OPTIMAL CODE FROM FLOW GRAPHS
OR
NOTES ON AVOIDING GOTO STATEMENTS

by

M. V. S. Ramanath and Marvin Solomon

Computer Sciences Technical Report #415
January 1981
Optimal Code from Flow Graphs

or

Notes on Avoiding Goto Statements

by

M. V. S. Ramanath and Marvin Solomon

ABSTRACT

This paper considers the problem of generating a linear sequence of instructions from a flow graph so as to minimize the number of jumps. We show that for programs constructed from atomic statements with semicolon, if-then, if-then-else, and repeat-until, the minimal number of unconditional jumps is bounded from above by $e+1$ and from below by $\max\{e-b+1, \sqrt{(e+1)/2}\}$, where $e$ is the number of if-then-else statements and $b$ is the number of repeat-until statements. We show that these bounds are tight and present a linear-time algorithm for finding the optimal translation of a flow graph.
Optimal Code from Flow Graphs

1. INTRODUCTION

Over the years, there has been considerable research in the area called "code optimization", which concerns itself with techniques for producing the best possible machine code from a high-level program. There are many possible definitions of "best possible", and the techniques are highly influenced by the natures of the source language and the target machine. In any realistic situation, the problem of producing optimal code is intractable, so researchers content themselves with producing good but not necessarily optimal code, or code that is optimal with respect to some restricted set of transformations or source programs.

The general class of "global" optimizations includes techniques for re-organizing the flow graph of a program, for example removing invariant expressions from loops. However, surprisingly little attention has been paid to the problem of mapping the resulting flow graph into the linear form required by most machine architectures. Careful attention to this step can result in substantial improvements in both space and time.

For example, consider the programs $H_n$, defined recursively on $n$ as follows:
\[
H_0 = S_0
\]
\[
H_n = \begin{cases} 
\text{if } B_n \text{ then } H_{n-1} \text{ else } S_n & \text{(n even)} \\
\text{repeat if } B_n \text{ then } H_{n-1} \text{ else } S_n \text{ until } C_n & \text{(n odd)}
\end{cases}
\]
(For each \(i\), \(S_i\) is some atomic statement and \(B_i\) and \(C_i\) are Boolean expressions.) Figure 1 shows the flow graph of \(H_6\). (Some nodes are labeled for future reference.) Standard techniques of code generation would translate \(H_n\) into the program \(P_n; \text{exit}\), where \(P_i\) is defined recursively by

\[
S_0
\]
\[
P_i \rightarrow P_{i-1} \text{ if not } B_i \text{ then } L_i \text{ goto } M_i
\]
\[
N_i \rightarrow N_{i-1} \text{ if not } B_i \text{ then } L_i \text{ goto } M_i
\]
\[
L_i : S_i
\]
\[
M_i : \begin{cases} 
\text{if } i = 0 & \text{if } i > 0 \text{ and } i \text{ is even} \\
\text{if } i > 0 \text{ and } i \text{ is odd}
\end{cases}
\]

The translation of \(H_6\) is shown in Figure 2a. A more sophisticated code generator would produce "goto M3" instead of "goto M2" and "goto M5" instead of "goto M4", but a much better translation is \(T_n\), where \(T_n\) is

\[
P_n \rightarrow \begin{cases} 
P_{n-1} & \text{if } B_n \text{ then } L_{n-1} \\
S_n & \text{goto } M_{n-1}
\end{cases}
\]
\[
L_n : \text{if } B_n \text{ then } L_{n-1}
\]
\[
M_{n+2} : \text{exit}
\]
\[
P_i \text{ is defined by}
\]
\[
M_{n+1} : \text{exit}
\]
\[
\text{n odd}
\]
\[
\text{n even}
\]
\[ L_0: S_0 \]
\[ M_1: \text{if } C_1 \text{ then } M_3 \]
\[ L_1: \text{if } B_1 \text{ then } L_0 \]
\[ S_1 \text{ goto } M_1 \]
\[ i = 1 \]

\[ L_{i-1}: \text{if } B_{i-1} \text{ then } L_{i-2} \]
\[ S_{i-1} \]
\[ M_i: \text{if } C_i \text{ then } M_{i+2} \]
\[ L_i: \text{if } B_i \text{ then } L_{i-1} \]
\[ S_i \text{ goto } M_i \]
\[ i > 1, \text{ i odd} \]

and the initial entrance to \( H_n \) is at \( L_n \). The translation of \( H_6 \) according to this scheme is shown in Figure 2b. There are \( n \) jumps in the first translation of \( H_n \) and only \( n/2 \) in the second.

