:

(1) The language associated with the current annotated flow graph is a superset of $L(R)$.
(2) There is a one-to-one correspondence between the nodes of R and the union of the sets of leaves of the annotations.

The first step of the construction is to annotate each node n of R with the leftist tree T_n consisting of a single (leaf) node labelled by $L(n)$, the label of n. It is easy to verify that properties (1) and (2) hold.

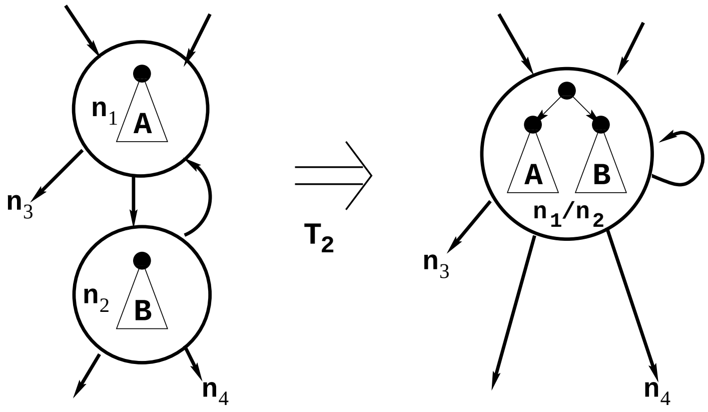

Fig. 3. After a $\mathbf{T}_2$ transformation, the annotation of the new fused node is a tree that has the old annotations as its left and right subtrees.

Whenever a $\mathbf{T}_1$ (self-loop elimination) transformation is applied to a node n of an annotated graph, we don't need to adjust the annotations. Property (1) will still hold since the only paths through the original flow graph that are no longer paths of the transformed one include repetitions of node n, but the fact that $L(T_n)$ is idempotent ensures that arbitrary repetitions are represented even after eliminating the self-loop.

A $\mathbf{T}_2$ transformation can be applied to nodes n_1 and n_2 only when the edge (n_1, n_2) is the only edge into n_2. When a $\mathbf{T}_2$ transformation is applied, we fuse the two nodes into a single node and update its annotation as shown in Figure 3.

In any path through the pre-$\mathbf{T}_2$ flow graph, every occurrence of n_2 must be immediately preceeded by an n_1. Thus, for each substring $s_{i-1}s_i$ where $s_{i-1} \epsilon L(T_{n_1})$ and $s_i \epsilon L(T_{n_2})$ that contributes to strings in the pre-$\mathbf{T}_2$ language, that same substring will be included in $L(T_{n_1/n_2}$ in the post-$\mathbf{T}_2$ flow graph. Thus, property (1) holds.[9] Property (2) follows from the construction.

Once we have reduced R to a single node via $\mathbf{T}_1$ and $\mathbf{T}_2$, the annotation of that node is T, the leftist tree we associate with R. As noted, $L(T)$ may be a superset of $L(R)$. However, if R is equivalent to $K_{\Sigma,e}$, then its language already contains *all* strings starting with e. Thus, $L(T) = L(R) = e\Sigma^*$. By Theorem 2, T has at least 2^{n-1} leaves. Since there is a one-to-one correspondence between the leaves of T and the nodes of R, the proof is completed. QED.

In the full paper, we will extend our results to show the even flow graphs which have outdegree 2 suffer from exponential blowup when converted to reducible flow graphs.

[9] New strings may be have been added to the language; for instance, in the example there was no edge from n_1 to n_4 or from n_2 to n_3 before $\mathbf{T}_2$ was applied.

Theorem 5 *For any n that is a power of 2, there is a (non-reducible) flow graph on $n(n-1)$ nodes such each node has outdegree two and any equivalent reducible flow graph requires at least 2^{n-1} nodes.*

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Frances E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[3] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[4] A. Erosa and L. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *International Conference on Computer Languages*, May 1994.

[5] Johan Janssen and Henk Corporaal. Controlled Node Splitting. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 44 – 58, Linkoping, Sweden, April 1996. volume 1060 of Springer Lecture Notes in Computer Science.

[6] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting vs. DJ-Graphs. Technical Report TR 01-146, Institut f. Informatik, Humboldt-University, January 2001.