In this paper, we confine our attention to translations that preserve the topology of the flow graph exactly, and ignore improvements that might result from techniques such as node splitting or loop unrolling [1]. Under this restriction, there is a one-to-one correspondence between nodes in the graph and instructions other than jumps in the translation. An optimal translation is thus one that minimizes the number of jumps. Since each goto-free segment of the translation corresponds to a simple path in the flow graph, the problem reduces to finding a partition of the graph into as few disjoint simple paths as possible.

A jump-free translation is possible if and only if the graph has a Hamiltonian path. The Hamiltonian path problem is known to be \( \text{NP-} \)complete, even for planar graphs with in-degree and out-degree bounded by 2 [2]. Since \( \text{NP-} \)complete problems are widely conjectured to require exponential time for their solution, we do not try to find optimal translations for arbitrary flow graphs, but restrict our attention to "structured" flow graphs that arise
from programs composed of if-then-else, if-then, and repeat-until statements.

The remainder of this paper is organized as follows: Section 2 sketches the definitions and formally states the basic problem. Section 3 presents a linear-time algorithm for finding the optimal translation of any program that uses only if-then-else and repeat-until statements. Section 4 states and proves bounds on the the cost of a partition and proves that the algorithm finds a optimal partition. Section 5 shows that the bounds are tight by exhibiting families of graphs for which the cost of an optimal partition attains the upper and lower bounds. Section 6 shows how to accommodate if-then statements (without an else clause). Section 7 compares our work to previous results and indicates the direction of our current research.

2. DEFINITIONS

We assume the reader is familiar with standard terms of graph theory such as directed graph (digraph), directed acyclic graph (DAG), node, arc, and simple path. By "path" we will mean "simple path".

A flow graph is a digraph $G = (N,A)$ together with a distinguished start node $s(G)$ and set EX($G$) of exit nodes, such that each node is reachable from the start node. A simple flow graph (SFG) is a flow graph constructed according to the following rules:
1. A single node \( n \) is an SFG with \( s = n \) and \( EX = \{ n \} \).

2. If \( P = (N_P, A_P) \) and \( Q = (N_Q, P_Q) \) are SFG's then a new SFG
\( T = (N_T, A_T) \) may be constructed from \( P \) and \( Q \) by any of the follow-
ing four operations (see Figure 3):

- **CAT** (write \( T = C(P, Q) \)): \( N_T = N_P \cup N_Q; \)
  \( A_T = A_P \cup A_Q \cup \{(x, s(Q)) \mid x \in EX(P)\}; \)
  \( s(T) = s(P); \)
  \( EX(T) = EX(Q). \)

- **IF** (write \( T = I(P, i) \)): Let \( i \) be a new node. Then
  \( N_T = N_P \cup \{i\}; A_T = A_P \cup \{(i, s(P))\}; \)
  \( s(T) = i; \)
  \( EX(T) = EX(P) \cup \{i\}. \)

- **ELSE** (write \( T = E(P, Q, i) \)): Let \( i \) be a new node. Then
  \( N_T = N_P \cup N_Q \cup \{i\}; A_T = A_P \cup A_Q \cup \{(i, s(P)), (i, s(Q))\}; \)
  \( s(T) = i; \)
  \( EX(T) = EX(P) \cup EX(Q). \)

- **REPEAT** (write \( T = R(P, t) \)): Let \( t \) be a new node. Then
  \( N_T = N_P \cup \{t\}; A_T = A_P \cup \{(t, s(P)) \cup \{(x, t) \mid x \in EX(P)\}; \)
  \( s(T) = s(P); \)
  \( EX(T) = \{t\}. \)

The number of applications of ELSE is called the **branching factor**
of \( G \), denoted \( e(G) \). The **back arcs** of \( G \) (denoted \( B(G) \)) are the
arcs of the form \( (t, s(P)) \) introduced by REPEAT; the **scope** of the
back arc \( (t, s(P)) \), denoted SCOPE\((t, s(P))\) is \( N_p \). The number of
back arcs is denoted \( b(G) \). It should be clear that every SFG is
reducible \([3,4]\) and that the set \( B(G) \) is precisely the unique set
of back-arcs \([3]\). Hence \( B(G) \) and SCOPE\((a)\), for each \( a \in B(G) \),
are independent of the construction of \( G \). The branching factor
is also an inherent property of G.

A restricted SFG is one constructed without any use of IF. If G be an SFG, the restricted SFG corresponding to G is the SFG obtained by replacing each use of I(P,i) in the construction of G with C({i},P).

Assume G is a restricted SFG.

The set \( A_G - B(G) \) is called the set of DAG edges of G. A partition \( p \) of G is a set of simple paths such that each node of G is in exactly one path. A path using only DAG edges is a DAG path; a DAG partition is one composed of DAG paths. The cost of the partition, \( c(p) \), is the number of paths in it. The cost of G, \( c(G) \), is the cost of a cheapest partition of G. Partition \( p \) is optimal if \( c(p) = c(G) \).

A path is a top hook if it starts at \( s(G) \) and a bottom hook if it ends at a node in \( EX(G) \). A partition is top-open if it contains a top hook, bottom-open if it contains a bottom hook, open if it contains both a top hook and a bottom hook, and nice if it contains a top hook and a bottom hook that are distinct.

The algorithm for finding an optimal partition of G produces two partitions for each subgraph in the construction of G; one is an optimal partition and the other is an optimal open partition. The next definition is used in building these partitions of a graph from partitions of its parts.
Let \( P \) and \( Q \) be restricted SFG's, and let \( p_P \) and \( p_Q \) be partitions of them (see Figure 4.)

(CAT) If \( T = C(P, Q) \), define the partition \( PC(p_P, p_Q) \) of \( T \) as follows: If \( p_P \) is bottom-open and \( p_Q \) is top-open, let \( h_P \) be a bottom hook of \( p_P \) (distinct from the top hook if possible) and \( h_Q \) be the top hook of \( p_Q \). Then \( PC(p_P, p_Q) = \{h_P, h_Q\} \cup (p_P - \{h_P\}) \cup (p_Q - \{h_Q\}) \). Otherwise \( PC(p_P, p_Q) = p_P \cup p_Q \).

(ELSE) If \( T = E(P, Q, i) \) and at least one of \( p_P, p_Q \) is top-open, define the partition \( PE(p_P, p_Q, i) \) of \( T \) as follows: If \( p_P \) is top-open, let \( h_P \) be its top hook. Then \( PE(p_P, p_Q, i) = (i, h_P) \cup (p_P - \{h_P\}) \cup p_Q \). Similarly, if \( p_P \) is not top-open, but \( p_Q \) is, \( PE(p_P, p_Q, i) = (i, h_Q) \cup (p_Q - \{h_Q\}) \cup p_P \).

(REPEAT) If \( T = R(P, t) \) and \( p_P \) is open, define partitions \( PR(p_P, t) \) and \( PR'(p_P, t) \) of \( T \) as follows: Let \( h_t \) and \( h_b \) be top and bottom hooks of \( p_P \) with \( h_t \neq h_b \) if \( p_P \) is nice. Then \( PR'(p_P, t) = \{h_b, t\} \cup (p_P - \{h_b\}) \) and \( PR(p_P, t) = \{h_b, t, h_t\} \cup (p_P - \{h_b, h_t\}) \) if \( p_P \) is nice and \( PR(p_P, t) = PR'(p_P, t) \) otherwise.

3. THE ALGORITHM

We are now ready to state the main algorithm of this paper:
3.1 Algorithm PARTITION

Input. A restricted SFG G.

Output. Two partitions p and p' for G.

Method. If $G = C(P, Q)$, call PARTITION recursively to get partitions $p'_P$ and $p'_Q$ for P and partitions $p'_Q$ and $p'_Q$ for Q. Let $p' = PC(p'_P, p'_Q)$. Let $p = p'$ if either $c(p'_P) = c(p'_P)$ or $c(p'_Q) = c(p'_Q)$, and let $p = PC(p'_P, p'_Q)$ otherwise.

If $G = E(P, Q, i)$, call PARTITION recursively to get partitions $p'_P$ and $p'_Q$ for P and partitions $p'_Q$ and $p'_Q$ for Q. Let $p = PE(p'_P, p'_Q)$ if $c(p'_Q) = c(p'_Q)$ but $c(p'_P) \neq c(p'_P)$. Otherwise, let $p = PE(p'_P, p'_Q)$. Let $p' = p$.

If $G = R(P, t)$, call PARTITION recursively to get partitions $p'_P$ and $p'_P$ for P. Let $p' = PR'(p'_P)$, and let $p = PR(p'_P)$ if $p'_P$ is nice; let $p = p'$ otherwise.

3.2 Theorem

The partitions $p$ and $p'$ computed for $G$ from Algorithm 3.1 have the following properties:

1. $p$ is optimal.

2. $c(p') \leq c(p) + 1$.

3. $p'$ is open.
4. If \( p' \) is not optimal then \( p' \) is nice and no optimal partition of \( G \) is top-open or bottom-open.

5. If \( G \) has a nice partition of cost \( c(p') \), then \( p' \) is nice.

**Proof.** The proof is by induction on the construction of \( G \).

The result is trivial if \( G \) is the one-node graph.

If \( G = C(P,Q) \), four cases arise:

**Case I.** \( p'_P \) and \( p'_Q \) are both optimal. By definition, \( p'_G = p_G \) and \( c(p_G) = c(P) + c(Q) - 1 \). If we could bet a cheaper partition for \( G \), we would be able to decompose it into partitions for \( P \) and \( Q \), one of which must be better than optimal. Thus property 1 is proved. Properties 2, 3, and 4 are easy. Property 5 follows from the fact that a nice optimal partition for \( G \) can be decomposed into partitions for \( P \) and \( Q \), one of which must be optimal and nice. Hence, using 5 inductively, either \( p'_P \) or \( p'_Q \) is nice and so is \( p'_G \).

**Case II.** \( p'_P \) is not optimal, but \( p'_Q \) is optimal. Here too, \( p'_G = p_G \) by definition, and \( c(p_G) = c(P) + c(Q) \). Properties 2 and 3 are obvious. Using 4 inductively, we see that \( p'_P \) is nice and hence so is \( p'_G \). So property 5 is proved. To prove 1, we note that any partition cheaper than \( p'_G \) can be used to yield an optimal partition for \( P \) which is bottom-open, violating property 4 for \( P \). Property 4 follows from 1.

**Case III.** \( p'_P \) is optimal, but \( p'_Q \) is not optimal. This case is very similar to case II.

**Case IV.** Both \( p'_P \) and \( p'_Q \) are suboptimal. From the definition, we
see that \( c(p_G') = c(P) + c(Q) + 1 \). By property 4, neither \( p_P \) nor \( p_Q \) is open at either end, so \( c(p_G) = c(P) + c(Q) \). Also, by an inductive use of 4, \( p_P' \) and \( p_Q' \) are nice and hence so is \( p_G' \). Thus 2, 3, and 5 are proved. To prove 4, suppose an optimal partition of \( G \) were top-open or bottom open. We could then get a top-open partition that is optimal for \( P \) or for \( Q \), violating property 4 for \( P \) or for \( Q \). Property 1 is proved as in case II.

If \( G = E(P, Q, i) \), we have the same four cases as for CAT. In all cases \( p_G' = p_G \) and so properties 2 and 3 are obvious and 4 follows from 1. Thus, only 1 and 5 need proof.

**Case I.** \( p_P' \) and \( p_Q' \) are both optimal. Here \( c(p_G') = c(P) + c(Q) \) and \( p_G' \) is nice. Thus 5 is proved. Property 1 follows from the fact that any partition for \( G \) better than \( p_G' \) can be used to produce a better-than-optimal partition for \( P \) or for \( Q \).

**Case II.** \( p_P' \) is not optimal, but \( p_Q' \) is optimal. Here \( c(p_G') = c(P) + c(Q) \). Property 1 follows as in Case I. To prove 5, suppose \( p_G' \) is not nice. Then \( p_Q' \) is not nice. An inductive use of 4 shows that any optimal nice partition for \( G \) yields an optimal nice partition for \( Q \), which is a contradiction, since \( p_Q' \) is not nice.

**Case III.** \( p_P' \) is optimal, but \( p_Q' \) is not. The proof is similar to case II.

**Case IV.** \( p_P' \) and \( p_Q' \) are both suboptimal. Here \( p_P' \) is nice by property 4 so \( p_G' \) is nice, proving property 5, and \( c(p_G') = c(P) + 1 \). Property 1 follows from the fact that any partition better than
$P'_G$ would yield an optimal top-open partition for $P$ or for $Q$, violating property 4 of the inductive hypothesis.

Finally, we consider the case that $G = R(P,t)$. Properties 2 and 3 are obvious. We have three cases:

**Case I.** $P'_G$ is optimal and nice. Clearly, $P'_G$ is nice, proving property 5. Since $c(P_G) = c(P'_P) - 1$, any partition for $G$ better than $P'_G$ would yield a partition for $P$ better than optimal. Hence property 5 is proved. The proof of 4 is similar.

**Case II.** $P'_G$ is optimal but not nice. In this case, $c(P'_G) = c(P'_G) = c(P)$. Property 4 follows from 1, which may be proved by arguments similar to case I above. Property 5 follows from the fact that an optimal nice partition for $G$ would imply an optimal nice partition for $P$.

**Case III.** $P'_G$ is not optimal. In this case $c(P'_G) = c(P'_P) + 1 = c(P) + 1$ and $c(P_G) = c(P)$. $P'_P$ is nice so $P'_G$ is nice and 5 is proved. Properties 1 and 4 follow by the usual arguments.

This completes the proof of Theorem 3.2. The algorithm is clearly linear in the length of the derivation of $G$, and hence in the size of $G$. 
4. BOUNDS ON COSTS

In this section, we derive upper and lower bounds on the cost of a restricted SFG.

4.1 Theorem

Let $G$ be a restricted SFG. Then

$$\max \{ e(G) - b(G) + 1, \lceil (e(G) + 1)/2 \rceil \} \leq c(G) \leq e(G) + 1$$

Before proving 4.1 we state and prove some preliminary results.

4.2 Lemma

If $p$ is any DAG partition of $G$, then $c(p) \geq e(G) + 1$; there is an algorithm to find a DAG partition such that $c(p) = e(G) + 1$.

Proof The usual code-generation algorithm produces a partition of cost $e(G) + 1$. The proof that this cost is the best possible is by induction on the construction of $G$.

If $G$ is a single node, the result is trivial. Otherwise, let $p_G$ be a DAG partition of $G$.

If $G = C(P, Q)$, then $p_G$ clearly decomposes into DAG partitions $P_P$ and $P_Q$ of $P$ and $Q$, respectively, such that $c(p_G) \geq c(p_P) + c(p_Q) - 1$. By the induction hypothesis, $c(p_P) \geq e(P) + 1$ and $c(p_Q) \geq e(Q) + 1$, so $c(p_P) \geq e(P) + 1 + e(Q) + 1 - 1 = e(P) + e(Q) + 1 = e(G) + 1$. 
If \( G = E(P,Q,i) \), then \( e(G) = e(P) + e(Q) + 1 \) and \( p_p \) can be decomposed into DAG partitions of \( P \) and \( Q \) such that \( c(p_G) \geq c(p_p) + c(p_Q) \). Once again, by the inductive hypothesis, \( c(p_p) \geq c(p_p) + c(p_Q) \geq e(P) + e(Q) + 1 = e(G) + 1 \).

If \( G = R(P,t) \), then there is a DAG partition of \( P \) such that \( c(p_G) \geq c(p_P) \). By induction, \( c(p_G) \geq c(p_P) \geq e(P) + 1 = e(G) + 1 \).

This proves lemma 4.2.

4.3 Corollary

For any partition \( p_G \) of an SFG \( G \),

(i) \( c(p_G) \geq e(G) - b(p_G) + 1 \)

(ii) \( c(p_G) \leq \lceil (e(G)+1)/2 \rceil \)

(iii) if \( c(p_G) = \lceil (e(G)+1)/2 \rceil \), then \( \lfloor (e(G)+1)/2 \rfloor \leq b(p_G) \leq \lceil (e(G)+1)/2 \rceil \)

Proof Deleting back arcs from \( p_G \) yields a DAG partition \( p_G' \) of cost \( c(p_G) + b(p_G) \). Hence, by Lemma 4.2, \( c(p_G) + b(p_G) \geq e(G) + 1 \), and (i) follows. To prove (ii), suppose \( c(p_G) \leq \lceil (e(G)+1)/2 \rceil \). Then \( b(p_G) \leq c(p_G) < \lceil (e(G)+1)/2 \rceil \), so by 4.2, \( \lceil (e(G)+1)/2 \rceil + \lfloor (e(G)+1)/2 \rfloor = e+1 \leq c(p_G') = c(p_G) + b(p_G) < \lceil (e(G)+1)/2 \rceil + b(p_G) < \lceil (e(G)+1)/2 \rceil + \lfloor (e(G)+1)/2 \rfloor + \lceil (e(G)+1)/2 \rceil \). Cancelling occurrences of \( \lceil (e(G)+1)/2 \rceil \) yields \( \lfloor (e(G)+1)/2 \rfloor < b(p_G) < \lceil (e(G)+1)/2 \rceil \), which is impossible.
The proof of part (iii) is the same as part (ii), except all occurrences of < should be replaced by ≤.

**Proof** of Theorem 4.1. The upper bound follows directly from the Lemma 4.2. One lower bound follows directly from 4.3(ii). The other lower bound is proved inductively:

If \( G \) is a single node, then \( e(G) - b(G) + 1 = 1 = c(G) \).

If \( G = C(P, Q) \), then

\[
\begin{align*}
\text{if } G & = C(P, Q), \text{ then } c(G) \geq c(P) + c(Q) - 1 \geq \\
& \geq (e(G) - b(G) + 1) + (e(Q) - b(Q) + 1) - 1 = (e(P) + e(Q)) - (b(P) + b(Q)) + 1 \\
& = e(G) - b(G) + 1.
\end{align*}
\]

If \( G = E(P, Q, i) \), then

\[
\begin{align*}
\text{if } G & = E(P, Q, i), \text{ then } c(G) \geq c(P) + c(Q) \geq \\
& \geq (e(G) - b(G) + 1) + (e(Q) - b(Q) + 1) = (e(P) + e(Q) + 1) - (b(P) + b(Q)) + 1 \\
& = e(G) - b(G) + 1.
\end{align*}
\]

If \( G = R(P, t) \), then

\[
\begin{align*}
\text{if } G & = R(P, t), \text{ then } c(G) \geq c(P) - 1 \geq (e(G) - b(G) + 1) - 1 = \\
& = e(P) - (b(P) + 1) + 1 = e(G) - b(G) + 1.
\end{align*}
\]

5. ADDING IF-THEN STATEMENTS

In this section we show that the results for restricted SFG's remain valid when **if-then** statements are added. Intuitively, the construction "**if B then S**" is modelled by "**if B then S else skip**". However, rather than introduce **skip** as a primitive concept, we model the **if-then** statement as \( C(i, P) \) (where \( i \) is a new node representing the condition \( B \) and \( P \) is the flow graph of \( S \)), and make \( i \) an additional exit node.
5.1 Theorem

Let $G$ be an SFG and $G'$ the corresponding restricted SFG. Any optimal partition $p$ of $G$ can be effectively transformed into a partition $p'$ of $G'$ such that $c(p') \leq c(p)$.

Proof (sketch). Call an arc $(i,n)$ a **forward arc** if $i$ is the node introduced by the operation $T = I(P,i)$, but $n$ is not $s(P)$ (see Figure 5). $G$ and $G'$ differ only in that forward arcs are present in the former and absent in the latter; hence, to transform $p$ to $p'$, we need only eliminate all forward arcs from $p$.

Choose an innermost forward arc $(i,n)$ used by $p$. Since the paths in $p$ are node-disjoint, $p$ does not use the arc $(i,s(P))$. That arc is the only arc entering the subgraph $P$, so $p$ can be decomposed into paths outside $P$ and paths inside $P$. The set of paths inside $P$ forms an optimal partition $P_P$ of $P$, so by Theorem 3.2, it may be replaced by an open partition $P'_P$ of $P$ with at most one more path. Modify the original partition of $G$ by replacing $P_P$ with $P'_P$. Then remove the path that uses $(i,n)$, say $u(i,n)v$, add $u$ to the top hook of $P'_P$, and add $v$ to a bottom hook of $P'_P$. (The latter operation is possible since the construction of an SFG ensures that any successor of any exit node of a subgraph is a successor of every exit node of that subgraph. Hence $n$ is a successor of each exit node of $P$.) This construction deletes the path $u(i,n)v$, so even if $c(P'_P) = c(P_P)+1$, the net increase in cost is zero.
5.2 Corollary.

If $p$ is an optimal partition for $G'$ then it is also an optimal partition for $G$.

6. TIGHTNESS OF BOUNDS

In this section, we show that the bounds derived in Section 4 are tight.

6.1 Theorem.

For any positive integer $e$, there are graphs $G_1$ and $G_2$ with branching factor $e$ such that $c(G_1) = e+1$ and $c(G_2) = \lceil(e+1)/2\rceil$. If $b$ is an integer such that $e-b+1 > \lceil(e+1)/2\rceil$, there is also a graph $G_3$ such that $e(G_3) = e$, $b(G_3) = b$, and $c(G_3) = e-b+1$.

Proof. Let $G_1$ be any graph with branching factor $e$ and no loops ($b(G_1) = 0$). By Theorem 4.1, $c(G_1) = e+1$.

Let $G_2$ be the graph $H_e$ defined in the introduction:

$H_0 = S_0$, $H_e = E(H_{e-1}, S_e, B_e)$ if $e$ is even, and $H_e = R(E(H_{e-1}, S_e, B_e))$ if $e$ is odd. Then the partition created by the algorithm is

$\{B_{2i}S_{2i}C_{2i+1}B_{2i+1}S_{2i+1} \mid 1 \leq i \leq \lfloor(e-1)/2\rfloor\} \cup \{S_0C_1B_1S_1, B_eS_e\}$

(The last path mentioned above is omitted if $e$ is odd.)
If $\lceil (e+1)/2 \rceil < e-b+1$, then $b < \lfloor (e+1)/2 \rfloor$. Let $G_3$ be $H_e$ with all but the $b$ innermost back arcs deleted. The partition of cost $e-b+1$ is the partition $p$ above, modified by removing the deleted arcs and splitting the paths that contained them in two.

7. SUMMARY AND CONCLUSIONS

Considering the amount of work that has been done on program optimization, it is surprising that more attention has not been paid to the problem tackled in this paper. Most literature on program optimization deals either with transformations on the flow graph of a program or with translation of an individual statement into machine code. The only other work we know of in this area is by Boesch and Gimpel [5]. They show that in the simple case that the flow graph is acyclic, the optimum partition problem can be reduced to the maximum matching problem for bipartite graphs. Hence any good algorithm for maximum matchings, such as the $O(n^{2.5})$ algorithm of Hopcroft and Karp [6], yields an algorithm for optimal partition of an acyclic flow graph. Since acyclic flow graphs are rare in practice, they present a heuristic algorithm for arbitrary graphs that proceeds by performing an interval analysis of the graph [3], finding optimal partition for the intervals, and pasting the partitions together. However, this procedure does not, in general, yield an optimal partition, and Boesch and Gimpel present no results on how close to optimal it comes.
Other related work involves investigations into the effect of long and short branch instructions on code length (see, for example, [7]) and the impact of restricting set of flow graphs to "structured programs" on program efficiency (for example, [8]).

The results here are only preliminary. We are currently extending the methods of this paper to cover other common constructs such as case, while and exit-loop statements. We conjecture that there is a polynomial algorithm for finding an optimal partition of any reducible flow graph.

8. REFERENCES


Figure 1

The Flow Graph of $H_6$
start:
  if not B₆ then N₆
L₅: if not B₅ then N₅
  if not B₄ then N₄
L₃: if not B₃ then N₃
  if not B₂ then N₂
L₁: if not B₁ then N₁
    S₀
    goto M₁
N₁: S₁
M₁: if not C₁ then L₁
    goto M₂
N₂: S₂
M₂: goto M₃
N₃: S₃
M₃: if not C₃ then L₃
    goto M₄
N₄: S₄
M₄: goto M₅
N₅: S₅
M₅: if not C₅ then L₅
    goto M₆
N₆: S₆
M₆: exit

L₀: S₀
M₁: if C₁ then M₃
L₁: if B₁ then L₀
  S₁
  goto M₁
L₂: if B₂ then L₁
    S₂
M₃: if C₃ then M₅
L₃: if B₃ then L₂
    S₃
    goto M₃
L₄: if B₄ then L₃
    S₄
M₅: if C₅ then M₇
L₅: if B₅ then L₄
    S₅
    goto M₅
start:
L₆: if B₆ then L₅
    S₆
M₇: exit

(a) (b)

Figure 2
Two Translations of H₆
Figure 3

SFG Operations

PC($P_P, P_Q$)

$P_P$ bottom-open and $P_Q$ top-open

otherwise

PE($P_P, P_Q$)

$P_P$ top-open

$P_Q$ top-open

PR($P_P$)

$P_P$ nice

otherwise

Figure 4

Operations for Combining Partitions
Figure 5

Eliminating Forward Arcs